HW1 Report
Real-time Analysis of a HW-SW platform

Kung Ta-Cheng
Student ID: 110652021
Dept. Computer Science
National Yang-Ming Chiao-Tung University

***Abstract*—This study will analyze the execution of Coremark using a software profiler, implement a hardware profiler on RISC-V processor, and compare the two outcomes to determine how the program's instructions affect the processor performance.**

Keywords—hardware profiling, RISC-V, Aquila

I. CLASSIFICATION OF PROFILING MECHANISMS

Profiling has become more important due to the increasing complexity of modern programs. Most programs spend their time in hotspots, which are the most frequently used code snippet. Optimizing these hotspots can bring the greatest performance improvements; thus, we need effective methods to profile programs and identify them.

A. Software profiling

To profile a program, the -pg flag must be added when compiling. It will insert some instructions to analyze runtime information, the relationship between caller and callee, or time usage, by observing the stack structure, allowing users to use gprof to profile the program's execution. One method in software profiling is generating timer interrupt every certain time slice, so profiler can collect runtime information like program counter or call stack during the interrupt and use them to estimate the actual execution of the program. However, software profiler cannot access the low-level architecture of the processor, so it cannot differentiate between memory access and computational cycle.

B. Hardware profiling

While SW profiling is easy to use, it is not accurate enough because inserting interrupts will change the structure of original code, and jumping into ISR(interrupt service routine) will also change the content in cache-system, so we need another method to profile the program's execution without impacting the original code, which is where hardware profiler excels.

With the help of hardware, we can place registers in the processor, using them to collect the runtime information in circuit-level. I capture the signals about hotspot analysis and connect them to profiling unit, where some counters are placed, so the hotspot information can be collected correctly. In this way, we can determine the percentage of time the processor spends on each function, so we can find the hotspots easily. In order to retrieve the profiling result stored in registers easily, I added some CSRs(control and status registers) into CSR file, so the result data can be retrieved by inserting assembly code into main function of Coremark, and the profiling result can be displayed on the terminal.

II. COMPARISON OF PROFILED RESULT

TABLE I.

Hotspots analysis by different profiler		
Function name	***SW-profiler***	***HW-profiler***
core_list_find()	25.51%	12.99%
core_list_reverse()	19.98%	8.79%
core_state_transition()	9.34%	17.88%
matrix mul matrix bitextract()	10.72%	16.94%
crcu8()	8.85%	12.05%

A. SW Profiler vs. HW Profiler

From the table above, it is obvious that there is a difference between the profiled result of SW-profiler and HW-profiler, there are some possible reasons:

1. Compiler :

   The program profiled by SW-profiler is compiled by default gcc(version 11.4.0), while the other one is compiled by riscv32-unknown-elf-gcc(version 13.2.0)

2. ISA:

   The SW result is obtained by executing on the x86-64 instruction set, while the HW result is on rv32-ima. As a result, the same behavior may be translated into different instructions during compilation, leading to differences in the number of cycles required for execution.

3. Microarchitecture:

   The SW result is running on my Intel i5-1135G7 core, which is an out-of-order processor, so the execution may be quite different then the Aquila's one.

TABLE II.

Function	Run time	Memory access		computation	Stall		
		store	load		Load hazard	Data fetch	From exe
core_list_find()	12.99%	0.00%	53.47%	0.68%	26.58%	26.73%	0.00%
core_list_reverse()	8.79%	27.61%	27.61%	42.51%	0.00%	27.61%	0.00%
core_state_transition()	17.88%	7.65%	19.36%	50.79%	2.73%	13.50%	0.00%
matrix_mul_matrix_bitextract()	16.94%	0.77%	13.66%	84.72%	0.00%	7.22%	46.73%
crcu8()	12.05%	0.00%	0.00%	89.39%	0.00%	0.00%	0.00%

A. *Computation cycles v.s. Memory cycles*

Table II. shows the cycles spent on each hotspot functions during the Coremark execution, so we can clearly see what type of the instructions consume most of the time in hotspot.

The main purpose of CPU, Central Processing Unit, is computing, so we expect the computation should account for most cycle during execution, however, some function - core_list_find() - even use less than 1% of its CPU time on computing, which is unreasonable. If we read the source code of it, it is clear that the reason is that that function spend time on making comparisons, so most of the instruction being executed are branches and jumps. The other two functions, core_list_reverse() and core_state_transition(), are also spend much time not on computation, it is because that they require more memory access than others, which cost at least two cycles to complete.

The memory system of Aquila stall on every load/store instruction, so the ratio of memory access will significantly impact the program execution efficiently. The function with many load/store instructions may not achieve a high computation ratio.

B. *Stall cycles analysis*

There are three types of stall cycles in Aquila, which are stall_load_hazard, stall_data_fetch, stall_from_exe.

The first type, stall_load_hazard, is caused by load use hazard, so the first two pipeline stages must stall to wait for the memory reading to be completed. The ratio of this type of stalling is very high in core_list_find(), since there are two load-use hazards in the assembly code.

The second type, stall_data_fetch, is caused by memory access. In Aquila, the memory request is generated after EX(execution) stage in pipeline, so during the first MEM(memory) stage, the request is sent to the memory system, and the data is returned on the second cycle of the MEM stage, causing the pipeline to stall for one cycle.

The third type, stall_from_exe, is because that aquila takes more than one cycle to complete a single mul/div/rem instruction, so the pipeline should wait for the result of executing stage. Since that multiplication is very important in matrix_mul_matrix_bitextract(), so it stall 46% of its execution time on waiting for the completion of multiplication.

III. SOME WAYS TO IMPROVE AQUILA

By previous discussions, the main reasons to cause the throughput of Aquila being limited are pointed out, so we can easily figure out some ways to improve Aquila.

A. *A better memory sub-system*

To avoid stalling on every load/store instruction, a improved memory access mechanism is necessary. For example, we can design a cache-system which can respond in the cycle it is requested. If the hit-rate of the cache can greater than 90%, then the AMAT(average memory access time) can be reduced from 2 cycles to 1.1 cycles, which will significantly improve Aquila's performance.

B. *Re-scheduling the instruction*

Some load-use hazard can be resolved by changing the order of execution, if we can manually find a way to schedule them so that there is no load-use hazard, the performance can be improved.

C. *Multiply and Division unit*

Multiplication and division take more cycles than other computational instructions, if we can figure out a better multiplication or division algorithm, we can reduce stall cycle spent on these stall cycle further.

D. *Advance branch predictor*

In some certain functions, the CPU are neither computing or waiting for memory respond, however, they are doing branch or jump, in this case, the computation ratio will be low.

To achieve higher computation ratio, the time spent on calculating branch condition and target address should be minimized, so we can resolve branches and jumps as early as possible. Originally, Aquila use a 2-bits saturating counter branch-predictor, which has 64 BHT(branch history table). Typically, a 2-bits counter's accuracy can be up to 93.5%, but if we adopt more complex model such as two-level branch predictor, the accuracy can be up to 97.1% theoretically, which can bring a huge impact on performance of Aquila.