Lab3, Cache Optimization

JUN-KAI CHEN, 111550001

***Abstract*—This paper focuses on the analysis and improvement of the data cache in the Aquila SoC implemented on an FPGA. We'll discuss how the cache controller operates and how we measure cache performance in the first section. The second paragraph will explain the implementation of different cache replacement policies and their results.**

***Index Terms*—Aquila, Cache, Cache Replacement Policy, FIFO, LRU, Random**

I. Data Cache Analysis

A. Data Cache Controller

An access to the data cache is divided into hit and miss. Since a cache hit is much simpler, we should focus on the cache miss. When a cache miss happens, the cache controller will select a victim, checking the dirty bit. If the dirty bit is asserted, it writes the cache line to data memory, and then it read the requested cache line. Thus, the time of accessing memory could be divided into two parts: writing and reading.

It is worth noting that the operation of writing to memory is not related to wrtiting to the cache. Writing to memory, that is, replacing cache blocks, is only related to dirty bits.

B. Data Cache Profiler

Since the cache is a bridge of processor and memory, we need to record the requests from both sides. The module needs to record the `p_rw` and `p_strobe` from the processor and the `c_cache_hit` and `c_ready` from the cache to determine the current request from the processor.

The cache controller of Aquila is quite simple and intuitive. For memory requests, we only need to use the state of the cache controller. In order to distinguish the uploading executable file from the actual city of execution, the PC value of the processor is additionally used.

Fig 1 is the block diagram drawn based on this idea. The signals on both sides will flow to and control their respective counters. In addition, in order to distinguish the operation of the cache by the bootloader and the program itself, I used the PC value to ensure that the uploaded program is currently operating the memory.

The number of cache misses should be equal to the number of read requests sent to the memory, and that of write requests should be less than or equal to the number of read request. We can use this to understand how often a cache blocks had been written before it was replaced. If the two closer, it means that additional writing is required, resulting in a longer latency for a processor request.

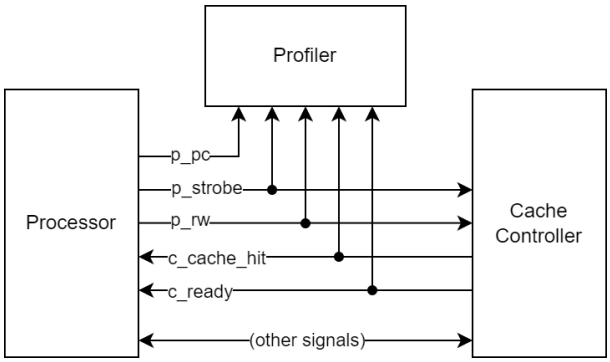

Fig. 1. The block diagram of the profiler module

C. Data Cache Performance

In this report, the software used for testing is to calculate the 5000-digit π value. The hardware limits the data cache size to 4 KB. Based on this situation, the datas of cache hit and cache miss are recorded in Table 1, while datas of memory requests are recorded in Table 2. The pi program spents 30,517 milliseconds to complete the calculating task with the 4-way set associative cache with FIFO policy.

For a cache hit, the processor sends a request and get the response after a cycle. The latency of a cache hit is always 1 cycle. Thus, it's not listed here.

The pi program accesses memory approximately 62 million times. Considering that the number of memory writes is almost equal to the number of reads, we can know that most cache lines were written before they were replaced. Another way to illustrate this is that the latency of a cache miss is very close to the sum of the latencies of reading and writing memory.

TABLE I
THE DATAS OF CACHE HITS AND MISSES WITH 4 WAYS FIFO CACHE.

	read	write
hit (%)	45.11%	34.86%
miss (%)	13.04%	6.99%
miss latency (cycles)	54.61	54.53

TABLE II
THE NUMBER AVERAGE LATENCY OF MEMORY REQUESTS.

	read	write
number	12,497,054	12,496,803
average latency (cycles)	34.54	20.04

TABLE III
EXECUTION TIME AND MISS RATE OF EACH DESIGN.

cache design	execution time (ms)	miss rate
2-way FIFO	30,518	20.029275%
4-way FIFO	30,517	20.029211%
8-way FIFO	30,517	20.029252%
2-way LRU	30,518	20.029062%
4-way LRU	30,518	20.029256%
8-way LRU	30,518	20.029294%
2-way Random	29,858	19.081489%
4-way Random	29,820	19.044288%
8-way Random	29,715	18.899493%

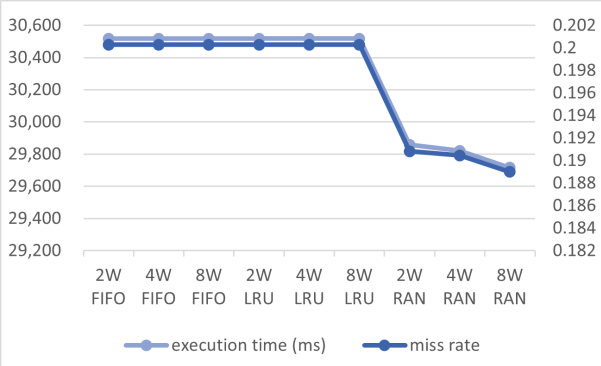

Fig. 2. Execution time and miss rate of each design.

II. Different Cache Design

To test which features and how they affect performance, I experimented with different associativities and replacement policies. To compare each design, I recorded the execution time and the miss rate in Table 3 and plotted as Fig. 2, as they're the most significant indicators. For diffrenet replacement policies, I only modified the method to select the victim cache line, and remained other part.

In addition, to understand the resource requirements of different implementations, I listed the resources used by different designs for the FPGA in Table 3.

A. FIFO

The FIFO policy is the original strategy used in the data cache. By changing the associativity, it's easy to test performance under different situations. However, the differences between 2, 4, and 8-way associativity are so subtle that it doesn't seem worthwhile to double the number. In the case of using this policy, perhaps 2-way is sufficient enough. Another interesting thing is that, in fact, the cache miss rate of 8-way cache is the higher than that of 4-way cache. I think this means belady's anomaly is happening here.

B. LRU

The way I implement LRU policy is to allocate a counter for each cache line. Each counter tracks the number of cycles since its last use and increments when any cache line with a higher number is hit. This is not the only way to implement LRU, nor is it the best, but it is the easiest method.

Also from Fig. 2, we can see that there is no significant performance difference between the LRU and FIFO designs. Judging from the same associativity, only 2-way has fewer cache misses than the FIFO policy, while the miss rates of 4-way and 8-way have increased slightly. However, these differences are so subtle that they have little impact on execution time.

Under the same associativity, the resources used by LRU far exceed FIFO, and grow exponentially as associativity increases. Although it should perform better in general purpose situations, using LRU for the pi programs is not worthwhile.

C. Random

To generate a sequence of random numbers, I used a 32-bit Fibonacci linear-feedback shift register (LFSR) with the feedback term be 0x827F4415 and the seed be 0xAAAAAAAA. For different associativity, the module will take the lowest few bits as the victim cache line number. The idea behind this random number generator is that if you use fewer bits, number 0 will be harder to generate and the ratio won't be even, so 32-bit is chosen. Though the numbers produced by LFSR are not truly random numbers, I think they are acceptable.

In fact, beyond my expectation, the performance of random numbers was the best among the three policies I tested. The execution time of the best-performaning 8-way cache is reduced by 0.8 seconds compared tp caches with the same associativities but using different policies, which is equivalant to an improvement of 2.63%.

The random number generator using LFSR uses fewer resources and the cache is used very efficiently, so it has certain use value here.

I'm not quite sure what actually affects the performance of random replacement cache, but I also tested different feedback and seeds, but it didn't have much impact on performance. Therefore, I think that although there is an optimal solution, LFSR's feedback and seed should not excessively affect cache performance. As long as the random numbers are dispersed enough, performance can be improved.

III. Impact of Replacement Policies

A. The pi program

Different access strategies will result in different miss rates for the program. Since the performance of LRU and FIFO is almost the same, I think the pi program should have the following features:

- The datas are accessed fairly regularly.
- The locality of the program is less obvious. The data will not be accessed multiple times in a short period of time, but new data will be processed.

If neither of these two points is met, the program should have a hot spot that is frequently accessed. In this way, the efficiency of LRU should be better than FIFO.

TABLE IV
THE RESOURCES OF EACH DESIGN.

Design Resource	Slice LUT	Slice Register	F7 Muxes	F8 Muxes	Slice	LUT as Logic	LUT as Memory	BRAM
2W FIFO	1,013	521	40	0	409	997	16	5
4W FIFO	1,720	521	32	0	641	1,704	16	10
8W FIFO	2,322	683	403	23	843	2,106	216	16
2W LRU	1,238	521	40	0	441	1,222	16	5
4W LRU	2,345	908	4	0	723	2,329	16	10
8W LRU	4,583	1,373	752	23	1,470	4,367	216	16
2W RANDOM	1,083	424	40	0	379	1,067	16	5
4W RANDOM	1,713	423	32	0	613	1,697	16	10
8W RANDOM	2,244	614	400	23	804	2,028	216	16

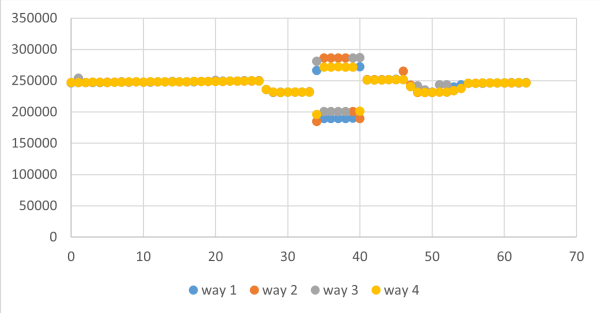

Fig. 3. The number of FIFO cache set access.

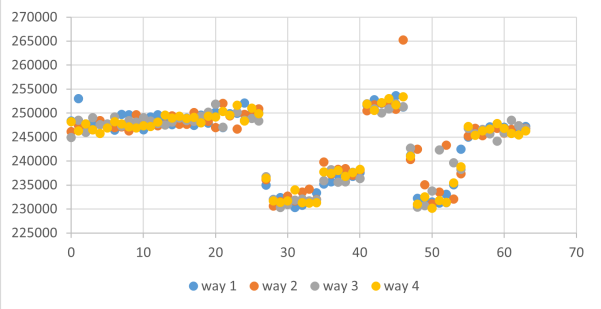

Fig. 4. The number of random cache set access.

B. Cache set access statistics

Athough we know the characteristic of the pi program, low locality and regular access cannot explain why the effect of the random policy is much improved. Therefore, I tried to count the number of times each cache line entry was accessed in different policies. What I tried here are 4-way FIFO and 4-way random policies. The complete results are presented in Fig. 3 and Fig. 4 respectively.

It can be clearly seen from the figure that the chart of FIFO policy is actually more regular, while the access distribution of random policy is more scattered. However, the distribution of the chart is actually determined only based on the replacement policy.

I think the random cache performs better than FIFO, probably because the cache entries in each cache line are used more evenly. Highly regular access behavior also meets a higher hit rate.