Real-time Analysis of a HW-SW Platform

Zih-Bo Yu(于子博), 111550016

***Abstract*—In this report, we primarily discuss the implementation of a hardware profiler on a RISC-V processor and analyze its results, as well as the differences between hardware profiler and software profiler.**

I. SOFTWARE PROFILER

I am running tests using the CoreMark environment provided by the professor in the email on a computer running Windows 11 with WSL 2.0. Following the steps, I used gprof2dot.py and obtained the top 5 hotspots of CoreMark, which are slightly different from those provided by the professor. This could be related to the operating system or compiler version.

core_list_find() 26.10%

core_state_transition() 10.44%

matrix_mul_matrix_bitextract() 10.56%

crcu8() 7.47%

II. HARDWARE PROFILER

A. *Caculate the CPU cycle*

The program counter (PC) from the EXE stage is used as the reference point because the EXE stage is where instructions consume computational resources, and instructions in the fetch and decode stages may be flushed due to branches. Additionally, using the EXE stage's PC as the reference facilitates the calculation of memory access and stall cycles later on. Subsequently, whenever the PC enters the start or end of each function, the counter for that function is incremented by one.

Function name	Cycle count	Ratio
core_list_find()	327671	5.14%
core_list_reverse()	221621	3.48%
core_state_transition()	450902	7.10%
matrix_mul_matrix_bitextract()	427164	6.70%
crcu8()	304624	4.78%
total	6372735	100%

B. *Calculate the ratio of computation versus memory cycle*

To distinguish whether an instruction accesses memory, we can check exe_re and exe_we. These two signals represent the memory's read/write enable control, respectively. When either of them is set to 1, it indicates that the instruction in the EXE stage needs to access memory. Therefore, when exe_re || exe_we == 1, the instruction represented by the program counter (PC) is a load/store instruction.

Function name	Mem access cycle	Ratio of mem access
core_list_find()	175200	53.47%
core_list_reverse()	122400	55.23%
core_state_transition()	121760	27.00%
matrix_mul_matrix_bitextract()	61640	14.43%
crcu8()	0	0%
total	2250010	35.30%

We can observe an interesting fact that all load/store instructions in crcu8() are immediate instructions, so there is no need to access memory.

C. *Calculate the stall cycles and determine whether they are caused by load/store instr.*

In Aquila, there are two signals that can stall the pipeline: stall_pipeline and stall_data_hazard. The former stalls all stages, while the latter only stalls the PC and fetch stage. Additionally, stall_pipeline is determined by the instructions in the EXE stage, whereas stall_data_hazard is determined by the instructions in the decode stage. Therefore, we need to connect the program counter of the decode stage to the profiler to identify the instructions causing stall_data_hazard.

Function name	stall cycle	Ratio (stall cycle/cycle count)
core_list_find()	174685	53.31%
core_list_reverse()	61200	27.61%
core_state_transition()	158933	35.24%
matrix_mul_matrix_bitextract()	230440	53.94%
crcu8()	0	0%
total	2333307	36.61%

How can we determine whether a stall comes from load/store instructions? We need to check the values of exe_re and exe_we when the stall occurs. If either of them is set to 1, it indicates that a load/store instruction is being executed. In some cases,

stall_pipeline = stall_data_hazard = 1 and

exe_re || exe_we = 1

may occur(load/store + load use hazard), indicating that the instructions in both the decode and EXE stages are stalling the pipeline. In this situation, there will be two consecutive stall cycles, the first will be consider caused by load/store instruction at exe stage, the second will be consider caused by load use hazard at decode stage.

Function name	Stall by load store	Ratio (Stall by load store/stall)
core_list_find()	87600	50.14%
core_list_reverse()	61200	100%
core_state_transition()	63800	40.14%
matrix_mul_matrix_bitextract()	30820	13.37%
crcu8()	0	0%
total	1129232	48.39%

III. Analyze profiled data

A. HW profiler v.s. SW profiler

We can observe that the top 5 hotspots identified by the software profiler have significantly lower proportions in the hardware profiler, and they may not even be in the top 5. The possible reasons are as follows:

1. Different ISAs

The program run by the software profiler is executed on the x86 instruction set, while Aquila is based on RISC-V.

2. Different compilers

The software profiler uses GCC, while the hardware profiler uses riscv32-unknown-elf-gcc.

3. Different sampling methods

The SW profiler samples at a fixed frequency (100 Hz), while the HW profiler can record every clock cycle.

B. ratio of computation versus memory cycle

If we directly examine the assembly code, the functions core_list_find() and core_list_reverse() involve frequent load/store instructions, whereas matrix_mul_matrix_bitextract() and crcu8() are computation-heavy functions. Therefore, the profiler data is quite reasonable.

C. stall cycles caused by load/store instr

The high stall ratio in core_list_find() can be observed in both simulation and assembly code due to numerous load and load-use hazards.

For matrix_mul_matrix_bitextract(), the multiplication and division operations in the Aquila core require long stall cycles, resulting in a higher stall ratio but low ratio of stall by load store (high ratio of stall by computation instruction).

IV. How to improve Aquila

A. Increase cache capacity and cache levels

Modern commercial CPUs have larger caches, including L3 cache. In previous tests, we found that memory access caused many stall cycles; enhancing the cache can help avoid this situation.

B. Optimizing the Multiplier and Divider

The existing multiplier and divider require a large number of clock cycles, causing significant pipeline stalls. By optimizing it with better algorithms and improving the state machine, the computation speed can be significantly enhanced.

C. Implementing More Complex Hardware Architectures

More complex hardware architectures can bring better performance, such as changing to a 2-issue design or increasing the number of pipeline stages. However, these changes are not easy to accomplish.

D. Optimizing Pipeline Utilization for Load/Store Instructions

In the current implementation, whenever a load/store instruction is encountered, the pipeline will stall. However, if the instruction following the load/store does not need to use the MEM stage, we do not need to stall the pipeline.

V. Improving the HW profiler

A. Inability to identify the actual top 5 hotspots:

The five functions found by the software are not the ones that actually consume the most time, so I wanted to find the true answer. However, aside from the brute-force method of inputting the range of every function into the profiler, I couldn't think of an efficient algorithm, so this problem remains unresolved.

B. Excessive use of registers:

Even though I reduced the bit width of each register, the usage is still relatively high.

C. Tring different method

At first, I wanted to directly analyze the instructions, but since the instructions are no longer passed through the pipeline registers after the decode stage, I adopted the current approach. However, if the pipeline were modified to keep passing the instructions through, it would probably allow for better analysis of the instruction types.

VI. Challenges Encountered

Initially, I intended to use the PC from the writeback stage as the reference point. However, when I calculated the memory access cycles, I realized that identifying which instruction in the pipeline stages is accessing memory is very important. Consequently, I discovered that it is the instructions in the EXE stage that access memory. The same applies when handling stalls. This made me understand the importance of trace code and gain a deeper understanding of this processor.