2010 10th IEEE International Conference on Computer and Information Technology (CIT 2010)

A Virtual Timing Device for Program Performance Analysis

Wen-Chang Hsu[2], Shih-Hao Hung[12], Chia-Heng Tu[1]
[1]Graduate Institute of Networking and Multimedia
[2]Department of Computer Science and Information Engineering
National Taiwan University
Taipei 106, Taiwan
{r96922119, hungsh and d94944008}@csie.ntu.edu.tw

Abstract

Functional virtual platforms have been popularly used to support system development without needing the actual hardware. While the emulation process is fast enough to model the behaviors of complex systems, performance assessment cannot be done accurately due to the lack of timing models for the simulated systems. To tackle the problem, we proposed a virtual timing device (VTD) for a functional virtual platform to advance simulated clock time based on the hardware/software events observed during the emulation process. As a case study, we implemented the VTD in QEMU, an open-source virtual platform, with a variety of timing algorithms offering trade-offs between the accuracy and speed of timing estimation. With a fast, but less accurate timing algorithm, quick performance analysis can be done on QEMU at approximately 67 million instruction per second and reported execution time for the MiBench with an average of 15.7% error. Highly accurate performance profiles can be obtained by elaborating the timing model, e.g. with the addition of cache simulation, at the cost of simulation speed.

1 Introduction[1]

Virtual platforms provide software designers with the environments to implement and evaluate their designs before the hardware becomes available by simulating or emulating the functions of the hardware. Figure 1 illustrates the concepts of a virtual platform which we use for system-level simulation environment. The *virtual platform software* at the middle layer offers the same interface to the target software on the top layer as the hardware being simulated and is responsible for interpreting/translating the instructions/functions of the simulated platform into the form that can be executed by the *host system* at bottom layer.

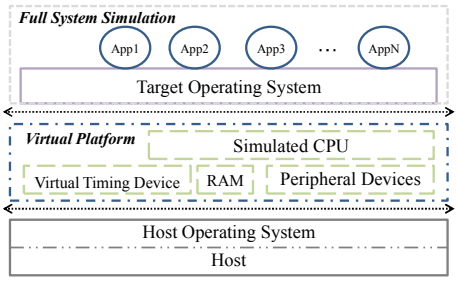

Figure 1. Overview of simulation environment

The simulation performance and accuracy depend on the modeling granularity of the virtualizing software. For example, a coarse grain *functional simulation* [2] [6] [7] [13] performs only the behaviors of the emulated platform as defined by the functional specifications, treating each hardware component as a black box without looking into the details of the hardware components. Thus, a functional simulation tends to be much faster than cycle-accurate simulation and can be adopted for software development. Nevertheless, due to the high-level abstraction of the hardware elements, it is difficult for a functional simulator to provide good estimates of execution time, which has made performance assessment virtually impossible for developing real-time applications, evaluating the efficiency of the system, and pinpointing performance bottlenecks.

The granularity of an *instruction-set architecture* (ISA) simulation [3] [2] [16] [8] is finer than functional

[1]This work was supported in part by a grant from the National Science Council (98-2220-E-002-020) and a grant from Ministry of Economic Affairs (MOEA) supported research project (98-EC-17-A-01-S1-034).

978-0-7695-4108-2/10 $26.00 © 2010 IEEE
DOI 10.1109/CIT.2010.389

IEEE computer society

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on February 25,2023 at 18:04:33 UTC from IEEE Xplore. Restrictions apply.

simulation as it models the processor pipeline, caches, and/or interconnection network in details. While it gives more accurate timing information, it usually takes more than thousands of instructions on the host to simulate one instruction, and the slow-down may render it unusable for interactive applications. Electronic system-level (ESL) simulation is of even finer granularity and is far too slow for analyzing software performance.

In this paper, we developed the virtual timing device, which is applicable to existing virtual platform software, for evaluating the software performance during the simulation of the target design. The VTD maintains a set of counters: *Event Counters*, a *Timing Interpreter*, and a *Virtual Time Stamp Counter (VTSC)*. The event counters record the operations which may influence the execution time when they occur. The timing interpreter uses a mathematical timing models [9] [14] [1] to transform the values from event counters into an estimated wall clock time for the emulated platform and update the VTSC.

In addition, by implementing a variety of timing models in the VTD on the virtual platform software, QEMU, we demonstrate that it is capable of managing the accuracy-speed trade-off. Our results show that the modified QEMU executed MiBench on an emulated XScale PXA270 system at 67 million instruction per second (MIPS) and reported execution time for the MiBench with an average of 15.7% error. With a cache simulator integrated to improved the consistency of performance estimates, the VTD slows down the virtual platform by 7.2 times, but was still able to perform at the speed of 17.1 MIPS, which is significantly faster than most cycle-accurate simulators. Hence, with the adaptive timing models provided by VTD, software development, performance analysis of target design, and hardware-software co-development and co-optimization can be done at early design stage of the system design and shorten time-to-market.

The rest of the paper is organized as follows. Section 2 provides the background of existing simulators and models for timing estimation. Section 3 gives the overview of the VTD and key components in the VTD. Section 4 shows the performance results from our experiments and discusses the trade-offs between the estimation accuracy and the simulation speed among different timing models. Finally, Section 5 summarizes this paper.

2 Background and Related Work

Simulation and *mathematical models* are two major approaches to the estimation of computer performance. Both have their limitations, and the choice would depend on the requirements of the user. Detailed simulation of a sophisticated computer system may take an enormous amount of time to achieve high accuracy. The level of details, i.e. the granularity, is the key factor which determines the speed and the accuracy of a simulation scheme. On the other hand, mathematical models offer fast estimation with statistical methods, pattern matching algorithms, or machine learning techniques. However, it is difficult for one mathematical model to cover a wide range of applications and platforms consistently as the characteristics of two applications can be very different on the same platform, and the characteristics of the same application can change significantly from one platform to another.

For example, *SimpleScalar* [8] is a popularly used cycle-accurate instruction-set simulator (ISS) for computer architects to model the behavior of a computer at the microarchitecture level. It is capable of simulating the pipelined execution of instructions within a processor by modeling the pipeline, the branch prediction, the processor caches, the memory management unit, and the translation lookaside buffer (TLB). Other cycle-accurate simulators such as *SESC* [3] and *Realview* [2] work similarly to simulate a target platform and report timing information. While the term "cycle-accurate" is commonly used to describe the degree of accuracy for the simulation of a processor, it does not guarantee the accuracy when the entire system is simulated with various hardware components such as the buses, the memory, the I/O devices.

A *function-accurate simulation* or *emulation* [6] [7] [13] mimics the functional behavior of the processor and other system components during the execution of a program. Since the internal behavior of the hardware components are not modeled in details, functional simulation is much faster than cycle-accurate simulation, but detailed hardware operations along with their timing information cannot be extracted with a functional simulator. To further speed up the emulation of a processor, several function-accurate simulators, e.g., AMD's *SimNow* [6] and *QEMU* [7], take advantage from the *dynamic binary translation* technology to translate the blocks of target instructions into an optimized sequence of native instructions.

Combining cycle-accurate simulation with functional simulation has been proposed to shorten the simulation time. For example, *PTLsim/X* [16] comprises of a cycle-accurate x86 processor simulator and a virtual machine based on *Xen* [5]. During the execution of an x86 program, the virtual machine, Xen, would execute the program directly on the physical x86 processors on the host system (a.k.a. native mode)

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on February 25,2023 at 18:04:33 UTC from IEEE Xplore. Restrictions apply.

and dynamically switches to the cycle-accurate simulation mode when the execution reaches the code sections specified by the user. This switch mechanism speeds up program execution by skipping the parts which are of no interests to the users.

Mathematical models [9] [14]have been proposed to reduce the time required to obtain accurate timing reports from a detailed cycle-by-cycle simulation. Giusto et. al. [9] categorized the instructions of the target processor into 7 types [12] and used an linear estimator to predict the execution time *statically* for a given program based on the distribution of instructions in each category by multiplying the pre-trained coefficients of each instruction type by the instruction counts of corresponding instruction type to obtain the estimated cycle counts. Their work focused on source code analysis, whereas our work is capable of profiling concurrently running programs on-the-fly during the simulated execution with support for conventional profiling tools such as gprof.

For system-wide emulation, QEMU is widely adopted by the computer industry and the software development community for its speed and platform support [7]. In particular, the Android SDK [1] included QEMU to assist the design and verification of software components. The QEMU functional emulator provides a facility to count the number of instructions executed by the emulation engine. Unfortunately, representing the performance of an application with the instruction count can be very misleading, as the number of clock cycles per an instruction (CPI) varies significantly from one application to another application on a modern computer system. Meanwhile, the developers of the Android SDK provided a primitive way to estimate program performance via multiplying the instruction counts by the executed cycles consumed by each type of instructions. The scheme may work fine for low-end embedded processors with very simple processor datapath, but the results are far from accurate for a modern embedded processor such as the ARM9, as the performance would be impacted heavily by cache and scratchpad memory [4], which is often the keys for performance optimization. For QEMU to account for execution cycles on a simulated platform, we proposed to integrate cycle-accurate simulation and mathematical models into the simulation process.

3 The Design of Virtual Timing Device

The VTD works by estimating the execution time on the simulated platform, and interfaces are provided for the user to access the execution cycle counters and event counters during the simulated execution. For example, conventional profiling tools such as Pin and gprof can take advantage of VTD. The VTD includes various front-end analysis modules which can be selectively activated to provide raw performance data by analyzing the executed instructions and/or simulating specific components.

As shown in Figure 2, several timing models can be developed to estimate the execution time and satisfy the requirements from the user. For example, if the user is only interested in the cache behavior, then the user may instruct the timing interpreter to select Timing Model 2, which activates the cache simulator and estimate the execution time with cache misses reported by the cache simulator. If the user chooses Timing Model 3, then the pipeline, cache, DRAM, and Disk simulators will all be activated and their results will be used to estimate the execution time. Obviously, Timing Model 3 should provide more accurate estimates than Timing Model 2 since it receives more data from the front-end modules. However, it also takes more time to execute the front-end modules activated by Timing Model 3 and slows down the virtual platform. It is important that our design allows the user to choose a timing model or switch from one to another.

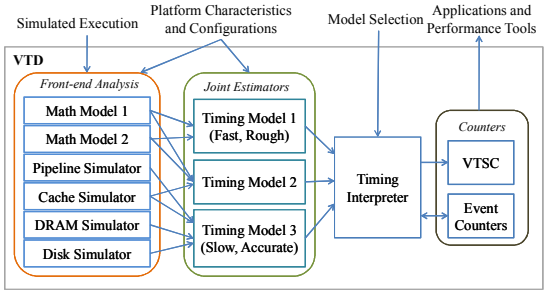

Figure 2. The internal of virtual timing device

During the simulation, the timing interpreter receives events and cycle counts from a timing model and advances the values in the VTSC and the event counters. The applications and profiling tools running on the virtual platform can access the counters to obtained simulated clock time and account for specific events the same way as they would access the time stamp counter (TSC) and performance monitors which might be available on the actual platform.

Note that the virtual platform still provides a TSC based on real-time clock on the host and generates timer interrupts regularly at the period specified by the OS running on the virtual platform, as there are some timing-sensitive I/O device drivers which depends on real-time clock. However, since our virtual platform may execute a program faster or slower than the actual

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on February 25,2023 at 18:04:33 UTC from IEEE Xplore. Restrictions apply.

platform (depending on the selected timing model), it is possible for the system timer to interrupt a program on the virtual platform less or more frequently than it would on the actual platform. Since this affects the performance estimate, one way to fix this issue is to have the timing interpreter monitor and control the overhead of the timer interrupt handler. Another solution is to adjust the frequency of timer interrupts, but it might affect the operations of certain device drivers. Thus, while both solutions are offered to the user, our virtual platform employs the first solution by default. The following subsections explain the three major components in our VTD design.

3.1 The Timing Interpreter and Timing Models

The timing interpreter monitors the simulated execution, estimates the execution time, and updates the VTSC and the event counters in the VTD. Several timing models which we developed are described below. As a case study, we evaluate the accuracy and speed of the following two timing models in this paper:

1. *Model A* analyzes the user-space and kernel-space instructions under Linux and estimates the execution time with a linear regression (LR) model established in advance.
2. *Model B* is also known as a *datasheet* model [9], which figures out the cycle counts for each instruction by consulting a table built upon the specifications datasheet from the simulated platform. The table lists the latency for each instruction, stall cycles in the case of pipeline hazards, and cache miss penalty.

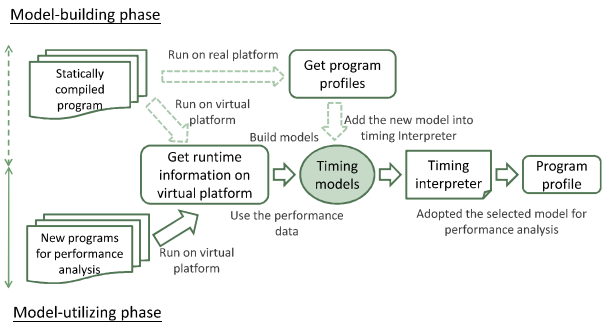

Figure 3. The workflow of building regression models and utilizing the models

Timing model A is based on LR method, where two phases are involved: *training* and *prediction*. As shown in Figure 3, in the training (model-building) phase , a set of programs is identified as the training set, which are executed on both the real platform and the virtual platform. Depending on the model, the profiles which contain the measurement of events on both platforms are used to construct a linear regression model. For example, with Model A, the cycle count for each program in the training set are measured from the real platform. The correlation between the cycle counts and instruction counts in user-space and kernel-space are analyzed to build the LR model, with the cycle counts as the dependent variable (Y), the user instruction counts as independent variables (N_i), and the kernel instruction counts as independent variables (M_i). The training phase quantifies the relationship between the dependent variable Y and independent variable N_i and M_i with the equation below:

$$Y = \sum_i P_i * N_i + Q_i * M_i, \quad (1)$$

where P_i and Q_i are the parameters used by the LR model to describe the mappings between Y and N_i and M_i, respectively. In the prediction (model-utilizing) phase, the independent variables N_i and M_i (i.e., user instruction counts and the kernel instruction counts) are input to the LR model to predict the cycle count Y for the simulated code block.

The datasheet model, unlike the LR models, does not require a training phase. Instead, the parameters shown in 1 (P_i or Q_i) are determined by the platform specifications. The user may also specify the range for the parameters to estimate the lower and upper bounds. This also allows the user to explore the system design space by varying the parameters.

3.2 Virtual Time Stamp Counter

The Time Stamp Counter (TSC), a 64-bit register in most x86 processors [11], is an efficient scheme for counting the number of clock ticks since the processor is reset. In our design, we offer a virtual time stamp counter (VTSC) to count the number of simulated cycles on the virtual platform. During the simulation process, the VTSC is updated by the timing interpreter thread from time to time, depending on the granularity of simulation. The timing interpreter may run asynchronously with the functional simulation and update the VTSC whenever (1) a code block finishes or (2) the VTSC is accessed. The functional simulation must wait for the timing interpreter only when the simulated execution access the VTSC or perform a timing-sensitive function such as bus synchronization, direct memory access, I/O operations, or interprocessor communication.

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on February 25,2023 at 18:04:33 UTC from IEEE Xplore. Restrictions apply.

3.3 Event Counters

The timing model adopted by the timing interpreter thread analyzes the simulated execution and keeps track of a variety of events with a set of counters. The event counters are available for the simulated software to access. In particular, a performance tool may run with the simulated software and use the event counters to monitor the performance of the simulated system. The following events are counted with Model B: *retired instructions*, *load instructions*, *store instructions*, *arithmetic instructions*, *branch instructions*, *logical instructions*, and *jump instructions*.

4 Performance Evaluation

We carried out experiments to evaluate the design of VTD by comparing the accuracy and speed of the two timing models mentioned in the previous section. This section describes our experimental setup, the performance index for evaluating the accuracy of a timing device, and the experimental results. Finally, we look into the speed of the timing models.

4.1 Experiment Setup

We ran the *MiBench* [10] benchmark programs on both the reference platform and the virtual platform whose configurations are listed in Table 1. We used QEMU to emulate the same PXA270 processor as on the reference platform. For the linear regression timing model (Model A), we performed *leave one out validation* (LOOCV) [15] to test the accuracy of the model. For the datasheet model (Model B), we built an instruction latency table and memory access latency table based on the specifications of the reference platform.

4.2 Performance Index

We evaluated the accuracy of a timing device for each benchmark program by calculating its *error rate* as:

$$ErrorRate = \frac{Predicted - Observed}{Observed} * 100\%. \quad (2)$$

We also calculated the *averaged error rate* for the entire MiBench benchmark suite as the following:

$$\frac{\sum_{i=0}^{K} |Error_i|}{K}, \quad (3)$$

where K is the total number of programs in testing set and $Error_i$ is the error rate for the i-th benchmark program.

4.3 VTD for Execution Time Estimation

Table 2. The error rate of the timing models

Benchmark	Error rate (%)	
Program	Model-A	Model-B
adpcm	11.3	24.3
crc32	1.6	9.6
sha	25	8.4
blowfish	-0.7	9.3
string search	3.7	44.7
ispell	-33.5	8.4
bitcnts	0.9	23.7
susan	1.5	11.4
jpeg	8.4	36.1
typeset	-54.2	38
dijkstra	-2.5	39
patricia	-44.7	21
Average	15.7	22.8

To evaluate the performance delivered by the timing interpreter, the error rates of the timing models implemented in the timing interpreter are shown in Table 2.

- By counting user-space and kernel-space instructions separately, Model A exhibited the averaged error rate of 15.7%, and the error rates for individual benchmark programs were around 25%, except for programs with extensive memory operations (*patricia* and *typeset*) and programs with frequent branches (e.g., *ispell*.)

- Though Model B (datasheet model) had large averaged error rate, 22.8%, the individual error rate ranged from 8.4% to 44.7%, which showed better consistency than the regression model. The execution time was overestimated since we picked the upper bound for memory access latency from the datasheet. The upper bound gave conservative estimates and allows us to safely design a system to meet real-time performance requirements.

4.4 The Speed of the Virtual Platform

Table 3 compares the speed of our design, QEMU and the reference platform (baseline). Using MiBench, we measured the speed in terms of million instruction executed per second. On average, the QEMU performed MiBench at a speed that is 77% of the reference platform. The overhead of Model A slows down the simulation speed to 54% of the reference platform. With cache simulation added, Model B reduced the

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on February 25,2023 at 18:04:33 UTC from IEEE Xplore. Restrictions apply.

Table 1. Experimental Setup

Platforms	Specifications
Reference platform	Creator Development Board (*Microtime Comp. Inc.*) XScale-PXA270 @520MHz + 64MB SDRAM Applications running on Linux 2.6.15
Virtual platform	Emulating XScale-PXA270 with QEMU-0.9.1 Executing entire applications/Linux Host: Intel-Q6600@2.6GHz + 3GB RAM

speed of simulation further to 13.8% (a factor of 7.2 times). Nevertheless, at 17.1 MIPS, the speed of Model B is still good enough to execute most embedded applications on simulated Linux platform smoothly, drive I/O devices correctly, and maintain network connection to the real world.

Table 3. Performance comparison between reference platform (Baseline) and timing models

	Baseline	QEMU	A	B
MIPS	123.6	95.1	66.9	17.1
Relative Perf.	100%	77%	54.1%	13.8%

5 Summary

In this paper, we describe the design of a VTD for measuring the simulated execution time of software running on a virtual platform. Our prototype implemented on QEMU has a timing interpreter thread to analyze the simulated execution with several timing models using mathematical models and component simulators. The VTD provided users with the execution cycle count and a variety of performance-related events via the VTSC and event counters. In our experimental study, we evaluated the accuracy of the simulated execution time reported by two timing models as well as the speed of simulation. With Model A, the VTSC reduced the simulation speed by 46% and reported estimated execution time with an average of 15.7% error relative to the measured execution on an actual platform. Model B delivered more consistent estimates, but it slowed down the virtual platform by a factor of 7.2 due to the overhead of cache simulation.

References

[1] Android Open Source Project. http://source.android.com/.

[2] RealView ARMulator. http://www.arm.com/products/devtools/realviewdevsuite.html.

[3] SESC: cycle accurate architectural simulator. http://sesc.sourceforge.net.

[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES*, pages 73–78, 2002.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, 2003.

[6] R. Bedichek. SimNow: fast platform simulation purely in software. *Hot Chips Symp.*, Aug. 2004.

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX*, pages 41–41, 2005.

[8] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[9] P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *DATE*, pages 580–589, 2001.

[10] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: a free, commercially representative embedded benchmark suite. In *WWC*, pages 3–14, 2001.

[11] Intel Corp. Using the rdtsc instruction for performance monitoring. Technical report, Intel Corp., 1997.

[12] M. Lazarescu, J. Bammi, E. Harcourt, L. Lavagno, and M. Lajolo. Compilation-based software performance estimation for system level design. In *HLDVTW*, pages 167–172, 2000.

[13] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, and B. Werner. Simics: a full system simulation platform. *Computer*, 35(2):50–58, 2002.

[14] M. S. Oyamada, F. Zschornack, and F. R. Wagner. Applying neural networks to performance estimation of embedded software. *J. Syst. Archit.*, 54(1-2):224–240, 2008.

[15] R. R. Picard and R. D. Cook. Applying neural networks to performance estimation of embedded software. *J. Am. Stat. Assoc.*, 79(387):575–583, 1984.

[16] M. Yourst. PTLsim: a cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS*, pages 23–34, 2007.

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on February 25,2023 at 18:04:33 UTC from IEEE Xplore. Restrictions apply.