HW#3 Cache Optimization

111550124 陳燁

***Abstract*— In this assignment, experiments with different cache size, associative ways, and replace policy is tested. Trying to find a best way to make the pi.c program as fast as possible. Moreover, explain the downgrade and upgrade of each different ways that I tried in this lab.**

I. INTRODUCTION

In this assignment, I test various cache sizes, associativity levels, and replacement policies to see how they affect efficiency. I start with the default FIFO policy and try it with different cache sizes and associativity settings. Then, I compare FIFO to other policies like LRU and RANDOM to figure out which works better. Finally, I explore ways to improve the dcache's finite state machine to boost performance.

II. PI.C ANALYSIS

In the pi.c program some key arrays such as one, pi, pwr, temp, and trm are frequently accessed to store intermediate results. This exhibit strong spatial and temporal locality of this program. Therefore, the performance of the program will heavily depand on the repetitive read and write operations in dcache. Optimizing cache behavior is therefore essential to enhancing the performance of this high-precision π computation.

III. DCACHE ANALYSIS

A. Dcache FSM

When there is a store or load instruction, the p_strobe_i from core_top will be set to one to tell dcache that there is a request from the core. Also we will check p_rw_i to see whether it is a read or write instruction. Once the FSM finds out a request occurs it will then transform into the analysis state. If a cache hit occurs it will then go back to the idle state and take the data back to the memory stage. On the other hand, when we did not hit the cache, it will then check whether the victim_sel is dirty. If the victim_sel is clean, it will directly go the RdfromMem Stage. Once the m_ready_i is set one, the data from main memory is ready, it then go back to the RdfromMemFinish state and Idle state. Otherwise, we need to write the data store in the victim_sel to main memory. Hence, we will transform to the WbtoMem state. Once the WbtoMem is finished it will transform to the WbtoMemFinsih State and finish the RdfromMem just as if the victim_sel is clean.

B. Read Hit and Write Hit

As we can see in the previous paragraph, when the state is in analysis and cache_hit equals to one. There is a hit in the cache, we can further check if it is a read miss or a write miss by checking the rw signal. If rw is one indicates that it is a write hit. On the other hand, if it is zero, it is a read hit.

C. Read Miss and Write Miss

When we need to read a data from the main memory, it indicate that there is a miss in the cache. Therefore, we can check the cache miss by if the state is in RdfromMemFinish. Also , just like read hit and write hit, we can distinguish if it is a read miss or a write miss by the signal rw. Moreover, as I test, writing back to the memory takes around 19-21 cycle and reading from memory takes around 34 to 36 cycle. Thus, once a cache miss happens, the whole CPU will stall around 55 cycle to wait for the data reading from the main memory.

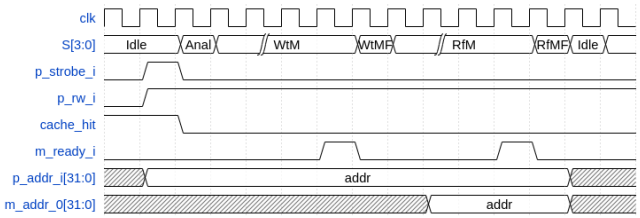

Fig. 1: timing diagram for dcache fsm

IV. IMPLMENTATION

A. Change Cache Size and Change Sst Associative

To change the cache size, we need to find the marco define DCACHE_SIZE in aquila_config.vh and change it to 8KB to test the difference between different cache size.

B. Change Set Associative

To change the set associativity in the given code, I change the N_WAYS in the DCACHE and increase the array of way_hit to help rewrite the cache_hit logic and hit_index logic.

C. LRU

The Least Recently Used (LRU) replacement policy operates on the principle of evicting the cache line that has not be used for the longest time. This method assumes that data accessed least recently is less likely to be reused soon. To implement the LRU, I used a matrix of registers called ORDER to track cache usage. On a cache hit, ORDER updates the usage order. Other lines' ranks shift up, and the accessed line moves to the end as most recently used. On the other hand, when a cache miss occurs, the least recently used line is evicted. It will be update to the new cache line data. The remaining data ranks adjust by shift upwards by one. Also, the new line goes to the end as most recently used.

D. Random

The random replacement policy indicates that when we need to read a new cache line, we need to randomly choose a victim_sel in the same line_index. I implement this by using two random number, which is called LSFR and LCG respectively. The LSFR works by shifting its bits to the right and create a new bit by using an Xor operation. In my implementation, the LSFR is four bits long, and the new bit is create by xoring LSFR[3] and LSFR[2]. This methid is leightweight and easy to implement in the hardware. On the other hand, I also create a LCG to increase it's randomness. The LCG is done by the formula $X_{n+1} = (a \cdot X_n + c)\%m$. To be more specific, a equals to 16'h41C6 and c equals to 16'h6073. Finally, we can select a victim_sel by this two randon number, and te formula I used can be see in below.

$$\text{LFSR}_{n+1} = (\text{LFSR}_n[2:0] \,||\, (\text{LFSR}_n[3] \oplus \text{LFSR}_n[2]))$$
$$\text{LCG}_{n+1} = (\text{LCG}_n \cdot a + c) \mod 2^{16}$$
$$\text{victim\_sel} = (\text{LFSR}_{n+1} \oplus \text{LCG}_{n+1}[3:0]) \mod 2$$

V. RESULTS AND ANALYSIS

A. Different Cache Size and Set associative

Tables I and II show how the cache performs for 4KB and 8KB sizes using the FIFO replacement policy. The tests compare different associativity levels: 2-way, 4-way, and 8-way. As we can see, increasing the cache size improves performance a lot. A bigger cache can hold more data, which means fewer replacements are needed.

On the other hand, adding more cache ways doesn't help—it actually hurts performance. In the 4KB data cachce, since the cache size is too small the difference between different set associative is not obvious. However, in the 8KB data cache, th smaller the set associative is, the better the performance is. This is because of how FIFO works. It treats frequently accessed hotspot entries the same as less important data, so those entries don't stay in the cache long enough. With fewer cache ways, there are more line indices available, which reduces competition for space. This makes it easier for hotspot entries to stick around longer. That's why lower associativity performs better in this situation.

Set Associative	2 way	4 way	8 way
Miss Rate	20.0293%	20.0292%	20.0292%
Read Miss Rate	22.4260%	22.4259%	22.4259%
Write Miss Rate	16.6992%	16.6992%	16.6992%
Total Time (ms)	30517	30517	30517

TABLE I: Cache Performance Results for 4KB with FIFO

Set Associative	2 way	4 way	8 way
Miss Rate	12.5817%	17.9181%	19.0770%
Read Miss Rate	16.8554%	19.4216%	20.7884%
Write Miss Rate	6.6437%	15.8291%	16.6992%
Total Time (ms)	25445	29081	29848

TABLE II: Cache Performance Results for 8KB with FIFO

B. DIFFERENT REPLACEMENT POLICY

The Least Recently Used (LRU) replacement policy performs better than First-In-First-Out (FIFO) in most cases. It works by keeping track of how recently each cache line was used. When it needs to evict a line, it chooses the one that hasn't been used in the longest time. This makes LRU a great fit for the hotspot that are likely to be reused soon in the later part of the program.

On the other hand, the eviction logic of FIFO is much simpler. It evicts the oldest cache line. No matter how often it was used. Because of this, it often removes data that's still needed. This will lead to more cache misses. For example, with 2-way associativity, LRU had a much lower read miss rate (14.54%) than FIFO (16.85%). This shows how well LRU keeps frequently used data in the cache. By reducing misses, LRU also cuts down on memory fetches and speeds up the program. Therefore, LRU's smarter approach makes it much better than FIFO.

Random replacement performs surprisingly well too. It doesn't track usage like LRU, but it avoids the problems FIFO has. Randomly picking lines spreads evictions out, which reduces the chance of removing important data too often. I saw that its miss rate (12.30%) was very close to LRU's (12.12%). Still, LRU has the advantage because it explicitly tracks access patterns. That makes it a better choice for workloads where data gets reused often.

Associativity	Policy	FIFO	LRU	Random
Two Way	Miss Rate	12.58%	12.13%	12.34%
	Read Miss Rate	16.86%	14.54%	15.17%
	Write Miss Rate	6.64%	8.78%	8.40%
	Total Time (ms)	25445	25123	25259
	Improvement	0.00%	0.91%	0.73%
Four Way	Miss Rate	17.92%	12.12%	12.30%
	Read Miss Rate	19.42%	13.59%	14.45%
	Write Miss Rate	15.83%	10.07%	9.31%
	Total Time (ms)	29081	25106	25231
	Improvement	0.00%	13.67%	13.24%
Eight Way	Miss Rate	19.08%	12.47%	12.58%
	Read Miss Rate	20.79%	14.29%	14.80%
	Write Miss Rate	16.70%	9.25%	9.48%
	Total Time (ms)	29848	25370	25416
	Improvement	0.00%	15.28%	14.85%

TABLE III: Cache Performance for Different Policies and Associativities

C. Hardware Utilization

The hardware requirements for FIFO, LRU, and RANDOM replacement policies differ based on how complex they are to implement. FIFO (First-In-First-Out) is the simplest. It uses a single counter (FIFO_cnt) to track the oldest cache line in each set. The counter increments every time

a replacement happens, making FIFO easy to implement and very efficient in terms of hardware. It doesn't need much logic or memory. For example, it only uses 650 Slice Registers and 41 F7 Muxes. The Block RAM Tile usage is the same for all policies because it depends on the cache size, not the replacement logic.

LRU (Least Recently Used) utilizes most resource because it has to track and update the access order for every cache line. This is done using an ORDER array, which stores priority data for each line. Every time there's a cache hit or miss, the priorities are updated, which takes a lot of logic and storage. This explains why LRU needs 2576 Slice LUTs, 1414 Slice Registers, and 176 F7 Muxes.

RANDOM is somewhere in between FIFO and LRU. It uses a pseudo-random number generator (LCG) and a linear feedback shift register (LFSR) to pick a random victim number. It doesn't need to track access patterns like LRU, but the LCG adds some extra hardware cost. RANDOM uses 2184 Slice LUTs and 115 F7 Muxes, which is less than LRU but more than FIFO. In short, the more complex the replacement logic, the more hardware it needs.

Metric	FIFO	LRU	RANDOM
Slice LUTs	2142	2576	2184
Slice Registers	650	1414	1414
F7 Muxes	41	176	115
F8 Muxes	10	40	42
Slice LUT as Logic	729	1041	1033
LUT as Memory	2110	2544	2152
Block RAM Tile	32	32	32

TABLE IV: Hardware Resource Utilization with 4 way set associative for FIFO, LRU, and RANDOM Policies

VI. OPTIMZIE FSM IN DCACHE

When the cache is writing a cache line from the memory it will spend one cycle in WbtoMemFinish. However, in the dcache logic, nothing is done in this state. Therefore, if we remove this state, we can reduce one cycle every time we need to write a cache line back main memory. The same logic also fits for the RdfromMemFinish. Nevertheless, in the dcache logic, there are some operation that is done based on this state. Thus, we need to replace the S == RdfromMemFinish logic to (S == RdfromMem && m_ready_i). By reducing this two state. We can save one cycle every time we read a cache line from main memory and wrtie a cache line back to the main memory. The result of the comparison is in table V and the new timing diagram of the new FSM is in figure 2.

VII. FUTURE WORK

A. Prefetch Logic

For a program like pi.c, which has clear patterns in how it accesses memory, prefetching can help increase the performance. This is done by predicting what data the program will need next and load it into the cache before the CPU asks for it. This reduces the delay caused by cache misses, especially in loops or when memory is accessed in regular steps. By making the data available ahead of time. The soc can work faster helps and avoid waiting, enhancing the performance of dcahe.

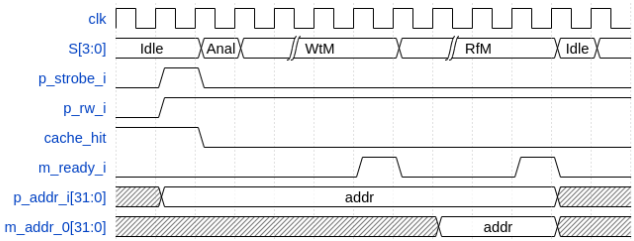

Fig. 2: Timing diagram for new FSM in dcache

Metric	FIFO (4-way)	FIFO (4-way) with new FSM
Miss Rate	17.92%	17.92%
Read Miss Rate	19.42%	19.42%
Write Miss Rate	15.83%	15.83%
Total Time (ms)	29081	28602
Improvement	0.00%	1.64%

TABLE V: Comparison of performance metrics between FIFO (4-way) and FIFO (4-way) with new FSM.

B. Writeback Buffer

A writeback buffer is a straightforward way to speed up memory writes. The principle behind this is simple. It wont write the data to memory immediate when a cache line is evicted. On the other hand, the buffer holds the evicted line temporarily. This means the CPU can keep working without waiting for the the signal m_ready_i. This helps the SOC run faster. By taking care of writes in the background, the buffer reduces delays and avoids slowing down the system. For programs with frequent writes, a writeback buffer is a simple and effective tool to improve performance.

VIII. CONCLUSION

In this assignment, we explored various strategies to improve dcache performance and analyzed their impact. While enhancing cache performance can reduce miss rates and improve efficiency, it often comes at the cost of increased hardware resource usage. Balancing performance and resource efficiency is the key challenge in cache design. Therefor, future work could focus on optimizing these trade-offs to achieve better performance without excessive hardware overhead.