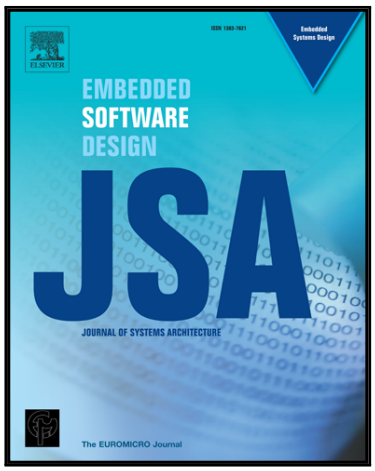


TQSIM: A Fast Cycle-Approximate Processor Simulator Based on QEMU

Shin-haeng Kang, Donghoon Yoo, Soonhoi Ha

PII: S1383-7621(16)30029-7
DOI: 10.1016/j.sysarc.2016.04.012
Reference: SYSARC 1356

To appear in: *Journal of Systems Architecture*

Received date: 19 August 2015
Revised date: 18 April 2016
Accepted date: 18 April 2016

Please cite this article as: Shin-haeng Kang, Donghoon Yoo, Soonhoi Ha, TQSIM: A Fast Cycle-Approximate Processor Simulator Based on QEMU, *Journal of Systems Architecture* (2016), doi: 10.1016/j.sysarc.2016.04.012

TQSIM: A Fast Cycle-Approximate Processor Simulator Based on QEMU

Shin-haeng Kang[a], Donghoon Yoo[b], Soonhoi Ha[a,*]

[a]*Seoul National University, 1 Gwanak-ro 301-456, Gwanak-gu, Seoul, Republic of Korea*
[b]*Samsung Electronics Co. Ltd., 130 Samsung-ro, Yeongtong-gu, Suwon-si, Gyeonggi-do, Republic of Korea*

Abstract

Timing simulation of a processor is a key enabling technique to explore the design space of system architecture or to develop the software without an available hardware. We propose a fast cycle-approximate simulation technique for modern superscalar out-of-order processors. The proposed simulation technique is designed in two parts; the front-end provides correct functional execution of the guest application, and the back-end provides a timing model. For the back-end, we developed a novel processor timing model that combines a simple-formula-based analytical model and a scheduling analysis of sampled traces so as to boost up the simulation speed with minimal accuracy loss. Attached with a cache simulator, a branch predictor, and a trace analyzer, the proposed technique is implemented over the popular and portable QEMU emulator, so named TQSIM (Timed QEMU-based SIMulator). Sacrificing around 8 percent of the accuracy, TQSIM enables one or two orders of magnitude faster simulation than a reference cycle-accurate simulation when the target architecture is an ARM Cortex A15 processor. TQSIM is an open-source project currently available online.

Keywords: `Superscalar out-of-order processor, Analytical simulation, Sampled simulation, Cycle-approximate simulation, QEMU`

1. Introduction

Simulation is one of the most critical techniques to facilitate the computer architecture design and the application development without using a real hardware [1]. A cycle-accurate simulator has been widely used to validate a target architecture at the cycle level. The cycle-accurate simulator focuses on the accuracy to predict the performance of the target architecture precisely, updating values of all the state elements of the machine at every clock cycle. Unfortunately, such high accuracy comes at the price of high development cost and long simulation time; It is known that single-core cycle-accurate simulators typically run at 0.01 to 0.3 million instructions per second (MIPS) which means that it would take several days of simulation time to simulate a couple of minutes of simulated time. Moreover, the simulation speed is degraded even more as the architectural complexity and the number of processors increase in a system.

*Corresponding author
Email addresses: `shkang@iris.snu.ac.kr` (Shin-haeng Kang), `say.yoo@samsung.com` (Donghoon Yoo), `sha@snu.ac.kr` (Soonhoi Ha)

The simulation speed is sometimes prioritized over the simulation accuracy. One case is a microprocessor or system level design space exploration. To compare the performance of several candidate architectures, fast simulation is necessary to simulate all of possible candidates. Another case is software development in the hardware/software codesign methodology. In this case, an application program has to be executed repetitively for software development and debugging, so simulation speed is a crucial factor that affects the productivity of software design. However, a certain level of accuracy is supposed to be guaranteed even in those cases. For example, accurate timing information about the events of each processor is essential to perform communication and synchronization of a multi-processor system.

Although recent CPU designs rely on multicore/multiprocessor designs to expose task-level parallelism, the reuse of the existing processor simulators with minor modification can save considerable development time and effort. Hence, many existing parallel simulation frameworks, such as MCEmu [2], Graphite [3], and HSIM [4], integrate existing single processor simulators or instruction set simulators (ISSs) to perform multicore/multiprocessor simulation. Thus improved single-core simulation performance is a key to make the simulation of larger multicore systems viable. Compromising the simulation speed and cycle accuracy becomes more challenging as the number of processors increases.

To serve those cases, in this paper, we propose a fast cycle-approximate simulation technique supporting modern superscalar out-of-order processors, boosting up the simulation speed with low accuracy loss. Like other modern simulators [5, 6, 7], the proposed simulator is designed in two parts; a front-end provides a functionally correct execution of the guest application, and a back-end provides timing models and recording data. For the front-end, we select one of the most popular and portable open-source emulators, QEMU. For the back-end, we combine the analytical and the sampled simulation techniques in a novel way. The baseline technique is an analytical simulation that models the processor timing with a simple formula. On top of that, the scheduling analysis of sampled traces improves timing accuracy by providing more accurate parameters used in the analytical formula. Since the sampling-based approach reduces the overhead of the scheduling analysis, we achieve faster simulation speed at minimal loss of accuracy. Since the proposed technique is implemented on top of QEMU, it is named TQSIM (Timed QEMU-based Simulator). Experiments show that TQSIM is one or two orders of magnitude faster than a cycle-accurate simulator, while maintaining high timing accuracy (average error approximately 8 % with MiBench programs). The simulator is only eight times slower than the baseline functional simulator, QEMU.

The following section provides a brief overview of various processor timing models and existing simulators based on QEMU. The detail description of the proposed technique is presented in Section III, and evaluated in Section IV. Finally, we summarize and conclude in Section V.

2. Related Work

2.1. Processor Timing Models

Over the years, there have been several timing simulation techniques proposed to guarantee an adequate level of accuracy and offer better performance than a cycle-accurate simulator at the same time. Extending the terminologies in [1], we summarize the representative timing simulation techniques as follows:

- ***k-CPI simulation*** assumes that it takes k cycles to execute one instruction. The number of cycles is given by one cycle for all instructions (1-CPI model) or given according to the

datasheet of the simulated processor (datasheet model). The datasheet model uses the cycle counts for each instruction by consulting a table built upon the specifications datasheet from the simulated platform. Such k-CPI model is readily implemented on a top of the functional simulator without sacrificing the simulation speed. It is known that commercial processor models such as Imperas OVP [8] and ARM FastModels [9] essentially use the k-CPI approach. However, the credibility of timing estimation from the k-CPI model is very low, because the k-CPI model neglects complex behavior of modern microprocessor architectures.

- ***Analytical simulation*** estimates the processor performance by using mathematical formulas. It can be mainly classified into two approaches, based on the analytical performance modeling method: mechanistic modeling [10, 11, 12], and empirical modeling [13]. A mechanistic modeling constructs a model based on the mechanics of the target processor, called white-box modeling. Empirical modeling uses a parameterized performance model where parameters are decided by machine learning or regression analysis, without any specific knowledge about the micro-architecture of the target processor, called black-box modeling. The proposed model is a variant of the mechanistic model.

- ***Sampled simulation*** [14, 15, 16] performs cycle-accurate simulation with a number of sampling units rather than the entire instruction streams. The sampling units are selected either randomly [14], periodically [15], or based on phase analysis [16]. To guarantee that those sampling units successfully represent the whole application, the statistical methods would be applied. We adopt the idea of sampled simulation only to derive some of the parameter values which are required for the mechanistic formula of the analytical simulation technique.

- ***Statistical simulation*** [17, 18] generates short-running synthetic traces or benchmarks that are representative for long-running benchmarks, and uses them to speed up architectural simulation. Statistical simulation is composed of two phases. Statistics collection phase collects base program characteristics (basic instruction mix and instruction dependency) and micro-architectural dependent statistics (cache, branch statistics). This information is then used to generate a synthetic instruction trace that is fed to a simple processor model in the synthetic simulator phase.

- ***FPGA-accelerated simulation*** [19, 20, 21] implements timing models onto field-programmable gate-arrays (FPGA) to exploit fine-grain parallelism in the FPGA. It is possible that FPGA-accelerated simulation is used in conjunction with the software-based techniques, such as mechanistic model. FPGA-accelerated simulation demands additional hardware and development time to synthesize the model into hardware.

- ***Hybrid simulation*** [22, 23, 24] uses bidirectional dynamic switching between a target cycle-accurate simulator and a functional simulator, while keeping the processor-centric state synchronized between both simulation modes. In HySim [22, 23], the target cycle-accurate simulator executes processor specific functions, whereas the host-compiled simulator executes target-independent parts of the application. Marss [24] supports seamless dynamic switching between the cycle accurate simulation mode and the native x86 emulation mode of QEMU. This switching mechanism speeds up program execution by skipping the parts which are of no interests to the users.

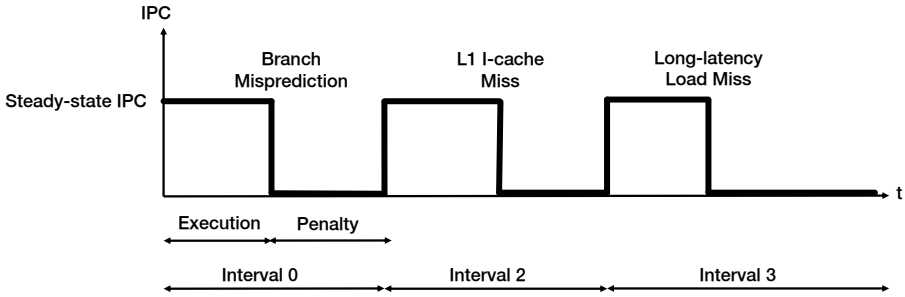


Figure 1: The basic idea of interval simulation: execution time is partitioned into discrete intervals by disruptive miss events such as cache misses and branch misprediction [11]

- ***Control-sensitive Simulation*** [25, 26, 27] makes use of context-sensitive estimates by keeping track of the execution history of the simulated target binary. Multiple execution times per basic block can be obtained at different contexts using a reference timing-accurate simulator [25] or static worst-case execution time analysis framework [26, 27] such as OTAWA [28] and Absint aiT [29]. The actual timing of basic blocks is defined dynamically depending on the previously executed basic blocks.

The main purpose of this study is to build a single core simulator that has good compromise between speed and accuracy, which will be used for building a many-core simulator. We define four requirements to serve our purpose: (1) the timing model does not require a reference timing-accurate simulator or a special hardware, because they may not be available; (2) a single run of the simulation is enough to extract timing information for communication and synchronization events between processors of a multiprocessor system; (3) the simulation speed is orders of magnitude faster than the cycle accurate simulator while guaranteeing an acceptable level of accuracy; (4) the processor model is very general, and can change any parameters easily.

Based on those conditions, existent simulation techniques are evaluated. *k-CPI model* shows lack of accuracy; *sampled* and *hybrid simulation* require a cycle-accurate simulator; *control-sensitive* and *statistical simulation* usually need preprocessing of simulated code. Therefore, we concluded that *analytical approach* is the best baseline technique. On top of that, we borrowed the philosophy of *sampled simulation* model, and devised the general scheduling analysis of sampled traces to improve the simulation accuracy of the analytical formula even further.

2.2. Analytical Model: Interval Simulation

Interval simulation [10, 12] has been proposed as the mechanistic modeling method, supporting out-of-order superscalar processors. In the interval analysis, execution time is partitioned into discrete intervals by disruptive miss events such as cache misses and branch misprediction as shown in Figure 1. It is based on two observations as follows:

- A superscalar out-of-order processor executes the number of instructions equal to a steady-state IPC (instruction per cycle), $SIPC$, at every clock cycle. Since the maximum number of instructions that enter the reorder buffer at every clock cycle is the dispatch width, the dispatch width will be the steady-state IPC in the ideal case. However, the actual value is smaller than the dispatch width and how to compute the steady-state IPC is the

key challenge of this method. The steady-state IPC is defined as the average number of committed instructions per clock cycle when cache accesses always hit, and all branches are correctly predicted. Then the total execution cycle becomes the number of committed instructions divided by the steady-state IPC under such ideal conditions.

- Disruptive miss events, such as cache misses and branch misprediction, interrupt the smooth flow of instruction execution. For example, when an instruction cache miss occurs, no more instructions are fetched until the cache miss is properly handled. Then cache miss penalty should be accounted for in the analysis.

Under these observations, the total execution cycle T is approximated as the following simple formula:

$$T = \frac{N_{total}}{SIPC} + \sum m_i \cdot p_i \tag{1}$$

N_{total} is the total number of simulated instructions, m_i is the number of disruptive events of type i, and p_i is the performance penalty of the event of type i. The disruptive events generally include L1 instruction cache misses, non-overlapping L2 cache misses, and branch mispredictions. Note that L1 data cache misses are not included. The performance penalty is determined based on the type of the event.

- **For instruction cache miss:** Instruction cache misses block the continuous inflow of new instructions until the miss event is handled. Hence, the first level instruction cache miss penalty becomes the performance penalty.

- **For data cache miss:** Data cache misses are divided into two categories [10]: short misses have latency significantly less than the maximum reorder buffer fill time, while long misses have latency significantly greater than the maximum reorder buffer fill time. Short miss does not block the instruction inflow to the reorder buffer, so a short miss instruction is treated as a long-latency instruction in the architectural point of view. On the other hand, a long miss usually blocks the continuous instruction inflow to the reorder buffer, because the reorder buffer becomes full until the miss event is properly served. Hence, the performance penalty of a long miss often becomes the cache miss penalty. It is generally assumed that a first-level data cache miss is a short miss while a higher-level (second- or third-level) data cache miss is a long miss.

- **For branch misprediction:** Branch misprediction penalty becomes the sum of branch resolution time and the front-end pipeline depth where the front-pipeline length denotes the latency between instruction fetch and instruction dispatch. Once the mispredicted branch enters the instruction queue, no more useful instructions enter the instruction queue until the mispredicted branch is resolved. After the branch misprediction is detected and the correct branch target instruction is fetched, the pipeline is flushed and fetching begins from the correct path. The correct path instructions take front-end pipeline depth cycles to reach the instruction queue.

If several long misses occur in a short period of time, the corresponding miss penalties may be overlapped since multiple cache misses can be handled concurrently in the architecture. We identify the overlapped case by checking if the distance between the first and second cache miss events is smaller than the size of reorder buffer. Overlapped long cache misses are counted

only once, thereby exposing memory-level parallelism (MLP). In contrast, consecutive branch mispredictions are not overlapped. Let us suppose that two consecutive branch mispredictions occur as an example of bursty branch mispredictions. In our model, the second mispredicted branch enters the instruction queue just after front-end pipeline refilling, and no more useful instruction enters the instruction queue until the second mispredicted branch is resolved. As a result, restoring IPC to the steady-state IPC is delayed again by the branch resolution time plus the front-end pipeline depth like the first misprediction. After all, bursty branch mispredictions do not affect the corresponding branch misprediction penalties.

In the analytical formula, cache miss penalties are easily known when the architecture configuration is settled. The numbers of cache misses and branch mispredictions are obtained by a cache simulator and a branch predictor, which are attached to the functional simulator. However, the branch misprediction penalty varies depending on the situation when the mispredicted branch is fetched. The steady-state IPC also varies depending on programs and the ranges of interest even in the same program.

The simplest way to calculate the steady-state IPC might be to perform cycle-accurate simulation with perfect caches and a perfect branch predictor. Since this method requires a cycle-accurate simulation, this approach is not appropriate for our purpose. Besides, a cycle-accurate simulator may not be available at the early stage of architectural design.

Another way to estimate the steady-state IPC is to use **IW** characteristics, based on the average functional unit latency and Little's law as proposed in an earlier work [10]. **IW** characteristic is a function that determines the number of instructions that **I**ssue in a clock cycle, given the number of instructions in the **W**indow. **IW** characteristic is represented by a curve $\mathbf{I} = \alpha \mathbf{W}^{\beta}/L$, where the values of α and β are specific to each benchmark, and L denotes the average instruction latency. However, estimating **W** without an accurate pipeline simulation is not trivial. Even worse, obtaining the values of coefficient α and β requires the analysis of applications.

Another approach has been proposed to use the critical path length of the instruction stream to obtain the steady-state IPC [12]. Conceptually, a ROB-sized window slides along the dynamic instruction stream. Intuitively, the window cannot slide faster than the rate that the processor is issuing instructions belonging to the critical path, which is the longest data dependence chain for that window. Since finding the exact critical path length is fairly time-consuming task, the authors of [12] also presented an approximation method to compute the critical path length in the window. Resource contention modeling for interval simulation has recently been proposed to consider the number of functional units [30].

In this paper, we propose to use a sampled simulation that determines the steady-state IPC. By using the in-house trace analyzer for a sampled window of instruction streams, we reduce the simulation performance loss, while maintaining the high accuracy of the steady-state IPC.

2.3. Timing Simulation based on QEMU

QEMU is a generic and open source machine emulator based on a portable dynamic translator to execute unmodified operating systems and application programs of a target architecture on a host [32]. QEMU is widely used for a number of reasons such as fast simulation speed, portability, and popularity. The most important benefit of using QEMU is the portability that supports multiple hosts and target CPUs including Alpha, ARM, CRIS, x86, x86-64, MicroBlaze, MIPS, PPC, and Sparc. Hence, many timing simulators [24, 33, 26, 27], including ours, are based on QEMU to enjoy its benefits.

Figure 2 presents the QEMU emulation model. The translation process of QEMU consists of two phases, front-end and back-end. The front-end translates the target binary code to inter-

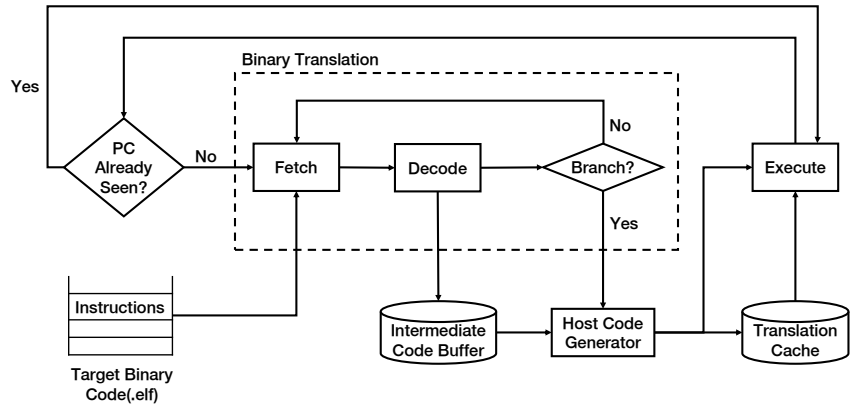


Figure 2: QEMU emulation model [31]

mediate code, and the back-end translates the generated intermediate code to the host executable code. The two-phase approach is particularly beneficial to support a new target or a new host machine. Translated host binary codes are stored in the translation cache, which enables QEMU to spend only 1% of total execution time for code translation [34].

For timing simulation, it is required to obtain information such as the instruction trace executed and the accessed memory addresses. Since QEMU executes the translated block as an atomic unit instead of instruction by instruction, extra instructions to collect such information are embedded when target code is being translated to the intermediate code. To this end, most existing QEMU-based simulators [24, 33], including ours, use QEMU's opcode helper functions to implement and embed complex guest instructions.

Google developed a QEMU-based Android Emulator, so software developers could develop Android applications in the absence of the Android smartphone [35]. The Android emulator is known to use a primitive k-CPI model to estimate program performance by multiplying the instruction counts by the executed cycles consumed by each type of instructions. The estimated timing is far from accurate for a modern embedded processor with the complex pipeline and cache memories.

In [36], the virtual timing device was proposed and implemented in QEMU, estimating the execution time with a linear regression model to find the correlation between the cycle counts and instruction counts. The linear regression model only needs to count the number of instructions by each type of instructions, which enables a fast simulation. However, timing-accurate simulations with a reference cycle-accurate simulator or a real hardware are required with an extensive training set that covers the huge range of applications. The datasheet model was also suggested to figure out the cycle counts for each instruction by consulting a table built upon the specifications datasheet from the simulated platform, but the model is fundamentally quite inaccurate for a modern complex processor.

The authors of [31] encapsulated the QEMU platform in a SystemC [37] module for transaction-level modeling. The number of cycles of instructions are obtained from the datasheet of the simulated processor and incremented during the simulation. In this approach the effect of out-of-order superscalar pipeline and the branch misprediction cannot be properly considered. In [33], QEMU is responsible for target system emulation while cycle-accurate simulation is

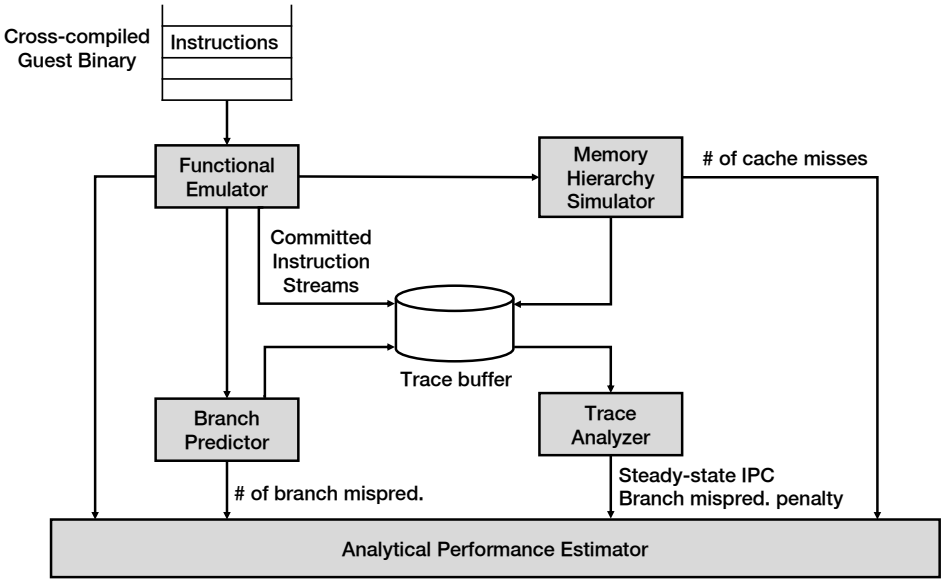


Figure 3: Structure of the proposed simulator

performed in the SystemC environment: QEMU sends the information of each instruction being executed in the SystemC environment, which simulates the HW at the cycle-accurate level based on that information. Unfortunately, the interaction with SystemC at every instruction considerably may slow down the simulation. To the best of our knowledge, the developed simulator was not validated through an accurate cycle-accurate simulator or a real hardware.

Fujitsu Labs proposed to include a pipeline model into QEMU [34]. They use a two-phase approach (offline and online phases) to estimate the application performance. In the offline phase the application execution time is pre-estimated. Pre-computed information is then used in the dynamic adaptation phase where CPU status and execution time of critical instructions are also taken into account, improving the estimation accuracy. Code generation and analysis are performed on a basic block basis. It is not clear, however, how the interblock effect and the performance penalty of branch misprediction are considered.

3. Proposed Timing Model

3.1. Overview

In this paper, we propose a combined analytical/sampled timing simulation technique, which is independent of the specific functional simulator. As the processor timing model, we adopt an analytical modeling technique, the interval simulation technique [12]. Essential parameters in the formula are estimated through sampled simulation with a trace analyzer, while the other parameters are obtained from architecture specification and functional simulation. As shown in Figure 3 with rectangle boxes, the structure of the proposed simulator consists of five components: functional simulator, branch predictor, memory hierarchy simulator, trace analyzer, and analytic performance estimator.

The base component of the simulator is the functional simulator. The functional simulator executes a cross-compiled target-machine binary on a host-machine. The functional simulator decodes target machine instructions, and sends minimum necessary information to the other components. The memory access information is transferred to the memory hierarchy simulator to detect the cache miss events. Branch instructions are sent to the branch predictor to detect the branch misprediction events. Sampled instruction streams are buffered to the trace buffer for the trace analyzer. The trace analyzer is invoked in a background thread only when the buffer is full, so that the time overhead of invoking the trace analyzer can be hidden. The analytic performance estimator calculates the time duration of the specific intervals according to the analytic formula, based on the parameter values collected from other simulator components. Each component is a modular, deployable, replaceable part of a simulator that could be configured to a specific processor by changing the parameter values.

3.2. Sampling Mechanism

Using accurate steady-state IPC and branch misprediction penalty is essential to perform interval simulation. In particular, the steady-state IPC is affected by dependencies between instructions, execution time of instructions, and micro-architecture of a processor. Hence, obtaining the accurate steady-state IPC in a reasonable time is a challenging task.

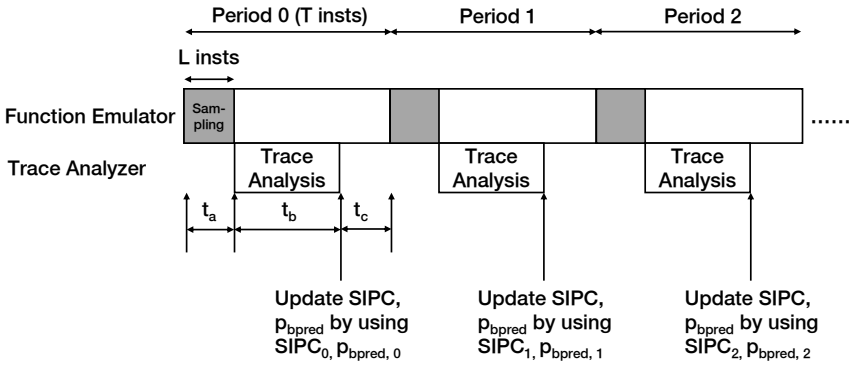
Analysis of the whole instruction stream is against the philosophy of analytical simulation, which is designed to predict the performance by a simple formula. Moreover, the speed of such analysis is significantly slower than a functional simulator augmented with a cache simulator and a branch predictor. Existing studies of sampled simulation have demonstrated the potential of using representative sampled traces. Thus we propose to use steady-state IPC and branch misprediction penalty obtained from the collected traces.

The proposed sampled simulation is different from conventional sampled simulation which depends on timing information of sampling units to predict the overall performance of an application. It is worth noting that we use sampled traces to estimate the steady-state IPC and the average misprediction penalty, so that the trace analyzer requires a minimal abstract model to schedule the sampled instruction trace. Hence, the trace analyzer has been implemented in only several thousand lines of codes with a minimal development effort. Most of discrepancies between the trace analyzer and a cycle-accurate simulator stem from omitted or unknown processor details. In addition, the choice of sampling units does not have a significant effect on the accuracy of timing estimation of the proposed technique if the sampling unit has enough instructions to warm up the window, which will be discussed with experimental results later.

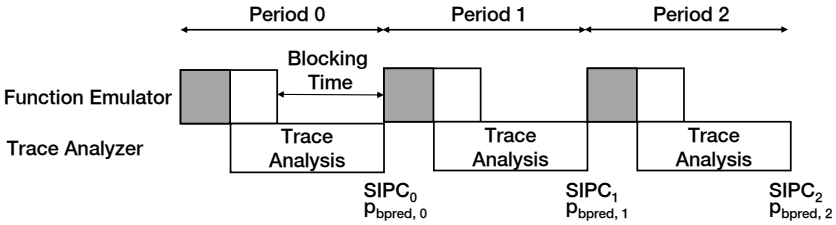
3.2.1. Sampling Configuration

In sampled simulation, the sampling units are selected either randomly, periodically, or based on phase analysis. Among them, we use periodic sampling, which is the simplest form to implement. Hence, a sampling configuration (L/T) is specified as the sampling size L and the sampling period T in terms of the number of instructions, and the sampling duty cycle is given by $L/T \times 100(\%)$. Once the specified number of instructions is collected in the buffer as the functional simulator proceeds, the trace analyzer is invoked for the buffer as a separate thread. The overhead of executing the trace analyzer is to some degree hidden by the use of an independent thread. This approach minimizes the overhead of running the trace analyzer, yet achieving reasonably accurate timing result. The overall simulation progress is described in Figure 4.

Excessively high duty cycle slows down the total simulation time, without improving the accuracy any more. Although the trace analysis is performed in a separate thread, we maintain



(a) How to make use of steady-state IPC and branch misprediction penalty obtained from the analysis of the sampled trace



(b) QEMU is blocked until the trace analyzer processes all instructions in the trace buffer

Figure 4: Simulation progress with the trace analyzer if L instructions are collected periodically every T instructions.

only one trace buffer. Once the duty cycle increases after a certain breakpoint and $t_a + t_b$ is longer than the sampling period, the functional simulator has to be blocked until the trace analyzer processes all instructions in the trace buffer as described in Figure 4(b). Therefore, the simulation time dramatically increases after the breakpoint.

We have to consider the trade-off relationship between the simulation speed and the accuracy when determining the L and T values, which will be discussed with experimental results in the next section.

3.2.2. Parameter Extraction

The analytical formula of the simulator uses the mean values of steady-state IPCs and branch misprediction penalties of sampled trace. Suppose that the application has n sampling periods $\{0, 1, \ldots, n-1\}$. Let N_i and $m_{bpred,i}$ the number of instructions and mispredicted branches that belong to i-th sampling period, respectively. If the steady-state IPC and the branch misprediction penalty of i-th period are given as $SIPC_i$ and $p_{bpred,i}$, the mean values of steady-state IPCs and

branch misprediction penalties are defined as follows:

$$SIPC_{mean} = \frac{n}{\frac{1}{SIPC_0} + ... + \frac{1}{SIPC_{n-1}}} \tag{2}$$

$$p_{bpred,mean} = \frac{p_{bpred,0} \cdot m_{bpred,0} + ... + p_{bpred,n-1} \cdot m_{bpred,n-1}}{m_{bpred}} \tag{3}$$

$SIPC_{mean}$ is the harmonic mean of $SIPC_0, SIPC_1, \ldots, SIPC_{n-1}$; $p_{bpred,mean}$ is the weighted arithmetic mean of $p_{bpred,0}, p_{bpred,1}, \ldots, p_{bpred,n-1}$ where each weight is determined by the ratio of the number of mispredicted branches in the sampling period to the total number of mispredicted branches.

Given analytical formula (1), $SIPC_i$ and $p_{bpred,i}$ of the i-th sampling period, the simulated cycles of n periods can be calculated as [1]:

$$T = \frac{N_0}{SIPC_0} + m_{bpred,0} \cdot p_{bpred,0} + \ldots \frac{N_{n-1}}{SIPC_{n-1}} + m_{bpred,n-1} \cdot p_{bpred,n-1} \tag{4}$$

As we use a fixed sampling length and period, $N_0 = ... = N_{n-1} = N_{total}/n$ holds. Let us combine $(N_i/SIPC_i)$-type terms in equation (4):

$$T' = \frac{N_0}{SIPC_0} + \frac{N_1}{SIPC_1} + \ldots + \frac{N_{n-1}}{SIPC_{n-1}} \tag{5}$$

$$= \frac{N_{total}}{n \cdot SIPC_0} + \frac{N_{total}}{n \cdot SIPC_1} + \ldots + \frac{N_{total}}{n \cdot SIPC_{n-1}} \tag{6}$$

$$= \frac{N_{total}}{n}(\frac{1}{SIPC_0} + \cdots \frac{1}{SIPC_{n-1}}) \tag{7}$$

$$= \frac{N_{total}}{SIPC_{mean}} \tag{8}$$

On the other hand, combining $(m_{bpred,i} \cdot p_{bpred,i})$-type terms in equation (4) gives the following result:

$$T'' = m_{bpred,0} \cdot p_{bpred,0} + \ldots + m_{bpred,n-1} \cdot p_{bpred,n-1} \tag{9}$$

$$= \frac{m_{bpred,0} \cdot p_{bpred,0} + \ldots + m_{bpred,n-1} \cdot p_{bpred,n-1}}{m_{bpred}} \times m_{bpred} \tag{10}$$

$$= p_{bpred,mean} \cdot m_{bpred} \tag{11}$$

Hence, equation (4) can be rewritten:

$$T = \frac{N_{total}}{SIPC_{mean}} + p_{bpred,mean} \cdot m_{bpred} \tag{12}$$

[1] Additional simulated cycles incurred by cache misses are not affected by the steady-state IPC or branch misprediction penalty, so we can ignore them here

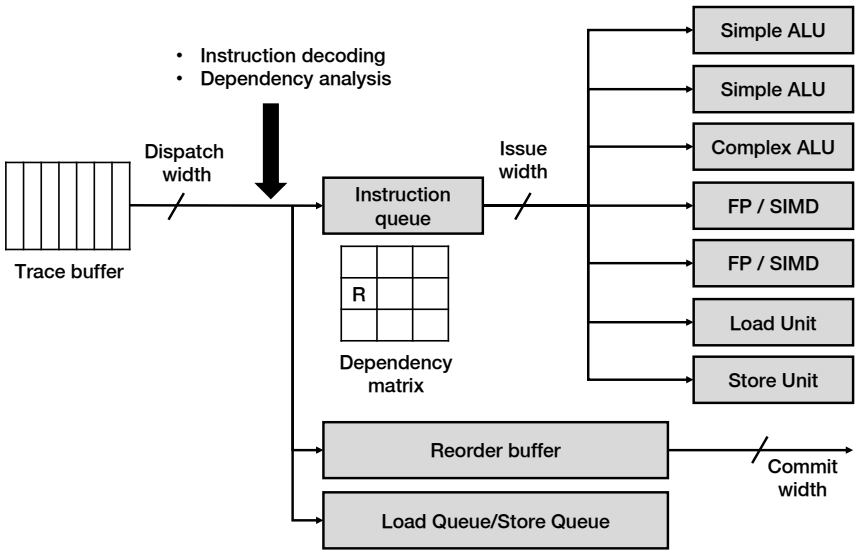


Figure 5: Core structure of the trace analyzer when it is configured for ARM Cortex A-15

It implies that we may use the mean values to obtain the simulated cycle.

Concurrent execution of the functional simulator and the trace analyzer enables faster simulation. However, the simulated cycle at a specific point of time has to be calculated based on parameters which do not include results of the current period. We illustrated this situation in Figure 4(a). If the sampling period is $t_a + t_b + t_c$ seconds, the analytical formula has to use old parameters during $t_a + t_b$ seconds, possibly degrading the accuracy of the simulated cycle estimated from the proposed technique. Furthermore, the point in simulated time when the updated parameters for the analytical model become available depends on the execution time of the trace analyzer. It would make the approach non-deterministic (results vary between runs on the same hosts) and simulation results depend on the performance of the simulation host (results vary between different hosts). However, we believe that its impact is not significant, compared with the accuracy error caused by the other factors. Non-determinism would be minimized if the simulator exclusively uses a homogeneous SMP (Symmetric multiprocessor system), which is the common simulation environment. The worst case scenario will occur when parameters of the current period are not available until simulating all instructions in the period is completed. Note that its impact is similar to the situation that we double the sampling period. Experiments in the paper confirmed that once a sampling duty cycle exceeds a certain point (≥ 0.001), the sampling duty cycle is mostly irrelevant to the simulation accuracy.

3.3. Trace Analyzer

Instead of modeling a specific micro-architecture of a processor, we developed a trace analyzer for generic out-of-order superscalar processors. We assume that the target out-of-order processor is well balanced so that as many instructions as the dispatch width can be put into the window at every clock cycle if the reorder buffer has available space and the cache miss or branch misprediction does not occur. These conditions are generally satisfied with the sufficient

Obtained from the functional simulator augmented with the cache and branch simulator				Obtained after instruction decoding			
Machine Code	Effective Memory Address	L1 Data Cache Miss	Branch Prediction Correctness	Instruction Type	Target Functional Unit	Source Registers	Target Registers
0a000001			Correct	Branch	SimpleALU	Cond. Flags	
e0854004				IntALU	SimpleALU	R4, R5	R4
e59f4064	0x1fac8	Y		MemRead	LoadUnit	PC	R4

Figure 6: Internal structure of the trace buffer

size of fetch, decode, rename width.

The trace analyzer focuses on the flow of instructions from the dispatch stage of a superscalar out-of-order processor; it models dispatch, issue, writeback, and commit stages only, while abstracting out fetch, decode, and rename stages. The core structure of the trace analyzer is illustrated in Figure 5. Since the trace analyzer uses the trace generated from a functional simulator, it needs not be concerned about functional correctness.

After an instruction is executed by the functional simulator that is augmented with the cache and branch simulator, a new trace entry is added to the trace buffer queue. The trace analyzer waits until the trace buffer is full. Three example entries in the trace buffer are illustrated in Figure 6. Left four columns of the table depict the entries (machine code, effective memory addresses, L1 data cache miss, and branch prediction correctness) that are obtained from functional simulation.

When an instruction in the trace buffer is dispatched into the instruction queue and the reorder buffer, the trace analyzer decodes the machine code of the instruction; so the right four columns of the table (instruction type, target functional units, source register, and target register) can be filled. At the same time, we perform the preliminary dependency checking.

How to use this information in the trace buffer is summarized as follows:

- *Source register and target register:* to find the register dependency.
- *Effective address:* to find the memory dependency.
- *L1 data cache miss:* to add an additional latency of the instruction which caused L1 data cache miss. Recall that a short miss instruction is treated as a long-latency instruction in the architectural point of view. Consequently, the effect of L1 data cache misses is included in the steady-state IPC.
- *Branch prediction correctness:* to track the branch resolution time of the instruction that makes a misprediction.
- *Instruction type and target functional unit:* to issue the instruction to the associated functional unit.

Once all instructions in the trace buffer are processed by the trace analyzer, the steady-state IPC and branch misprediction penalty of the specific period are derived: the steady-state IPC is the value of the sampling length (in the number of instructions) divided by cycles required to process the trace buffer; the branch misprediction penalty is the value of the accumulated misprediction penalties divided by the number of mispredicted branches throughout the specific period. Those values are used to calculate the mean values of steady-state IPC and misprediction penalty for the entire application.

Reorder buffer

Idx	0	1	2	3	4	5	6	7	8	9
	inst_8	inst_9	inst_0	inst_1	Inst_2	Inst_3	Inst_4	Inst_5	Inst_6	Inst_7

Relationship between Instructions

Register Dependency
inst_0 (2)-> inst_2 (4)
inst_6 (8)-> inst_8 (0)
inst 5 (7)-> inst_9 (1)

Control Dependency
inst_3 (5)-> inst_7 (9)

Memory Dependency
inst_2 (4)-> inst_8 (0)

Dependency Matrix

	0	1	2	3	4	5	6	7	8	9
0										
1										
2					R					
3										
4	M									
5										C
6										
7		R								
8	R									
9										

Figure 7: Dependency matrix

3.3.1. Dependency Analysis

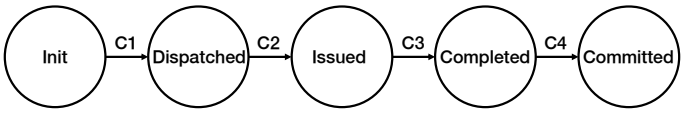
The important technique involved in the trace analyzer is to manage dependency between instructions. In addition to data dependencies, we have to consider control and memory dependencies carefully. We define a dependency matrix to represent the relationship between instructions.

The reorder buffer is a circular queue which contains all in-flight instructions, namely, all instructions that have been dispatched but have not yet committed. We use the fact that the index of an instruction in the reorder buffer is unique and invariable once the instruction is allocated to the reorder buffer. Suppose that the size of the reorder buffer is W_{rob}. Then the dependency matrix is a square $W_{rob} \times W_{rob}$ matrix M such that $M[i, j]$ is 'R' when there is a register dependency from the i-th instruction to the j-th instruction in the reorder buffer. 'C' and 'M' are used to denote control and memory dependency, respectively. At the point of dispatching an instruction, the initial dependency analysis of the instruction is conducted to fill the dependency matrix. The dependency matrix is, later, used to look up the state of previous instructions that the current instruction depends on.

Figure 7 illustrates the simple dependency matrix when $W_{rob} = 10$ and $inst_i$ is older instruction than $inst_j$ if $i < j$. For example, register dependency from $inst_0$ to $inst_2$ is expressed as $M[2, 4] =$ 'R'.

3.3.2. Life Cycle of an Instruction

The life cycle of an instruction can be specified by a finite-state machine (FSM) as shown in Figure 8. Each instruction may have five states from the set {*Init, Dispatched, Issued, Completed, Committed*}. An instruction is initially in *Init* state. If an instruction in the *Init* state is allocated to the instruction queue and the reorder buffer, the state of the instruction becomes *Dispatched*. If an instruction in *Dispatched* state becomes executable and the corresponding functional units are available, the instruction is issued to the corresponding functional unit, entering into the *Is-*



	Conditions
C1	The remaining dispatch width is available / ROB & IQ & LDQ & STQ have an available buffer slot
C2	The remaining issue width is available / all types of dependencies are met / the functional unit is available
C3	The execution time is elapsed
C4	The remaining commit width is available / the instruction is the oldest instruction in the ROB

Figure 8: Life cycle of an instruction. The set of transition conditions is associated with each edge.

Table 1: Simulated target system characteristics

Parameter	Value
CPU Codename	Cortex-A15
ARM ISA	ARMv7-A (32-bit)
Core clocks	2.0 GHz
Front-end width	3
Back-end width	8
Front-end pipeline depth	12
Branch predictor	Bi-Mode
IQ enties	48
LSQ entries	16 each
ROB entries	60
Functional Units	2 simpleALU, 1 complexALU, 2 FP/SIMD unit, 1 load unit, 1 store unit
L1 I-cache	32KB 2-way set-associative, 64B lines
L1 D-cache	32KB 2-way set-associative, 64B lines
L2 cache	unified, 1MB, 16-way set-associative, 64B lines, 10 cycle access time
Main memory	100 ns access time

sued state. Instructions of *Issued* state are immediately removed from the instruction queue. The number of instructions issued per cycle may not exceed the maximum issue width. Once the execution time of the issued instruction elapses, the instruction is claimed completed and released from the functional unit, moving to the *Completed* state. As many completed instructions as the commit width may leave the window if they are the oldest instructions in the window. Instructions of *Committed* state are immediately removed from the reorder buffer.

4. Experimental Results

Simulation target system: To evaluate the proposed technique, we first configure the simulation target system based on the Cortex A15 processor as closely as possible. Table 1 shows the system characteristics with key parameter values. By modifying some parameters, we will evaluate the adaptability of the proposed simulation technique, compared with the reference simulator in terms of accuracy.

Reference simulator / simulation error: For the reference simulator to compare the accuracy and speed of the proposed simulator, we chose a state-of-the-art open-source cycle-accurate

Table 2: Benchmarks description with input sets and ratios (%) of each instruction type

Benchmark	Configuration	Arith(Short)	Arith(Long)	Branch	Memory
basicmath	small	83.49	1.24	14.54	15.17
bitcnts	75000 items	90.90	0.00	13.62	9.10
qsort	small	63.84	0.14	19.78	36.02
susan	input_small.pgm -s	55.25	16.60	10.53	28.12
jpeg	-dct int -progressive -opt	57.84	1.39	11.17	40.77
dijkstra	small	78.48	0.07	22.76	30.86
patricia	small.udp	67.45	0.53	17.48	32.00
ispell	-a tests/small.txt	64.23	0.08	15.77	35.69
stringsearch	large	69.89	0.00	20.14	30.11
rijndael	input_small.asc	63.73	0.11	6.39	36.16
sha	input_small.asc	74.56	0.00	5.70	25.44
fft	4 4096	78.59	1.73	16.54	19.58
adpdm	data/small.pcm	84.25	0.00	6.86	15.75
gsm	-fps -c data/small.au	45.44	12.44	7.96	42.12

simulator GEM5 [38]. Using open-source cycle-accurate simulator as a reference simulator has two advantages: 1) It is easy to identify the source of simulation errors since the reference simulator provides important system statistics; 2) It is easy to change the parameters of the system and CPU. If we use only one microarhictecture, it is easy to tune a simulator or a model for the target microarhictecture. It is reported that the maximum error of GEM5 is 15% with some exceptions in the SPEC benchmarks (gems, milc, namd) [39]. Therefore, we decided that comparison with GEM5 is a good starting point for simulation development. However, we aware that comparing the results of the presented approach to GEM5 may lead to more simulation error when the proposed simulator is compared to the real hardware. We leave it as a future work to compare with a real processor.

We use the number of instructions that execute per cycle (IPC) to evaluate the overall accuracy of the proposed simulation. We define the IPC error:

$$IPC\ error = \frac{IPC_t - IPC_r}{IPC_r} \tag{13}$$

IPC_r is the reference IPC measured by the reference simulator; IPC_t comes from TQSIM. The average absolute IPC error is also defined as:

$$Average\ absolute\ IPC\ error = \frac{1}{n}(\sum^{n} \| \frac{IPC_t - IPC_r}{IPC_r} \|), \text{ where } n \text{ is the number of benchmarks} \tag{14}$$

In the context of design space exploration, the accuracy of speedup/slowdown prediction is also important, which indicates the ability of a simulation technique to predict how well our processor model tracks the results of detailed simulation by the reference simulator. Hence, we define the prediction error as the relative error of the predicted speedup/slowdown:

$$Prediction\ error = (\frac{IPC_{t,B}}{IPC_{t,A}} - \frac{IPC_{r,B}}{IPC_{r,A}})/(\frac{IPC_{r,B}}{IPC_{r,A}}) \tag{15}$$

$IPC_{t,B}$, $IPC_{t,A}$ denotes the IPC values obtained from TQSIM when running on two different microarhictectures A and B; $IPC_{r,B}$, $IPC_{r,A}$ are the similarly defined IPCs for the reference

simulator. This definition quantifies the degree of difference in IPC increase/decrease obtained from TQSIM versus the IPC increase/decrease obtained from the reference simulator.

Benchmark application: We use 14 of the MiBench benchmarks [40]. See Table 2 for more details on these applications and the inputs that we have used. The frequency of occurrence of each instruction type is given to represent the application characteristics. The application selection procedure is based on the following steps: First, we choose only one application among applications with similar code characteristics. For example, we use only one application between FFT and IFFT applications. Second, we remove applications which failed to be built on our host system. Third, we remove applications which failed to be simulated with GEM5. Finally, we have 4 automotive, 2 networking, 1 consumer, 2 office, 2 security, and 3 telecom applications as the benchmark set. We believe that 14 applications and 5 different system configurations, 70 combinations, are enough to validate the generality of the proposed approach.

Simulation environment: All guest binaries were cross-compiled with *arm-linux-gnueabi-gcc* tool-sets for the ARM processor. The simulation host machine is equipped with an Intel i7-3770K processor clocked at 3.50GHz, 32 GB main memory, and Ubuntu Linux 64bit.

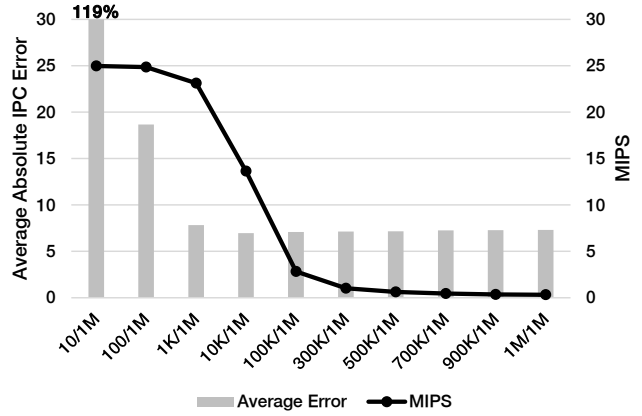
4.1. Time-accuracy Trade-off Controlled by the Sampling Configuration

First, we explore various sampling configurations to find the best compromise between the simulation error and the simulation speed. To separate the impact of the sampling size from that of the sampling period, we fix the sampling period as one million instructions, and vary the sampling size from ten to one million instructions. The simulation accuracy and speed are plotted in Figure 9(a). Increasing the sampling size results in lower error by having more instructions to be analyzed. From the experimental results, we observed that the average simulation error is stabilized under 7.5% after 1,000 instructions. Inevitably, the window is empty when each trace analysis begins with the sampled instruction trace. Hence we need to warm up the window to estimate the steady-state IPC in the trace analysis. The estimation error becomes more significant when the length of sampled instructions is too short to warm up the window. Meanwhile, the simulation speed decreases slowly until 1,000 sampling size, after which it decreases rapidly. Such a rapid drop is attributable to the congested trace buffer. The trace analysis becomes the speed bottleneck since writing new instructions to the trace buffer is blocked until the previous analysis finishes.

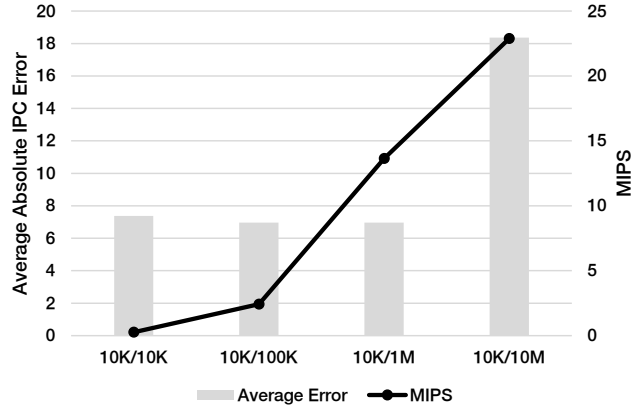
To measure the impact of various sampling periods, we use the constant sampling size, 10,000 instructions, and vary the sampling period from 10,000 to 10 millions. Figure 9(b) shows that a long sampling period does not necessarily cause the accuracy loss until it becomes too large. Unlike the sampling size, the simulation accuracy is not sensitive to the sampling period unless the number of sampled traces is too small or the application's IPC characteristic changes too much. On the other hand, the simulation speed is significantly degraded if sampling the instruction trace is too frequent such as (10K/10K) and (10K/100K).

Figure 9(c) indicates that sampling configuration (1K/1M) or (10K/1M) results in the best compromise between simulation time and accuracy for the given experimental setting and the benchmark applications.

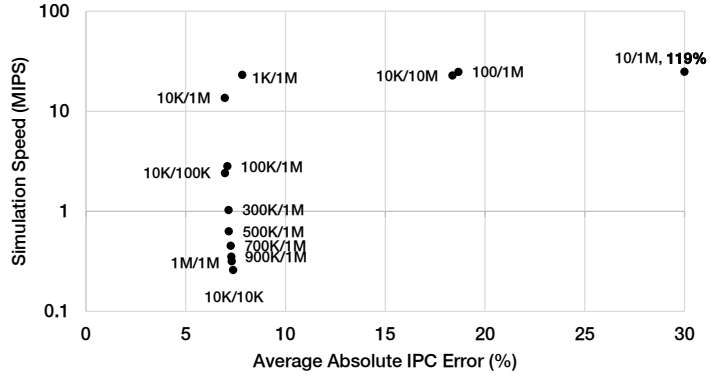
We observed that the breakpoint is around 0.1-1% duty cycles, where the simulation time dramatically increases. It is mainly due to the trace buffer blocking described in Figure 4(b). To mitigate the blocking penalty, we may maintain multiple trace buffers in order to overlap multiple trace analyses. However, it will not improve simulation accuracy.



(a) Various sampling sizes and the fixed sampling period (one million instructions)



(b) Various sampling periods and the fixed sampling size (10,000 instructions)



(c) Simulation error and time controlled by the sampling sizes and sampling period

Figure 9: The time-accuracy trade-off controlled by the sampling configuration

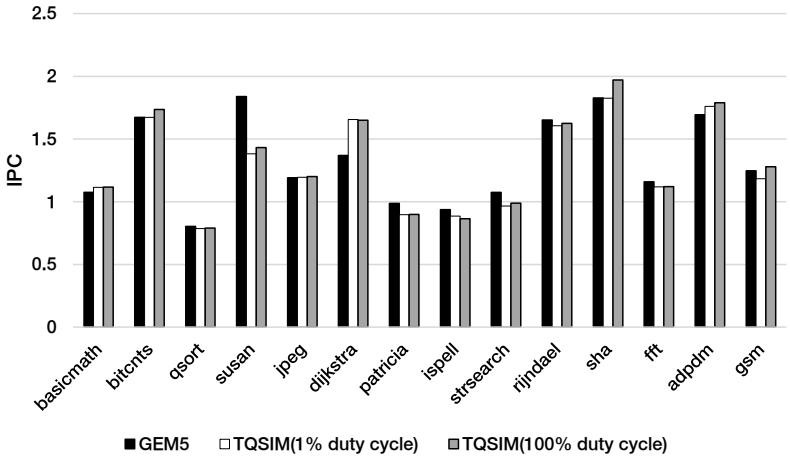


Figure 10: IPC (instructions per cycle) obtained from GEM5 and TQSIMs

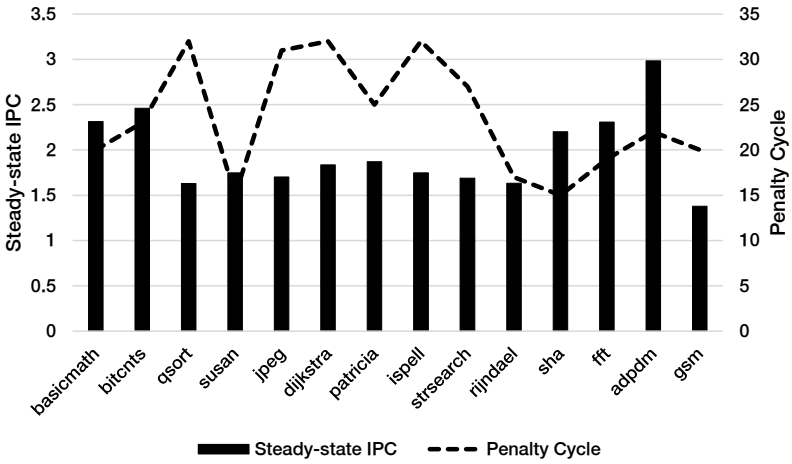


Figure 11: Steady-state IPC and branch misprediction penalties for benchmarks

4.2. Accuracy of TQSIM

4.2.1. IPC Error

To validate the accuracy of TQSIM, we first validate the trace analyzer in isolation by changing the duty cycle of sampling from 100% duty cycle (sample always) to sparse sampling (1%). We measured the total simulated cycles and the overall IPCs obtained from TQSIM and GEM5. The result is shown in Figure 10. TQSIM with 100% duty cycle gives a simulation cycle error of -17.00% - 28.44%, with an average absolute error of 7.31 %. We believe that this error is acceptable since we model the architecture at a very high level unlike general cycle-accurate simulators usually do. For reference, the error of 1-CPI approach was measured to have 36.23% on average and 83.77% at maximum. It is also seen that IPC differences from duty cycle 100% and 1% represent a negligible amount, which justifies, to some degree, our sampling approach.

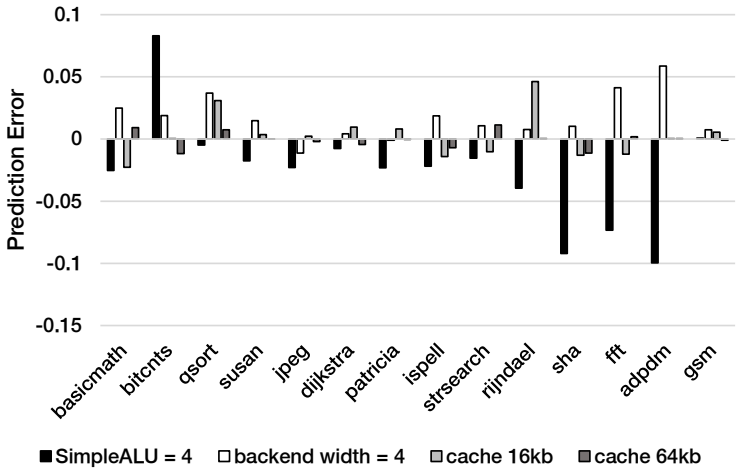


Figure 12: Prediction error for four different configurations

Figure 11 shows that the steady-state IPC and the branch misprediction penalty significantly vary with each benchmark. For example, the steady-state IPC of *adpdm* is nearly double of that of *qsort*. Hence, estimating those values for each application is demanded.

4.2.2. Prediction Error

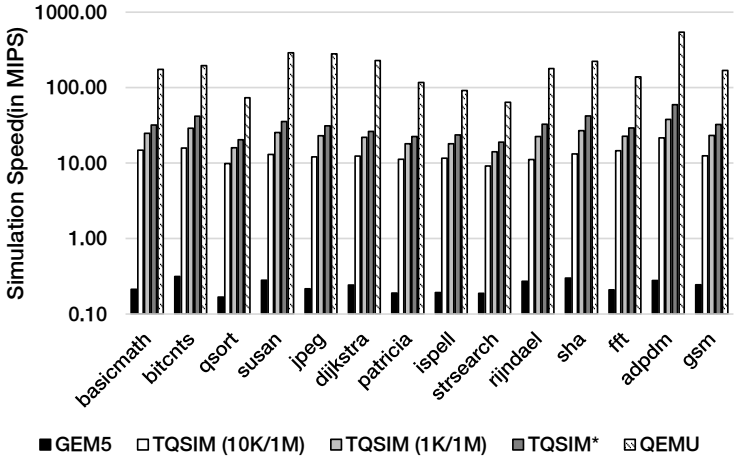
To show how well the proposed technique predicts the performance difference between various configurations, we measured the prediction errors for the following four configuration changes: 1) the number of SimpleALUs is increased to 4, 2) the back-end pipeline width is reduced to 4, 3) instruction/data L1 cache size is decreased to 16KB, and 4) instruction/data L1 cache size is increased to 64KB.

The prediction error for four different configurations is presented in Figure 12, where the architecture A and B represent Cortex A15 and a new configuration respectively in the prediction error formula. According to the formula, the positive value indicates overestimation of IPC increase or underestimation of IPC decrease, whereas the negative value indicates underestimation of IPC increase or overestimation of IPC decrease.

It is observed that the average absolute prediction error is 1.9%, with a maximum of 11.1% (SimpleALU = 4). High prediction error observed with (SimpleALU = 4) implies a discrepancy of the instruction issue mechanism of the trace analyzer and the reference simulator. Further improvement is needed to ensure more reliable simulation results.

4.2.3. Source of Errors

The main source of error is definitely the highly abstracted processor model. In addition, some accuracy loss is inevitable if a functional emulator is used as the base simulator. For instance, as explained earlier, a functional simulator does not execute instructions on the mispredicted path, which may influence the cache simulator and branch predictor. Another cause of accuracy loss is discrepancies between the trace analyzer and the cycle accurate simulator. Timing accuracy would be improved if a more detailed mechanism of the target processor is modeled in the trace analyzer. For fair comparison with GEM5 syscall emulation mode that does



	GEM5	TQSIM(10K/1M)	TQSIM(1K/1M)	TQSIM*	QEMU
Average	0.24	13.09	23.11	32.04	197.74
Minimum	0.17	9.19	14.08	18.99	63.74
Maximum	0.32	21.56	37.81	59.47	543.72

Figure 13: Simulation speed (in MIPS)

not consider the system call overhead, we also ignored the system call overhead in the current implementation while it is possible to define a fixed system call overhead for each kind of system call.

4.3. Speed of the TQSIM

We measured the simulation speed of TQSIM, along with GEM5 and the original QEMU. The simulation speed of TQSIM without the trace analyzer (TQSIM*) is also measured to examine the overhead of sampling and the trace analysis. The simulation speed is presented in Figure 13. The speed of TQSIM is 41 (10K/1M) - 135 times (1K/1M) faster than that of GEM5. Comparing the original QEMU and the TQSIM without the trace analyzer, we observed that instrumenting and calling helper functions to use cache/branch predictor slows down simulation by 3.36 - 9.14 times. Executing the trace analyzer with sampled instructions slows down simulation further 1.20 -1.57 times more. As we saw in the previous experiments, TQSIM demonstrates reasonable accuracy (below 8% error), while showing one or two orders of magnitude faster than a cycle-accurate simulator. This result confirms the usefulness of the proposed technique and TQSIM.

5. Conclusion

In this paper, we propose a fast timed-simulation technique supporting modern superscalar out-of-order processors. The simulator is developed on QEMU that is an open source machine emulator. For timing estimation, we use a novel combined analytical/sampled method that computes the simulation cycles analytically by using the steady-state IPC, which is obtained by the scheduling analysis of sampled traces. QEMU is extended with a cache simulator, a branch

predictor, and the trace analyzer with which we estimate the dynamically varying steady-state IPC metrics. Experimental results with MiBench benchmarks prove that the proposed simulator TQSIM shows about 13.09 - 23.11 MIPS performance (up to 135 times faster than a cycle-accurate GEM5 simulator) with an average absolute error of approximately 8 %. TQSIM is also capable of compromising between simulation time and accuracy by adjusting the sampling period and size. TQSIM is suitable for early design space exploration of system architectures and design/evaluation of software without hardware in the hardware/software codesign methodology. TQSIM is an open-source project currently available online [2].

As a future work, we consider the following possible extensions:

- *Compile-time analysis:* For a list of instructions in a basic block never changes, static instruction-related information can be obtained from static processing of instructions in the block. As proposed in [34], compile-time analysis may help us to reduce the number of helper function calls for static information. On the other hand, dynamic information such as effective data addresses for load/store instructions can only be obtained during the program execution by calling helper functions at run time. How to reduce the number of helper function calls by compile-time analysis is left as a future work since it has more technical challenges than expected.

- *Multicore/multiprocessor simulation:* While we mainly focus on a single core, in this paper, we expect to extend it for multi-core simulation, which we leave as a major future work. The main purpose of this study is to build a single core simulator that has good compromise between speed and accuracy, which will be used for building a many-core simulator. To be used for multiprocessor simulation, the proposed uniprocessor simulator needs to be modified to generate synchronization events and measure the simulated time between those events. Then, using old-parameter values described in Figure 4 may bring up synchronization problem. Moreover, the heavy communication between processor simulators would significantly slow down simulation performance [41]. A multicore simulator based on the proposed simulator is currently under development.

Acknowledgement

This work was supported by Samsung Advanced Institute of Technology, Samsung Electronics Co., Ltd.

References

[1] J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, J. E. Smith, The future of simulation: A field of dreams, Computer 39 (11) (2006) 22–29.

[2] C.-H. Tu, S.-H. Hung, T.-C. Tsai, Mcemu: A framework for software development and performance analysis of multicore systems, ACM Transactions on Design Automation of Electronic Systems (TODAES) 17 (4) (2012) 36.

[3] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, A. Agarwal, Graphite: A distributed parallel simulator for multicores, in: Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2010, pp. 1–12.

[4] D. Yun, J. Kim, S. Kim, S. Ha, Simulation environment configuration for parallel simulation of multicore embedded systems, in: Proceedings of the 48th Design Automation Conference (DAC), ACM, 2011, pp. 345–350.

[2] As of April 2016, download at: https://github.com/mseed2/tqsim

[5] T. E. Carlson, W. Heirman, L. Eeckhout, Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2011, p. 52.

[6] J. Wang, J. Beu, R. Bheda, T. Conte, Z. Dong, C. Kersey, M. Rasquinha, G. Riley, W. Song, H. Xiao, et al., Manifold: A parallel simulation framework for multicore systems, in: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS),, IEEE, 2014, pp. 106–115.

[7] D. Sanchez, C. Kozyrakis, Zsim: fast and accurate microarchitectural simulation of thousand-core systems, in: ACM SIGARCH Computer Architecture News, Vol. 41, ACM, 2013, pp. 475–486.

[8] Ovp homepage.
URL `http://www.ovpworld.org/technology_ovpsim`

[9] N. Romdan, Arm fastmodels–virtual platforms for embedded software development, Information Quarterly Magazine 7 (4) (2008) 33–36.

[10] T. S. Karkhanis, J. E. Smith, A first-order superscalar processor model, in: ACM SIGARCH Computer Architecture News, Vol. 32, IEEE Computer Society, 2004, p. 338.

[11] S. Eyerman, L. Eeckhout, K. De Bosschere, Efficient design space exploration of high performance embedded out-of-order processors, in: Proceedings of the conference on Design, automation and test in Europe, European Design and Automation Association, 2006, pp. 351–356.

[12] D. Genbrugge, S. Eyerman, L. Eeckhout, Interval simulation: Raising the level of abstraction in architectural simulation, in: Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2010, pp. 1–12.

[13] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, M. Schulz, Efficiently exploring architectural design spaces via predictive modeling, Vol. 41, 2006, pp. 195–206.

[14] T. M. Conte, M. A. Hirsch, K. N. Menezes, Reducing state loss for effective trace sampling of superscalar processors, in: Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD), IEEE, 1996, pp. 468–477.

[15] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, J. C. Hoe, Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling, in: Proceedings of the 30th Annual International Symposium on Computer Architecture, IEEE, 2003, pp. 84–95.

[16] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, Automatically characterizing large scale program behavior, ACM SIGARCH Computer Architecture News 30 (5) (2002) 45–57.

[17] S. Nussbaum, J. E. Smith, Modeling superscalar processors via statistical simulation, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques., IEEE, 2001, pp. 15–24.

[18] L. Eeckhout, S. Nussbaum, J. E. Smith, K. De Bosschere, Statistical simulation: Adding efficiency to the computer designer's toolbox, Ieee Micro 23 (5) (2003) 26–38.

[19] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, H. Angepat, Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators, in: Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture, IEEE Computer Society, 2007, pp. 249–261.

[20] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, K. Asanovic, Ramp: Research accelerator for multiple processors, IEEE Micro 27 (2) (2007) 46–57.

[21] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, J. Emer, Hasim: Fpga-based high-detail multicore simulation using time-division multiplexing, in: Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2011, pp. 406–417.

[22] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, H. Meyr, Hysim: a fast simulation framework for embedded software development, in: Proceedings of the 5th IEEE/ACM international conference on Hardware/-software codesign and system synthesis, IEEE, 2007, pp. 75–80.

[23] L. G. Murillo, J. Eusse, J. Jovic, S. Yakoushkin, R. Leupers, G. Ascheid, Synchronization for hybrid mpsoc full-system simulation, in: Proceedings of the 49th Design Automation Conference (DAC), IEEE, 2012, pp. 121–126.

[24] A. Patel, F. Afram, S. Chen, K. Ghose, Marss: a full system simulator for multicore x86 cpus, in: Proceedings of the 48th Design Automation Conference (DAC), ACM, 2011, pp. 1050–1055.

[25] R. Plyaskin, A. Herkersdorf, Context-aware compiled simulation of out-of-order processor behavior based on atomic traces, in: VLSI and System-on-Chip (VLSI-SoC), 2011 IEEE/IFIP 19th International Conference on, IEEE, 2011, pp. 386–391.

[26] S. Ottlik, S. Stattelmann, A. Viehl, W. Rosenstiel, O. Bringmann, Context-sensitive timing simulation of binary embedded software, in: Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2014, pp. 14:1–14:10.

[27] S. Stattelmann, S. Ottlik, A. Viehl, O. Bringmann, W. Rosenstiel, Combining instruction set simulation and wcet analysis for embedded software performance estimation, in: 7th IEEE International Symposium on Industrial Embedded Systems (SIES), 2012, pp. 295–298.

[28] H. Cassé, P. Sainrat, Otawa, a framework for experimenting wcet computations, in: 3rd European Congress on

Embedded Real-Time Software, 2006.

[29] C. Ferdinand, R. Heckmann, ait: Worst-case execution time prediction by static program analysis, in: Proceedings of the 18th International Parallel and Distributed Processing Symposium, Springer, 2004, pp. 377–383.

[30] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, L. Eeckhout, An evaluation of high-level mechanistic core models, ACM Transactions on Architecture and Code Optimization (TACO) 11 (3) (2014) 28.

[31] M. Gligor, N. Fournel, F. Pétrot, Using binary translation in event driven simulation for fast and flexible mpsoc simulation, in: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis, ACM, 2009, pp. 71–80.

[32] F. Bellard, Qemu, a fast and portable dynamic translator., in: USENIX Annual Technical Conference, FREENIX Track, 2005, pp. 41–46.

[33] T.-C. Yeh, G.-F. Tseng, M.-C. Chiang, A fast cycle-accurate instruction set simulator based on qemu and systemc for soc development, in: Proceedings of the 15th IEEE Mediterranean Electrotechnical Conference (MELECON), IEEE, 2010, pp. 1033–1038.

[34] D. Thach, Y. Tamiya, S. Kuwamura, A. Ike, Fast cycle estimation methodology for instruction-level emulator, in: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, 2012, pp. 248–251.

[35] Google, Android emulator.
URL `http://developer.android.com/tools/help/emulator.html`

[36] W.-C. Hsu, S.-H. Hung, C.-H. Tu, A virtual timing device for program performance analysis, in: Proceedings of the IEEE 10th International Conference on Computer and Information Technology, IEEE, 2010, pp. 2255–2260.

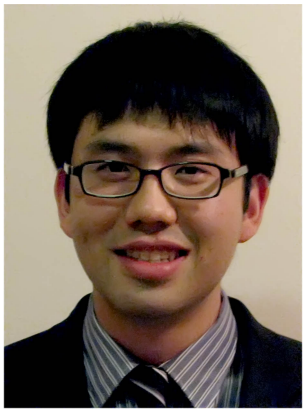
[37] S. Liao, G. Martin, S. Swan, T. Grötker, System Design with SystemC^{TM}, Springer Science & Business Media, 2002.

[38] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al., The gem5 simulator, ACM SIGARCH Computer Architecture News 39 (2) (2011) 1–7.

[39] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, N. Paver, Sources of error in full-system simulation, in: Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on, IEEE, 2014, pp. 13–22.

[40] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, Mibench: A free, commercially representative embedded benchmark suite, in: Proceedings of the IEEE International Workshop on Workload Characterization (WWC), IEEE, 2001, pp. 3–14.

[41] M.-H. Wu, P.-C. Wang, C.-Y. Fu, R.-S. Tsay, An extended systemc framework for efficient hw/sw co-simulation, ACM Transactions on Design Automation of Electronic Systems (TODAES) 17 (2) (2012) 11.



Shin-haeng Kang received his B.S. degree in computer science and engineering from Seoul National University in 2010. Currently, he is in the Ph.D course of the M.S.-Ph.D integrated program at Seoul National University. His research interests include design and simulation methodologies for parallel embedded systems.

Donghoon Yoo is a technical manager of GPU software at Samsung Electronics. His research interests include processor architecture, device driver, compiler, simulator and graphics & compute APIs. He has a PhD degree in computer engineering from Gwangju Institute of Science and Technology.

Soonhoi Ha is a full professor in the School of Computer Science and Engineering at Seoul National University. From 1993 to 1994, he worked for Hyundai Electronics Industries Corporation. He received his Bachelors (1985) and Masters (1987) in Electronics Engineering from Seoul National University, and PhD (1992) degrees in Electrical Engineering and Computer Science from University of California, Berkeley. He has worked on the Ptolemy project and the PeaCE (development of a HW/SW codesign environment) project. Currently he is leading the HOPES (development of an embedded S/W design environment for MPSoC) project. His research interests include hardware-software codesign, design methodology for embedded systems and embedded S/W. He is a senior member of the IEEE Computer Society.