Real-time Analysis of a HW-SW Platform

Chih-Hsuan Chen, 110550029

Keywords—Profile; Aquila; Processor; Hardware; Software; Xilinx; Vivado; CoreMark

I. INTRODUCTION

To analyze the performance of a program, numerous profiling methods are available that can provide programmers with detailed information about a program, thus they can further optimize the program or perform other operations. The task for this assignment is to identify the top five hotspots of CoreMark and analyze them using both software and hardware methods. Afterward, compare their similarities and differences, and analyze the ratio of computation and memory cycles.

II. PROFILING MACHENISM

A. Software

The GNU profiler, gprof, is used to conduct a profiling analysis of CoreMark in this assignment. Gprof is a sample-based profiler, which will insert a timer ISR that interrupts the program at specific frequency and detect the program counter's location. Finally, it will show how frequently each function execute and their cost of time.

B. Hardware

Based on the block diagram provided in the homework spec, I added a profiler module into the top module. This profiler receives three signals: instruction, PC and stall, and all of these signals are sent from the RISCV CORE0 module in Aquila SoC module. Except for the PC signal is sent from the writeback stage of the core, the instruction and stall signals are sent from the instruction decode stage, and pass through the execute, memory access and writeback stage.

From the map file of CoreMark, the start addresses and end addresses of the top five hotspots functions and the text section of CoreMark can be identified. According to this information, in each clock cycle, the profiler checks which interval the PC falls into, and determines whether CoreMark is executing and which of its functions is executing. If the PC is in a memory interval of a hotspot function, the profiler will identify whether this cycle is a stall cycle, and which type the instruction is according to the last seven bits of the instruction signal (opcode). i.e., if the opcode is "0000011", "0100011" or "0110111", it is a load/store instruction; otherwise, it is an arithmetic instruction.

After executing the whole program on the FPGA board, I used Vivado ILA to analyze the real behavior of the Aquila SoC. I didn't set any trigger condition here but ran trigger immediately after the CoreMark completed its execution. All the values of the counters wouldn't change anymore at this stage.

III. RESULT

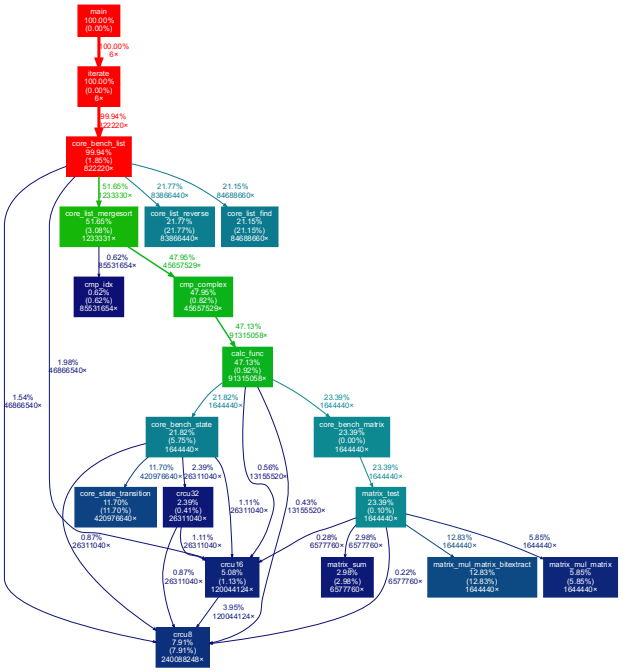

Fig. 1. The profiling result of gprof (software)

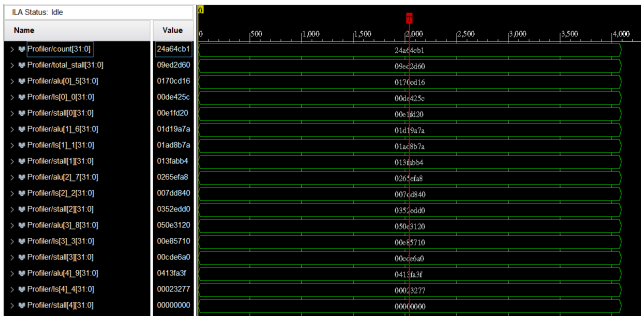

Fig. 2. The profiling result of the profiler (hardware)

A. Software & Hardware

Table I. shows the top five hotspots (by software analysis) with their CPU cycles and CPU ratio. It is evident that the profiling results by software and hardware have significant differences. Since gprof profiles a program by inserting timer ISR, the program will be interrupted frequently when executing, and it must spend lots of time to store a large amount of profiling data into the main memory, while the hardware profiler profiles a program by electric signals, which doesn't influent the program self thus can obtain much less error. Also, their ratios of the sample rate to the clock rate are quite different (1/1 on the FPGA board while 100/2.3G with gprof on my PC), indicating that the hardware profiler can obtain a more precise result.

TABLE I.

Index	Function	Software	Hardware	
		CPU Ratio	*CPU Cycles*	*CPU Ratio*
0	core_list_reverse	21.77%	53546130	8.71%
1	core_list_find	21.15%	79614376	12.95%
2	matrix_mul_matrix_bitextract	12.83%	104248760	16.95%
3	core_state_transition	11.7%	113536720	18.46%
4	crcu8	7.91%	68562102	11.15%

TABLE II.

Index	Function	Arithmetic	Load/ Store	Stall
0	core_list_reverse	45.14%	27.2%	27.66%
1	core_list_find	38.33%	35.36 %	26.31%
2	matrix_mul_matrix_bitextract	38.6%	7.91%	53.49%
3	core_state_transition	74.7%	13.41 %	11.89%
4	crcu8	99.79%	0.21%	0%
Total		72.92%		27.08%

B. Computation & Memory Cycles

As what was mentioned above, gprof must use plenty of I/O resource to store the profiling data, which may lower the I/O performance of CoreMark itself. From Table I. and Table II., we can find that the function "core_list_reverse" and "core_list_find" have about 30% of memory access instrcutions, and their profiling result differences between software and hardware are also the most significant. The function "matrix_mul_matrix_bitextract" and "crcu8" have least proportion of load/store instructions, and they have the least their profiling result differences, too. Thus, the memory access may account for a significant portion of the profiling result error from the software, leading to the observed differences between the software and the hardware.

C. Stall Cycles

Since the memory access instructions will cause lots of stall cycles, we can consider that the more the load/store cycles are, the more stall cycle there will be. From Table II., we can observe that most functions among the top five hotspots have close proportion of load/store instructions and stall cycles, excluding the function "matrix_mul_matrix_bitextract". This function does multiplication of matrices, which is much more complex than integer multiplication. Moreover, the multiplier and divider from the MulDiv module are implemented by the slowest shift-add and shift-sub algorithms. The above reasons can explain why the function "matrix_mul_matrix_bitextract" has only a few memory access instructions but has over half of its cycles stalled.

We can also observe that the function "crcu8" has no stall cycle. From the objdump file, we can find that "crcu8" is a small function with a few branch and load/store instructions. This may be why it has no stall cycle.

IV. DISCUSSIONS

From Table II., the stall cycles account for nearly 30% of all cycles, which indicates that the Aquila processor idled at many times.

Figure 3 shows a portion of the detailed signal changes of the function "core_list_find". We can observe that the stall cycles were produced not only by the load/store instructions but also by the arithmetic instructions, and the number of stall cycles generated by load/store instructions and arithmetic instructions have no significant difference. Also, most of the stall cycles of the function "matrix_mul_matrix_bitextract" were from the arithmetic instructions.

This observation indicates that the Aquila doesn't perform well on dealing with data hazards and control hazards. Therefore, Aquila can be improved by modifying or changing the algorithm of branch prediction and loading/storing. Also, it is better to use a faster multiplication/division algorithm as previously mentioned.

REFERENCES

[1] RISC-V, ISA Specifications, vol. 1, December 2019

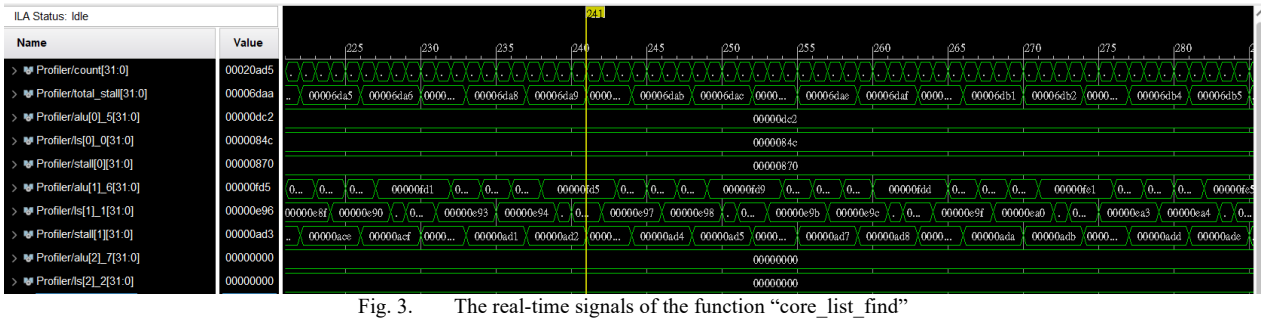

Fig. 3. The real-time signals of the function "core_list_find"

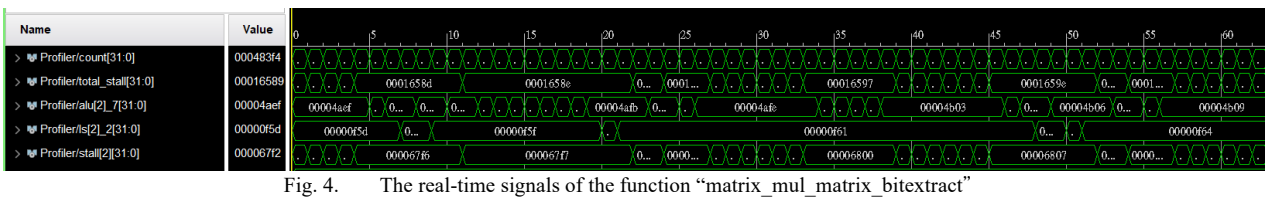

Fig. 4. The real-time signals of the function "matrix_mul_matrix_bitextract"