Function Profiling for Embedded Software by Utilizing QEMU and Analyzer Tool

Tran Van Dung*, Ittetsu Taniguchi†, Takuji Hieda‡ and Hiroyuki Tomiyama†
*Graduate School of Science and Engineering, Ritsumeikan University, 1-1-1 Noji-higashi, Kusatsu, Shiga 525–8577 Japan.
†College of Science and Engineering, Ritsumeikan University, 1-1-1 Noji-higashi, Kusatsu, Shiga 525–8577 Japan.
‡Research Organization of Science and Engineering, Ritsumeikan University, 1-1-1 Noji-higashi, Kusatsu, Shiga 525–8577 Japan.
E-mail: {gr0150pk@ed, i-tanigu@fc}.ritsumei.ac.jp, takuji.hieda@ieee.org, ht@fc.ritsumei.ac.jp

***Abstract*—Function profiling is crucial for optimized embedded software which needs to have resource constraint, low level power consumption and real-time ability. In this work, we provide a fast and reliable solution by utilizing an instruction set simulator named QEMU and creating an analyzer tool. We developed a tracing module inside the simulator to trace execution information of software in run-time and record it in a log file. Our implementation takes advantages of the dynamic binary translation of QEMU to keep its speed fast and use an analyzer tool to analyze the log file and creates a function profile. We implemented this methodology for ARM architecture, and evaluated many kinds of embedded applications.**

Keywords*—*Function profile, QEMU, dynamic binary translation, ARM architecture.

I. Introduction

The function-level profiling is considered to be helpful and important in optimizing big and compacted software. Since developers usually have to improve software with which they are not familiar, they need some function profiling tools to find out bottlenecks of software which are the starting point of optimization process. The bottlenecks are a set of functions of the software which spends the most clock cycle of CPU and also consumes the most power. After optimizing these functions, the developers use the function profiling tools again to evaluate performance improvement.

However, it is not always the case since implementing function profiling for embedded systems is difficult and time-consuming. Among approaches for this problem, GNU profiler (*gprof*) [1] is the most popular but it has significant disadvantages. Firstly, applications need recompilation with -pg flag so they may not have the same profile as the original ones. Because the profiling adds code to executable files, it increases function call overhead and execution time. Secondly, *gprof* assumes that all calls of a routine take the same time, and therefore time should be distributed according to the number of times the routine was called from a particular call site. For many routines, this assumption of a uniform call time is erroneous. Last but not least, *gprof* does not work with embedded applications which run in user mode — without an operating system.

Besides Gprof, other approaches provide more detailed evaluation but still have disadvantages while being applied to embedded systems. The first example is Oprofile — a system-wide profiler for Linux systems [2]. It can profile all parts of a running system, from the kernel to shared libraries to binaries. Several post-profiling tools are available to turn profile data into human readable information. It even can create call-graph profiling just like Gprof does without recompilation. The second example, Valgrind [3], not only has the same ability as Oprofile, but also provides cache simulation and branch prediction to discover further information about runtime behavior of an application. However, both of them cannot work in user mode which is quite popular in embedded systems.

To solve this problem, we propose to utilize an instruction set simulator and an analyzer tool to get function profile of embedded software. The simulator simulates target systems and executes software. During run-time, it logs necessary execution information to file, and stores this file on a host computer. After that, the analyzer tool analyzes the log file, calculates evaluation result and outputs a function profile. The advantages of this method are no requirement of embedded hardware, no change in applications, and no limitation in user mode.

Among various instruction set simulators, QEMU is an outstanding one [4]. It emulates most of the CPU architectures (x86, ARM, MIPS,) on many hosts (x86, ARM, MIPS,). QEMU can run an unmodified target operating system and all its applications in a virtual machine. Moreover, it is very fast because of the implementation of the dynamic binary translator. Thanks to these advantages, QEMU has attracted many researchers to develop it into a cycle-accurate simulator. A group from Fujitsu Laboratories Ltd successfully achieved that cycle simulation error averages is only 10% while the emulation latency is 3.37 times that of original QEMU [5]. Their methodology is to add 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. Because of this impressing research, we developed our simulator from their result as a cycle accurate simulator.

The overview of our function profiling is described in Fig. 1. The embedded software is executed by QEMU with or without operating system. In the former case, which the Figure 1 shows, it is more difficult to get function profile because execution information of an application is confused by one of OS. The ”call/return event trace” module inside QEMU traces call or return events of embedded applications and logs to file, including cycle count information from ”cycle

978-1-4799-0066-4/13/$31.00 ©2013 IEEE

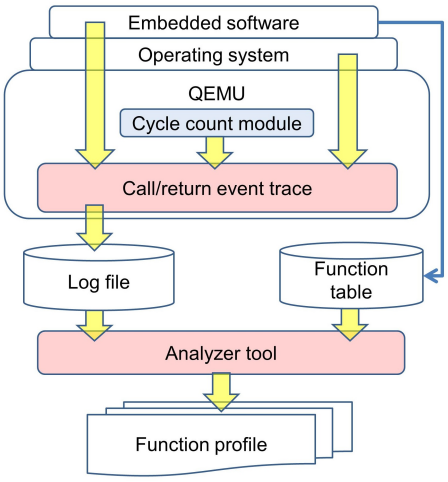

Fig. 1: Overview of proposed system

count module" which is explained in [5]. After run-time, the executable file is used to create the function table — the mapping of function names and function addresses. The analyzer tool analyzes the function table and the log file to create the function profile result.

The next section explains the methodology and implementation of the "call/return event trace" module added inside QEMU. Section 3 presents how analyzer tool works and its result. In the subsequent Section 4, we report our experiments for the function profiling environment. Finally, the conclusion and potential further studies are presented in Section 5.

II. Trace of Call/Return Event

Call/return event trace is the new module that we add to QEMU to trace every call and return event of applications. To know how many cycles each function takes, developers must know the time of entry and exit of functions as well as those of their subroutines. The problem becomes more complicated when there is an operating system because of the process switch and interrupt. Between the function entry and exit, the interrupt may occur and it must be excluded in function duration. Also, the operation system may switch to other processes during run-time, so traced information becomes confusing. To solve the first problem, we developed two sub modules named "function call/return trace" and "interrupt call/return trace" which trace the call/return events of functions and interrupts respectively. For the second problem, we utilize a variable of QEMU named context ID which is updated by OS to distinguish user mode and system mode. Our simulator only traces the execution information of applications, not that of the operating system.

A. Function call/return trace

It is important to define which information this module has to trace to avoid any redundancy. Besides tracing function call/return events, it must log the timing information of events to calculate the duration of each function. However, QEMY cannot get the names of executed functions. Instead, it can get the function addresses so that we can find the function names later. In short, this module has to trace not only function call/return events, but also clock cycle of events, and function addresses.

The idea of this module is that each CPU architecture uses specific instructions for the function call and return. The subroutine call has to store its current address so that the program control can return after the subroutine is finished. Since the subroutine return does not have target address, it gets the stored address to jump to. For example, in ARM architecture [6], the function call is executed by: *BL address*, *BLX address*. The function return is executed by: *BX LR (LR is link register)*, *POP* {*registers, PC*}, *MOV PC, LR*, *LMD* {*SP*} {*!*}, {*registers*}. Therefore, this module always checks branch instructions and creates a trigger if a function call/return event is detected.

The dynamic binary translation is the key point of QEMU. The target binary code is divided into basic blocks (BB) by branch instructions. QEMU translates and executes them one by one. First, a BB is interpreted to intermediate code, then to translated block (TB) of host code which is stored in a translation cache (TC) . When QEMU receives a new BB, it checks if the BB is interpreted. If yes, the simulator just gets the translated code from TC to execute. Otherwise, the simulator translates the BB to host code and stores in TC. The function call/return trace is developed by following this methodology.

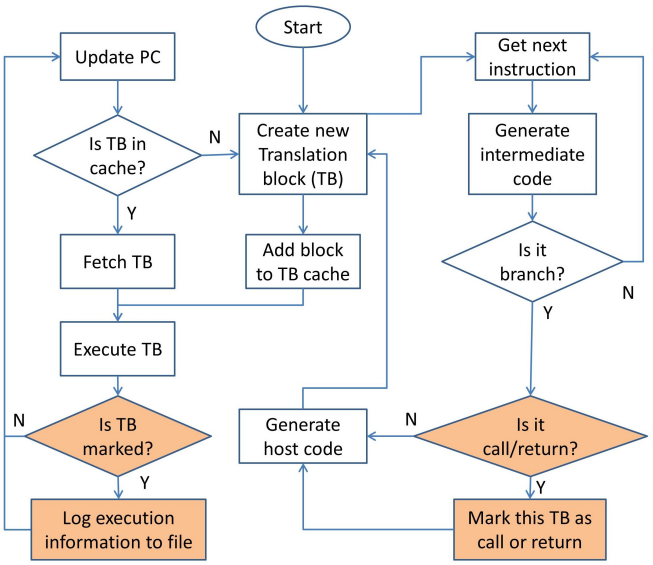

Fig. 2: Function call return trace method

Figure 2 describes the method of "function call/return trace" module. The filled blocks are our work which is added to the dynamic binary translation of QEMU. Because the simulator has to look for the specific instructions, this job must be done when translating guest code to intermediate code. However, module is not required to check all instructions. It just waits for the end of BB, and then examines if the end of BB is a normal branch or a function call/return. If yes, our module marks the corresponding TB by enabling a flag. At the executing part of the dynamic binary translation, the function

call/return trace module examines whether a TB is marked. If yes, it logs the execution information to file.

B. Interrupt call/return trace

This module traces the call and return events of interrupt in case QEMU runs with an operating system. We focus on software interrupt caused by the OS. In QEMU, interrupts are handled in the *cpu-exec()* function (located in cpu-exec.c) by calling the function *do_interrupt()*. To trace the interrupt call, the logging function is added inside this function. The interrupt return is treated differently. It not only jumps to an address like a branch instruction, but also restores all status which CPU was before this interrupt. Therefore, each architecture has specific instructions for this job. For instance, the ARM v7 architecture uses *"MOVS PC, R14_svc"*. As a result, the method to trace the interrupt return is the same as one of the function call/return trace.

Figure 3 describes in details how to implement the interrupt trace in QEMU. The filled blocks are important functions of the dynamic binary translation in the simulator which are modified. Translating to intermediate code is done by the function *gen_intermediate_code_internal()* which is located in *target-xxx/translate.c* (xxx is target architecture). Each instruction is interpreted by the function *disas_xxx_insn()*. So we add new function in this place to wait for the interrupt return instructions and mark the corresponding TB. The function *tcg_qemu_tb_exec()* is in charge of executing every TB and it is called by the function *cpu_exec()*. So in *cpu_exec()*, after calling *tcg_qemu_tb_exec()*, the logging function is added to trace the interrupt return. For interrupt call, it is caught and logged at the function *do_interrupt()* located in *target-xxx/helper.c*.

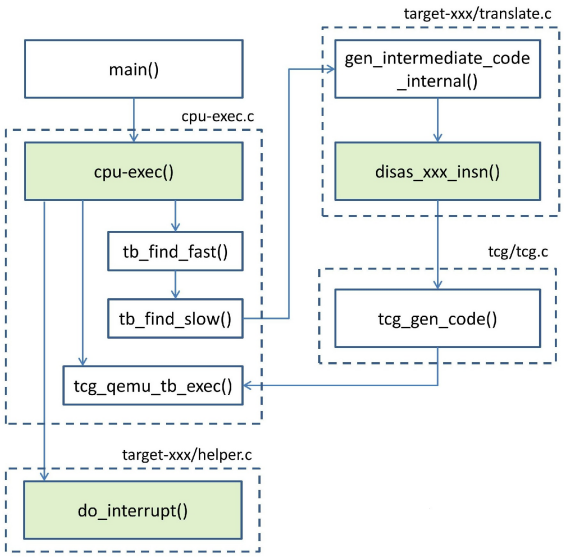

Fig. 3: Interrupt call/return trace implementation

C. Format of log file

It is crucial to define necessary information of the log file to reduce its size. We propose a format as below example:

Format of log file			
Cycle	Event	Current address	Target address
1	CALL	0x80001000	0x800000f0
5	IRQ	0x80001020	0x00000100
10	RTI	0x80000410	0x00001020
20	RET	0x80000104	0x80001004

The first column is the cycle when an event happens. The second column is the event's attribute in a short form. CALL and RET are function call and return event respectively. Similarly, IRQ and RTI are interrupt call and return event. The current address is the address of a current instruction. The target address is the address of a next instruction. We can understand the example of the log file as below:
- Line 1: At the cycle 1 when PC is 0x80001000, the function whose address is 0x800000f0 is called.
- Line 2: At the cycle 5 when PC is 0x80001020, an interrupt occurs and program control jumps to 0x00000100.
- Line 3: At the cycle 10 when PC is 0x80000410, the interrupt is ended and PC is restored as the one just before interrupt (0x80001020).
- Line 4: At the cycle 20 when PC is 0x80000104, the function whose address is 0x800000F0 is ended.

III. Analyzer tool

The analyzer tool analyzes the log file to get the function profile of software. From now, we consider IRQ as CALL which calls an unknown function at specified address and its RTI as RET. This change is easily done by a simple Perl or C program. Because all calls and returns of functions and their subroutines are clear, we can apply the analyzing method to get the function profile. However, function names are unknown because there are only function addresses in the log file. Therefore, target is to find the mapping between function addresses and function names which is called *function table*.

A. Function table

The function table can be obtained from a compiled file of software. Most of embedded applications are in the executable and linkable format (ELF) [7]. Linux provides an useful utility named $readelf$ to display information of ELF file. The mapping information is listed in the symbol table with lots of other information. We created a tool by Perl to filter and take out only mapping information. Functions of software are taken from lines which have the attribute "FUNC" in the "Type" column. Function names are in the "Name" column and function addresses are in the "Value" column.

B. Method of analyzer tool

Figure 4 describes how analyzer tool works. The key idea is to find the return event of each call event and calculate its duration by utilizing stack. Each function call is pushed to stack, so that when we get function return, its corresponding call must be at the top of stack. Also, each function call has its father function call in the top of stack. During the process of mapping call and return events, the duration of functions is calculated.

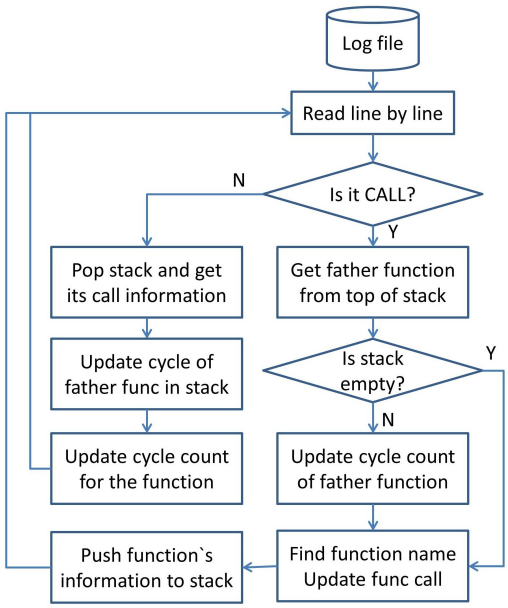

Fig. 4: Analyzing method to get function profile

IV. Experience result

We evaluated three levels of optimization of the ARM GNU cross-compiler by our function profiling environment. Each software was compiled 3 times with optimization options: -O1, -O2, -O3. Our result shows that for simple applications which do some calculation, there is no difference. However, for applications which execute a big number of loops, results of the optimization option -O3 are better while results of the optimization option -O1, -O2 are similar. It makes sense because the level -O3 enables vectorization to reduce execution time on loops.

We used our work to evaluate the optimization ability of multimedia applications. Most of them are optimized by using ARM NEON which is the advanced SIMD instruction set of Cortex-A8 to speed up audio/video processing [8]. One famous example is VP8 — the promising video codec for HTML5. We compiled VP8 with and without NEON support, encoded/decoded video files, and compared function profiles. The multimedia application was run in system mode to include various software interrupts into this experiment. The results shows that NEON optimization helps to increase speed of video processing twice. Moreover, we compared results of the function profiling with that of GNU profiler by running the VP8 encode and decode on the ARM Linux on the simulator. Both of them show the same bottleneck functions but the GNU profiler can profile only 12 functions when decoding and 25 functions when encoding — around 5% of the number of functions profiled by our profiling tool in each case. This is because of GNU profiler only profiles in the main thread of an application. Figure 5 shows an example of our output profile which lists 10 bottleneck functions of VP8 decoder.

We compared the speed of our simulator with the original QEMU using two methods. Firstly, we measured the time of booting ARM Linux on both simulators. Secondly, after ARM Linux was booted, we compared their *bogomips* obtained

Function	Count	Ave	Total	% of total
vp8_decode_mb_tokens(void)	504	8563	4315655	35.3
vp8_predict_intra4x4(void)	1552	474	735947	6.0
vp8_decode_frame(void)	10	66800	668003	5.5
memcpy(void)	20095	33	653609	5.3
vp8_decode_mode_mvs(void)	9	43514	391625	3.2
vp8_build_uvmvs(void)	990	350	346244	2.8
_wordcopy_fwd_aligned(void)	2943	99	292686	2.4
fwrite(void)	2880	94	270871	2.2

Fig. 5: 8 bottleneck functions of VP8 decoder

by the command *$cat /proc/cpuinfo*. Our result shows that the speed of our simulator is decreased 10 times of that of the original QEMU. To limit this impact, we developed an additional module named "trace control" inside QEMU so that all tracing modules can be turned on or off by applications. We implemented a pseudo device at a fixed address to which applications can write any value by using a memory map instruction. For example, it is *mmap* in ARM instruction set. If the enabling value or disabling value is written, all tracing code is turned on or off respectively. Therefore, the speed of our simulator is close to that of the original QEMU if we do not use the function profiling.

V. Conclusion

This paper introduced a method to evaluate embedded software in a simpler way. We utilize a fast cycle accurate simulator to execute and trace execution information of applications and then analyze it to create the performance profile. Our implementation for ARM architecture provides accurate and reliable result. It possibly become a useful supporting tool for software developers to optimize embedded applications.

In the future, we can make some improvements in this method to evaluate dynamic linking applications. In this case, function addresses are changed during execution so the analyzer tool cannot find corresponding function names. Therefore, we have to trace additional execution information and upgrade the analyzer tool to calculate loaded function addresses. This evaluation method is considered to be applicable to other CPU architectures such as MIPS and PowerPC in order to provide various options for software developers.

References

[1] GNU profiler, Jay Fenlason and Richard Stallman, *http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html*

[2] Oprofile, *http://oprofile.sourceforge.net/doc/index.html*

[3] Valgrind, *http://valgrind.org/*

[4] Fabrice Bellard, *QEMU, a Fast and Portable Dynamic Translator*. Proceedings of USENIX Annual Technical Conference, June 2005

[5] David Thach, Yutaka Tamiya, Shinya Kuwamura and Atsushi Ike, *Fast Cycle Estimation Methodology for Instruction-Level Emulator*. Design, Automation & Test in Europe Conference & Exhibition, 2012

[6] ARM, Cortex-A8 Technical Reference Manual, Revision r2p1 2006. http://infocenter.arm.com.

[7] ELF file format, *http://www.skyfree.org/linux/references/ELF_Format.pdf*, retrieved 09 January 2013.

[8] ARM Architecture & NEON, Ian Rickards, Standford University, *http://xecanson.jp/ARM_PDF/ARM_NEON_STANFORD_lect10arm_soc.pdf*.