A Dynamic Linking Framework for Efficient QEMU Peripheral Development and Maintenance

Gihyeon Jeon, Jisu Kwon and Daejin Park*
School of Electronic and Electrical Engineering, Kyungpook National University, boltanut@knu.ac.kr

***Abstract*—When developing a new system, preliminary verification can be performed using an emulator before actual hardware production. Among the emulators needed in the development process, QEMU is the most well-known. Although defining and implementing a new peripheral's operation can be challenging, in this paper, we propose using dynamic linking to implement the peripheral's operation conveniently. Specifically, by connecting a separate program called NeuroSim to QEMU through dynamic linking, we can emulate a system in which a separate accelerator is connected to the MCU. Moreover, when expanding QEMU in this way, even if modifications to the peripheral's operation are required, it is better in terms of time and convenience because only the shared object file needs to be compiled without rebuilding the entire QEMU. Our study demonstrates that building with dynamic linking resulted in a reduction of approximately 91% in time compared to the build process without dynamic linking.**

I. Introduction

A. QEMU

QEMU is a generic and open source machine emulator and virtualizer [1]. It can be used in several ways, but in our study, it is used for system emulation, in which it provides a virtual model of a machine. Although this machine has some built-in peripherals by default, there are also peripherals that are not implemented or special peripherals that users might need. Therefore, research has been conducted on development tools for implementing new peripherals [2].

To define a new peripheral in QEMU, we need to modify a json file and write the peripheral's operation in C code. The json file is a modified version of the SVD file provided by the chip vendor. This file contains information about various registers and fields. Therefore, we add the new peripheral's registers and fields in this file and implement the peripheral's operation in a separate C code.

B. NeuroSim

NeuroSim is an intergrated framework, which is developed in C++ and wrapped by Python to emulate on-chip performance on the hardware accelerator based on near-memory computing or in-memory computing architectures [3], [4], [5]. Specifically designed for hardware accelerators based on near-memory computing and in-memory computing architectures, NeuroSim provides hierarchical design options spanning device-level, circuit-level, and algorithm-level configurations. The framework supports weight-duplication for optimizing on-chip memory utilization and accommodates various neural network topologies, enabling comprehensive benchmarking from VGG-8 to ResNet architectures across datasets ranging from CIFAR to ImageNet. Furthermore, NeuroSim interfaces seamlessly with popular machine learning platforms such as PyTorch and TensorFlow, providing detailed hardware performance metrics including area estimation, latency analysis, dynamic energy consumption, and leakage power assessment.

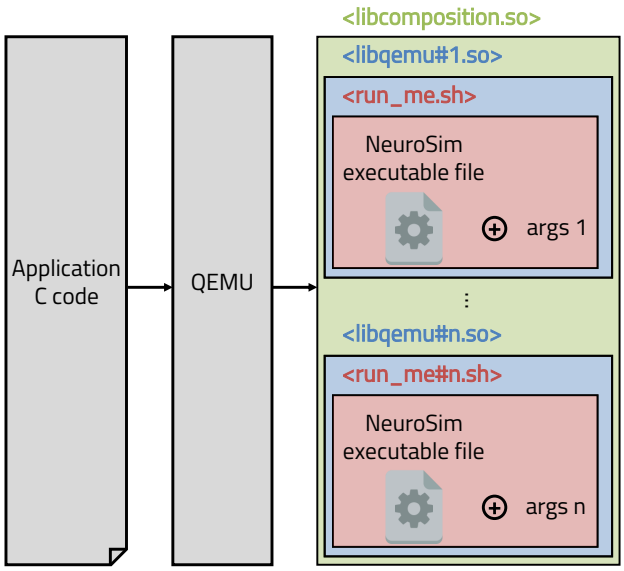

Fig. 1: Structure of shared object files

In our study, we aimed to define NeuroSim as a new QEMU peripheral for efficient accelerator development. Through the implementation of dynamic linking, we have enhanced the development process by reducing component coupling and making it more robust. This approach provides a foundation for future AI accelerator design by enabling efficient integration and testing of NeuroSim's hardware design capabilities within the QEMU environment.

II. Proposed Method

A. Implementation Using Dynamic Linking

Two major dynamic linking processes are required. The first dynamic linking process occurs during NeuroSim operation. To run NeuroSim from QEMU's code, the executable

This study was supported by the BK21 FOUR project (4199990113966), the Basic Science Research Program (NRF-2022R1I1A3069260, 10%), (NRF-2022R1I1A3069260, 10%) through the National Research Foundation of Korea (NRF) funded by the Ministry of Education. This work was partly supported by an Institute of Information and communications Technology Planning and Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2022-0-01170, PIM Semiconductor Design Research Center, 30%) and (No. RS-2023-00228970, Development of Flexible SW-HW Conjunctive Solution for On-edge Self-supervised Learning, 50%). The EDA tool was supported by the IC Design Education Center (IDEC), Korea.

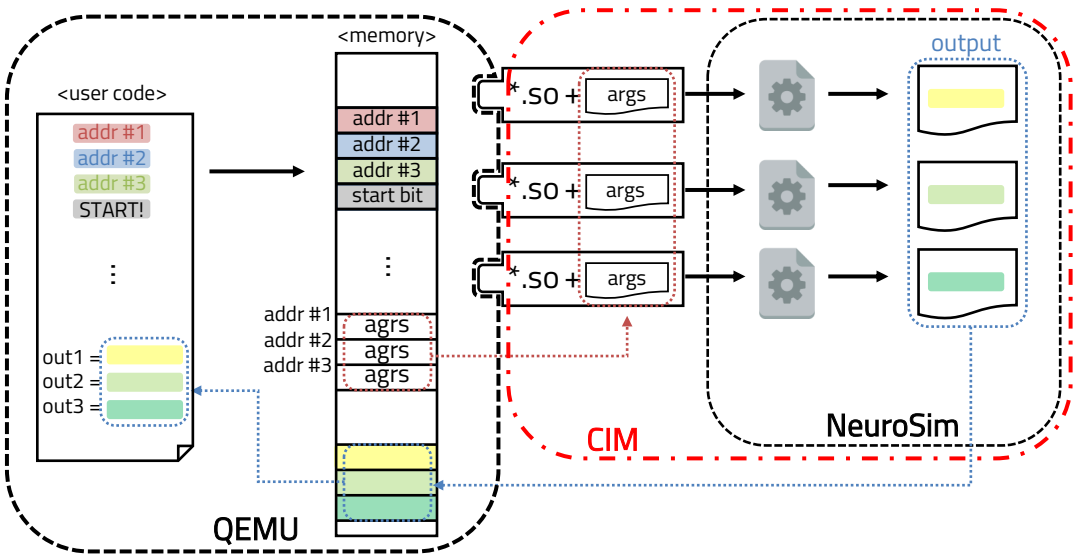

Fig. 2: Overall structure of proposed method

file generated when building NeuroSim must be executed with parameters to be processed in NeuroSim. For this, a C function that calls the external executable file with parameters needs to be implemented and compiled into a shared object. This shared object is represented as the libqemu.so file in Fig. 1.

The parameters passed to Neurosim by users can vary, for they represent the values to be computed in the peripheral. As a result, multiple libqemu.so files exist, each combining the same executable with different parameters. The second dynamic linking is used to call the functions that perform desired computations from these libqemu.so files at the intended execution points. When functions from the second shared object file are called at specific points in QEMU, these functions gain the ability to invoke functions from the libqemu.so file. The second shared object file is represented as libcomposition.so in Fig. 1.

B. Order of Operations

The application code the users write will be operated on the microcontroller. First, the user inputs the start address and end address of the weight array defined in the application code into the WSR (weight start register) and WER (weight end register) of the CIM peripheral, respectively. When "1" is written to the control register, it creates a separate CSV file containing the weight array that exists at the address the user previously wrote to the WSR and WER. By applying this CSV file as an argument to the NeuroSim program, users can apply the weights they desire. The results of NeuroSim operating this way are stored in the ODR (Output Data Register) of the CIM register, and users can access this register to produce the desired results.

Fig. 2 illustrates the content mentioned in this chapter. The arguments written at addresses specified in the user code are set as arguments for another shared object file by the shared object called CIM, which is the same as in the libcomposition.so file in Fig. 1. This CIM shared object (libcomposition.so) executes various shared objects depending on the case, and corresponds to the libqemu.so files in Fig. 1. The multiple shared objects (libqemu.so) operating within CIM (libcomposition.so) correspond to the first dynamic linking mentioned in subsection A. These various shared objects can obtain output by executing the corresponding executable files for each case.

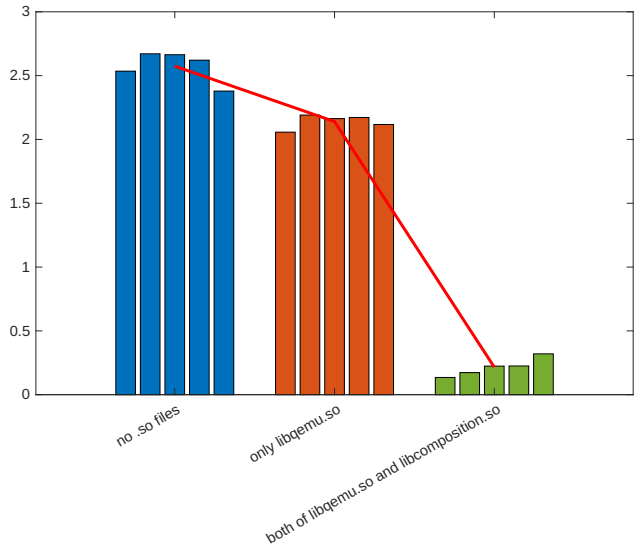

Fig. 3: Comparison of build time

III. Experimental Results

We compare three cases, as shown in Fig. 3:firtst, the case without dynamic linking in which the neurosim source code is included in QEMU (blue bar); second, the case using dynamic linking with only libqemu.so (orange bar); and finally, the case using dynamic linking libqemu.so and libcomposition.so files (green bar). For each case, the build time was measured 5 times, with a new neurosim operation added in each iteration.

The red line in Fig. 3 connects each case's average values, which are 2.574s, 2.14s, and 0.2154s. This demonstrates that

utilizing dynamic linking enables faster builds during code development, reducing build time by approximately 91% compared to the build without dynamic linking. Additionally, it shows that the separation between QEMU source code and the peripheral source code to be implemented becomes possible, making it more advantageous for maintenance.

IV. Conclusion

Adding a new peripheral to an existing microcontroller has financial and time-related burdens, and failure in this process can result in significant losses. Therefore, emulation during the development phase is essential, but adding a new peripheral to the existing QEMU can also be a significant challenge.

Using dynamic linking, as proposed in this study, enables more convenient addition of peripherals. Furthermore, we demonstrated that when an executable file already exists from external implementation, creating a separate function to call that file allows it to be dynamically linked to QEMU at runtime, implementing the desired functionality. Along with this convenience, the results showed that the build time was reduced by approximately 91% compared to cases without dynamic linking.

References

[1] D. Gutierrez, W. Ocampo, A. Perez-Pons, H. Upadhyay, and S. Joshi, "Virtualization and validation of emulated stm-32 blue pill using the qemu open-source framework," in 2023 11th International Symposium on Digital Forensics and Security (ISDFS), 2023, pp. 1–6.

[2] V. Efimov and V. Padaryan, "Peripheral device register support for source code boilerplate generator of qemu development toolkit," in 2018 Ivannikov Memorial Workshop (IVMEM), 2018, pp. 36–39.

[3] X. Peng, S. Huang, Y. Luo, X. Sun, and S. Yu, "Dnn+neurosim: An end-to-end benchmarking framework for compute-in-memory accelerators with versatile device technologies," in 2019 IEEE International Electron Devices Meeting (IEDM), 2019, pp. 32.5.1–32.5.4.

[4] J. Lee, A. Lu, W. Li, and S. Yu, "Neurosim v1.4: Extending technology support for digital compute-in-memory toward 1nm node," IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 71, no. 4, 2024, pp. 1733–1744.

[5] P.-Y. Chen, X. Peng, and S. Yu, "Neurosim+: An integrated device-to-algorithm framework for benchmarking synaptic devices and array architectures," in 2017 IEEE International Electron Devices Meeting (IEDM), 2017, pp. 6.1.1–6.1.4.