Rapid Simulation Tool for Both Software and Hardware Performance Analysis and Evaluations on Heterogeneous Platforms

In heterogeneous or composable systems, an application might change its behavior during the runtime based on the available computing resources in the system [1][2][3][4]. For example, in order to integrate accelerators and big.LITTLE [5] cores, ARM has introduced dynamiQ [6] as a way to optimize the energy efficiency of a program by providing machine learning specific instructions and also by offering cluster flexibility within a mix of up to 8 different processors. The composability of big.LITTLE cores, the flexibility of its power management, and a new dedicated port for accelerators together bring this application to a new level of performance efficiency. However, understand the performance and energy profiles at all levels, from the application to the hardware, is the key to optimize the entire system, and our research team has been developing methodologies and tools to assist the system developers in this regard.

Before the actual hardware products, e.g. a new generation of processors, become available for application developers, simulation tools are often used to perform the hardware functions and model the performance of the hardware components. However, a detailed full-system simulation with accurate timing models is extremely time-consuming for complicated applications and is not intended for performance profiling purposes. To be used for application-level performance profiling and optimization on developing heterogeneous multicore platforms, we intend to provide a simulation tool that is fast enough for the user to perform the entire application and evaluate design alternatives in time to shorten the development cycles. Unlike conventional simulation tools, we want the simulation tool to provided advanced performance analysis capabilities by identifying repeating program phases, finding major performance bottlenecks in an application, and suggesting ways to accelerate the program. With such capabilities, we hope to pave the road to advanced dynamic optimization for time-varying phase behavior due to hardware resources [8][9][10][11], which has been the weaknesses of static compiler optimization methods.

As shown in Figure 1, we have designed a simulation tool, called Snippets, which leverages the cycle-approximate timing simulation tool on QEMU [7] to achieve a fast simulation and provide performance counters for online phase detection. There are four main components in our proposed design. With all three components, we propose a phase profiling mechanism to help a programmer optimize the program or help the runtime to optimize the resource for the application.
1. **Timing Simulation** provides the counters and timing and feeds the calculated wall clock time to the emulated guest environment.

2. **Event Tracker** provides the ability to monitor and track all the function calls and events in both kernel and user space.

3. **Phase Detector** collects the counter and timing information when a window is formed, then classifies the phase and pushes the information into the phase.

4. **Device Driver** provides an interface to guest applications and controllers for the guest system to access our Virtual Performance Monitoring Unit (VPMU) as if it is an I/O device. The communications between the guest and the host system are done via pre-defined shared memory space.

### Enabling Techniques of Timing Simulation

As depicted in Figure 2, VPMU is the coordinator which controls all the components in the system. VPMU is responsible for issuing synchronizations, collecting performance counters, and communicating with the helper applications on the guest system.
Multi-model ring buffer (MMRB) [14]

In our experience, QEMU can reach around 600 MIPS for executing native instructions without simulating any hardware component. For QEMU to model certain hardware components on the fly, simulation (timing) models for the hardware components must be attached. We have developed a scheme called Multi-model ring buffer (MMRB) for the user to attach timing models with optimized throughput for single-writer-multiple-reader communication patterns. The design of MMRB uses local buffer to improve private cache performance as well as the memory performance with bulk data transfer. With MMRB, users can choose which timing models to be used and how many different configurations to be explored in run time. In addition, user can also hook an external process (timing simulator) onto our simulation tool with shared memory techniques provided by Linux. The results of simulation speed with MMRB reaches 45 MIPS which is around 40 times faster than timing accurate simulators.

Just-In-Time Model Selection [14]

Since many data-intensive programs have regular data access patterns, we designed a scheme called just-in-time (JIT) model selection to accelerate the simulation by switching a timing accurate model to an approximated model, which could yield higher simulation speed with approximated performance estimation. By identifying the hot basic blocks during the simulation, we can roughly catch the hot regions (usually short loops) and use approximate timing models to those hot regions for speed. With a simple arithmetic model and fallback mechanism when it’s not a regular accesses, the JIT model selection reaches 70 MIPS of simulation speed. Furthermore, when compiling with Intel icc, the simulation speed under the same configurations can reach 100 MIPS. In some special cases (target applications), the performance was even up to 160 MIPS (recall that native QEMU is

---

1 The workload tested here is matrix multiply application with size 200 by 200
The experimental results were running on Intel i7-4790k CPU with gcc version 6.3.1 and icc version 2016.3.210; the optimization flag was -O2. The benchmark suit was MiBench [16], a representative embedded benchmark suit, and we further used matrix multiply to demonstrate the memory intensive signal processing applications. For a fare comparison, we used MIPS as the basic unit instead of execution time since the emulation speed varies with application behaviors. Also, to test the scalability of our MMRB, we run multiple cache configurations at the same time for each test, scaling from 1, 2, 4, 6, to 8. As shown in Figure 3, with our MMRB, concurrent simulation of eight cache configurations was improved by three times comparing to direct pass implementation. With JIT model selection, the simulation speed was further increased by 2.6 times with negligible errors on the simulation results. Combining the proposed schemes, our framework provided 7.8 times of speedup which can improve the speed of DSE process dramatically.

Event Tracker

By parsing the symbol table from vmlinux, the binary of a Linux kernel without compression, before emulation, the event tracker builds a table of function addresses. We use the softMMU in QEMU to translate the guest virtual addresses to the host virtual addresses for reading the values of input arguments in a function call. With function call and input arguments, we can track important kernel events, such as context switches. We also implemented the callbacks to perform the tracking of the following events for phase detection: __switch_to, mmap_region, do_execv, wake_up_new_task, do_mmap, and do_exit.

With the capabilities of monitoring both kernel events, the event tracker provides break points for collecting performance data. If the binary of the profiled binary is presented, event tracker can read its symbol table and use it to track user function calls/returns. With all the capabilities, one can profile a program with full function call stacks and performance counters of each function.
Phase Detection

We utilized the power of phase detection algorithms in [10] to explore the important phases of profiled applications/systems using performance information gathered from our virtual platform. To help a developer find program bottlenecks rapidly, we implemented a profiling mechanism that can (1) explore different hardware parameters, (2) evaluate the impact on a program phase if hardware changes, (3) discover the causes of bottlenecks and the possibility of using accelerators, and (4) understand the program's behavior.

To showcase the advantages of program phases, we conducted an experiment to find the program phases which are friendly to GPU, i.e., might gain a performance boost on GPU. We used the Rodinia Benchmark Suite [13], a representative benchmark for heterogeneous accelerations, to train the machine learning model. Nineteen OpenCL-based programs were selected from the benchmark suite and were run on the evaluation platform, which was a desktop computer equipped with Intel CPU i5-4790, 16GB DD3 Memory, and Nvidia GPU GTX780, where the OpenCL kernels were accelerated by the GTX 780. We compared the delivered performance of the OpenCL
and non-OpenCL (serial) versions, and labeled the OpenCL kernels (accelerated regions) as **GPU-friendly** if the parallel version is 2x faster than the serial version; otherwise, they were labeled as **non-GPU-friendly**. In summary, there were 371 training data, consisting of 139 GPU-friendly data and 232 GPU-non-friendly data in the data set.

We used the *F-measurement* as the metric to evaluate the performance of the built model. The results show that utilizing phases as the basic units of recording performance counter events achieves 90.56% of accuracy and outperformed the results from the referenced design that was 80% [12]. *Precision* shows that 87.68% of the phases which was predicted as GPU-friendly were actually GPU-friendly, while *Recall* shows that 87.05% of the total numbers of actual GPU-friendly phases were found.

**CPU-GPU Co-Simulation [15]**

To provide a simulation environment for heterogeneous systems, we leverage the *Multi2Sim* framework to account for the timing of GPU applications. With the Multi2Sim binary toolchain and the Multi2Sim OpenCL runtime library, the OpenCL programs that can be executed by the Multi2Sim framework can also run on our proposed framework. As shown in Figure 4, we implemented three key components to achieve the fast co-simulation with physically shared memory which is like an integrated GPU in ARM dynamIQ [6].

1. *Virtual IOCTL* is designed to replace the Multi2Sim GPU Driver. The design of Virtual IOCTL avoids the need of using system calls to manipulate the GPU. In fact, it is done by creating a QEMU hardware device on the emulated hardware platform and using memory mapped I/O to control GPU from the modified Multi2Sim runtime library.
2. *GPU Plugin Module* is responsible for interfacing with the GPU simulators. It acts as the adaptation layer which translates the OpenCL operations taken from the Virtual IOCTL into the commands that accepted by GPU simulators.
3. *Memory Access module* determines the GPU is allowed to access to either the shared memory or the private memory. Together with the support of Virtual IOCTL and GPU Plugin Module, the applications that conform OpenCL standard are able to run on the proposed emulation system.
Illustrative Case Study

To illustrate the performance and accuracy of our simulation framework, we choose a scalable computing intensive program, *kmeans*, which is an algorithm for cluster analysis as the target application. As illustrated in Figure 5, our CPU-GPU co-simulation framework achieves 6.6x of speedup in average. Breaking down the main components of simulation, the emulation time of our framework consists of CPU, GPU, race-detection, and memory subsystems. For Multi2Sim, the emulation time of executing a program includes the dynamic library loading which took about 2 seconds for eight shared libraries, the CPU emulation time which took the majority of emulation time, and the GPU emulation time which should be similar to ours. The CPU emulation time varies depending on how many codes were executed on CPU side which is hard to formally and fairly compare with, especially Multi2Sim is not a full system emulator.

Reference:

About the Author

Chih-Wei Yeh, Ph.D. Candidate
Shih-Hao Hung, Professor
Department of Computer Science and Information Engineering
National Taiwan University