J Sign Process Syst (2019) 91:247–259
https://doi.org/10.1007/s11265-017-1285-z

Rapid Hybrid Simulation Methods for Exploring the Design Space of Signal Processors with Dynamic and Scalable Timing Models

Chih-Wei Yeh[1] · **Chia-Heng Tu**[2] · **Shih-Hao Hung**[1]

Received: 15 November 2016 / Revised: 11 May 2017 / Accepted: 14 September 2017 / Published online: 27 September 2017

Abstract As today's state-of-the-art signal processing systems often require heterogeneous computing and special-purpose accelerators to offer highly efficient performance for mixed application workloads, including not only traditional signal processing algorithms, but also the demands to enable smart applications with data analytics, machine learning, as well as the capability interacting with both physical and cyber worlds via sensors and networks. Thus, the complexity of such systems has been increasing, and the focus of designing has been shifting to exploring the design space with a mixture of processing cores/accelerators and the interconnection networks between the components to optimize the performance and efficiency at the system level. Traditional simulation tools may offer accurate performance estimation at micro architectural level, but it is highly complicated to combine the simulators for various components to perform complex applications, and they fall in short in terms of their capabilities to profiling application workload. Furthermore, the speed of such complex simulation would be unacceptably slow with traditional system-level simulation framework such as SystemC. To solve the problem, we develop a rapid hybrid emulation/simulation framework that allows the user to execute full-blown system and application software and plug in emulators, simulators, and timing models for various components in the prototype system, switching the timing models dynamically with our just-in-time model selection mechanism, and connect the emulated/simulated components with scalable communication channels, so that the framework can be accelerated effectively by a multicore host. Our just-in-time model selection mechanism is capable of detecting and skipping regular program patterns to save the simulation time dramatically. In addition, our framework is capable of estimating the performance of different system configurations with concurrent multiple timing models, which further saves the time needed for traversing the design space. Our experimental results have shown that our dynamic model selection and multi-model approach collectively can speed up the design space exploration by 13.4 times on a quad-core host for cache simulation.

Keywords Embedded system · Efficient data transfer · Simulation · Approximate timing model · Acceleration · Design space exploration

✉ Chih-Wei Yeh
medicinehy@gmail.com

Chia-Heng Tu
chiaheng@mail.ncku.edu.tw

Shih-Hao Hung
hungsh@csie.ntu.edu.tw

[1] Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan

[2] Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan

1 Introduction

Multicore systems are widely used in image processing and signal processing areas for various complex tasks such as recognizing human faces and voice dictations. With the emergence of smart *Internet of Things* (IoT) applications, today's signal processing systems demands additional capabilities such as data mining, deep learning and

collaboration with other computing devices on the Internet, which pose new challenges as it requires tightly coupled heterogeneous processors and application-specific accelerators for more computing power while keeping low power [35, 40]. Furthermore, as the complexity of such systems has been increasing, the focus of designing systems has been shifting to exploring the design space with a mixture of heterogeneous processor cores/accelerators and the interconnection networks between the components to optimize the performance and efficiency at the system level. For each targeted application domain, the designer needs to combine different types of processor cores and makes design tradeoffs on caches, memories, and the interconnections between heterogeneous cores and accelerators.

However, the question is, how to *co-design the application and the system*? First, such a work would need a tool that enables the designer to write/execute a complex application on a system that is still being developed. Secondly, the tool should provide abundant and fairly accurate information to reveal hardware-software interactions by showing relevant software and hardware events. Thirdly, the tool should better support design space exploration (DSE) by allowing the user to control and vary parameters in the system configuration manually or automatically. Finally, in practical, the tool must be fast enough to finish the work in time since design varies often at early stage development.

Exploring different design parameters, architects rely on this understanding to perform cost-benefit analysis among alternative design options. However, especially in a heterogeneous computing environment, a large system has millions or billions of possibilities, and so enumerating every point in the design space is prohibitive. Several DSE algorithms [12, 13, 28, 39] are proposed to systematically explore the design space in a cost-effective manner, but the DSE process is still bound by the speed of simulating a single system configuration.

In the past, simulation tools are widely used for circuits, processor chips and electronic system design. Cycle-accurate simulators are capable of evaluating the performance impact of hardware changes, however, as the complexity of hardware increases, the complexity of cycle-accurate simulation also increases and becomes unacceptably slow. A general practice is to divide a system into components and design each component with individual simulators. Unfortunately, it is very difficult, if not impossible, for the user to evaluate the application performance on the entire system unless there is a framework to put together all the hardware component simulators and execute the application. Without such system-level evaluation capability, the interactions between components would not be analyzed and the designers are often limited to work on incremental design changes.

For processor design, traditional simulation tools may offer accurate performance estimation at micro architectural level, but it is highly complicated to combine the simulators for various components to perform complex applications due to the lack of a simulation framework to effectively support such works. While one could use traditional system-level simulation framework such as SystemC, the speed of such complex simulation would be unacceptably slow. Furthermore, they fall in short in terms of their capabilities to profiling application workload and reveal relevant hardware and software events. For instance, it would take a very long time for a cycle-accurate processor simulator to boot the Linux operating system), one of the most widely used operating systems for smartphones and intelligent IoT devices.

Previously, we developed a *cycle-approximate hybrid emulation/simulation approach* [24] to estimate the performance of Linux-based Android smartphones with performance models which can be selectively and dynamically switched on and off to speed up simulation by fast-forwarding through the boot sequence.

With the fine-grain control, the speed of our cycle-approximate simulator can be several orders of magnitude higher than traditional cycle-accurate simulators.

In conjunction with parallel DSE algorithms, our ADSET work successfully reduces the duration of DSE from 35 days to 12 hours [23].

In addition, our *Virtual Performance Analyzer* (VPA) framework [24] enables tracing and profiling of hardware and software events. Recently, we added the capability of emulating/simulating graphics processing unit (GPU) to support the *Heterogeneous System Architecture* (HSA) specifications [16, 22].

From our past experiences, we found that the communications and synchronizations between the component simulators cause great overheads and slow down the entire system level simulation.

In this paper, we propose several methods with its performance evaluations to further speed up our cycle-approximate hybrid emulation/simulation framework. First, we developed a scheme to reduce the communication overhead between component simulators. Then, we extended a new communication scheme to support *multi-model simulation*, allowing concurrent simulation of multiple component simulators in one shot, which further saves the duration of DSE.

Finally, we develop a *just-in-time model (JIT) selection* technique that enables the system level simulation to automatically adjust the level of simulation for a component by shifting between faster cycle-approximate models and slower cycle-accurate simulation models based on the statistics that are collected on-the-fly.

In our case study, with our *MMRB*, concurrent simulation of eight cache configurations was improved by 3 times comparing to using eight previous VPAs. 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 new framework provided 7.8 times of speedup which can improve the speed of DSE process dramatically. [1]

The rest of the paper further introduces related works in Section 2, describes the proposed framework and its implementation in Section 3, discusses our experimental results in Section 4, and finally concludes with remarks on potential future works in Section 5.

2 Background and Related Work

To assist the development of signal processing systems in the early stage, simulation and DSE tools are widely used for evaluating design alternatives and finding the optimal design. In this section, we describe existing simulation/DSE tools and compare our work to them.

2.1 Simulator Tools for System Design

A variety of simulation tools exist for evaluating processors or system designs by modeling the hardware details at the micro-architectural or the system level. For evaluating processor performance, SimpleScalar [10], PTLsim [43] and SESC [36] are popular examples of *cycle-accurate microarchitecture simulators* in the public domain, which is capable of modeling microarchitecture details, including processor pipelines, branch predictors, and caches to give accurate performance estimates.

More recently, ZSim [38] is developed to simulate the performance of x86-based multicore processors.

For non-mainstream or application-specific processor design, such as digital signal processors, designers often create their own simulation tools. Lin et al. [29] developed the cycle-accurate simulator for modeling the performance and power of the PAC DSP to evaluate the performance and power consumption of signal processing applications. As GPU becomes increasingly popular, various GPU simulators have surfaced, including GPGPUSim [34] and Multi2Sim [42], but are limited to specific vendors, such as NVIDIA and AMD.

In addition to the whole processor simulators, tools exist for simulating specific parts in a processor or other components in a computing system. Taking examples from popular public-domain tools, Dinero IV is a cache simulator [20], DRAMSim2 [37] can be used for simulating DRAM, and NS2 [27] simulates for networks.

Unlike the whole processor simulators, which often take the application executable as input, the aforementioned *component simulators* are usually fed with the *event traces*, which could be generated from physical machines or whole processor simulators.

For simulating a full system, one may have to combine processor simulators with component simulators to perform the target application/system software as close as an actual system does, which has been a challenging task for designers. Now with the addition of heterogeneous multicores and application-specific accelerators, the task becomes even harder.

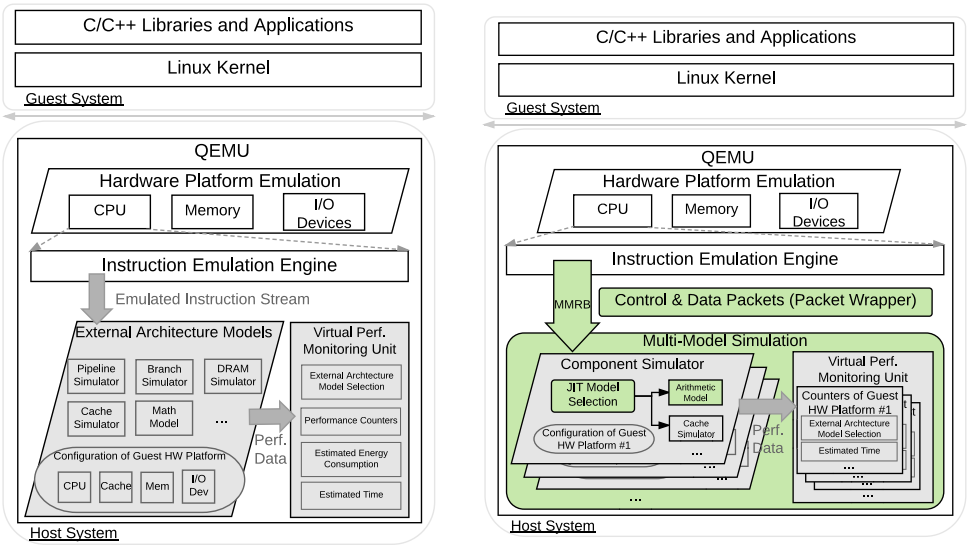
For example, gem5-gpu [7] incorporates the CPU simulator (gem5 [7]) with the GPU simulator (GPGPUSim [34]) and M5 simulator [8] modeling the performance of both processors. Our earlier work, MCEmu [41], provides a simulation framework that is capable of modeling both the ARM-based application processor and the digital signal processing cores in the PAC Duo system-on-chip.

While cycle-accurate simulators offer detailed timing information, they are several orders of magnitude slower than the functional simulators, a.k.a. emulators, which do not model the hardware details at all. For a complex system design, cycle-accurate simulators are generally too slow and are inadequate for running the complex software systems that involve timing constrains or interactive I/O events. In recent years, the *cycle-approximate simulators* are developed to tackle the issue by adding the timing models to full system functional simulators [15, 24, 31]. Both FAST [15] and our VPA [24] employ the QEMU emulator to execute the system/application software and feed the executed instruction stream to timing models to estimate the execution time. FAST implements its timing model with field programmable gated array (FPGA) to model microarchitecture details and resulted an emulation speed of 1.2 MIPS (Million Instructions per Second), roughly an order of magnitude faster than cycle-accurate simulators. Our VPA full-system simulation framework allows the user to plug-in needed component simulators and activates some of them selectively to increase the execution speed.

VPA leverages QEMU's dynamic binary translation scheme to accelerate the execution of instructions. By plugging in cycle-accurate cache simulators and simple pipeline timing models, we had demonstrated that VPA was useful in finding performance bottlenecks in system and application activities for Android smartphones at about 10 MIPS. Even power consumption can be estimated with good accuracies [25].

We use VPA as an example to illustrate the key concept of the cycle-approximate simulators. As illustrated in

[1] We have shared this work as an open source project on https://bitbucket.org/paslab/qemu_vpmu_opensource, https://github.com/snippits/qemu_vpmu.



(a) An overview of the VPA cycle-approximate simulator [24].

(b) The proposed framework.

Figure 1 An architectural comparison of our previous VPA design and the new features proposed in this paper.

Fig. 1a, QEMU emulates the virtual hardware platform, including the CPU, Memory and I/O devices, so that the Linux-based, full-blown software stack is able to run on the virtualized platform (i.e., *guest system*). During the software execution, the executed instruction stream is fed to the external architecture models so as to account for the performance events and deriving the timing information, with regard to the configuration of the guest hardware platform. The generated performance data are consolidated by the virtual performance monitoring unit (VPMU), which acts as the performance monitoring units available on the real machines. For example, the estimated cycles done by the CPU, the cache misses rate simulated by the cache simulator, and the branch miss count are all available in the VPMU. Since then, we had developed several techniques to accelerate the simulation framework by parallelizing the simulation of coherence protocols in distributed caches [14] and the works described in this paper.

2.2 Design Space Exploration Tools

Given the user-defined objective(s) for the target system, DSE mainly involves the algorithms/methodologies for rapid search of good design points in a very large design space. Mathematical models or simulation tools are utilized for estimating the performance of certain design points. Many studies have been done in the literature to reduce the time for DSE with rapid design space searching algorithms, mathematical models to project the performance, and/or parallelized DSE process executed on multiple computers. The following paragraphs describe the related works and point out how our framework facilitates and accelerate the DSE process.

Mohanty et al. [32] proposed the hierarchical DSE methodology, which is commonly used in the recent DSE tools. Unnecessary design points are filtered out based on user-defined constraints for the target system, and detailed simulation runs are performed to pinpoint desired design points. Angiolini et al. [5] emphasized on the integration of existing tools to facilitate system-level DSE. Schatz et al. [39] proposed a constraint-based model-transformation scheme, which helps reduce the search time by conducting the search in the transformed parameter space to avoid the detailed simulations as much as possible.

Several learning-based methods have been developed to predict the performance of the given design point [6, 17, 30, 33]. Beltrame et al. [6] adopted the Markov Decision Process to help DSE. Yu et al. introduced the transductive experimental design (TED) method [44] that judiciously sampled representative and hard-to-predict microarchitecture choices for training the learning models .

Still, the above learning based methods are tightly coupled with the applications that are used to train the model, which means that the models need to be re-trained for new applications or when application behavior changes.

Dubach et al. [17] developed an architecture-specific approach that allows the prediction of any new program across the entire microarchitecture configuration space.

Chen et al. [13] proposed the ArchRanker framework to improve the accuracy of learning-based DSE tools by using a pair-wise performance prediction method and achieving 29.68% to 54.43% fewer incorrect predictions, compared to a traditional approach [26]. While the use of well-trained machine learning models could shorten the space searching process, detailed simulation results are still required for training the machine learning models. Typically, more

data would result in better models, which also leads to the demand of faster simulation tools.

With abundant computing resources, parallelized and distributed DSE may effectively shorten the search time [11, 12, 19, 23]. For example, Calborean et al. proposed a framework [12] that utilizes multi-objective search algorithms to perform the DSE on multiple computers. Our ADSET [23] framework leverages the jMetal [18] as the DSE engine to guide parallel search with multiple system-level simulators, which was able to shorten the DSE time from 128 days (using the cycle-accurate simulator) to 2 days (using the cycle-approximate simulator).[2] In this paper, we further propose a fine-grain parallel DSE scheme, which uses our multi-model scheme to take advantage of today's multicore computers to obtain multiple results in one simulation run.

3 The Multi-Model and JIT Approach

As shown in Fig. 1a, our original VPA leverages QEMU to achieve cycle-approximate simulation and analysis by plugging in external architecture models (hardware component simulators) and a virtual performance monitoring unit (VPMU). To further speed up DSE, we extend VPA to (1) obtain the results of multiple simulation configurations with different hardware design parameters and (2) speed up the simulation of repeated regular patterns.

We have developed three key features in our newly proposed framework, as illustrated in Fig. 1b. The new features are:

- *Packet Wrapper* is a module for handling communications and synchronizations between the instruction emulation engine in QEMU and the *external architectural models* (component simulators or simple timing models) through a pre-defined unified interfaces.
- *Multi-Model Ring Buffer (MMRB)* is responsible for configuring component simulators, *i.e.* Dinero IV cache simulator, and serves as a scalable, efficient communication channels for multiple component simulators to obtain input events from QEMU with reduced data transfer overheads.
- *Just-in-Time (JIT) Model Selection* is dedicated for speeding up the architectural simulation by detecting repeating patterns and converting them into faster timing models.

With *Packet Wrapper*, the user can attach multiple hardware component simulators with a simple registration function and selectively route the event packets generated by the instruction emulation engine of QEMU via the *MMRB*. Moreover, one can use multiple hardware models to simulate processor pipelines, caches, TLB, I/O devices, etc. in parallel, or to simulate components in different design parameters. Each simulator receives packets from *MMRB* and stores its results in a set of event counters in the enhanced VPMU's, which is why multiple simulation results can be obtained in one run. The *JIT model selection* method can be included in some of the external component simulators, e.g. cache simulation, to simplify the simulation of repeated patterns. The following subsections further discuss the details for each component we designed for scalability and efficiency.

3.1 Packet Wrapper

In our previous work, we have instrumented the instruction emulation engine and virtual devices in the QEMU to extract events such as memory references, branch references, I/O requests, etc. Packet Wrapper further defines two types of packets, *data packet* and *control packet*, to connect the QEMU and the external component simulators.

- A specific type of *data packets* is used to transfer the events required by each component simulator in a predefined format. With a set of predefined data packets, we can improve the efficiency of data transfer in the MMRB.
- A predefined set of *control packet* are used for the VPA to send commands to the external architecture models for maintaining, synchronizing, and displaying the state of the architecture models.

As the simulation starts, *Packet Wrapper* issues the *initialization packet*, i.e. the control packet which passes the configuration to timing simulators. During the simulation, it sends hardware events to the component simulators via various types of data packets. From time to time, *barrier packets* and *synchronization packets* are needed for synchronizing the execution states of the architecture models to derive or receive cycle counts data in VPMU. When a simulator receives a *dump packet*, it displays the status/statistics of the simulation on the console to enable performance analysis and debugging. Notice that we use a token-based mechanism to prevent multiple simulators from dumping information to the console at the same time. Thus, to handle the aforementioned control packets, each simulator needs a *Packet Processor* to handle the semaphore, pass the token, and synchronize performance data. We have provided several templates in our open source release.

An external architecture model often comes with its own input data format, so the packet processor needs to convert our predefined data format for the architecture model, which incurs some overhead. However, *Packet Wrapper* and the

[2] In this case study, we explored the cache designs for ARM-based smartphone systems by considering the execution time, die area and power consumption.

predefined packet formats provide a standardized interface and make it easier for the users to attach existing simulators even without the source codes. Through the ring buffers described in the next subsection, the packets of events are broadcasted to all the packet processors attached to *Packet Wrapper* with reduced communication overheads.

3.2 Multi-Model Ring Buffer (MMRB)

Our framework uses the shared memory to construct a Multi-Model Ring Buffer (MMRB) for configuring component simulators and broadcasting events to the hardware component simulators, as illustrated in Fig. 2. The MMRB is partitioned into following segments:

- The *Config* segment specifies the configurations for the component simulators in *json*-format [9]. For example, this can be used by the user to configure the size and associativity for the cache simulator(s). Each component simulator is required to find its configuration by parsing the *Config* segment.
- The *Data* segment provides an array of data communication channels for individual component simulator to update/report the results of simulation. Through these channels, VPA stores the performance-related results in the virtual performance monitoring unit (VPMU) and use these data to estimate the execution time and analyze the performance.
- The *Token* segment is used to serialize the reports of simulators by having the component simulators dump the results to the console one by one for debugging purposes.
- The *Ring Buffer* scheme reduces the communication overheads and improve the scalability as the number of component simulators increases. The data packets are stored in the local buffer within QEMU before pushing into the ring buffer in the shared memory. Then each component simulator acquires an identical block of data packets and stores the acquired packets in its local buffer. The redesign and implementation is due to the characteristics of our application which requires a huge bandwidth of broadcasting packets while synchronizing from time to time, but overall concept is the same as every ring buffer designs.

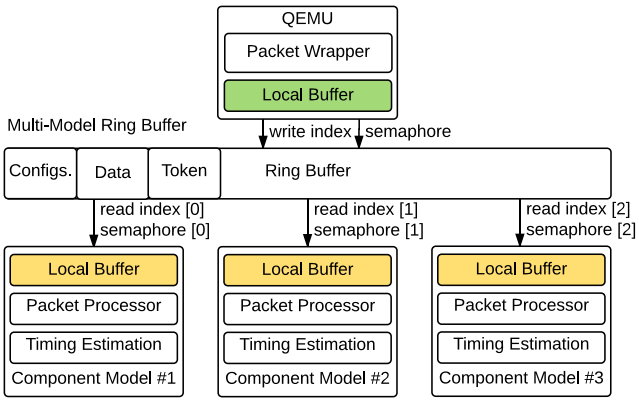


Figure 2 The architecture of Multi-Model Ring Buffer (MMRB).

Note that each component simulator runs as a separate process to enable the attachment of an existing simulator without its source code. In addition, we did not use CPU affinity to bind the process of a component simulator to a dedicated CPU core because we found it would sometimes cause slowdown when running on our environment. We use the Linux shared memory scheme to implement the ring buffer, which improves the efficiency of broadcasting traced events to multiple component simulators, but we also need to pay attention to the cache pollution issues in the buffering scheme. As illustrated in Fig. 2, QEMU and the component simulators have their own local buffers, instead of accessing the ring buffer directly, a local buffer is created for each component simulator to improve the scalability for running multiple configurations/models. Section 4 further explores the alternative buffering designs and presents experimental results to show the benefits of our MMRB design.

3.3 Just-In-Time (JIT) Model Selection

As QEMU can execute applications at hundreds of millions of instructions per second, adding component simulators to QEMU often slows down the execution speed significantly. For example, a typical cache simulator is capable of consuming 20∼50 millions memory references per second in our experiences. Especially for a multicore system, in addition to simulating the distributed caches, modeling the cache coherence protocol between the distributed caches and simulating the shared cache can further slowdown the execution speed. Thus, we proposed the *JIT Model Selection* mechanism to accelerate the cycle-approximate simulation by switching a timing accurate model to a approximated model, when the execution speed is favored for performance estimation.

Figure 3 illustrates the workflow of the JIT model selection mechanism that is integrated into the dynamic binary translation (DBT) engine within the QEMU. When the DBT engine executes a basic block (BB) initially, the JIT model selection mechanism marks the BB as *cold*. This is done by adding a field in the code cache of QEMU. When a BB is executed the second time within a specific period of time (e.g. 0.5K instructions), the BB is marked as *hot* and triggers the selection of arithmetic timing model over the cache simulation to speed up the execution. For example, *BB 1*

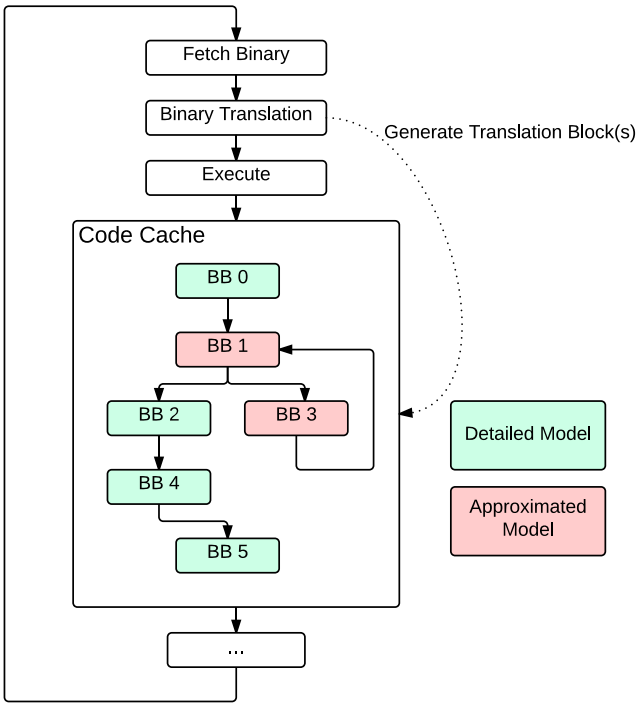


Figure 3 A demonstration of working flow of proposed JIT model selection. In a period of time, repeatedly executed basic blocks are hot BBs.

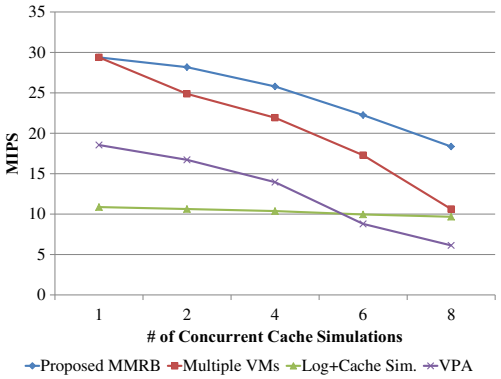


Figure 4 Comparison of running multiple VMs simultaneously (with the proposed ring buffer) and VPA (with direct pass ring buffer) and the proposed MMRB.

and *BB 3* are hot BB's in the figure, and the other BB's are cold. A BB becomes cold if it has not been executed within another period.

To be more specific, the approximated model we demonstrated in this paper is a simple mathematical model of L1 cache models. The time period is set according to the size of the instruction cache, thus, for a hot BB, it is obvious that the instructions in the BB should remain in the instruction cache. In this case, we can naively assume the instruction fetches are all hits and bypass the simulation of the instruction cache. For the data memory accesses generated by a hot BB, we implement a high-speed data access tracking model which detects if any data memory access is hit within the same cache block as its previous four memory accesses. If a data memory access is not hit, it is marked as miss temporarily and is sent to the cache simulator to verify if it is actually a cache miss. Since only a small portion of memory accesses would require such a verification step in a typical memory access pattern, it has a minor impact on the simulation speed.

4 Evaluation

Designing an efficient implementation of the ring buffer for modern processors is not a trivial task due to the complex hardware micro-architectures. In our previous VPA framework, as the number of simulators increased from one to eight, the simulation speed dropped from 18.5 MIPS to 6.1 MIPS, which was much slower than logging down traces and running it on cache simulator separately. Even with the proposed ring buffer implementation, running eight VMs concurrently with a single cache model in each VM, marked as distributed simulation in Fig. 4, was still 1.7x slower than running with our *MMRB*. The best way of running multiple configurations and component simulators is running multiple component simulators concurrently in parallel with the communication channel designed for broadcasting as we did in the proposed MMRB. With profiling information, the implementation details and benefits would be introduced in the later subsections.

The following experiments were mostly done on Intel i7-4790k CPU which has 32 KB L1 data cache and 256 KB L2 cache equipped with 32GB DDR3 Memory running on Linux kernel version 4.4, except Fig. 5 was done on AMD Opteron 6174 server platform which has 64 KB L1 data cache and 512 KB L2 cache. We used *Buildroot* [4] to compile the root file system for both real and simulated systems, in order to control the software running on the systems and size of the system image. As for the cache configuration, we repeated the same configuration when running multiple cache simulators in parallel. Finally, the compiler of *gcc* was version 6.3.1 and *icc* was 2016.3.210 and the compiler options were the default configurations of *QEMU* which used *-O2*. In the following experiments, the size of local buffer was 3 KB (256 entries) and the size of ring buffer was 64 MB for *MMRB*. The benchmark suit was MiBench [21], a representative embedded benchmark suit, and we further used matrix multiply to demonstrate the memory intensive signal processing applications. For a fare comparison, we

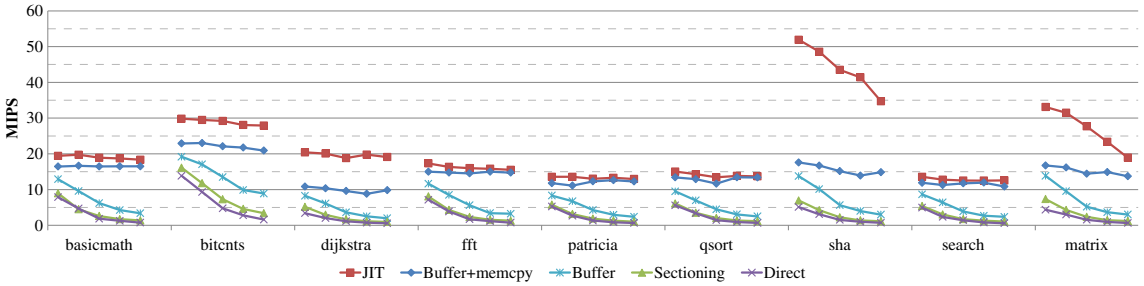


Figure 5 MIPS trend of different implementation of ring buffer mechanism on AMD Opteron 6174 (An 48 cores server processor, two sockets). We run 1,2,4,6,8 cache configurations concurrently in parallel for each benchmark.

used MIPS as the basic unit instead of execution time since the emulation speed varies with application behaviors.

4.1 The Bottleneck Shifts

In most timing approximate simulators, communication channel is not a critical issue since the bottleneck usually lies on timing models and functional simulations. However, when we looked into the overhead of VPA, we found that VPA spends 91% of time on communicating with cache simulator when we force all the timing models skip every traces and do nothing. This attracted our interests to further diagnose the cause and optimize it, in order to enable the fast performance evaluation for embedded multi-core processors.

In our case study of collecting memory access traces of a matrix multiply program, 0.22 billion of memory access traces were generated which was around 2.3GB/s of memory accesses on sending the traces. We implemented the proposed *MMRB* design with its *Direct Pass* implementation of ring buffer in our VPA design. Marked as VPA in Fig. 4, the simulation speed was 18.5 MIPS which is faster than logging down traces and running cache simulation separately. However, running eight forked Dinero simulators in parallel was around 6.12 MIPS which was a little bit slower than logging traces and simulates it concurrently. We look into the performance of the naive approach and found that the communications between QEMU and Dinero was the bottleneck and had a really high CPI of 3.58, as listed in Table 1. When collecting the performance counters, the CPI of main thread only counted functions of our framework which excludes other QEMU functions in order to understand the effects of implementations.

4.1.1 Direct Pass

The first naive design of broadcasting cache references is *direct pass* which puts data into ring buffer directly and wakes Dinero simulator when the amount of data exceeds a threshold. On the receiver side, Dinero cache simulator gets data as long as the buffer is not empty. If the buffer is empty, the receiver will then sleep until QEMU sends a signal waking up this process. The direct pass design was very simple in code and minimizes the number of signals, however, the speed of direct pass dropped to a third when simulating 8 Dinero simulators which shows the bad scalability of this naive approach. The first possible bottleneck was the overhead of system calls caused by semaphore which wakes the child Dinero processes up. But, in the modern Linux versions, semaphores are built upon futex (fast user-level mutex) which does not involve a system call when the semaphore number is bigger than 1. With the design of futex, we noticed the slowdown was not from involving system call frequently. After profiling the program, we found a high CPI resulting in sending/receiving data to/from the ring buffer in the *direct pass* implementation. The cause of CPI might be the inefficient use of shared memory and cache

Table 1 The counter information reported from Intel Vtune of running *matrix* with eight cache simulators.

Implementation	Cycles per Instruction		Retired Instructions	MIPS
	Main Thread	Cache Thread		
Direct	3.583	1.516	44.5 G	6.13
Sectioning	2.448	1.520	32.3 G	7.67
Buffering	2.691	0.932	32.7 G	10.54
Buffering + memcpy	0.582	0.839	31.6 G	18.35
icc on VPMU and DineroIV	0.619	0.626	31.7 G	26.90

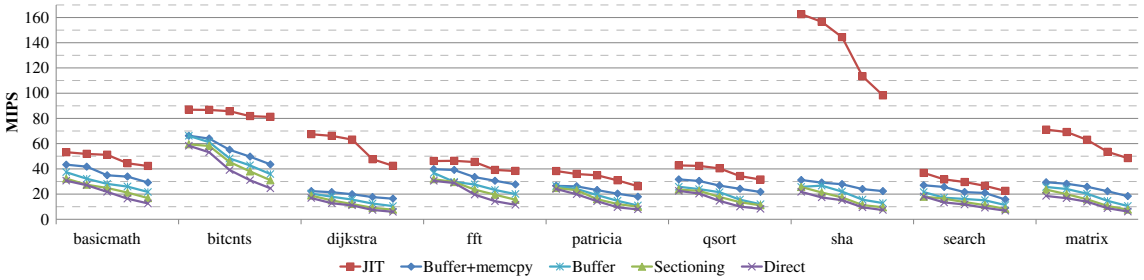


Figure 6 MIPS trend of different implementation of ring buffer mechanism on i7-4790k. We run 1,2,4,6,8 cache configurations concurrently in parallel for each benchmark.

coherence, but it was not clear from the profiling results we had. In the following sections, we will introduce different implementations to understand the cause of bottleneck and improve the performance of *MMRB*.

4.1.2 Sectioning

In order to minimize the CPI, we partitioned the ring buffer into several sections where the size of each section was 12 KB, which was about a third of L1 data cache size. By having an ownership concept on each section of the ring buffer, this approach avoids the cache invalidation, which is also known as false sharing, when sender and receiver are accessing concurrently. As listed in Table 1, the profiling counters showed that this implementation improved the CPI of main thread and the number of retired instructions. The design of sectioning, however, had only slightly improvement on the modern desktop CPU as shown in the Fig. 6. Noting this problem, we analyzed the utilization of buffer sections and found that the speed of generating memory accesses was similar to the speed of consuming the traces in average. Normally, the meaning behind this phenomenon is that cache timing simulation bounds the main threads, but we noticed that the CPI on both main thread and cache simulation thread was way too high.

4.1.3 Local Buffer with Memcpy

According to the result of profiling information, the high CPI was still an unsolved problem even sectioning helped false-sharing between the sender and receiver. Tracing the codes, we noticed another possible problem on the contention of shared memory which might cause low cache utilization in our design. Though using sectioning to avoid invalidation was a proper solution, the performance of passing data was still suffered from cache/memory bandwidth when using shared memory. To find more evidences of our thought, we used a local buffer to utilize L1 cache resource before sending data onto shared memory which might be the bottleneck of passing data. The local buffer was a small buffer with 3 KB in size, trying to hold data nearby processor cores and avoid bus contention when accessing a huge amount of data. In addition, the concept of sectioning was embedded in the design of local buffer since the data were copied chunk by chunk, also called bulk-transfer. Implementing local buffer on both sender and receiver side, we found a significant improvement of speed reaching 10.54 MIPS when running eight configurations simultaneously, shown in Table 1. As listed in Table 1, the CPI on cache thread had a significant improvement, but the main thread still remained high CPI. Due to the execution of functional simulation, the local buffer seems to not fit in L1 cache in main thread well without write-backs. To conquer the problem of copying data by CPU cores, we used the highly optimized *memcpy* from the standard C library to improve the copy process by utilizing the hardware resources as well as vector operations. Resulting in both execution time and CPI, using local buffer with *memcpy* further boosted the performance to 18.35 MIPS which was 3x faster than direct pass approach, shown in Table 1. From the performance counter information, the CPI dropped to 0.58 and 0.84 for main thread and cache thread, respectively. The values of CPI were already promising and acceptable on a modern computers, but we still found some other issues from the profiling information, which would be introduced in Section 4.1.4.

Table 2 The profiling information of CPU related bottlenecks reported from Intel Vtune on buffer+memcpy implementation.

Compiler	Loads Blocked by Store Forwarding	4K Aliasing
Default *gcc*	0.225	0.207
icc on VPMU and DineroIV	0.068	0.017

The value is the number of percentages CPU stalls from the cause.

Table 3 The error rate of using JIT comparing to original counter values, where negative value means under-estimated.

	Instruction cache counters		Data cache counters		
Benchmarks	Accesses	Read Misses	Accesses	Read Misses	Write Misses
basicmath	0.01%	−1.76%	0.00%	0.50%	−0.12%
bitcnts	0.00%	2.16%	0.03%	1.25%	2.20%
dijkstra	0.00%	1.03%	0.00%	0.00%	−2.15%
fft	0.00%	0.36%	0.01%	1.14%	0.03%
patricia	0.00%	−0.71%	0.00%	0.25%	0.02%
qsort	0.00%	0.13%	0.01%	−0.07%	−0.10%
sha	−0.01%	3.58%	0.00%	0.05%	0.32%
search	0.08%	9.10%	0.47%	2.21%	0.64%
matrix	0.00%	−1.25%	0.00%	−0.06%	0.01%

4.1.4 The Last Mile of Performance

With the best implementation of ring buffer we proposed, the profiler still reported a bottleneck of *load blocked by store forwarding* and *4K aliasing* on cache simulation thread. Coincidentally, the size of local buffer was 3KB, which is not 4K but might be the cause of *4K aliasing*.

Here is the explanation of these two bottlenecks extracted from Intel Developer Zone [1, 3]:

1. **Load Blocked by Store Forwarding**: A store forward block describes a situation when a recent store is unable to forward to a load. If a load follows a store and reloads data that the store has written to memory, Intel microarchitectures can in many cases forward the data directly from the store to the load. The penalty for store-forwarding is 10-15 cycles [1].
2. **4K Aliasing**: When an earlier (in program order) load issued after a later (in program order) store, a potential WAR (write-after-read) hazard exists. To detect such hazards, the memory order buffer (MOB) compares the low-order 12 bits of the load and store in every potential WAR hazard. If they match, the load is reissued, penalizing performance for about 5 cycles in common cases [2].

Since the problem of *load blocked by store forwarding* and *4K aliasing* were not from our implementation of *MMRB*, we introduced the optimization from compiler. By using the Intel C Compiler, *icc*, compiling the code of our framework and Dinero IV cache simulator, the proposed *MMRB* gained another 1.5x of speed up when simulating 8 cache configurations concurrently in parallel. Furthermore, incredibly, *JIT model selection* reached 104 MIPS of simulation speed when simulating one cache model. The speedup was from having a faster cache timing simulation with better compiler on Intel platform. As shown in Table 2, both bottlenecks got significant improvements with *icc* compiler optimizations. This showed that timing simulation was still a big bottleneck and needs to be take care of.

4.2 JIT Model-Selection

As shown in Fig. 6 and Table 3, the speedup on a memory intensive program, *matrix*, achieved 2.6 times faster than the MMRB with 0.06% relative error on data cache and 1.25% relative error on instruction cache. Diving into details, as the results showed in Fig. 7, the ratio of *hot BB* were 94% and 52% for instruction and data cache, respectively. In other words, 48% of memory accesses was fed into Dinero IV cache simulator for simulating data cache that might cause a miss. Moreover, *sha* demonstrated an extreme example which only 17.84% of instruction traces and 9.29% of data traces were fed into component simulators. Resulting in super high speed-up, the simulation speed of simulating single cache parameter reached 162.6 MIPS while the limit of pure QEMU was 600MIPS on our intel-i7 platform.

From the results of *sha* and *matrix*, we can tell that the speedup of JIT was mostly from accelerating data cache simulation since the access counts and *hot BB* ratios of instruction cache were similar. In other word, generally, if the application has heavy data accesses, it would be most likely benefit from the approximated model of JIT. Even if an application has low data accesses, like *bitcnts*, the JIT can still help when the code are regularly executed in a dense loop, which signal processing applications usually have.

As shown in Table 3, all of the test cases have negligible errors on each counter value. Since the cache state is

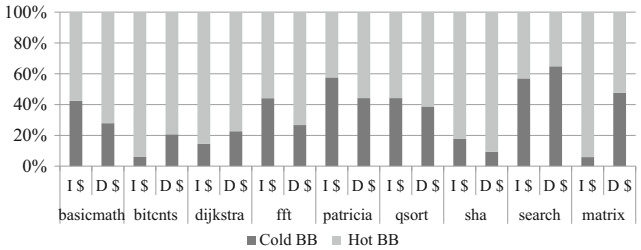


Figure 7 Ratio of cold BB and hot BB of each application. Generally, the more percentage of hot BB, the more speedup.

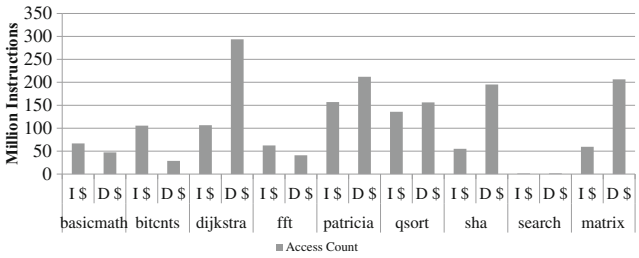


Figure 8 The number of loads/stores on instruction cache and data cache.

not always clean in the real system, the error rate of our results is similar to the error rate of a real system in most cases. In the *search* application case, the main cause of 9.1% error of instruction cache misses was the size of executed instructions and accessed data which were only about 1 million accesses, as shown in Fig. 8. Due to some randomness of cache pollution and OSs, the range of instruction cache misses actually varied a lot in our experiments. In other word, 9.1% error rate was actually within the variance and negligible in common scenarios.

With the proposed *MMRB* and its optimized ring buffer implementation, the emulation speed of simulating eight cache parameters for *matrix multiply* was 18.35 MIPS which was 3 times faster than using the design of previous VPA. Combined with *JIT Model-Selection*, which proveded another 2.6x of speedup, the performance of simulating eight cache parameters was boosted to 7.8x.

5 Conclusion

Targeting embedded system and application designer in early stage development, we proposed a framework to accelerate the process of timing simulation for both application profiling and design space exploration. Providing an elastic framework for customized add-ons with its timing model, we designed two main algorithms to accelerate the simulation speed while keeping the accuracy. Firstly, scalable and efficient ring buffer enables exploring multiple design parameters concurrently with memory efficient ring buffer design which gives 3 times of speedup. In addition, with the proposed efficient ring buffer design, our framework was 1.7x faster than our previous work VPA when running single cache simulation and 3x faster when running eight cache models. Secondly, in addition to improving the speed, we designed a *JIT model selection* algorithm to further shorten the timing simulation when data/instruction accesses are regular with an approximated model. Taking the concept of JIT compilation, hot/cold basic blocks are adopted into the design to increase the accuracy on hot codes for best performance gain with minimum loss of micro architectural states. With both *multi-model* and *JIT model selection*, the proposed framework reaches 7.8x of speedup and provides a fairly fast simulation for DSE tools comparing to using our previous VPA framework. We believe the proposed framework would be very useful on design space exploration as well as timing simulation for application developers.

Acknowledgements This work was financially supported by the Ministry of Science and Technology of Taiwan under Grants MOST 105-2622-8-002-002, and sponsored by MediaTek Inc., Hsin-chu, Taiwan. We specially thank our colleagues, Tsung-Han Chiang and Jen-Chieh Wu, for proofing the concept of this work.

References

1. Intel performance bottleneck (2012). Loads blocked by store forwarding https://software.intel.com/en-us/forums/intel-performance-bottleneck-analyzer/topic/333586.
2. Intel developer forum: 4k aliasing. https://software.intel.com/en-us/forums/intel-vtune-amplifier-xe/topic/606846 (2016).
3. Intel performance bottleneck: 4k aliasing. https://software.intel.com/en-us/node/544395 (2016).
4. Andersen, E. Buildroot: making embedded linux easy. https://buildroot.org/.
5. Angiolini, F., Ceng, J., Leupers, R., Ferrari, F., Ferri, C., & Benini, L. (2006). An integrated open framework for heterogeneous mpsoc design space exploration. In *Proceedings of the conference on design, automation and test in Europe: proceedings* (pp. 1145–1150.) European Design and Automation Association.
6. Beltrame, G., Fossati, L., & Sciuto, D. (2010). Decision-theoretic design space exploration of multiprocessor platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *29*(7), 1083–1095.
7. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., & et al. (2011). The gem5 simulator. *ACM SIGARCH Computer Architecture News*, *39*(2), 1–7.
8. Binkert, N.L., Dreslinski, R.G., Hsu, L.R., Lim, K.T., Saidi, A.G., & Reinhardt, S.K. (2006). The m5 simulator: modeling networked systems. *IEEE Micro*, *26*(4), 52–60.
9. Bray, T. (2014). The javascript object notation (json) data interchange format. https://en.wikipedia.org/wiki/JSON.
10. Burger, D., & Austin, T.M. (1997). The simplescalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, *25*(3), 13–25.
11. Calborean, H., Jahr, R., Ungerer, T., & Vintan, L. (2011). Optimizing a superscalar system using multi-objective design space exploration. In *Proceedings of the 18th international conference on control systems and computer science (CSCS)*, (Vol. 1, pp. 339–346). Bucharest.
12. Calborean, H., & Vintan, L. (2010). An automatic design space exploration framework for multicore architecture optimizations. In *2010 9th on roedunet international conference (RoEduNet)* (pp. 202–207). New York: IEEE.
13. Chen, T., Guo, Q., Tang, K., Temam, O., Xu, Z., Zhou, Z.H., & Chen, Y. (2014). Archranker: a ranking approach to design space exploration. In *2014 ACM/IEEE 41st International symposium on computer architecture (ISCA)* (pp. 85–96). New York: IEEE.
14. Cheng, J.J., Hung, S.H., & Yeh, C.W. (2015). Rapid analysis of interprocessor communications on heterogeneous system architectures via parallel cache emulation. In *Proceedings of the 2015 conference on research in adaptive and convergent systems* (pp. 418–423). New York: ACM.

15. Chiou, D., Sunwoo, D., Kim, J., Patil, N.A., Reinhart, W., Johnson, D.E., Keefe, J., & Angepat, H. (2007). Fpga-accelerated simulation technologies (fast): fast, full-system, cycle-accurate simulators. In *Proceedings of the 40th Annual IEEE/ACM international symposium on microarchitecture* (pp. 249–261). Washington, D.C.: IEEE Computer Society.
16. Ding, J.H., Hsu, W.C., Jeng, B.C., Hung, S.H., & Chung, Y.C. (2014). Hsaemu: a full system emulator for hsa platforms. In *Proceedings of the 2014 international conference on hardware/software codesign and system synthesis* (p. 26). New York: ACM.
17. Dubach, C., Jones, T., & O'Boyle, M. (2007). Microarchitectural design space exploration using an architecture-centric approach. In *Proceedings of the 40th Annual IEEE/ACM international symposium on microarchitecture* (pp. 262–271). Washington, D.C.: IEEE Computer Society.
18. Durillo, J.J., Nebro, A.J., & Alba, E. (2010). The jmetal framework for multi-objective optimization: design and architecture. In *IEEE congress on evolutionary computation* (pp. 1–8). New York: IEEE.
19. Dutta, R., Roy, J., & Vemuri, R. (1992). Distributed design-space exploration for high-level synthesis systems. In *Proceedings of the 29th ACM/IEEE design automation conference* (pp. 644–650). Washington, D.C.: IEEE Computer Society Press.
20. Edler, J., & Hill, M.D. (1998). Dinero iv trace-driven uniprocessor cache simulator.
21. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., & Brown, R.B. (2001). Mibench: a free, commercially representative embedded benchmark suite. In *2001 IEEE International workshop on workload characterization, 2001. WWC-4* (pp. 3–14). New York: IEEE.
22. Hsu, H.C., Yeh, C.W., Hung, S.H., Hsu, W.C., King, C.T., & Chung, Y.C. (2016). Hsaemu 2.0: full system emulation for hsa platforms with soft-mmu. In *Proceedings of the international conference on research in adaptive and convergent systems* (pp. 230–235). New York: ACM.
23. Hung, S.H., Chen, J.H., Tu, C.H., & Shieh, J.P. (2012). Adset: a framework of rapid design space exploration for android-based systems. In *2012 IEEE 1st Global conference on consumer electronics (GCCE)* (pp. 586–587). New York: IEEE.
24. Hung, S.H., Kuo, T.W., Shih, C.S., & Tu, C.H. (2012). System-wide profiling and optimization with virtual machines. In *2012 17th Asia and South Pacific design automation conference (ASP-DAC)* (pp. 395–400). New York: IEEE.
25. Hung, S.H., Liang, F.T., Tu, C.H., & Chang, N. (2013). Performance and power estimation for mobile-cloud applications on virtualized platforms. In *2013 Seventh international conference on innovative mobile and internet services in ubiquitous computing (IMIS)* (pp. 260–267). New York: IEEE.
26. Ipek, E., McKee, S.A., Singh, K., Caruana, R., Supinski, B.R.D., & Schulz, M. (2008). Efficient architectural design space exploration via predictive modeling. *ACM Transactions on Architecture and Code Optimization (TACO)*, *4*(4), 1.
27. Issariyakul, T., & Hossain, E. (2011). *Introduction to network simulator NS2*. Berlin: Springer Science & Business Media.
28. Kang, E., Jackson, E., & Schulte, W. (2010). An approach for effective design space exploration. In *Modeling, development, and verification of adaptive systems foundations of computer software* (pp. 33–54). Berlin: Springer.
29. Lin, C.Y., Chen, P.Y., Tseng, C.K., Huang, C.W., Weng, C.C., Kuan, C.B., Lin, S.H., Huang, S.Y., & Lee, J.K. (2010). Power aware sid-based simulator for embedded multicore dsp subsystems. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis, CODES/ISSS '10* (pp. 95–104). New York: ACM, DOI https://doi.org/10.1145/1878961.1878981, (to appear in print).
30. Liu, H.Y., & Carloni, L.P. (2013). On learning-based methods for design-space exploration with high-level synthesis. In *Proceedings of the 50th annual design automation conference* (p. 50). New York: ACM.
31. Miettinen, A.P., Hirvisalo, V., & Knuuttila, J. (2011). Using qemu in timing estimation for mobile software development.In *1st international QEMU users' forum* (Vol. 1, pp. 19–22).
32. Mohanty, S., Prasanna, V.K., Neema, S., & Davis, J. (2002). Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. *ACM SIGPLAN Notices*, *37*(7), 18–27.
33. Ozisikyilmaz, B., Memik, G., & Choudhary, A. (2008). Efficient system design space exploration using machine learning techniques. In *Proceedings of the 45th annual design automation conference* (pp. 966–969): ACM.
34. Power, J., Hestness, J., Orr, M.S., Hill, M.D., & Wood, D.A. (2015). gem5-gpu: a heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, *14*(1), 34–36.
35. Pullini, A., Conti, F., Rossi, D., Loi, I., Gautschi, M., & Benini, L. (2016). A heterogeneous multi-core system-on-chip for energy efficient brain inspired vision. In *2016 IEEE International symposium on circuits and systems (ISCAS)* (pp. 2910–2910). New York: IEEE.
36. Renau, J., Fraguela, B., Tuck, J., Liu, W., Prvulovic, M., Ceze, L., Sarangi, S., Sack, P., Strauss, K., & Montesinos, P. (2005). Sesc: cycle accurate architectural simulator. Retrieved November 19 2013.
37. Rosenfeld, P., Cooper-Balis, E., & Jacob, B. (2011). Dramsim2: a cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, *10*(1), 16–19.
38. Sanchez, D., & Kozyrakis, C. (2013). Zsim: fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer architecture news*, (Vol. 41 pp. 475–486): ACM.
39. Schatz, B., Holzl, F., & Lundkvist, T. (2010). Design-space exploration through constraint-based model-transformation. In *2010 17th IEEE International conference and workshops on engineering of computer based systems (ECBS)* (pp. 173–182). New York: IEEE.
40. Stoif, C., Schoeberl, M., Liccardi, B., & Haase, J. (2011). Hardware synchronization for embedded multi-core processors. In *2011 IEEE International symposium of circuits and systems (ISCAS)* (pp. 2557–2560). New York: IEEE.
41. Tu, C., Hung, S., & Tsai, T. (2012). Mcemu: a framework for software development and performance analysis of multicore systems. *ACM Transactions on Design Automation of Electronic Systems*, *17*(4), 36. https://doi.org/10.1145/2348839.2348840.
42. Ubal, R., Sahuquillo, J., Petit, S., & López, P. (2007). Multi2sim: a simulation framework to evaluate multicore-multithread processors. In *IEEE 19th International symposium on computer architecture and high performance computing* (pp. 62–68). New York: Citeseer.
43. Yourst, M.T. (2007). Ptlsim: a cycle accurate full system x86-64 microarchitectural simulator. In *IEEE International symposium on performance analysis of systems & software, 2007. ISPASS 2007* (pp. 23–34). New York: IEEE.
44. Yu, K., Bi, J., & Tresp, V. (2006). Active learning via transductive experimental design. In *Proceedings of the 23rd international conference on Machine learning* (pp. 1081–1088). New York: ACM.

Chih-Wei Yeh received his B.S. in Electronic Engineering from National Taiwan University of Science and Technology, Taiwan in 2012. Currently he is a Ph.D student in Computer Science and Information Engineering, National Taiwan University, Taiwan. His research interests include performance analysis tools, emulation tools, application acceleration on heterogeneous systems and methods for system behavior modeling.

Chia-Heng Tu is an assistant professor with department of Computer Science and Information Engineering (CSIE), National Cheng Kung University (NCKU). Before joining NCKU CSIE, he worked as Postdoctoral Researcher in MEDIATEK-NTU Advanced Research Center, National Taiwan University (NTU) in 2015, and as R&D Manager in Institute for Information Industry from 2012 to 2015, after he completed his Ph.D. training from NTU in 2012. His research interests are developing tools (e.g., computer architecture simulators, performance analyzers/optimizers, and parallelizing compilers) for designing/optimizing specialized computer systems.

Shih-Hao Hung joined National Taiwan University in 2005 and is currently a professor in the Department of Computer Science and Information Engineering. His research interests include cloud computing, parallel processing, embedded systems, and pervasive applications. He worked for the Performance and Availability Engineering group at Sun Microsystem Inc. in Menlo Park, California (2000–2005) after he completed his PostDoc (1998–2000), Ph.D. (1994–1998) and MS (1992–1994) training in the University of Michigan, Ann Arbor. He graduated from National Taiwan University with a BS degree in Electrical Engineering in 1989.