Lab1, Real-time Analysis of a HW-SW Platform

JUN-KAI CHEN, 111550001

***Abstract*—In this report, I demonstrate the result of the CoreMark benchmark executing on a standard PC and the Aquila core. There is also a discussion of the reason of difference between the two results and propose some idea to improve the processor.**

***Index Terms*—Aquila, hardware profiler, CoreMark**

I. Introduction

In analyzing or optimizing a program, it's necessary to know the time ratio of each components. For software, the profiling tool inserts the check point to the program to calculate the hotspot. However, a more ideal and more expensive approach is to design a circuit to record the behavior of the tested program. This method could recognize different situation the processor faced and count the cost time of each components precisely.

II. Module Design

The module using to implementing hardware profiling is simple. It consists of several counters, each counters correspond to one function and count the cost time of that function. To trigger the counter correctly, the module uses the address of the instruction of the entry point as the flag. In addition, this module has an enable signal. I could change the input and collect different data without design a new circuit.

III. Result

A. Software

The software profiling is execute on my PC, with the environment is WSL 2.3.24.0 and Ubuntu 22.04 LTS version The compiler I used is gcc 11.4.0 . In order to compare with the hardware profiler, the top 5 hotspots of the CoreMark benchmark are listed in table I.

B. Hardware

The cross compiler I used to is gcc 13.2.0 . Using the design of section II, the numbers of time ratio and spent cycles are listed in table II.

IV. Comparison

A. Hardware and Software

Although the order and ratio differ, the time-consuming function are common. In fact, the only difference between the functions in the two tables is the core_list_reverse in the software and crcu8 in hardware. Based on this situation, I think software profiling tool still provides reliable results analyzing hotspot.

Checking these function in opposite result, the `core_list_reverse` function is in sixth place of hardware result. Furthermore, its time ratio is 8.89%. In comparison to 15.32% in the software result, I believe the reason lies in the regularity of the CoreMark program. If the invocation of the `core_list_reverse` function is regular, it is likely that the samples will consistently fall in the same place.

On the other hand, the time ratio of the crcu8 function in software result is 6.89%. I assume the reason is same as the previous one.

B. Computation vs. Memory

According to the hardware result, the time spent on data fetching is longer that spent on execution. Though the difference is small. The ratio of data fetching stall is roughly 14.27%, and execution stall is 12.94%. However, this execution stall only determines how long Aquila takes to complete multiplication and division instructions.

More appropriately, we can conclude that Aquila spends more time on generally computation but not data fetch.

C. Stall Cycles

In Aquila, stall cycles arise from two major situtations, `stall_from_exe` or `stall_data_fetch` being asserted. The top 5 memory stall cycles and their ratios in the overall stall cycles are listed in table IV, while those of the execution part are listed in table V.

TABLE I
Software Result

function name	time ratio
core_bench_list	20.89%
core_list_find	19.62%
core_state_transition	14.99%
matrix_mul_matrix_bitextract	11.26%
matrix_mul_matrix	7.27%

TABLE II
Hardware Result

function name	spent cycle
core_state_transition	108348240
matrix_mul_matrix_bitextract	103389660
core_list_find	79150934
crcu8	73520435
matrix_mul_matrix	62146738

TABLE III
HARDWARE RESULT(SORTED BY SPENT CYCLES)

function name	spent cycles	stall cycles	time ratio	stall ratio
core_state_transition	108348240	13493920	0.1785	0.1245
matrix_mul_matrix_bitextract	103389660	55766480	0.1703	0.5394
core_list_find	79150934	20842250	0.1304	0.2633
crcu8	73520435	0	0.1211	0
matrix_mul_matrix	62146738	32156960	0.1024	0.5174

TABLE IV
MEMORY STALL CYCLES

function name	cycles	ratio
core_list_find	20842250	1
core_list_reverse	14918090	1
core_state_transition	13493920	1
matrix_mul_matrix	7458440	0.2319
matrix_mul_matrix_bit...	7458440	0.1337

TABLE V
EXECUTION STALL CYCLES

function name	cycles	ratio
matrix_mul_matrix_bit...	48308040	0.8663
matrix_mul_matrix	24698520	0.7681
matrix_mul_vect	2744280	0.7673
matrix_mul_const	2744280	0.7778
__divdf3	9480	0.9454

From Table IV, it's clear that most come from functions related to lists. The main reason is that these kinds of function heavily use load instructions. Furthermore, some less efficient instructions such as `lbu` and `lh`, are used in certain functions.

On the other hand, the stall from execution comes from multiplication and division. This is the main reason why matrix-related functions are listed in the table. It can also explain why the execution stall of these instructions has such a high proportion of pipeline stall.

V. IMPROVE AQUILA

A. Execution

According to source code, it record that aquila spent 3 cycles on multiplication and 32 cycles on division. If we try to optimize these circuits, the execution stalls might decrease.

Based on the results of the CoreMark program, optimizing the multiplier seems to be a better choice since multiplication is more common in the CoreMark program. However, the multiplier in Aquila is synthesised by the hardware IP provided by Xilinx, which might be hard to optimize. On the other hand, optimizing the dividor would be an easier approach to improve the performance, but the number of division operation is less and won't provide a significant improvement.

B. Data Fetch

The strategy of how Aquila access data memory relies on the enable signal. At first, decode module will decode an instruction and decide whether it needs memory access or not. Once it needs, module asserts the write or read enable signals. These signals are passed to the memory stage and used by a finite state machine. If the signals are asserted, this state machine waits for data, and pipeline stall signal will be asserted. After data access is completed, the state machine sets a flag, indicating to the CPU that the pipeline can resume operation.

This method is not easy to optimize. If we hope to reduce the data fetch stall cycles, a method is to use LUTRAM to synthesis the memory instead of BRAM. I believe a more effective approach would be to optimize the program rather than the hardware, though this might not be suitable for the CoreMark program.