HW3 Report
Cache Optimization

Tzu-Han Hsu
dept. Computer Science
National Yang Ming Chiao Tung University
Student ID: 081358

***Abstract*—Experiments with different data cache size, cache ways, and replacement policies are presented in this work. Aiming to find a balanced design between hardware resource and performance. Data cache setup with 4KB size, 8 way cache running FIFO replacement policies reaches the same performance (difference $< 10^{-6}$) than the default setup while saving half of the memory space.**

***Index Terms*—Cache, Replacement policy, RISC-V, Aquila**

I. About the Cache system

Cache systems play an important role in modern computing architectures. The main memory, usually manufactured using DRAM technologies are slow compared to the transistor-built processing units. Direct memory access is so time-consuming which brings down the performance of the overall system. Cache systems acts as a buffer between the processing unit and the main memory, providing high speed memory access whilst utilize the large volume of capacity provided by DRAM at a cheaper cost.

Different from the scratchpad memory which programmer could directly access via a linker script. Cache systems works automatically, abiding the fetching and flushing algorithms programmed by the designer. Programmers view the memory system with cache as a whole, neglecting the complicated data-exchange happening between the two memory parts. However, such design is a double-edged sword, now the overall system performance heavily relies on the cache system for fast memory access. A high-performance cache system becomes a key component of a modern processor. Another downside of cache system is that they are power-consuming and large in chip area, making the system more expensive to manufacture.

II. Profiling the cache behaviour

Most of the profiling work is done in the data cache controller. However, we still need the current programming counter coming from the Aquila core to start and stop data collection. The signal is carefully sent from the execute stage out to the data cache controller. By sampling the program counter, we could obtain the precise clock cycle spent to execute the program, which is a far more precise way to measure performance than CoreMark score.

There are separate counters to collect read and write statistics. Every time a request come in to the cache controller, a request counter would be incremented. Another counter would

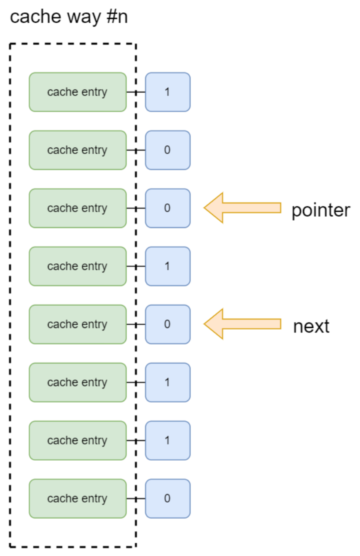

Fig. 1. One chance replacement policy

then we activated, stops when the data request is answered. The counter would be added to a request clock cycle pool to tell on average how many cycles was spent on R/W request. The maximum and minimum R/W cycles is also tracked. The statistics collected altogether would be used to evaluate how the cache system work.

III. Optimizing the data cache

In this homework, we aim to optimize the data cache system by experimenting with different cache system setups. We would try distinct replacement policies on various cache size, the two parameters dominate how a caches system works. Cache ways are also included in our experiment, aim to find the optimal cache system parameter at a fixed size of data cache.

A. Replacement Policy

The default replacement policy is FIFO. A circular counter records the memory address for the next entry to insert, which is its neighbor cache slot. The FIFO replacement policy is built upon the assumption of temporal locality. However, the

TABLE I
DATA CACHE READ & WRITE STATISTICS COLLECTED DURING COREMARK EXECUTION

Dcache size	2 KB			4 KB			8 KB		
Cache way	2	4	8	2	4	8	2	4	8
Replacement policy	FIFO								
Coremark	68.855	68.856	68.973	69.279	69.279	69.279	69.280	69.280	69.280
Run cycles	734,880,530	734,839,447	733,616,207	730,390,283	730,388,935	730,388,451	730,388,357	730,388,378	730,388,363
Read	68,247,807 (77.599%)								
Avg. cycles	1.067%	1.058%	1.038%	1.001%	1.001%	1.001%	0.094%	1.001%	1.001%
Miss rate	0.1356%	0.1170%	0.0768%	0.0025%	0.0024%	0.0023%	0.0002%	0.0023%	0.0023%
Max miss cycles	65%	61%	64%	59%	60%	60%	59%	61%	59%
Min miss cycles	31%	31%	31%	31%	31%	31%	31%	31%	31%
Write	19,702,089 (22.401%)								
Avg. cycles	1.010	1.040	1.047	1.010	1.010	1.010	1.010	1.010	1.010
Miss rate	0.0212%	0.0817%	0.0951%	0.0210%	0.0208%	0.0208%	0.0209%	0.0208%	0.0208%
Max miss cycles	61	63	68	61	61	60	60	60	60
Min miss cycles	31	31	31	31	31	31	31	31	31

Dcache size	2 KB		4 KB		8 KB	
Cache way	4	8	4	8	4	8
Replacement policy	One Chance					
Coremark	68.923	68.132	69.279	69.228	69.279	69.279
Run cycles	734,170,525	742,692,453	730,390,450	730,943,873	730,388,548	730,390,226
Read	68,247,807 (77.599%)					
Avg. cycles	1.046	1.138	1.001	1.007	1.001	1.001
Miss rate	0.0941%	0.2791%	0.0023%	0.0152%	0.0022%	0.0023%
Max miss cycles	61	64	65	61	59	61
Min miss cycles	31	31	31	31	31	31
Write	19,702,089 (22.401%)					
Avg. cycles	1.045	1.162	1.010	1.016	1.010	1.010
Miss rate	0.0916%	0.3293%	0.0208%	0.0334%	0.0208%	0.0208%
Max miss cycles	61	62	59	61	60	61
Min miss cycles	31	31	31	31	31	31

biggest problem of such replacement policy is that every entry stays approximately equally long in the cache. Hot-spot entries would be washed out no matter how popular it is. On the other hand, entries that was used only once would also stay in the cache for a full cache cycle.

The Least Recently Used (LRU) replacement policy would be ideal to avoid common accessed entry be washed out. Albeit, the hardware cost of implementing a counter for each entry is pretty high. An algorithm called one-chance replacement policy is used instead in our experiment and it works as follow: every cache entry has an associated 1-bit status bit, indicating if the entry is accessed during a full replacement cycle. If the entry is ever used, the bit is wiped out and the entry was given another cycle to live (one chance). A new entry would be inserted at the slot where the status bit is empty. Fig.1 shows the components used and how the pointers work in a one chance replacement algorithm. One advantage of such replacement policy is that the design is hardware-level friendly, saving a lot of effort for hardware designers.

B. *Data cache size*

Enlarging the cache size is the most instinctive way to improve the cache system performance. The more data you store in the cache, the higher the hit probability. Note that increasing cache size would not only require more storage space, usually using more SRAM. But also increase the size of other tables and demand more complicated components for the caching algorithm to work. For example, a multiplexer is super expensive to synthesize in an FPGA if the multiplexer grows large in I/O.

Despite all the downsides of a large cache, modern CPUs tend to increase cache size for better performance. For example, the latest Intel® Core™ i9-13900K has 2.875MB L1 cache, 32MB L2 cache and 36MB L3 cache. Showing a large cache is the trend of modern CPU design and is an effective strategy. In our experiment, data cache size of 2KB, 4KB and 8 KB is tested. Aiming to find the relation between Coremark performance and different cache size.

C. *Cache way*

Under a fixed size of memory, different cache ways would also affect the performance of the cache system. With a smaller total cache way, a wider range of memory would be hashed into the same cache way. If the number of cache way is huge, it resembles a fully associative cache. It is reported that a larger cache way would perform better than a smaller on. We would verify the saying in our experiment.

IV. DISCUSSION ON STATISTICS COLLECTED

Table I shows statistics about data cache collected during the execution of Coremark. The upper major column is using default FIFO cache replacement policy. And the bottom major column is running our self-implemented one chance algorithm. We did a grid search on data cache size 2, 4 and 8 KB and cache way with 2, 4 and 8. Note that with cache way of two, cache slots are few to choose from so FIFO and One chance converges to the same policy.

From the analysis result of HW1, we know that load instruction takes about 15% of the overall instructions. Store instruction takes about 4.5%. Coremark overall is not a memory intensive program. The Read and write count records in Table I show how many read write request is sent to the cache controller, thus is a constant across different cache parameter setup.

A. *Cache size does matter*

If there's not enough cache, get yourself some more! Table I sends a clear message that enlarging the cache size would almost immediately increase the data cache performance. Under the same cache way and replacement policies, data cache with greater volume would outperform the smaller sized cache. The results are unsurprising, due to both FIFO and one chance algorithm are stack based algorithm, meaning a smaller sized cache is actually a subset of a larger cache.

Nonetheless, the law of diminishing marginal utility also applies in our experiment. When the data cache doubles from 2 KB to 4 KB, miss rate of a write operation dropped from about 0.1% to 0.0025%. A 40 times lower cache miss rate! If we continue to double the cache size from 4KB to 8KB, the performance between the two are almost identical. Indicating that cache size of 4KB could already meet the needs of CoreMark execution.

From Table I, we learn that a threshold exists in a cache system. And before the cache system reaches the capacity threshold, increasing the cache volume is the most effective way of squeezing extra performance out of the system.

B. *How cache way influence performance*

If the cache size is limited, try different cache ways may bring you surprise. Table 1 shows under identical 2KB cache and FIFO replacement policy, a larger cache way would bring better performance. A 2 cache way system suffers almost double the miss rate than a 8 cache way system when a read request is sent. In the meantime, the miss rate of the write request is almost the same. Similar phenomenon could also be seen at 4KB cache running a FIFO replacement policy.

It would be too naïve of us to believe that a cache system with more cache ways always perform better. 4 way cache performs worse than a 2 way cache when the overall data cache size reaches 8 KB. Furthermore, with one chance replacement policy seems to prefer smaller cache way designs. Statistics from Table 1 shows all 8 way cache has a higher read & write cache miss rate than a 4 way cache.

CoreMark data cache statistics imply that different replacement policies may prefer contrast data cache way designs. We may need to collect more use cases and perform more testbenches to determine which cache way design fits most of the use cases. Overall, CoreMark is only one of the testbench program with less memory access.

C. *Brain of the cache: Replacement policy*

Occam's razor encourages use to keep our solutions simple. The rule seems to apply when we switched from FIFO replacement policy to one chance replacement policy. Aside from size of 2KB and with 4 cache way, all set ups with a more complicated one chance replacement policy is inferior to a simpler FIFO replacement policy cache. Under a 4 KB 8 way cache, FIFO replacement policy suffers from 0.0023% of read instruction miss rate. If one chance algorithm takes over the job, 0.0152% of the read instruction would miss, almost 6 times worse.

It would be too early for us to conclude that one chance replacement policy is inferior than the straightforward FIFO. After all, more complicated system often works better on a more sophisticated environment. It is also astonishing that a worse replacement policy would compromise the cache performance by such a large extend. With an unfit replacement policy, we may even waste all the benefits from enlarging the cache size. In other words, on a more sophisticated and memory intensive real-time use case, picking a smart replacement policy is essential for a high performance cache system.

V. Conclusion and discussion

If the cache hits, the cache would return cached data one clock cycle later. If the cache misses, it would take at least 31 cycles to fetch the data from main memory back to the cache. The data cache takes on average 48 to 50 cycles each cache miss. I'm still not clear why each request would take different response time from the main memory. I presume that it has something to do with DDR memory design.

From the experiment, we understand that increasing the data cache size is the most effective approach of reducing cache miss rate before the threshold is met. Cache way design is related to what replacement algorithm applied. After all, CoreMark is not an ideal target of testing the cache system in my opinion. The memory access in Coremark is not intense thus the statistics could not fully reflect the cache design capabilities.