Extending gem5 for Accurate PCIe-Based Device Modeling in Heterogeneous Architectures

Seunghyun Song
Electrical and Computer Engineering
Seoul National University
Seoul, Republic of Korea
seunghyun.song@snu.ac.kr

Junehyuk Boo
Electrical and Computer Engineering
Seoul National University
Seoul, Republic of Korea
junehyuk@snu.ac.kr

Yeongwoo Jang
Electrical and Computer Engineering
Seoul National University
Seoul, Republic of Korea
yeongwoo.jang@snu.ac.kr

Daye Jung
Electrical and Computer Engineering
Seoul National University
Seoul, Republic of Korea
daye.jung@snu.ac.kr

Jangwoo Kim
Electrical and Computer Engineering
Seoul National University
Seoul, Republic of Korea
jangwoo@snu.ac.kr

***Abstract*—The growing demand for heterogeneous computing in AI workloads has heightened the need for efficient CPU–accelerator communication. For this purpose, PCI Express (PCIe) has emerged as the dominant interconnect in recent systems. However, current architectural simulators provide limited support for accurate performance evaluation of PCIe devices. In this paper, we present a guideline to extend the gem5 simulator for arbitrary PCIe device integration. While gem5 is widely used for CPU simulation and architectural exploration, it only supports a limited set of PCI devices and lacks a standardized framework. By analyzing the mechanisms of gem5, particularly its support for AMD GPUs, we identify key components involved in communication: MMIO, DMA, and interrupts. Based on these insights, we propose a methodology for integrating arbitrary PCIe device simulators into gem5, enabling development of heterogeneous systems.**

I. Introduction

Modern computer systems have heterogeneous architectures, where CPUs operate together with a variety of accelerators and I/O devices. This trend is especially driven by the rapid growth of AI workloads, making data transfer between CPUs and I/O devices an important factor. Among the interconnects, Peripheral Component Interconnect Express (PCIe) is widely used as an essential interface for connecting devices to CPUs.

Simulators can play a crucial role in the architectural exploration of heterogeneous systems. They allow researchers to evaluate the system performance without the need for costly and time-consuming hardware implementation. Among various simulators, gem5 [1] is one of the most widely used CPU simulators in the computer architecture community. It has been extensively adopted in both academia and industry for its open-source nature and flexibility. It is renowned for its flexibility and extensibility, allowing users to model a wide range of CPU architectures.

However, modeling heterogeneous systems in gem5 still poses challenges. While gem5 provides a robust framework for CPU modeling, it lacks comprehensive support for PCIe-based devices. Although gem5 provides models for several I/O devices, they are limited to specific implementations. As new device simulators continue to emerge, integrating them into the gem5 ecosystem requires additional effort. Therefore, a general-purpose framework that allows seamless integration of arbitrary PCIe device simulators into gem5 is necessary.

In this paper, we present an extension to gem5 that enables accurate modeling of PCIe-based devices. To this end, we analyze gem5 CPU's communication mechanisms, with a particular focus on its internal support for AMD GPUs. Finally, we propose a methodology for integrating arbitrary PCIe device simulators into gem5.

II. Background

A. The gem5 simulator

gem5 is a widely used computer system simulator capable of modeling both microarchitecture-level and system-level behavior. It is mainly written in C++ and Python, allowing users to define complex system configurations and behaviors. gem5's simulation engine is built on a discrete-event simulation model, where events are scheduled to occur at specific times. It supports two main execution modes that enables efficient and reasonably accurate modeling of complex computer systems. First, system emulation (SE) mode emulates system calls for faster simulation but lacks accurate modeling of OS interactions. Second, full system (FS) mode boots a real OS using a disk image and kernel, providing more accurate system behavior at the cost of complexity and slower speed.

B. PCIe overview

PCIe is a high-speed interface for connecting various devices, such as GPUs, NPUs, NICs, SSDs, and other peripherals, to CPUs. It is composed of a root complex, PCIe switches, and endpoints. The root complex connects directly to CPU memory and manages PCIe traffic, while PCIe switches

allow multiple devices to share a PCIe hierarchy. Endpoints are devices that communicate with the root complex, such as GPUs. PCIe devices communicate with the CPU through mechanisms like memory-mapped I/O (MMIO), direct memory access (DMA), and interrupts.

Communications between the host CPU and a PCIe device typically involves two main phases. First, to efficiently communicate with the device, the device driver initially allocates DMA buffers in the host memory. To send a task to the device, the CPU writes commands in the command queue and signals the hardware with an MMIO write to the device memory (i.e., BAR region). This MMIO write is known as the doorbell, and it tells the device that new commands are ready. When the device receives the doorbell, it uses its DMA engine to read the commands in the command queue and begin the workload.

Next, once the device completes the task, it notifies the CPU. It does this by writing a completion record into a completion queue, also in the DMA buffer. To notify the CPU, the device can send an interrupt, which triggers the driver to read the result from the completion queue. For higher performance, the driver may skip interrupts and poll the completion queue instead.

C. PCIe model in gem5

Currently, gem5 implements PCIe devices through a PCI bridge-based model. Key existing models include NICs, IDE controllers, UARTs, DMA engines. These models are designed to work with gem5's existing CPU communication mechanisms, allowing for basic PCIe functionality. However, these models are often limited to specific implementations and lack general support for integrating arbitrary PCIe device simulators.

III. Motivation

A. Limitations of gem5's current PCIe model

In gem5, PCIe devices are modeled based on a simple PCI bridge, which does not accurately reflect PCIe architectures. The devices are directly connected to the I/O bus rather than PCIe's hierarchical, point-to-point topology. Although PCIe includes key components such as root complex, PCIe switches, and endpoints, gem5's current model lacks the ability to represent these components accurately. To model PCIe-based devices accurately, some recent studies have proposed works to extend gem5's PCI model for PCIe devices [2].

B. Integration needs for external PCIe device simulators

As heterogeneous architectures are becoming mainstream, communication between GPUs, NPUs, and NICs have a significant impact on overall system performance. In recent AI workloads, data transfer between the CPU and GPU becomes a clear bottleneck, highlighting the need for accurate PCIe modeling. While gem5 supports a few internal devices, it lacks a flexible interface for integrating external PCIe simulators. To address this problem, we propose a solution that enables architects to connect arbitrary PCIe simulators to gem5 with minimal modifications to its code.

IV. Proposed Solution

We analyze how gem5 supports AMD GPU integration in its full-system mode. A simple PyTorch GEMM application is executed. We trace log outputs and examine how gem5 binaries and Python files co-operate to instantiate and run the GPU system. We analyze the PCI setup sequence, MMIO mappings, and DMA interactions between the CPU and GPU. Then, we generalize this communication flow to create a reusable methodology for integrating PCIe devices into gem5.

A. gem5's AMD GPU integration

Recently, the full-system mode of gem5 was enabled for AMD GPU integration, where a real operating system is booted. While the CPU part of gem5 is accelerated by using the Kernel-based Virtual Machine (KVM) for fast execution, the GPU model is simulated in software.

```
system._dma_ports.append(gpu_hsapp)
system._dma_ports.append(gpu_cmd_proc)
system._dma_ports.append(system.pc.
    south_bridge.gpu)
for sdma in sdma_engines:
    system._dma_ports.append(sdma)
...
gpu_hsapp.pio = system.iobus.mem_side_ports
gpu_cmd_proc.pio = system.iobus.
    mem_side_ports
system.pc.south_bridge.gpu.pio = system.
    iobus.mem_side_ports
for sdma in sdma_engines:
    sdma.pio = system.iobus.
        mem_side_protocols
...
```

Listing 1. Python code for GPU setup

Initially, the GPU is connected to the CPU by binding necessary ports. Programmable I/O (PIO) ports are connected to the CPU's southbridge I/O bus for memory-mapped I/O (MMIO) access. Also, DMA ports are connected to the CPU's memory controller for data and parameter transfer.

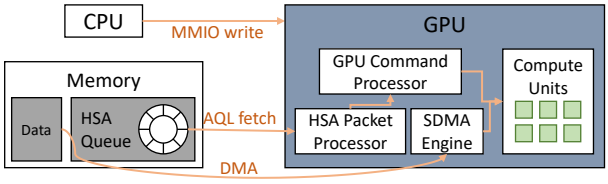

Fig. 1. gem5's AMD GPU kernel launch sequence.

After the GPU is connected, the communication between the CPU and GPU is established through MMIO and DMA. Figure 1 illustrates the communication flow during a kernel launch. The GPU has a Heterogeneous System Architecture (HSA) interface, which allows the CPU to manage GPU resources and launch kernels. The HSA queue is used to receive Architected Queuing Language (AQL) packets from the CPU for kernel execution. The HSA packet processor polls the HSA queue and fetches AQL packets by periodically performing PCIe DMA reads. The packets are then decoded by the GPU's

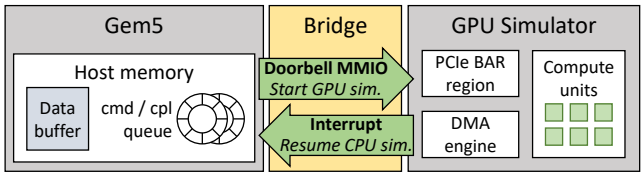

Fig. 2. Overview of our solution.

command processor, which executes the corresponding kernels to activate computing units. The data and parameters for the kernel execution are transferred from the CPU to the GPU through DMA. Finally, after the kernel execution the GPU send an interrupt to the CPU to notify completion.

B. PCIe device integration methodology

From the thorough analysis of gem5's AMD GPU model, we propose a general methodology for integrating arbitrary PCIe device simulators into gem5. Our approach is built around a new component within gem5 that acts as an interface, or bridge, between the host system and the external device simulator. Figure 2 shows the overview of our solution.

This bridge solves the two main challenges of integration: knowing when to start the simulation and when to signal its completion. First, to start the device, the bridge listens for specific doorbell MMIO writes from the CPU. When the bridge detects this signal, it commands the external device simulator to begin its task. Second, to handle completions, the device simulator notifies the bridge when it finishes. The bridge then informs the host CPU by sending an interrupt or writing to a completion queue. For DMA operations, our framework fully reuses gem5's existing memory system and the PCI model.

The modifications needed for the device simulator is to accept external commands from our gem5 bridge, such as `start_simulation()` and `check_status()`, to control its execution. In addition, it has to follow the gem5's DMA path instead of its own DMA model.

The following section describes the core architectural components and implementation details required for integrating such devices into gem5's full-system simulation framework.

1) Implementing the execution trigger: To start the device simulation at the correct time, our bridge model must first identify the specific doorbell MMIO write that triggers the hardware execution. We determine this doorbell address using one of two methods. First, the most direct way is by consulting the device's official specifications or an open-source driver, if available. These sources typically document the key register address. Second, if the device protocol is unknown, we use a tracing-based approach. We run a simple, known workload on the real hardware (e.g., that completes immediately) and use a tool like mmiotrace [3] to log all MMIOs. The doorbell is then identified as the specific MMIO write that acts as the trigger for the work. This trigger is the write that consistently occurs right before the device becomes active, and eventually sends a completion interrupt.

In either case, the bridge model monitors for a write to this specific address from gem5. When the doorbell write is detected, the bridge records its timestamp and starts the external device simulator to begin its execution.

2) Implementing device completion: Handling task completion follows a logic similar to the execution trigger. When the external device simulator finishes its work, it notifies the bridge model inside gem5. The bridge is then responsible for alerting the host CPU so it can proceed. This notification process involves two steps: (1) writing a completion record to the completion queue and (2) sending an interrupt to the CPU. Just like the doorbell, the exact completion protocol, such as the format of the completion record and the specific conditions for sending an interrupt, is determined either from device documentation and open-source drivers, or by analyzing traces from real hardware execution.

3) DMA operations: To support DMA operations, the device simulator should implement logic for initiating memory reads and writes to the CPU's memory. In gem5, this is performed using the dmaRead() and dmaWrite() functions, which operate over a master port (commonly named dmaPort). These functions take a memory address, a buffer pointer, and a completion event callback. The DMA engine must track outstanding transactions and signal completion either internally or through MMIO-accessible flags. The MemObject subclass typically includes a port connection:

```
1 Port &getMasterPort(const std::string &name
      , PortID) override {
2     if (name == "dma") return dmaPort;
3     return MemObject::getMasterPort(name,
          id);
4 }
```

Listing 2. Port connection in C++

Similarly, the slave ports should be registered and connected during system initialization.

V. Conclusion

This work presents an approach that allows arbitrary PCIe device simulators to be integrated into gem5 with minimal modifications to the code. Our methodology enables more accurate modeling of modern heterogeneous computing systems with PCIe-based devices. Future work includes validation of the proposed framework through detailed case studies and performance evaluations. By improving PCIe modeling in gem5, we move one step closer to enabling realistic simulation and performance analysis of modern AI-driven heterogeneous systems.

Acknowledgments

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean Government (MSIT) (No.RS-2024-00402898, Simulation-based High-speed/High-accuracy Data Center Workload/System Analysis Platform). We also appreciate the support from Automation and Systems Research Institute (ASRI) and Interuniversity Semiconductor Research

Center (ISRC) at Seoul National University. The EDA tool was supported by the IC Design Education Center (IDEC), Korea.

REFERENCES

[1] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," 2020. [Online]. Available: https://arxiv.org/abs/2007.03152

[2] M. Alian, K. P. Srinivasan, and N. S. Kim, "Simulating pci-express interconnect for future system exploration," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 168–178.

[3] "mmiotrace: In-kernel memory-mapped i/o tracing." https://www.kernel.org/doc/Documentation/trace/mmiotrace.txt.