2025 Design, Automation & Test in Europe Conference (DATE 2025)

RTHeter: Simulating Real-Time Scheduling of Multiple Tasks on Heterogeneous Architectures

Yinchen Ni[1], Jiace Zhu[1], Yier Jin[2], An Zou[1]
[1]Shanghai Jiao Tong University, [2]University of Florida

***Abstract*—The rising popularity of AI applications is driving the adoption of heterogeneous computing architectures to handle complex computations. However, as these heterogeneous architectures grow more complex, optimizing the scheduling of multiple tasks and meeting strict timing constraints becomes significantly challenging. Current studies on real-time scheduling on heterogeneous processors lack agile and flexible simulation tools that can quickly adapt to varying system settings, leading to inefficiencies in system design. Additionally, the high costs associated with evaluating real-time performance in terms of human and facility efforts further complicate the development process. To address these challenges, this paper introduces a comprehensive hierarchical simulating approach and a corresponding simulator designed for flexible heterogeneous computing platforms. The simulator supports ideal or practical, off-the-shelf or customizable heterogeneous architectures, upon which the simulator can execute both parallel and dependent tasks. Utilizing this simulator, we present two case studies that were time-consuming previously but are now easily achieved by the proposed simulator. The first case study reveals the possibility of using policy-based reinforcement learning to explore novel scheduling strategies; the second explores the dominant processors within heterogeneous architectures, providing insights for optimizing the heterogeneous architecture design.**

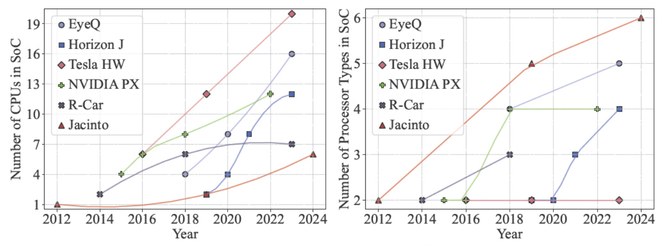
***Index Terms*—Simulator, Heterogeneous Architectures, Real-Time Scheduling, Reinforcement Learning**

I. Introduction

The rapid evolution of artificial intelligence (AI) applications has driven the demand for more sophisticated and diverse computing architectures. Modern real-time AI applications increasingly embrace extreme heterogeneity, utilizing systems with varying processor types to meet performance demands [1], [2]. This trend is particularly evident in the development of automotive system-on-chips (SoCs) for autonomous systems, as shown in Fig. 1, where the number of CPUs and the diversity of processor types have increased significantly in recent years [3]–[7]. As the number and variety of processors within a single SoC continue to expand, the challenge of accurately modeling and predicting system behavior becomes more and more critical.

Real-time AI applications, such as object detection, particle filtering, and audio recognition, often operate on a millisecond scale, where precise timing and responsiveness are crucial for the quality of services [8]–[11]. Fig. 2 shows the scheduling of these applications on heterogeneous computing systems involving different processors and task models. Traditional offline analytics methods, however, often lead to pessimism

An Zou is the corresponding author. This work is supported by NSFC 62202287.



(a) Number of CPUs (b) Number of Processor Types

Fig. 1: Automotive SoCs are embracing extreme heterogeneity: both in the number of CPUs and processor types.

and inaccuracy in estimating response times due to the growing complexity of these systems [12], [13]. This gap highlights the critical demand for more reliable and accurate tools and methods for predicting and evaluating timing performance for such systems.

Moreover, the non-trivial costs associated with real-time testing and validation—which require significant time, specialized skills, and facility resources—pose a substantial barrier to the efficient development and deployment of real-time applications on the heterogeneous system [14], [15]. To address these challenges, we propose a simulation approach and present a timing simulator dedicated to exploring scheduling algorithms and evaluating the real-time performance on customizable architectures with ideal and practical processor models. This simulator aims to provide an agile and practical solution for understanding and optimizing the performance of complex real-time heterogeneous computing systems. The contributions of this paper are four-fold:

- A hierarchical simulation approach is proposed for simulating the real-time scheduling of multiple tasks on heterogeneous architectures.
- A simulator, RTHeter, is developed, which supports both ideal and practical heterogeneous architectures with both parallel and dependent computation tasks.
- A case study demonstrates the potential of using policy-based reinforcement learning to discover innovative scheduling strategies.
- A case study explores the dominant component within heterogeneous architectures, providing insights into future heterogeneous architecture designs.

The rest of the paper is organized as follows. Sec. II summarizes the background and related works; Sec. III introduces the simulating hierarchy and Sec. IV discusses the design and implementation of the simulator. Sec. V presents two case studies utilizing RTHeter, and Sec. VI concludes the paper.

II. Background and Related Works

A. Heterogeneous Architectures and Computation Tasks

Heterogeneous computing architectures, which consist of CPUs and different accelerators, are gaining popularity in both embedded computing, such as the NVIDIA Jetson series, and high-performance computing, such as Oak Ridge's Titan supercomputer. These systems can improve performance at lower energy costs than homogeneous systems. Typically, when an application runs on a heterogeneous system, the CPU handles I/O and serial computation while parallel executions are offloaded to the accelerators (GPU, FPGA, DSP, etc.). To enable efficient data transfer between the CPU and peripherals, some heterogeneous platforms feature data copy engines (e.g., CE in NVIDIA GPU [16], [17]) to manage to copy data to the accelerators and return results to the CPU cores.

One of the mainstream task models on heterogeneous architecture computing is the Direct Acyclic Graph (DAG) task model [18], [19]. In this model, each task τ_i is represented by a graph $G_i = (V_i, E_i)$, as well as its deadline D_i and period T_i. Each vertex in V_i corresponds to a subtask execution time and its processor affinity, while each directed edge represents the constraints that a subtask can only be executed after the completion of preceding nodes.

The multi-segment self-suspension model [20] is a special case of the DAG task model, where the only dependencies are linked between each segment (analogous to subtask in DAG). A task τ_i has m_i execution segments and $m_i - 1$ suspension segments between the execution segments. Thus, task τ_i with deadline D_i and period T_i is expressed as a 3-tuple $\tau_i = \big((L_i^1, S_i^1, L_i^2, ..., S_i^{m_i-1}, L_i^{m_i}), D_i, T_i\big)$, where L_i^j and S_i^j are the lengths of the $j+1$-th execution and suspension segments, respectively. From the CPU perspective, the execution segments L_i^j are the computation segments, and the suspension segments S_i^j are the workload offloaded to accelerators. In this work, we will stick to the term "segment" to represent the subordinate task regardless of the task model.

B. Timing Simulation on Heterogeneous Architectures

To meet the blossom of the accelerator, many heterogeneous microarchitecture simulation frameworks are proposed. For example, GEM5 [21] is an architectural simulation infrastructure with a flexible CPU model, system mode, and memory system. GPGPU-Sim [22] models the parallel architecture in contemporary GPU and supports CUDA application simulation. Shao et al. present Aladdin [23], a pre-RTL, power-performance accelerator modeling framework, and demonstrate its application to system-on-chip simulation. Subhankar et al. propose HETSIM [24] for large-scale heterogeneous systems to estimate pre-silicon performance and power accurately.

In recent years, researchers have worked on event scheduling in heterogeneous architectures. For example, Dreimann et al. propose HetSim [25], a modular simulator for event-driven scheduling on heterogeneous hardware. Guan et al. present an RL-based scheduling algorithm applied to an FPGA

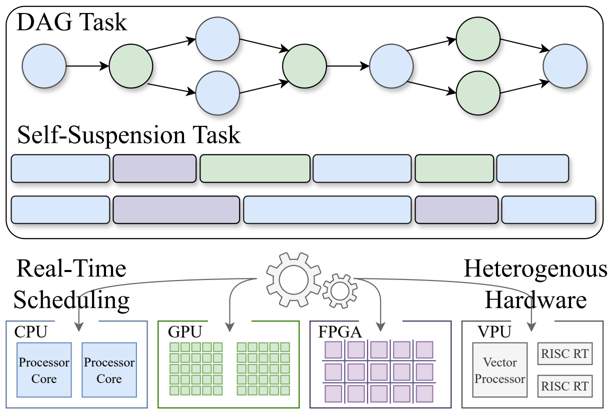


Fig. 2: Scheduling AI tasks on heterogeneous computing platforms, where tasks follow the DAG model.

to generate scheduling arrangements with limited resources [26]. However, most of these works focus on modeling the order of events, which is different from the real scheduler in the OS triggered at processor ticks. Thus, they ignore the ability to schedule parallel and runtime tasks with real schedule granularity. Therefore, bridging the gap between the hardware processors and runtime scheduling of parallel runtime applications, this work introduces RTHeter, a comprehensive scheduling simulator for parallel tasks executed on heterogeneous computing architectures.

III. Simulating Hierarchy and Temporal View

This section presents the simulating hierarchy for the real-time scheduling of heterogeneous computing platforms and the temporal simulation mechanisms.

A. Simulation Hierarchy

The proposed RTHeter simulates the **processor core-level** scheduling and execution of multiple tasks on heterogeneous architectures. As shown in Fig. 3, it bridges the application tasks with the underlying hardware. RTHeter supports both ideal and practical hardware processor models. In the ideal model, microarchitecture-level dynamics, such as cache misses and branch prediction, are ignored. Therefore, a processor finishes the same amount of computational workload given a fixed time interval and RTHeter can work as a standalone simulator without the extra runtime dynamics inputs. In the practical model, these dynamics are abstracted into statistical parameters, causing the computation speed to vary but be bounded by the microarchitecture-level statistics. In the practical model, the RTHeter is plugged with the micro-architecture simulators, such as GEM5 [21], GPGPUSim [22], and Aladdin [23], to obtain the microarchitecture-level statistical.

B. Temporal Simulation Mechanisms

As depicted in Fig. 2, the execution of multiple tasks in heterogeneous architectures naturally follows a hierarchical structure, where the real-time scheduler schedules computation tasks at higher levels to lower-level processors. The proposed RTHeter sequentially divides this hierarchical process into continuous simulating slices. The slice is a simulation cycle (stride), which is the schedule granularity and its length can be configured from tick to any user-specific time unit.

In each slice, RTHeter provides status of both the higher-level tasks and lower-level processors and renders an interactive interface for the median-level scheduler. Then, the question is: What essential activities and information should an effective scheduling process include? Fig. 3 categorizes the hierarchical process into the following three activity groups, organizing them in temporal order to form a complete slice.

- **Step I: Queries** In these activities, the median-level scheduler inquires information without making any modifications. To make effective scheduling, it is necessary to check the status of both higher-level tasks (τ_1, τ_2, $\dots$), and the lower-level processors (p_1, p_2, $\dots$). Therefore, our simulator supports querying each processor's status (busy or idle, preemptive or not, the current task executing, microarchitecture uncertainty, etc.) and the status of each task (period, deadline, execution progress, etc.).
- **Step II: Scheduling** This activity—which is the scheduling command from the median-level scheduler—comes with a 3-tuple parameter (i, j, k), representing "schedule processor i with task j segment k". Our simulator first checks the validness of the command, blocking issues such as preceding segments being uncompleted, the processor and segment affinity not matching or the processor not supporting preemption. If the scheduling is valid, RTHeter binds the status variable of the lower-level processor with a given higher-level task. Since there are N processors, there are at most N scheduling commands in each slice.
- **Step III: Executions** All the lower-level processors advance one simulation cycle and update the status of lower-level processors and high-level tasks in these activities. In each time simulation cycle, the scheduler may skip scheduling some processors (for example when the processor is non-preemptive or reserves the computing power for future tasks), therefore it should invoke this function manually to indicate the finishing of scheduling.

In summary, inside each simulation cycle, RTHeter first performs queries, then schedules tasks, followed by executing the tasks. After these steps, it advances to the next simulation cycle. This process continues until either the user-specified simulation time limit is reached or a task misses its deadline. At that point, the simulator generates corresponding traces and a summary report.

IV. Simulator Implementations and Usages

This section discusses the detailed implementation of the proposed RTHeter, to realize the simulation depicted in Sec. III. First, Sec. IV-A introduces the functionality flow of our simulator, and then Sec. IV-B further illustrates how we construct the internal attributes of each processor and task to achieve those functions. Last but not least, Sec. IV-C introduces the building-up structure of our simulator.

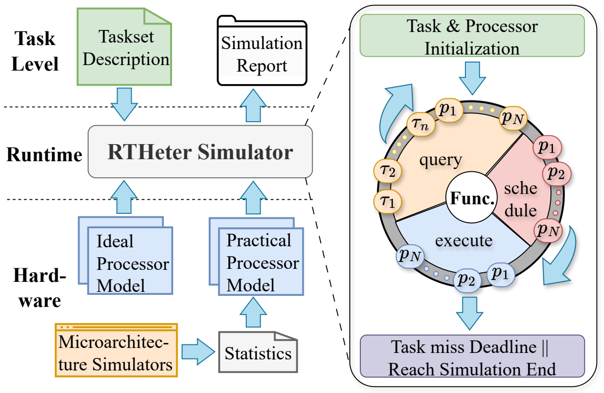


Fig. 3: Simulation hierarchy and working mechanism.

A. Functionality Overview

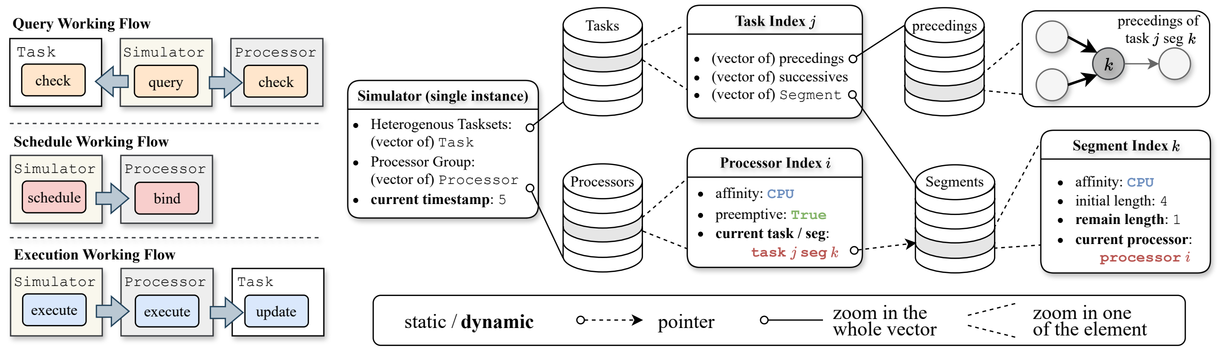
To standardize the program interface and maintain the degree of extensibility, we implement every key component in C++ classes: `Segment`, `Task`, `Processor`, and `Simulator`. Fig. 4b shows the layering and relationships between these classes in our simulator. On the very left, the `Simulator` is a class that works in a single instance incorporating the `Task` and `Processor` group in vectors. It mainly supports the four types of functions: ❶ `query`, ❷ `schedule`, and ❸ `execute`, as described in Sec. III, as well as additional interfaces for the ❹ creation and initialization of processors and task sets.

The class functions provided by `Simulator` will further invoke the underlying sub-function in the `Processor` or `Task` class. Fig. 4a introduces the call stack of the three major functions. ❶ `query` will invoke the `Task` checking, including the execution progress of tasks and whether the task is missing a deadline, etc. It also checks the processor status, such as idle or busy. ❷ `schedule` will further bind (as a decisive command) the given segment with this processor, while improper scheduling behaviors are rejected automatically. ❸ `execute` invokes the execute function of all the processors, which further invokes the update function of the task that binds to the processor in ❷. Of course, all the classes contain interfaces for initialization, which are called at the beginning of the simulation process.

B. Key Components Boiling Down

The realization of external functions is founded on the construction of internal properties. Inside the implementation of `Processor` and `Task`, we classify the attributes into two categories: static attributes describing fixed properties and dynamic attributes representing runtime conditions. Those internal attributes are essential for the logical correctness of all the external functions. Following this design scheme, we implement them as follows:

1) `Processor`: Each instance of a `Processor` simulates the runtime behavior of one processor core. A wide range of processors can be abstracted by two static attributes, its processor type (stored in enum) and whether supporting preemption (a boolean). We define two pointers as dynamic attributes to represent the working condition: `currentTask`



(a) Working flow of three major functionalities (b) Component inside the simulator, where the simulator contains a vector of tasks and processors and the task contains a vector of segments and dependency graphs

Fig. 4: Key functionality and component layering of the simulator.

and `currentSegment`. When the processor is scheduled with one task, these pointers change accordingly. Upon finishing the segment, two pointers are set back to null.

2) `Task`: Each `Task` instance simulates the execution of a task τ_i in the taskset, matching the DAG task model. Further breakdown: An essential component is modeling the behavior of a `Segment`. For the static part, the `Segment` class uses initial length and processor affinity to describe a segment. For the dynamic part, we use one variable to model the task execution progress and a pointer pointing to the current processor.

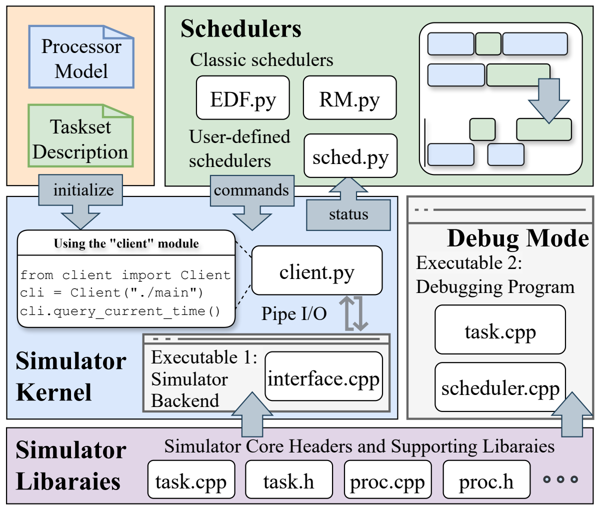
Treeifying the segments by dependencies forms a heterogeneous task τ_i. We utilize the vector to store the instance of segments inside the `Task` class. In addition, two vectors depict the preceding and successive dependencies for each segment. When a segment is completed, our simulator visits all its consecutive segments in the list, checks whether their proceedings are all completed, and marks them ready.

C. Simulator Building Up

One of the most critical components of the simulator is providing an easy-to-use but powerful user interface. Fig. 5 shows the architecture for building schedulers in our simulator, which contains three layers: Simulator Libraries, Simulator Kernel, and User-definable Schedulers. At the bottom layer, the Simulator Libraries comprise all the necessary headers and libraries, as introduced in Sec. IV-B, deriving two functional building paths: (i) the Debug Mode (debugging program) and (ii) the Simulator Kernel (the simulator backend).

1) Debug Mode: In Debug Mode, we directly compile two additional source files—a taskset generator and a scheduler—with the simulator libraries into Executable 2. This program performs single-case simulations, which is essential for developers to make further changes to the simulator libraries, debug potential issues, and verify their implementations.

2) Simulator Kernel: A full function path is compiling the library files with a tailored I/O handler (`interface.cpp`) into Executable 1. A corresponding Python frontend module in `client.py` encapsulates the simulator backend, by creating a subprocess and sending commands via pipe I/O. This module runs in a statically interactive manner in that it waits for the



```
from client import Client
cli = Client("./main")
cli.query_current_time()
```

Fig. 5: Building-up structure of our simulator, where users can choose classic schedulers or customize their own.

querying and scheduling from the upper-layer scheduler and returns the corresponding result or status.

The uppermost layer contains user input including task descriptions, processor models (ideal or practical models), and schedulers. Users can either choose classic schedulers such as Earliest Deadline First (EDF) and Rate Monotonic (RM) provided by our simulator or freely define their scheduler.

V. Application Case Studies

Utilizing RTHeter, we present two application case studies, which facilitate real-time scheduling in modern heterogeneous architectures. These two case studies are difficult or extremely time-consuming without RTHeter.

- Application 1: Training reinforcement learning based static scheduling agent for heterogeneous computing platforms with self-suspension tasks (Sec. V-A).
- Application 2: Benchmarking different schedulers and identifying the dominating processors in the heterogeneous architecture with DAG tasks (Sec. V-B).

A. Case 1: Reinforcement Learning for Static Scheduling

Researchers have been devoted to optimizing end-to-end real-time scheduling on heterogeneous architectures. Xu et al. [12] introduce a scheduling algorithm and response time analysis for multi-core CPU-based heterogeneous architectures, failing to incorporate data copy overheads. Saha et al. [13] introduce a software-hardware solution for efficient spatial-temporal scheduling for GPU, but the analytic pessimism is non-negligible. The existing methods are tailored to specific computing architectures and suffer from considerable analytic pessimism compared to real systems. However, our simulator is eligible to simulate general heterogeneous architectures and complex tasksets, upon which we can explore new-type schedulers. In this case study, we leverage RTHeter as a reinforcement learning environment and train a static real-time scheduling agent.

1) Problem Formulation: The problem of finding a static scheduling policy for the agent can be formalized using the framework of Markov Decision Processes (MDPs). The MDP starts with an initial state s_1, representing the initial status of all processors and tasks, the current task ID, and the segment ID to schedule. By following a policy $\pi(a|s) = p(a|s)$, the agent takes action a_1 representing the targeted processor for scheduling, receives a reward r_1 evaluating the quality of the scheduling behavior, and then transfers to state s_2. This learning process will continue until all tasks are well scheduled or any task violates its deadline, generating a sequence of states, actions, and rewards: $[s_1, a_1, r_1; \dots ; s_T, a_T, r_T]$, which is referred as an episode of length T. The agent updates the parameters θ of its policy $\pi_\theta(a|s)$ by policy gradient:

$$\theta \leftarrow \theta + \alpha \sum_{k=1}^{T} \nabla_\theta \log \pi_\theta(s_k, a_k) \left(\sum_{k'=k}^{T} r_{k'} \right),$$

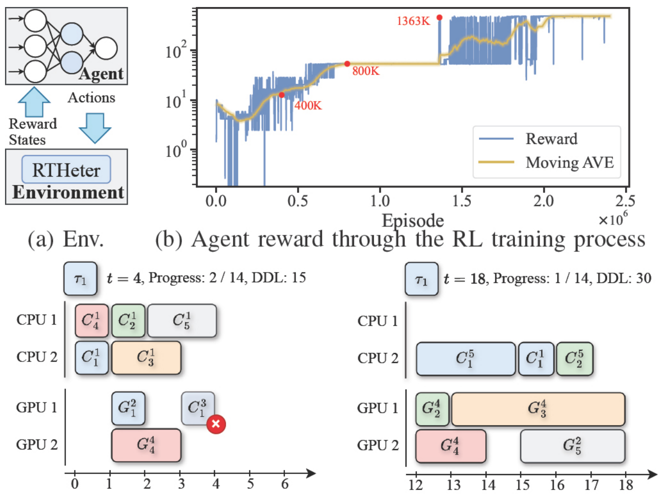
where α is the learning rate.

As shown in Fig. 6a, the action, reward, and state transitions are all supported by the RTHeter environment. We translate the agent action into `schedule` commands and encode the `query` results as state s_i. For the reward, three metrics returned from RTHeter evaluate the scheduling behavior: Does the agent decide to schedule or skip this round (`lazy`)? Is the scheduling legal or not (`legal`)? Does it cause preemption (`preemption`) or not? A linear combination of these metrics calculates the reward:

$$R = a \times \texttt{lazy} + b \times \texttt{legal} - c \times \texttt{preemption},$$

where a, b, and c are experimentally determined parameters.

2) Training Results: We train the agent under the hardware setting of 2 preemptive CPUs and 2 nonpreemptive GPUs with a fixed taskset consisting of 5 tasks under the self-suspension model and simulation time-bound is set to the least common multiple of all the tasks' periods. Hence, the action a_k will be an integer ranging from 0 to 4, where 1 and 2 represent scheduling on CPU 1 and 2 respectively, 3 and 4 represent scheduling on GPU 1 or GPU 2, and 0 stands for skipping this round. The agent is implemented by a 64×64 ResNet block



(a) Env. (b) Agent reward through the RL training process

(c) 400K-th episode: τ_0 misses deadline due to the agent attempting illegal scheduling (d) 800K-th episode: τ_0 misses deadline because the agent didn't reserve any GPU resource

Fig. 6: Reinforcement learning reward over episodes and two cross-sections representing the learning process of the agent.

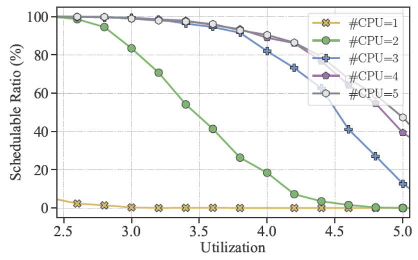
and multilayer perceptron (MLP) with 3 hidden layers, each including 48,32,16 neurons and rectified linear unit (ReLU) as activation function.

Fig. 6b shows the reward that our agent receives within the first million episodes, exhibiting a stair-like generally rising trend. Those stairs represent a critical time in the scheduling process, where a non-optimal choice may lead to a missed task deadline. Initially, the agent is likely to perform illegal scheduling, such as putting CPU segments (C_i^j represents the $j+1$-th CPU segment of τ_i) on GPU as shown in Fig. 6c. After several episodes, the agent advances to "survive" more time stamps. In Fig. 6d, the agent successfully handles the first release of τ_1, but scheduling is still not feasible. From $t = 13$ to $t = 17$, since the only ready GPU tasks are G_3^4 and G_5^2, our agent schedules them on the two GPUs. Unfortunately, those two segments are long and the GPU is non-preemptive, thus τ_1 has to wait until $t = 18$ to access the GPU, causing a deadline violation. The global EDF scheduler also fails to schedule this taskset for the same reason.

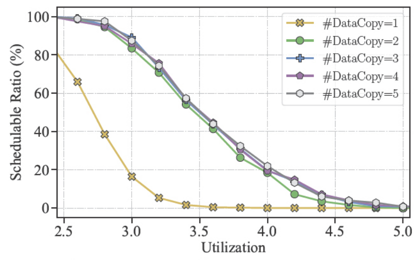
After 1363K episodes, our scheduling agent successfully identified a feasible scheduling solution. Utilizing RTHeter, we finish the simulation and reinforcement learning within 20 hours on an i7-11700 processor with NVIDIA GTX 1660 Super.

B. Case 2: Benchmarking and Identifying Dominate Proc.

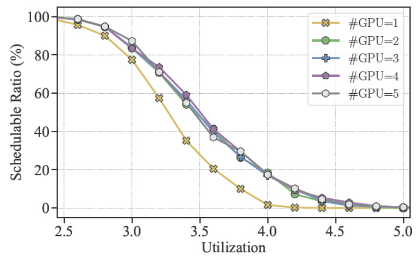
Agile benchmarking of scheduler performance and identifying dominant processors are critical for both processor architects and embedded software developers. Processor architects need to reveal the bottlenecks in terms of real-time schedulability and enhance architectural designs for future generations. Meanwhile, software developers need to test real-time scheduling strategies and optimize processor resource allocation efficiently. For instance, what are the minimal



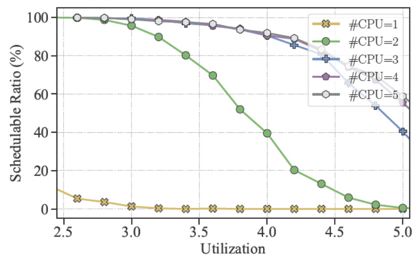
(a) RM: Number of CPU



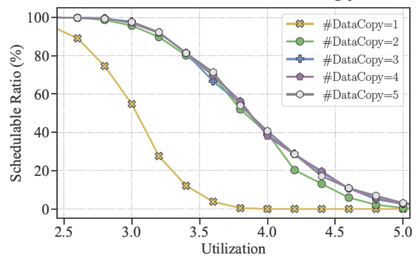
(b) RM: Number of Datacopy



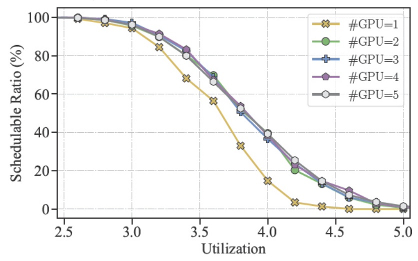
(c) RM: Number of GPU



(d) EDF: Number of CPU



(e) EDF: Number of Datacopy



(f) EDF: Number of GPU

Fig. 7: Synthetic schedulability of RM / EDF scheduling under various hardware configurations.

hardware resources required for a specific taskset to meet its deadlines? If the taskset cannot meet its deadlines, would adding CPU cores or integrating other accelerators be more effective? Since on-chip validation is often time-consuming and costly, our simulator provides a fast real-time verification tool for processor architects and software developers, allowing them to optimize computing platform designs or configurations to suit their tasksets.

We take the NVIDIA Jetson Serie's SoC as an example, which features multiple CPU cores, data copies, and GPUs. With our simulator, we can easily configure the hardware setting and scheduling strategies for pre-silicon tests. In this case study, we conduct extensive experiments to evaluate real-time scheduling on these architectures, focusing on the effectiveness of different scheduling algorithms and the benefits of increasing the number of processors with different types.

1) Setup: To present schedulability under different configurations and scheduling strategies, we measured the schedulable ratio with respect to targeted utilization, which is defined as the number of schedulable tasksets divided by the number of tasksets. We apply the same method to generate utilization of n tasks in existing works as in [12], [13], [27], and then we randomly generate all segment lengths, uniformly distributed within the range of 1 to 10. The deadline D_i of task i is set according to the generated segment lengths and its utilization rate. To form a DAG task, we use the same method as [28], which sets edge density p representing the possibility of having dependency between two segments and adds edges if necessary to ensure connectivity. The default heterogeneous computing platform has 2 CPU cores, 2 parallel data copies, and 2 GPUs. Table I presents a detailed summary of task generation.

2) Results: We first investigate the impact of the number of processor cores under the Rate Monotonic scheduling algorithm. The results are presented in Figs. 7a, 7b, and 7c, respectively. As the number of CPU cores, data copies, or GPU cores increases, the schedulable ratio increases correspondingly. However, the number of CPUs has a greater impact on schedulability than the other two processor cores. Compared to the case with only one CPU, the schedulable ratio doubles with one more CPU. A similar trend is exposed in Fig.7d-7f, but the EDF scheduler performs better than the RM scheduler since EDF is a dynamically prioritized scheduler while RM sticks to fixed priorities. In those figures, the increments converge after $N_{CPU} \geq 4$, $N_{Copy} \geq 2$ or $N_{GPU} \geq 1$. Therefore, we can conclude that a suited proportion between CPU, data copy and GPU cores is 4:2:1, given the premise of similar workloads.

TABLE I: Parameters for random task generation.

Parameters	**Value**
#Tasks in a taskset	$n = 5$
Task period and deadline	$T_i = D_i$
#Tasksets in each experiment	1000
Scheduling Algorithms	RM / EDF
Edge density	$p = 0.2$
Successor of CPU node	50% Datacopy, 50% CPU
Successor of Datacopy node	50% CPU, 50% GPU
Successor of GPU node	50% Datacopy, 50% GPU

VI. Conclusion

This paper introduces a comprehensive hierarchical simulating approach and a corresponding simulator tailored for flexible heterogeneous computing platforms. The simulator supports both commercial off-the-shelf and customizable architectures, capable of executing parallel and dependent tasks. Using this simulator, we present two case studies that were previously time-consuming but are now easily achievable. The first case study demonstrates the potential of policy-based reinforcement learning to explore new scheduling strategies. The second case study investigates dominant components within heterogeneous architectures, offering insights into optimizing architecture design.

References

[1] Jeff Anderson, Armin Mehrabian, Jiaxin Peng, and Tarek A El-Ghazawi. Extreme heterogeneity in deep learning architectures., 2019.

[2] Saibal Mukhopadhyay, Yun Long, B Mudassar, CS Nair, Bartlet H DeProspo, Hakki Mert Torun, M Kathaperumal, V Smet, Duckhwan Kim, Sudhakar Yalamanchili, et al. Heterogeneous integration for artificial intelligence: Challenges and opportunities. *IBM Journal of Research and Development*, 63(6):4–1, 2019.

[3] Renesas Electronics. Automotive products. https://www.renesas.com/us/en/products/automotive-products. Accessed: 2024-08-16.

[4] NVIDIA. Jetson orin. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/. Accessed: 2024-08-16.

[5] Horizon Robotics. Horizon journey series. https://en.horizon.cc/horizon-journey-series/. Accessed: 2024-08-16.

[6] Mobileye. Eyeq chip technology. https://www.mobileye.com/technology/eyeq-chip/. Accessed: 2024-08-16.

[7] Texas Instruments. Tda4vm. https://www.ti.com/product/TDA4VM. Accessed: 2024-08-16.

[8] Mingoo Ji, Saehanseul Yi, Changjin Koo, Sol Ahn, Dongjoo Seo, Nikil Dutt, and Jong-Chan Kim. Demand layering for real-time dnn inference with minimized memory usage. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 291–304, 2022.

[9] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[10] Falk Wurst, Dakshina Dasari, Arne Hamann, Dirk Ziegenbein, Ignacio Sañudo, Nicola Capodieci, Marko Bertogna, and Paolo Burgio. System performance modelling of heterogeneous hw platforms: An automated driving case study. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 365–372, 2019.

[11] Jinghao Sun, Jing Li, Zhishan Guo, An Zou, Xuan Zhang, Kunal Agrawal, and Sanjoy Baruah. Real-time scheduling upon a host-centric acceleration architecture with data offloading. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 56–69, 2020.

[12] Yuankai Xu, Tiancheng He, Ruiqi Sun, Yehan Ma, Yier Jin, and An Zou. Shape: Scheduling of fixed-priority tasks on heterogeneous architectures with multiple cpus and many pes. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[13] Sujan Kumar Saha, Yecheng Xiang, and Hyoseung Kim. Stgm: Spatio-temporal gpu management for real-time tasks. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–6. IEEE, 2019.

[14] Jaume Abella, Jon Perez, Cristofer Englund, Bahram Zonooz, Gabriele Giordana, Carlo Donzella, Francisco J. Cazorla, Enrico Mezzetti, Isabel Serra, Axel Brando, Irune Agirre, Fernando Eizaguirre, Thanh Hai Bui, Elahe Arani, Fahad Sarfraz, Ajay Balasubramaniam, Ahmed Badar, Ilaria Bloise, Lorenzo Feruglio, Ilaria Cinelli, Davide Brighenti, and Davide Cunial. Safexplain: Safe and explainable critical embedded systems based on ai. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2023.

[15] Shibo Chen, Yonathan Fisseha, Jean-Baptiste Jeannin, and Todd Austin. Twine: A chisel extension for component-level heterogeneous design. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 466–471, 2022.

[16] NVIDIA Corporation. *NVIDIA MIG User Guide*, 2024. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.

[17] An Zou, Jing Li, Christopher D. Gill, and Xuan Zhang. Rtgpu: Real-time gpu scheduling of hard deadline parallel tasks with fine-grain utilization. *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[18] Maria A. Serrano and Eduardo Quiñones. Response-time analysis of dag tasks supporting heterogeneous computing. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[19] Micaela Verucchi, Ignacio Sañudo Olmedo, and Marko Bertogna. A survey on real-time dag scheduling, revisiting the global-partitioned infinity war. *Real-Time Syst.*, 59(3):479–530, aug 2023.

[20] Wen-Hung Huang and Jian-Jia Chen. Self-suspension real-time tasks under fixed-relative-deadline fixed-priority scheduling. In *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2016.

[21] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[22] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE international symposium on performance analysis of systems and software*, pages 163–174. IEEE, 2009.

[23] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. *ACM SIGARCH Computer Architecture News*, 42(3):97–108, 2014.

[24] Subhankar Pal, Kuba Kaszyk, Siying Feng, Björn Franke, Murray Cole, Michael O'Boyle, Trevor Mudge, and Ronald G. Dreslinski. HETSIM: Simulating large-scale heterogeneous systems using a trace-driven, synchronization and dependency-aware framework. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 13–24, 2020.

[25] Marcel Lütke Dreimann, Birte Friesel, and Olaf Spinczyk. Hetsim: A simulator for task-based scheduling on heterogeneous hardware. ICPE '24 Companion, page 261–268, New York, NY, USA, 2024. Association for Computing Machinery.

[26] Y. Guan, Bd. Zhang, and Z. Jin. An frtds real-time simulation optimized task scheduling algorithm based on reinforcement learning. *IEEE Access*, 8:155797–155810, 2020.

[27] Lea Schönberger, Wen-Hung Huang, Georg Von Der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. Schedulability analysis and priority assignment for segmented self-suspending tasks. In *Conference on Embedded and Real-Time Computing Systems and Applications*. 2018.

[28] Zahaf Houssam-Eddine, Nicola Capodieci, Roberto Cavicchioli, Giuseppe Lipari, and Marko Bertogna. The hpc-dag task model for heterogeneous real-time systems. *IEEE Transactions on Computers*, 70(10):1747–1761, 2021.