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

Tzu-Han Hsu
dept. Computer Science
National Yang Ming Chiao Tung University
Student ID: 081358

***Abstract*—In HW1, a hardware profiler is implemented on a RISC-V processor platform, where Benchmark program Coremark serves as the target software. We plan to compare between S.W and H.W profiler, analyse instructions in a program, and discuss how stall cycles affect the execution of a modern microprocessor.**

***Index Terms*—hardware profiling, RISC-V, Aquila**

I. Profiling Method

A. Software Profiler

I'm running WSL 1.0 Ubuntu 20.04.5 LTS on my Windows machine. After compiling the Coremark code by gcc version 9.4.0 with the -pg flag, only call analysis is provided. I have tried to switch to WSL 2.0, but that crashed my computer, so I switched back. Thus, Software Profiling statistics would be the numbers given in the HW1 slide.

B. Hardware Profiler

The goal of HW1 is to build a hardware profiler. Counters are placed beside the execute stage pipeline to monitor electrical signals. Why the execute stage? First, the execute stage is the workhorse of the CPU. Second, the execute stage is right after the decode stage, so signals to tell apart instruction types are easier to transfer. Instructions under profile are:

1) Load
2) Store
3) Branch
4) Jump (include jal)
5) Fence
6) System Call
7) CSR
8) Computation (Shift & Arith. & Logical & Compare)

Whenever the program counter at the execute stage points at the monitored function range, counters starts to accumulate. Separate counters are set up for stall cycles aiming to compare program behaviours under stalling conditions. The range of program counters of each function could be found in objdump file. Thanks to the abundant BRAM in A7-100T, probing mass number of signals are possible. 1794 nets are connected to the debug core and about 70% of the BRAM is used. We have to limit the data depth to 1024, abandon advanced trigger (we won't use it anyway) and optimize the counters in order to fit synthesized ILA in the FPGA.

TABLE I
Clock cycles spent on each Function

Item	S.W Profiler	H.W Profiler
core_bench_list( )	25.74%	1.89%
core_list_find( )	25.36%	13.23%
matrix_mul_matrix_bitextract( )	11.21%	17.11%
core_state_transition( )	9.21%	18.28%
crcu16( )	8.07%	1.19%
sum	79.59%	52.7%

TABLE II
Clock cycles executed by each Instruction type

Instruction Type	Cycle count	Percentage
Load	90,090,594	15.09%
Store	25,937,156	4.35%
Branch	124,285,151	20.82%
Jump	11,874,771	1.99%
Fence	0	0%
CSR	12	$\approx$ 0%
System Call	0	0%
Computation	344,710,569	57.75%
Sum	596,898,253	100%

II. Discussion of profiling statistics

A. H.W Profiler v.s. S.W Profiler

Clock Cycles spent on each functions by different profilers are shown in Table I. Differences between them are huge. Some reasonable explanation may be using different compilers, separate target ISA and hardware micro-architecture, distinct platform OS... From Table I and Table III, S.W profiler tends to overestimate the execution time of functions with longer stall cycles. core_list_find( ) function was mostly made of Load and Branch instructions which uses 26.84% of overall stall cycles, was overestimated 2$\times$ by the S.W profiler.

Result between S.W profiler and H.W profiler could reach a magnitude of difference. As an example, H.W and S.W profilers estimates the execution cycle of core_bench_list( ) and crcu16( ) very differently. Since theoretically H.W profiling turthfully reflects what genuinely happens inside the hardware, the experiment advise that data collected by S.W profiler could be very inaccurate.

TABLE III
PROPORTION OF DIFFERENT INSTRUCTIONS EXECUTED UNDER MONITORED FUNCTION

Instructions	core_bench_list( )		core_list_find( )		matrix_mul_mat...( )		core_state_tra...( )		crcu16( )		Program Avg.	
	Avg.	Stalled	Avg.	Stalled	Avg.	Stalled	Avg.	Stalled	Avg.	Stalled	Avg.	Stalled
Load	20.47%	2.76%	44.60%	33.37%	7.30%	47.31%	13.18%	22.14%	9.98%	25.00%	15.09%	26.56%
Store	5.68%	5.46%	0%	0%	0.39%	0%	5.21%	1.71%	9.98%	0%	4.35%	10.05%
Branch	10.11%	7.79%	54.29%	66.63%	3.89%	0%	29.78%	27.73%	0%	0%	20.82%	23.66%
Jump	9.88%	32%	0.48%	0%	0.01%	0%	4.12%	0.20%	14.96%	25.00%	1.99%	0.87%
Computation	53.87%	51.99%	0.64%	0%	88.41%	52.69%	47.70%	48.23%	65.09%	50.00%	57.75%	38.87%
Stalled Cycle	17.50%		26.84%		7.30%		13.50%		19.95%		14.81%	

B. Instruction types

Table II shows the clock cycle accumulated during Coremark runtime and the distribution of clock cycles among different categories of instructions. True work is done by computational instructions, so they should occupy the most clock cycles. Apart from computational instructions, Loading and branching takes the most clock cycles. Both types of instructions are infamous for their stalling properties. Fence, Control State Register (CSR), and System Call instructions seldom pop up during Coremark execution, we would ignore them in our further discussion.

C. Computation v.s. Memory Cycles

In Coremark, there are more computation instructions than memory cycles. However, executing a single memory instruction takes more cycle in average than a computation instruction. Most computational instruction would take a single cycle to complete, multiplication and division would take more if DSP isn't available (Aquila uses DSP). Memory instructions would cause long stall if cache misses. To reduce the stalling cycles spent on memory instructions, a well-designed cache system comes in handy.

D. Stalled cycles

Table III shows detail about stalling cycles counts in each monitored function and instruction types they would happen in. Around 15% of execution cycles are stalled on average. According to Table III, memory access (load/store) and branching instructions tend to generate stall cycles, they caused over 60% of the stalling combined. Furthermore, Store instructions would cause longer stall cycles than load instruction. Coremark has more load instruction than store, so load instruction still tops the instruction type to generate stalls. Branching instructions would cause a lot of stalls too, probably due to incorrect branch prediction. As a result, function with more memory access and branching instructions, like core_list_find( ), would hold a lot of stall cycles.

E. Better H.W profiler

H.W profiler of our design is still flawed. we only get to know the average clock cycle stalled on each function or instruction. Get to know the minimum/maximum value and variance of stalls would also help illustrate the execution of Coremark program. Design based on counting executio cycles has no idea when a function is called or returned. Further Research shall be done to design a more capable hardware profiler.

III. HOW TO IMPROVE AQUILA?

A. Software

From H.W#0, We know that software library and compiler would effect the performance of benchmark programs. Updating them would boost the benchmark performance, though no hardware improvements are made.

B. Memory system

One possible way of improving Aquila is to build a more efficient memory system, further reduce the average cycles needed to execute memory-related instructions. From statistics collected above, we understand that memory instructions tends to spend more cycles than computation instructions. Speeding them up shall reduce stall cycles, pushing more instructions through the pipeline at a fixed amount of time.

C. Branch predictor

Branching takes a lot of time and stalls often in Coremark. Branch prediction is vital in modern micro-architecture designs, misprediction in branching direction would waste some executed pipeline stages, reducing instruction throughput. I think by implementing a mighty branch predictor, the performance of Aquila could reach a new height.

D. Overclocking

PC Enthusiasts love overclocking. By adding some crazy expensive cooling systems, they would raise the clock frequency to the CPU's limit to squeeze the last bit of computing power of a piece of hardware. Though it's not literally "Improving Aquila" (more like torturing H.W), nor is it improving performance per MHZ, it's still something you shall try if you need more performance. Some possible way may be switching to more expensive FPGA boards with higher clock rates available.