Cache Optimization

Zih-Bo Yu(于子博), 111550016

***Abstract* —This report will first introduce the cache system of Aquila and test the changes brought about by modifying certain configuration parameters. Then, further modifications will be introduced to improve memory performance.**

I. CACHE SYSTEM IN AQUILA

A. *Implementation*

By default, the data cache has a size of 4KB and is 4-way associative, with a state machine used to represent the state of the data cache controller. The data cache size and the number of ways can be modified, but when changing the data cache size, it is also necessary to adjust `way_hit`, `c_valid_o`, `c_tag_o`, and `hit_index` to match the corresponding sizes.

The cache line size is 128 bits, which means that when a cache miss occurs, the cache will request 128 bits of data starting from the target address in memory. The address is byte-aligned, and for instructions like `lb`, the `byte_enable` signal indicates which specific byte should be accessed.

B. *Analysis of data cache*

When `p_strobe_i` is triggered, it indicates that the CPU needs to access data from memory. The next clock cycle determines whether it is a hit or a miss. Therefore, the cycle count starts from the clock cycle following `p_strobe_i`. Due to the characteristics of `p_strobe_i`, it can be delayed by one clock cycle as `p_strobe_i_d`. At this point, not only can a cache hit or miss be determined, but the operation type (read or write) can also be identified based on `p_rw_i`. The cycle counting stops when `p_ready_o` is triggered.

II. OPTIMIZING THE DATA CACHE

A. *Cache size*

From the statistics on the next page, it is evident that increasing the cache size provides the most significant improvement. Ironically, this is also the simplest approach. As a result, many CPUs on the market emphasize their cache size, as the performance gains are substantial. In gaming, cache bottlenecks are often encountered, which is why AMD's recently launched 9800X3D is considered the most powerful gaming CPU for consumers.

Interestingly, when we increase the cache size to 16KB, the hit rate reaches 99%. This indicates that the cache size is large enough to accommodate almost all the data required by the program. Once the data is moved from memory to the cache, it will not be replaced.

Strangely, when I adjusted the cache size to 16KB, the simulation failed to run properly, getting stuck before entering the `pi` program. I'm not sure if this is normal, but it doesn't affect the actual hardware implementation. All the test results in this report are based on running on board up to 5000 decimal places.

B. *N-way associative*

We can observe that, under the same conditions, changing the number of ways results in only minimal impact. I believe the reasons are as follows:

Increasing the number of ways is primarily aimed at reducing conflict misses (caused when multiple memory addresses map to the same set, leading to premature cache line replacement). Therefore, if the memory access pattern of a program does not exhibit significant conflicts (e.g., the accessed data is distributed relatively evenly), increasing the number of ways has a limited effect on performance improvement.

C. *Replacement policy*

1) *FIFO*

In Aquila's default configuration, FIFO is used as the replacement policy. The `FIFO_cnt` is incremented after handling a cache miss, ensuring that the way that was brought in first will be replaced.

2) *LRU*

If we want to replace FIFO, the most intuitive choice is LRU. My implementation approach is as follows: each line keeps track of the number of times it has been accessed. When one of the ways is accessed (on a hit or miss), the access count of the line is stored in a register representing the corresponding data. This way, by identifying the way with the smallest number in each line, we can determine which one is the least recently used.

3) *Random number*

An LFSR is used to generate a 2-bit random number, which serves as `victim_sel`.

D. *Other Optimizations*

Before introducing pre-fetching, I noticed that during a cache miss, signals such as `m_strobe_o` and `m_addr_o`, which the cache sends to memory, would lag by one clock cycle. To address this, I modified these signals so that they change as soon as `S_nxt` is detected. This optimization reduces the clock cycles required for each miss by one. Since this optimization was implemented later, it was only tested with two configuration. Please refer to the "with Miss Optimization" section below.

E. *Possible Optimizations: write buffer (Not Implemented)*

By default, if a write cache miss occurs and the block to be replaced is dirty, the data needs to be written back to memory. The process waits until the memory write is complete before reading from memory and writing new data. However, for a write operation, the original memory data is not required. Therefore, a **write buffer** can be used to store the data to be written. Once the data is written to the buffer, the CPU can continue execution, while the write buffer handles the memory write operation in parallel.

Cache size	4KB					8KB				
Replacement policy	FIFO			LRU	random	FIFO			LRU	random
Way number	2	4	8	4	4	2	4	8	4	4
Run time(ms)	30533	30517	30517	30518	28996	25457	29087	29848	29018	24928
Read hit rate	77.57%	77.57%	77.57%	77.57%	80.05%	83.14%	80.57%	79.21%	80.53%	85.92%
Write hit rate	83.30%	83.30%	83.30%	83.30%	85.13%	93.34%	84.17%	83.30%	84.45%	91.25%
Read latency	13.26	13.24	13.24	13.24	11.87	10.21	11.61	12.33	11.63	8.67
Write latency	10.11	10.10	10.10	10.10	9.09	4.63	9.63	10.09	9.47	5.75

Cache size	4KB with miss Optimization	8KB with miss Optimization		16KB with miss Optimization
Replacement policy	FIFO	LRU	random	FIFO
Way number	4	2	2	4
Run time(ms)	29989	26020	24688	16903
Read hit rate	77.57%	83.03%	84.44%	99.97%
Write hit rate	83.30%	90.13%	93.05%	99.91%
Read latency	12.74	9.87	9.15	1.01
Write latency	9.79	6.18	4.63	1.04

III. Test Data Analysis

A. Overall

The arrangement of the test results table is based on the degree of impact: cache size has the greatest effect, followed by the replacement policy, and finally, the way number. However, in certain cases, the way number can have a significant impact.

Comparing different configurations reveals a strong correlation between hit rate and latency: the higher the hit rate, the lower the latency. Runtime, on the other hand, is influenced by the hit rates of both read and write operations. In the program that calculates to the 5000th decimal place of π, there are a total of 36,281,369 read requests and 26,146,366 write requests. As a result, even when the write hit rate is lower, a higher read hit rate can lead to a shorter runtime. This can be observed, for example, in the relationship between the 8KB-2way-FIFO and 8KB-4way-random configurations.

With miss optimization, latency can be reduced without changing the hit rate. The lower the hit rate, the greater the improvement. However, overall, the improvement is limited to about 1–2%. The advantage of this approach is that it provides a stable and predictable performance boost, whereas modifying the way number or replacement policy may result in varying levels of improvement depending on the program being executed.

B. Cache size

There is an interesting phenomenon with different cache sizes: at 8KB, the impact of the way number and replacement policy is more significant than at 4KB. I consider that this is because the number of lines increases, leading to fewer collisions, which in turn enhances the effectiveness of the optimizations.

C. Way number

Modifying the way number usually does not result in significant changes, but there is one exception: at 8KB, reducing the cache to 2-way results in a substantial improvement. I initially thought increasing the number of ways would enhance performance, but it may not be that simple. To investigate further, I tested CoreMark running on DDR memory. At 8KB-4way, the score was 100.067218, while at 8KB-2way, the score slightly increased to 100.067473. This suggests that reducing the number of ways at 8KB could potentially improve performance. However, it is also possible that the memory access patterns of these two programs caused this phenomenon.

D. Replacement policy

From the table, it can be observed that LRU provides almost no improvement compared to FIFO and is sometimes even worse, whereas random shows significant improvement. Of course, I believe this is related to the characteristics of the program. To verify this, I tested CoreMark. Under the 4KB-4way configuration, the score for random was 100.0671, while the score for LRU was 100.067491. This was quite reassuring, as I finally understood that LRU generally performs better than random in typical scenarios.

IV. About Pre-fetching

Even though I have not successfully implemented it yet, I have spent a significant amount of time on it. Therefore, I will explain my implementation approach and the potential causes of the errors.

A. *Reason for Wanting to Do Prefetching*

When examining the waveform, I discovered that the CPU's memory access addresses follow a pattern, and the addresses with cache misses also show regularity. Often, a miss at address x is followed by a miss at address x+16. This is because the cache retrieves 128 bits of data from memory at once. When address x experiences a miss, the data from x to x+15 will be stored in the cache. Therefore, once address x encounters a cache miss and the miss is processed, it becomes possible to prefetch the data for address x+16 into the cache.

B. *Implementation Idea*

First, I don't want to add another state to the state machine because its characteristics would highly overlap with Idle. Moreover, adding more states would increase the workload, so I decided to perform prefetching during the Idle state, using a variable to indicate whether prefetching is in progress. Then, I divided the work into two parts: 1. Ensure that the prefetching m_strobe does not affect the program's correctness 2. Store memory output data in the cache. The second part is not difficult; the key is the first part.

C. *Part.1*

In the first part, I decided to strobe when the state transitions from *RdfromMemFinish* to *Idle*. At this point, the strobe signal is used to request the prefetching address from memory. In the ideal case, if the dcache controller remains in *Idle*, when *m_ready_i* is triggered, the data can be stored into the cache. However, the state will not stay in *Idle* indefinitely. If the controller enters another state and a strobe is needed again, problems may arise. By examining the waveform, I found that if a strobe is triggered while the DRAM is processing an ongoing request, the DRAM will ignore the new request and continue executing the old one. Therefore, when this situation occurs, it is necessary to strobe again after prefetching finishes to ensure the correctness of the data. Additionally, because states related to the *fencei* instruction and *AMO* instructions also have an impact, handling these states is too complex. Therefore, I decided to start prefetching only after entering the main function.

This is where I got stuck. After implementing the design according to my idea, I was able to run the program for 50 or 500 decimal places of π on the board, and the results were correct. However, when running for 5000 decimal places, the output was incorrect. I have confirmed that the issue is due to the strobe signal, which causes incorrect register values and, in turn, leads to erroneous results. However, this kind of bug is very difficult to resolve because I cannot compare the contents of the registers one by one during the board test, making it impossible to pinpoint the exact location of the fault.

D. *Part.2*

If part 1 can be completed, the remaining work becomes relatively simple. When triggering the strobe for prefetching, we need to store relevant information such as *cache_write, valid_write*, and other related signals. This is because when *m_ready_o* for prefetching is triggered, these signals may be overwritten by those of the new request. Therefore, when prefetching is finished and data needs to be stored in the cache, the saved signals should be used. In the absence of prefetching, the original signals can be used instead.

V. Things I Learned

Although pre-fetching was not successfully implemented, it was not without its gains. In fact, the miss optimization is something I discovered during the pre-fetching implementation that could be improved. Additionally, examining the waveform helped me understand how the DRAM operates. Furthermore, the process of implementing pre-fetching gave me a more comprehensive understanding of the entire dcache controller.