

# Fast Cycle Estimation Methodology for Instruction-Level Emulator

David Thach

Fujitsu Laboratories Ltd  
Kawasaki, Japan  
thach.david@jpf.fujitsu.com

Yutaka Tamiya

Fujitsu Laboratories Ltd.  
Kawasaki, Japan  
tamiya.yutaka@jpf.fujitsu.com

Shin'ya Kuwamura

Fujitsu Laboratories Ltd.  
Kawasaki, Japan  
kuwa@jpf.fujitsu.com

Atsushi Ike

Fujitsu Laboratories Ltd.  
Kawasaki, Japan  
ike@jpf.fujitsu.com

**Abstract**—In this paper, we propose a cycle estimation methodology for fast instruction-level CPU emulators. This methodology suggests achieving accurate software performance estimation at high emulation speed by utilizing a two-phase pipeline scheduling process: a static pipeline scheduling phase performed off-line before runtime, followed by an accuracy refinement phase performed at runtime. The first phase delivers a pre-estimated CPU cycle count while limiting impact on the emulation speed. The second phase refines the pre-estimated cycle count to provide further accuracy. We implemented this methodology on QEMU and compared cycle counts with a physical ARM CPU. Our results show the efficiency of the trade-offs between emulation speed and cycle accuracy: cycle simulation error averages 10% while the emulation latency is 3.37 times that of original QEMU.

## I. INTRODUCTION

Recent market trends reveal that the demand for electronic devices offering services of always higher quality keeps growing progressively. In the case of embedded systems, such a demand has a huge impact on design methodologies; engineers are obliged to continuously perform software (SW) scale-up. They have to face two main difficulties: on one hand, high SW complexity makes functional verification more difficult. On the other hand, performance of embedded systems becomes extremely critical; recent embedded systems are required to deliver quasi-instantaneous responsiveness regardless of how heavy the services that run in parallel are.

While traditional SW debug environments still remain an alternative for SW functional verification, SW performance verification represents a great challenge. Developers have difficulties dealing with SW performance constraints due to the lack of efficient technique to evaluate SW performance. Furthermore, it is very likely that SW complexity will keep increasing rapidly. This creates an urgent need for fast and accurate SW performance evaluation methods.

Our approach consists in implementing a SW performance estimation methodology in a just-in-time (JIT) compiled fast instruction-level CPU emulator called QEMU [1]; owing to its high emulation speed, responsiveness of the emulated CPU is very close to reality. Electronic system-level designers consider cycle simulation error to be acceptable when it does not exceed 10% [2]. Therefore, based on QEMU, we aim at keeping cycle simulation error lower than this limit.

## II. RELATED WORKS

Several attempts in building an environment for SW performance evaluation were made. Among all, the two followings can be pointed out: (1) utilization of on-chip SW functional debugging environments and (2) utilization of register transfer level (RTL) simulation environments such as FPGAs. Unfortunately, both fail to provide debug easiness and high simulation speed; on-chip debugging environments give the advantage of high execution speed but uncontrollability of external factors (e.g. chip voltage, temperature, network load, peripherals, etc.) makes performance bug hardly reproducible. Contrary to on-chip environments, FPGA does give access to reproducible performance data but setup of FPGA simulation environments is time consuming and simulation itself is quite slow compared to on-chip execution. Moreover, such devices are expensive and therefore hardly accessible.

A recent interesting work suggests using the QEMU-SystemC emulation framework to implement a fast cycle-accurate instruction set simulator [3]. In this method, QEMU is responsible for the target system emulation while cycle-accurate simulation is performed in the SystemC environment: for each instruction executed, QEMU sends information regarding the latter to the SystemC environment; this information is used to simulate the HW at cycle-accurate level. Unfortunately, interaction with SystemC and the amount of code added in QEMU slows down simulation considerably.

Another method for cycle-accurate simulation is presented in [4]. It is based on a fast instruction-level emulator similar to QEMU: the emulator uses JIT compiled dynamic binary translation techniques. The method consists in computing cycle count based on a model of the target CPU's execution pipeline. The pipeline model is updated at runtime, every time an instruction is executed. Updating this pipeline model using information regarding executed instruction enables it to be always accurate. As a result, the cycle count computed out of this model is highly accurate. However, similarly to [3], the cost of accuracy is speed; invocation of pipeline model update functions between instructions execution significantly reduces simulation speed.

All these previous works illustrate the difficulty of finding a method that provides speed, accuracy, and debug easiness at low costs. CPU emulators such as QEMU certainly provide an environment easily accessible and in which debug can be performed simply. But the speed and accuracy issues remain.

The method presented in this paper manages to achieve the all requirements. To satisfy both speed and accuracy constraints in QEMU, we propose a cycle estimation methodology based on a two-phase pipeline scheduling technique; a first off-line static scheduling phase, followed by a runtime accuracy refinement phase

The next section gives a brief description of QEMU. It also explains why pipeline scheduling is important for accurate cycle estimation. In the subsequent Section IV, we describe our proposed method to perform cycle accurate simulation. Section V presents an implementation for this method, and the related experimental results are given in Section VI. Finally, Section VII concludes this paper.

### III. BACKGROUND

#### A. QEMU: a fast CPU emulator

QEMU is an open-source fast instruction-level CPU emulator. It uses the target CPU’s binary code to perform emulation on a host machine. QEMU is extremely flexible; owing to its portable JIT dynamic code generator, it is capable of emulating many different types of CPU targets on many different types of host machines.

QEMU divides the target binary code into chunks of code called basic blocks (BBs), using branch instructions as separators. Code generation is performed on a BB basis: when the program counter of the emulated system reaches a specific BB for the first time, the entire BB is translated into equivalent block of host code called translated block (TB) (Fig. 1). The generated TB is stored in a translation cache (TC), from which it is repeatedly accessed by the host CPU for execution.

The use of a TC is the reason why QEMU is so fast; the TC enables the host system to skip code generation for TBs that are already stored in it. As a result, when switching between BBs, QEMU needs to perform code generation about 1% of the time, while nearly 99% of the time it accesses the TB directly from the TC.

#### B. Pipeline scheduling

SW performance evaluation consists in evaluating the number of CPU cycles necessary to achieve execution of the SW. In fact, assuming that the CPU micro-architecture is known, the CPU cycle count gives information about the SW execution speed and the energy consumption as well.

CPU cycle estimation is performed via pipeline scheduling. For a given CPU micro-architecture, such scheduling depends mainly on two factors: the instruction type and the CPU status. The instruction type gives information about how its execution is scheduled in each stage of the pipeline, while the CPU status gives information about the current pipeline status and instruction execution conditions.

Knowing about the CPU status is the key for accurate estimation; pipeline scheduling, as well as the resulting CPU cycle count, varies considerably depending on the execution conditions. Especially, scheduling factors such as register hazards (conflict that occurs when several instructions attempt to access a same register at the same time), cache misses, branch mispredictions and super scalar executions (parallelized instruction execution) have a significant impact on pipeline scheduling and it is therefore necessary to monitor them.

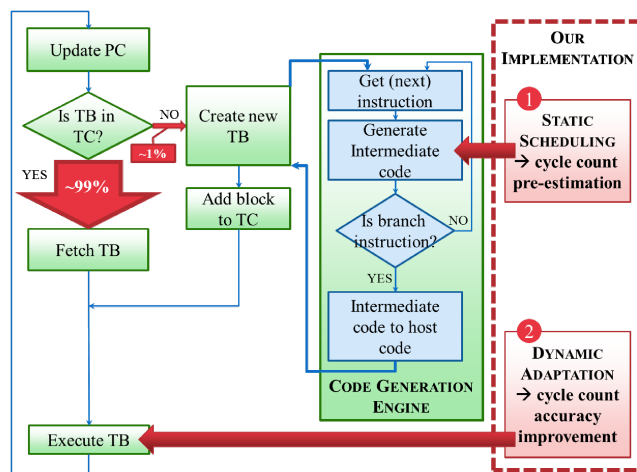


Figure 1. QEMU’s emulation flow

### IV. THE PROPOSED CYCLE ESTIMATION METHOD

The method proposed in this paper has two objectives: elevate cycle estimation accuracy as much as possible and keeping QEMU’s emulation speed as high as possible.

When performing CPU emulation using QEMU, the CPU status can be known at runtime only. Thus, in order to certify accuracy, it seems inevitable to perform cycle simulation at runtime; we will refer to this scheduling method as dynamic scheduling. With this method, conserving QEMU’s emulation speed is difficult; performing dynamic pipeline scheduling and cycle estimation between each instruction execution would reduce emulation speed significantly. Thus, this method is not acceptable.

On the opposite, embedding cycle simulation operation prior to translation would definitely limit impact on emulation speed since QEMU would execute it only about 1% of the whole time spent during emulation. But this time, because CPU status cannot be known prior to execution, scheduling accuracy cannot be guaranteed. It is still possible to compute pipeline scheduling prior to translation using assumptions about the CPU status; we will refer to this method as static scheduling. But again, it is difficult to determine how accurate the resulting cycle estimation is.

Neither of dynamic scheduling or static scheduling seems to be capable of providing both high emulation speed and cycle estimation accuracy. Conserving QEMU’s emulation speed and implementing an accurate SW performance estimation feature is somewhat contradictory. A compromise between these two objectives is definitely required.

To solve this issue, we propose a method that consists in splitting up pipeline scheduling computation into two phases: a static scheduling phase where the essential of cycle simulation operation is performed followed by a dynamic “adaptation” phase where scheduling is refined to improve accuracy.

#### A. Static scheduling

The static scheduling phase gives the possibility to reduce impact on emulation speed while obtaining a pre-estimation of the cycle count for each BB. To do so, assumptions on the CPU status are needed. Even though CPU status is not exactly predictable at static scheduling phase, experience tells that instructions are very likely to be executed in the most probable

conditions. For example, the hit rate of caches/TLB and the success rate of branch predictions are both high. Thus, we believe that rough cycle estimation is still reachable when making the following CPU assumptions: any memory access leads to a cache/TLB hit and any branch is correctly predicted.

Upon these assumptions, static scheduling is performed prior to the translation of each basic block, delivering what we consider to be a reasonable cycle pre-estimation.

### B. Dynamic adaptation

The basic idea of dynamic adaptation consists in refining the cycle pre-estimation obtained from static scheduling to elevate accuracy. To compensate the static scheduling assumptions, adequate scheduling modifications are initiated based on observations of the CPU status at runtime; critical cases such as cache misses or branch misses imply a certain amount of penalty cycles, which lead to significant pipeline scheduling changes.

During static scheduling each memory access is assumed to lead to a cache hit and each branch prediction to be always successful. To compensate these assumptions, dynamic adaptation is performed by adding cache miss and branch misprediction penalty cycles to the execution latency of the critical instruction when such events are detected at runtime. Fig. 2 illustrates one possible adaptation method. In this example, cycle penalty is computed so that scheduling of the next dependent instruction after dynamic adaptation is identical to the real scheduling.

By allowing refinements of pertinent cases only, the dynamic adaptation phase limits its impact on emulation speed while delivering highly accurate cycle estimation.

## V. IMPLEMENTATION

For our proposed implementation, we choose to use the ARM Cortex-A8 [5][6] CPU as target and implemented models to simulate the following characteristics of this CPU:

- Two integer pipelines for execution of ARM, Thumb and Thumb-2 instruction sets
- Super scalar pipeline execution (dual issue here)
- L1 and L2 caches

For the static scheduling phase, we implement a pipeline model that simulates the two integer pipelines and the super scalar pipeline executions. This model plays an important part in static scheduling since pipeline scheduling and cycle pre-estimation are performed based on it; the pipeline model analyses the type of instructions, tracks register hazards and super scalar executions while scheduling all memory access instructions as if they all led to a cache hit, and conditional branch instructions as if they were all successfully predicted.

For the dynamic adaptation phase, we build a cache simulator that models the L1 and L2 cache behavior as the ARM Cortex-A8 micro-architecture defines. This cache simulator uses virtual tag lists to keep track of the content of each cache line. Whenever a memory access instruction is encountered at runtime, tag lists are checked to determine at each cache level whether the instruction leads to a cache hit or miss. When a cache miss occurs, the cache simulator triggers dynamic adaptation: the pipeline scheduling is refined by adding cache miss penalty cycles. For now, using hints given by the CPU implementation note and external memory

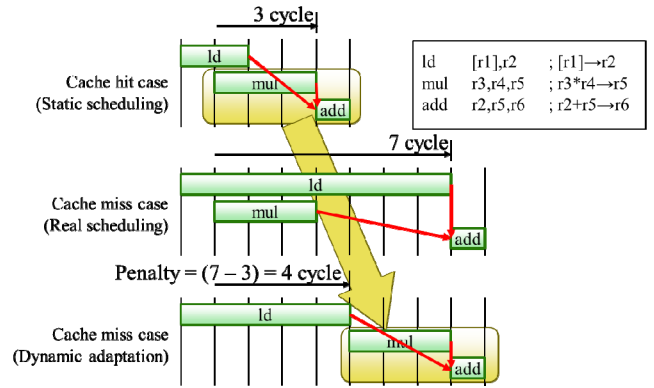


Figure 2. Dynamic adaptation of a cache miss case

datasheet, we set arbitrary cycle values for L1 and L2 cache miss penalty cycles.

We implement our methodology in QEMU by mainly modifying the source code of the code generation engine. Static scheduling is performed throughout the code generation process of a BB and delivers a pre-estimated cycle count which will be computed with the emulation overall cycle count at runtime (see ① in Fig. 1). Further code generation is performed for at-runtime dynamic adaptation purpose. This code is triggered by dynamic factor simulators when necessary (see ② in Fig. 1). Note that simulators for the dynamic branch predictor and the NEON pipeline are not included.

## VI. EXPERIMENTAL RESULTS

### A. Experimental environment

For our experiments, we use version 0.13 of QEMU as a base to implement our method. We use the Intel® Xeon® X5570 2.93GHz as host CPU. Also, we choose to use the evaluation board called “BeagleBoard-xM” as reference. It embeds a system-on-chip which integrates the Cortex-A8 CPU. We set up the board with version 11.04 of the Ubuntu Linux OS and burn an image of the system to run it on QEMU. We decide to focus on the integer execution pipeline of the Cortex-A8 and therefore use several different integer benchmarks.

We run each benchmark on four different environments: ① the real CPU (ARM Cortex-A8 here), ② original QEMU, ③ QEMU with static scheduling phase only and ④ QEMU with both static scheduling and dynamic adaptation phases.

For each benchmark run, we record the CPU cycles count obtained after a complete run, as well as the time elapsed. To perform CPU cycle count on the real CPU, we use the performance monitor embedded in the Cortex-A8.

### B. Results

We analyze the results in TABLE I from two different points of view: the emulation speed aspect and the cycle estimation accuracy aspect. To evaluate emulation speed variation, we compare execution times of original QEMU (②) with the ones obtained using our implementations ③ and ④. For accuracy analysis, we compare the cycle counts obtained from the real CPU (①) with the ones obtained using again our implementations ③ and ④.

TABLE I. CYCLE COUNT AND EXECUTION TIME OBTAINED FROM RUNS OF INTEGER BENCHMARKS

Program		Beagle Board ①	Original QEMU ②	QEMU w/ static scheduling w/o dynamic adaptation		QEMU w/ static scheduling w/ dynamic adaptation			
				③	④	error = $((③-①)/①)$		error = $((④-①)/①)$	
						speed down = $(③/②)$		speed down = $(④/②)$	
perl 1	cycle count	33,745,611	-	12,438,370	0.63	33,628,957	0.00		
	run time (s)	0.056	0.423	0.489	1.16	0.813	1.92		
perl 2	cycle count	1,291,258	-	369,339	0.71	996,629	0.23		
	run time (s)	0.002	0.140	0.035	0.25	0.030	0.21		
perl 3	cycle count	376,269,656	-	174,313,581	0.54	381,677,424	0.01		
	run time (s)	0.627	0.966	1.795	1.86	5.398	5.59		
perl 4	cycle count	874,958	-	351,436	0.60	923,126	0.06		
	run time (s)	0.002	0.003	0.010	3.33	0.015	5.00		
perl 5	cycle count	1,573,391	-	501,969	0.68	1,320,542	0.16		
	run time (s)	0.003	0.005	0.015	3.00	0.019	3.80		
perl 6	cycle count	18,206,544	-	6,946,545	0.62	16,187,397	0.11		
	run time (s)	0.030	0.082	0.115	1.40	0.211	2.57		
go 1	cycle count	673,642,824	-	107,473,065	0.84	728,227,505	0.08		
	run time (s)	1.123	2.502	2.878	1.15	9.168	3.66		
go 2	cycle count	303,782,694	-	6,383,928	0.98	297,103,591	0.02		
	run time (s)	0.507	0.890	1.717	1.93	3.749	4.21		
go 3	cycle count	260,492,304	-	6,400,762	0.98	253,521,094	0.03		
	run time (s)	0.435	1.052	1.305	1.24	2.661	2.53		
go 4	cycle count	1,405,488,304	-	6,474,496	1.00	1,500,526,183	0.07		
	run time (s)	2.343	4.576	7.572	1.65	15.563	3.40		
quantum	cycle count	281,126,149	-	88,386,130	0.69	344,721,154	0.23		
	run time (s)	0.469	0.568	1.021	1.80	2.733	4.81		
xml	cycle count	924,756,639	-	57,033,290	0.94	688,489,546	0.26		
	run time (s)	1.542	3.717	5.654	1.52	9.988	2.69		
Average				error	0.77	error	0.10		
				speed down	1.69	speed down	3.37		

Comparison of the execution times of original QEMU (②) with the implementation ③ gives us an idea of the minimum necessary speed sacrifice to obtain cycle estimation. In particular, we can observe that execution time is multiplied by 1.69 in average. When we perform the same observation and compare execution times of original QEMU (②) with implementation ④ average time is of about 3.37 times that of original QEMU.

From the cycle estimation accuracy point of view, using cycle counts obtained from the real CPU (①) as reference, we see that implementation ③ with static scheduling gives a cycle simulation error of 77% while estimation refinement via dynamic adaptation in implementation ④ considerably lowers error; in average, the simulation error is 10%.

Experiments show that, even though there is still room for improvement, our method already offers accurate cycle simulation. Experiments also show that our implementation of the dynamic adaptation phase has indeed limited influence on emulation speed.

## VII. CONCLUSION

This paper presented a method for accurate cycle estimation in fast instruction-level CPU emulators. While the cost for accurate cycle simulation is emulation speed reduction, we showed that, with proper trace-off, it is possible to achieve high level of cycle accuracy while keeping impact on emulation

speed small.

We plan to focus on improving the dynamic adaptation phase as part of our future work. In particular, we plan to perform tuning of the compensation for cache/TLB miss cases. Finally, we are confident that additional improvement of the dynamic adaptation phase with the implementation of models such as the branch predictor simulator and the memory/BUS simulator will lead us to our goal: keeping SW performance estimation error lower than 10%.

## REFERENCES

- [1] F. Bellard. QEMU, a fast and portable dynamic translator. Proceedings of USENIX Annual Technical Conference, June 2005, pp.41-46.
- [2] OSCI, Requirements specification for TLM 2.0 ver.1.1, September 2007. <http://www.accelera.org/downloads/standards/systemc/tlm>
- [3] 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 15<sup>th</sup> IEEE Mediterranean Electrotechnical Conference, April 2010, pp. 1033-1038.
- [4] I. Böhm, B. Franke, N. Topham, "Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator," in Proceedings of the International Conference on Embedded Computer Systems (SAMOS), 2010, pp.1-10.
- [5] ARM, ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition, 2008. <http://infocenter.arm.com/help/index.jsp>
- [6] ARM, Cortex-A8 Technical Reference Manual, 2006. <http://infocenter.arm.com/help/index.jsp>