HW1_CoreMark Real-Time Analysis of a HW-SW Platform

郭宗信, 111550105

Abstract—This assignment explores two methods for profiling the CoreMark benchmark on a hardware-software platform. The first method employs software profiling using `gprof`, while the second involves hardware-based profiling integrated into the Aquila core using Verilog in Vivado. The work focuses on comparing these methods, analyzing program execution behavior, and identifying performance bottlenecks. The profiling data collected provides insights into program optimization, particularly in addressing stall cycles and memory access delays, which significantly impact overall performance. The results demonstrate how profiling can guide optimizations that improve the computational efficiency of the system.

Keywords—Profiling, FPGA, Aquila core, CoreMark, real-time analysis, gprof, hardware profiling.

I. Introduction

Profiling is essential for identifying performance bottlenecks in software and hardware. This study uses the CoreMark benchmark to compare two profiling methods: Gprof for software-based profiling and a hardware profiler on the Aquila core. Gprof generates call graphs and runtime statistics, while the hardware profiler measures clock cycles on an FPGA. By analyzing execution behavior, including memory accesses and stall cycles, this work targets optimizations to improve Aquila's performance.

II. Profiling Methodologies

A. Gprof

Gprof is used to profile the CoreMark benchmark on a Linux system. The program is compiled with the **-pg** flag, enabling the collection of function call data and execution times. After execution, the data is saved in **gmon.out** and analyzed by Gprof, which can generate performance reports and call graphs. Analizing these results can point highlight key hotspots helping identify areas for optimization.

B. Profiler in verilog

A hardware profiler is implemented in Verilog within the Aquila SoC to track clock cycles for specific CoreMark functions. Profiling counters monitor the program counter (PC) during function execution, counting the cycles spent in each function. The captured data is read out using the Xilinx Integrated Logic Analyzer (ILA), allowing precise performance analysis of functions at the hardware level.

III. Implementation

A. Gprof Profiling

In a Linux environment, the CoreMark benchmark is compiled using GCC with the **`-pg`** flag to enable profiling. During execution, the profiling code collects runtime data, which is stored in the **`gmon.out`** file. This file is subsequently processed by the **`gprof`** tool to generate a detailed performance report in **`profile.txt`**. To visualize the call graph, **`gprof2dot.py`** and **Graphviz** are employed, resulting in an image that illustrates the program's execution flow (Figure 1). The top five hotspots identified from this call graph provide valuable insights and serve as the basis for hardware profiling in Verilog.

B. Verilog Profiler

Using the top five hotspots identified by Gprof, the start and end addresses of these CoreMark functions can be found in **coremark.map**. A counter increments whenever the program counter (PC) falls within the function's address range, tracking clock cycles. The PC is sampled after the writeback stage, ensuring accurate profiling by excluding branch mispredictions.

Total clock cycles are tracked similarly across the text section, while memory cycles are counted when **`write_enable`** or **`read_enable`** signals are asserted. Stall cycles are recorded using the `stall_pipeline` signal. Computation cycles are calculated by subtracting stall and memory cycles from the total.

The profiling data is captured using Vivado's Integrated Logic Analyzer (ILA), with **`mark_debug = "true"`** ensuring signals remain available during synthesis. The ILA is configured to monitor the program counter and profiling counters in real-time as CoreMark runs on the FPGA, allowing for performance analysis of the functions.

IV. Results

A. Gprof Profiling Results

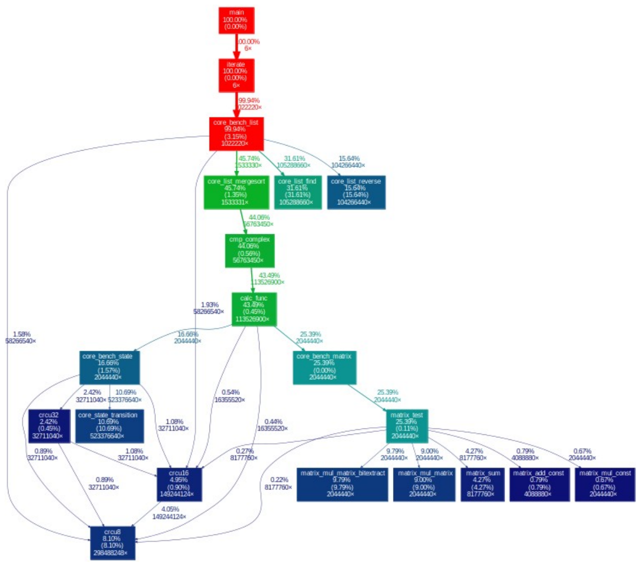
This table and figure presents the performance data gathered using Gprof, including the call graph and runtime statistics for CoreMark's top 5 hotspots.

TABLE I. TOP 5 HOTSPOTS IN COREMARK (UBUNTU HOST)

Rank	**Top 5 Hotspots**	
	Function Name	***Precentage***
1	core_list_find()	31.61%

Rank	**Top 5 Hotspots**	
	Function Name	***Precentage***
2	core_list_reverse()	15.64%
3	core_state_transition()	10.69%
4	matrix_mul_matrix_bitextract()	9.79%
5	matrix_mul_matrix()	9.00%

Fig.1. *figure generated by gprof2dot for CoreMark in ubuntu 22.04*


main 100.00% (0.00%)
100.00% 6×
iterate 100.00% (0.00%) 6×
99.94% 1022220×
core_bench_list 99.94% (3.15%) 1022220×
45.74% 1533330×
31.61% 105288660×
15.64% 104266440×
core_list_mergesort 45.74% (1.35%) 1533331×
core_list_find 31.61% (31.61%) 105288660×
core_list_reverse 15.64% (15.64%) 104266440×
44.06% 56763450×
cmp_complex 44.06% (0.56%) 56763450×
43.49% 113526900×
calc_func 43.49% (0.45%) 113526900×
1.93% 58266540×
1.58% 58266540×
16.66% 2044440×
25.39% 2044440×
core_bench_state 16.66% (1.57%) 2044440×
core_bench_matrix 25.39% (0.00%) 2044440×
2.42% 32711040×
10.69% 523376640×
0.54% 16355520×
25.39% 2044440×
crcu32 2.42% (0.45%) 32711040×
core_state_transition 10.69% (10.69%) 523376640×
1.08% 32711040×
0.44% 16355520×
matrix_test 25.39% (0.11%) 2044440×
0.89% 32711040×
1.08% 32711040×
0.27% 8177760×
9.79% 2044440×
9.00% 2044440×
4.27% 8177760×
0.79% 4088880×
0.67% 2044440×
0.89% 32711040×
crcu16 4.95% (0.90%) 149244124×
0.22% 8177760×
matrix_mul_matrix_bitextract 9.79% (9.79%) 2044440×
matrix_mul_matrix 9.00% (9.00%) 2044440×
matrix_sum 4.27% (4.27%) 8177760×
matrix_add_const 0.79% (0.79%) 4088880×
matrix_mul_const 0.67% (0.67%) 2044440×
4.05% 149244124×
crcu8 8.10% (8.10%) 298488248×

B. *Hardware Profiling Results*

These table provides the results of hardware profiling on the Aquila core, focusing on clock cycles, memory accesses, and stall cycles. The measured performance of the top five functions.

TABLE II. PROFILING COREMARK IN AQUILA FOR 5 HOTSPOTS

Rank	**Top 5 Hotspots**	
	Function name	***Percentage***
1	matrix_mul_matrix()	10.08%
2	core_state_transition()	7.08%
3	matrix_mul_matrix_bitextract()	6.71%
4	core_list_find()	5.21%
5	core_list_reverse()	3.48%

TABLE III. PROFILING COREMARK IN AQUILA FOR 5 HOTSPOTS

Top 5 Hotspots And There advance analysis			
Function name	***computation***	***stall***	***memory***
matrix_mul_matrix()	18.88%	27.03%	54.06%
core_state_transition()	27.61%	27.62%	44.77%

Top 5 Hotspots And There advance analysis			
Function name	***computation***	***stall***	***memory***
matrix_mul_matrix_bitextract()	59.10%	13.49%	27.01%
core_list_find()	31.80%	53.95%	14.42%
core_list_reverse()	28.81%	52.90%	18.9%

V. DISCUSSION

A. *Differences Between Software and Hardware Profiling*

Gprof identifies CPU-bound bottlenecks like **core_list_find()** and **core_list_reverse()**, while hardware profiling highlights memory-bound functions such as **matrix_mul_matrix()**. The difference may due to Gprof measuring CPU time without considering memory delays or stalls, while hardware profiling captures them, offering a more comprehensive system performance view.

B. *Analysis of Computation vs. Memory Cycles in HW profiling*

Hardware profiling results indicate that a significant proportion of clock cycles in functions like **matrix_mul_matrix()** are spent on memory operations (54.06%), demonstrating that memory access is a key performance bottleneck. In contrast, functions such as **core_list_find()** and **core_list_reverse()** show lower memory cycle percentages but higher stall cycles, indicating that pipeline stalls, rather than computation time, are the primary cause of performance degradation. Optimizing memory access patterns and reducing load/store latency could alleviate these bottlenecks.

C. *Stall Cycle Analysis*

Stall cycles have a significant impact on Aquila's performance, with functions like **core_list_find()** and **core_list_reverse()** exhibiting stall rates above 50%. These high stall percentages may suggest that performance degradation is due to pipeline inefficiencies. Addressing these issues through pipeline optimization, such as reducing dependencies and improving instruction scheduling, could significantly improve performance by minimizing the time spent in stall cycles.

D. *Strategies for Improving Aquila's Performance*

Improving

To enhance Aquila's performance, optimizations should target both memory and stall-related bottlenecks. Improving cache utilization and increasing memory bandwidth could mitigate memory cycle bottlenecks, while optimizing pipeline efficiency, through better instruction scheduling and dependency resolution, could reduce stall cycles. Additionally, In the case, optimizing matrix operations and improving data access patterns may further improve performance, particularly for this platform. Future work should focus on these areas to maximize system efficiency and minizing the cost to achieve better overall performance on Aquila.