Topic

    (Programming Project)

    The cache behavior simulation

Goal

    i.   Study and implement a cache policy (NRU replacement policy). [70%]

    ii.   Choose a cache indexing scheme to minimize cache conflict miss. [30%]

Introduction

In hierarchy memory system The cache is a fast but small SRAM (static random access memory) that stores recently using data. It tries to "guess" and store the data that will be used by the processor in the near future. When the processor issues memory instructions, e.g. *lw* or *sw*, the cache could save long DRAM (dynamic random access memory) latency, if the data is found in the cache, so called cache hit. If the data does not in the cache, i.e. cache miss, the processor must wait for an additional cache miss penalty. Since the latency of DRAM is hundred times slower than that of SRAM, the cache hit rate, i.e. the hit count ratio, becomes a critical factor of overall system performance. Thus, for decades, computer architects have developed many cache policies and tried to identify those recently using data that will be accessed in the near future to improve the cache hit rate.

There are several cache related performance policies, including cache allocation policies, cache replacement policies, cache write policies, cache indexing schemes, etc. All of them have significant impact on the cache performance. In this project, we are going to implement a cache behavior simulator targeting specific cache policies. Given a series of memory accesses, a cache configuration (including cache capacity, number of cache associative, etc.). Your program should trace the cache behavior and report the cache performance.

The cache allocation policy defines possible places where a data block could

occupy. For example, direct-map policy defines a single location for each data block. Figure 5.7 from the textbook [1] illustrates a cache with 32 address bits, 4-byte cache block size, 1024 cache sets, and using direct mapping scheme. In this case, there are 2 offset bits; the cache indexing needs another 10 bits for 1024 cache sets; and the other 20 bits are cache tag. Under a cache access, cache hit or miss can be simply derived by comparing the cache tag on the corresponding allocation. If the cache tag matches the one of the address, then it's a cache hit. Otherwise, the cache entry should be replaced by the new data block, and the cache tag information should be updated correspondingly.
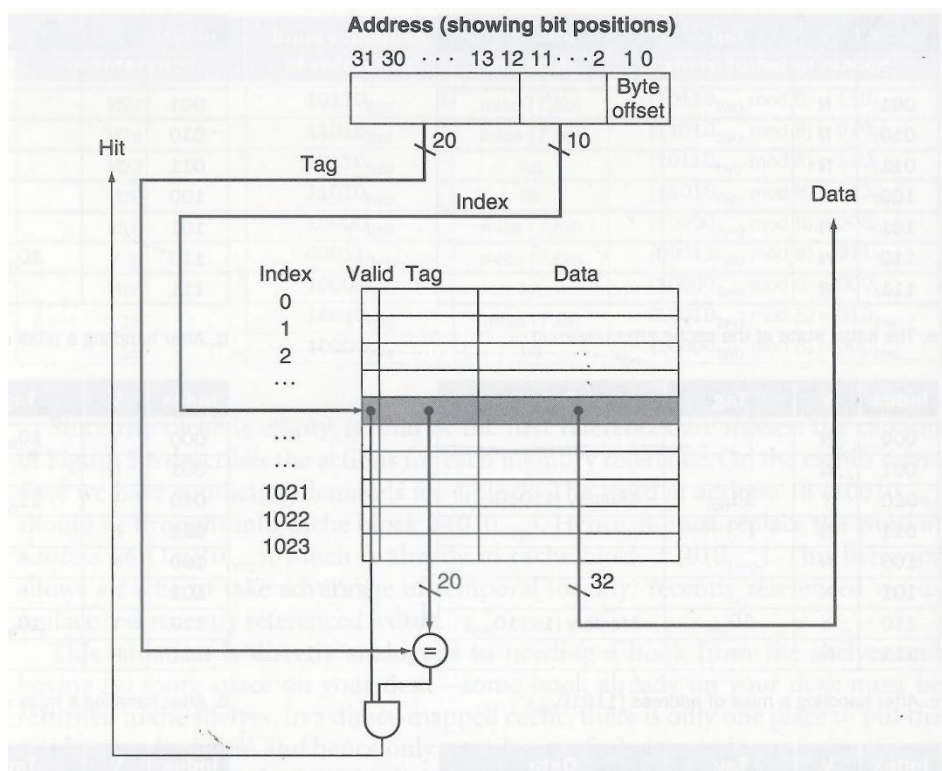
FIGURE 5.7 For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag. This cache holds 1024 words or 4 KB. We assume 32-bit addresses in this chapter. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address. Because the cache has 2^{10} (or 1024) words and a block size of one word, 10 bits are used to index the cache, leaving $32 - 10 - 2 = 20$ bits to be compared against the tag. If the tag and upper 20 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.

Take a 4-way associative cache as another example of cache allocation policy could be. Each data block has 4 possible allocations that can be occupied. Figure 5.17 from the textbook [1] shows a cache with 32 address bits, 4-byte cache block size, 256 cache sets, and 4-way associative. In this scenario, there are 2 bit offset bits, 8 bits indexing bits, and the other 22 bits are cache tag. Apart from the direct map policy, there are four possible allocations for a data block. Each cache tag needs to be compared separately to decide whether it is a cache hit or miss under a cache access. If one of them is identical to the one of the address, then it is a cache hit. Otherwise,
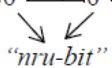
the cache replacement policy will decide a specific location among the possible locations for the new data block to occupy. **In this project, your program should take the number of associativity as an input variable and simulate the corresponding behavior. We may have hidden test cases with different associativity scenario for your program.**
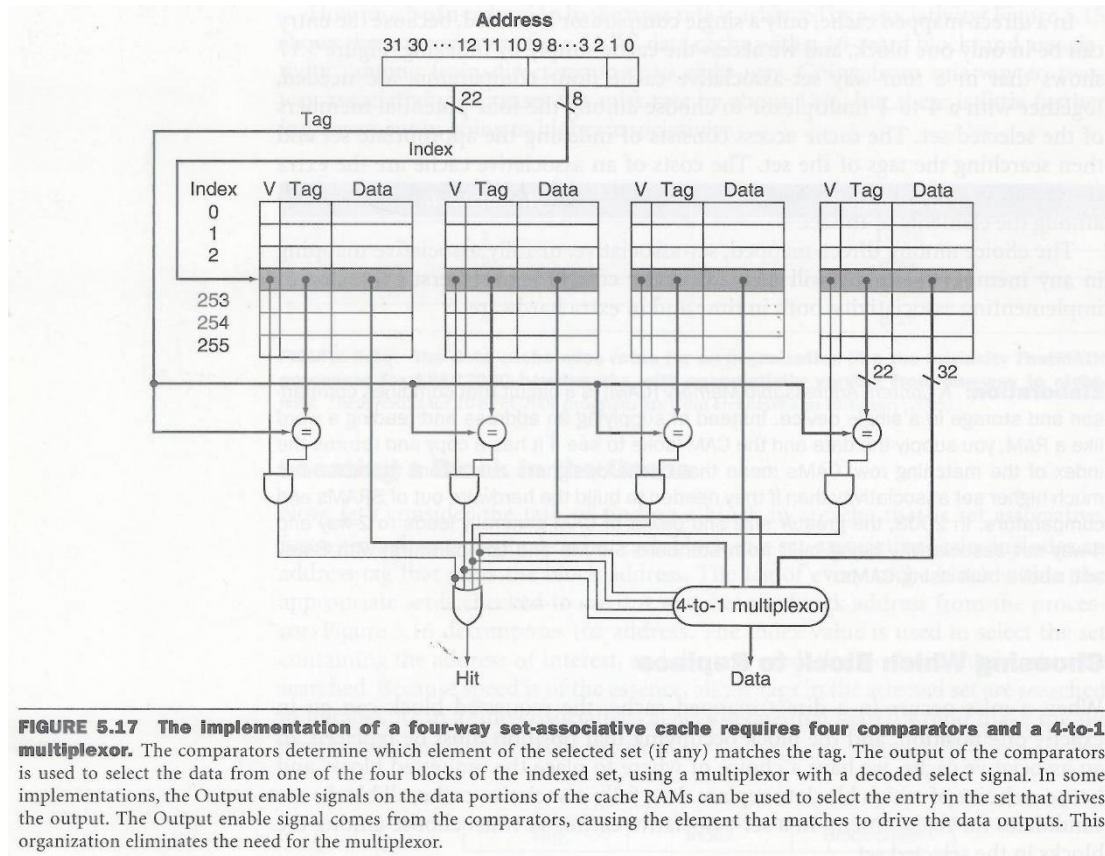
FIGURE 5.17 **The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor.** The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed set, using a multiplexor with a decoded select signal. In some implementations, the Output enable signals on the data portions of the cache RAMs can be used to select the entry in the set that drives the output. The Output enable signal comes from the comparators, causing the element that matches to drive the data outputs. This organization eliminates the need for the multiplexor.

When allocating a data block, it will firstly choose an empty place, i.e. an invalid cache block. If all possible places are occupied by valid data, the cache system has to follow a cache replacement policy to decide which one should be replaced. The cache replacement policy will choose a data block that is least likely to be used in the near future to achieve a higher hit count. Many replacement policies, e.g. the least recently used (LRU) replacement policy, have been proposed seeking to achieve a better hit rate considering hardware complexity and costs [4]. However, implementing the least recently used (LRU) replacement policy in hardware is costly due to the high hardware complexity. One of the most famous alternatives is the not recently used (NRU) replacement policy, which has the advantage of hardware efficiency. You are required to implement an 1-bit NRU replacement policy in this project.

In this project, we employ the single bit not recently used (NRU) replacement policy [3], in which one NRU-bit is associated with each cache block. At the beginning, all NRU-bit are set to 1. When the cache block is filled or re-referenced, the NRU-bit of that block is set to 0. As to cache block replacement, the evicted cache block is selected while its NRU-bit is 1. If there are multiple cache blocks whose NRU-bit are 1, the block that is closest to the head will be selected for eviction. If there is no cache blocks whose NRU-bit is 1, all cache blocks in the same cache set will set their NRU-bit to 1 at the same time. Then, the evicted cache block can be selected as mentioned before. The following figure shows an example of the single bit NRU replacement policy. The NRU has been proven to be very effective considering the cost and performance.

Cache Hit:
    (i) set nru-bit of block to '0'

Cache Miss:
    (i) search for first '1' from left
    (ii) if '1' found go to step (v)
    (iii) set all nru-bits to '1'
    (iv) goto step (i)
    (v) replace block and set nru-bit to '0'

As mentioned previously, cache holds a subset duplicated data from the main memory. After we update a data in the cache by a *sw* instruction, the data will no longer be the same as the one in the main memory, i.e. data consistency issue. The cache write policies target this issue. It decides when to write the new data back to main memory. There are two basic cache write policies, namely write-through and write-back. The write-through policies always updates the new data all the way through cache and memory to guarantee that all data in cache and memory are consistent. The write-back policy, on the other hand, only updates the main memory when the data block is replaced (or evicted). To do so, the write-back policy requires additional bits, called dirty bits, to indicate whether the data block has been updated or not. **Since we target a single level cache behavior in this project, your program does not need to consider the cache write policy.**

Lastly, the cache indexing scheme decides a cache set where the data should be stored. If different data blocks accessed frequently are mapped into the same location, those data blocks will keep kick others out from cache, and results in lots of cache misses, i.e. conflict misses. In this scenario, the system performance will be seriously degraded. For general cases (Figure 5.7 and Figure 5.17 from the textbook [1]), designers usually choose the least significant bits (LSBs) of block address (skipping offset bits) to decide the cache set. However, some applications may have potential performance improvement under other indexing schemes [2]. **For this project, you are required to implement the LSB indexing scheme as baseline implementation. You will get the basic score for the baseline implementation (70%+). In addition, the second topic of this project is to design a cache indexing scheme with the minimal cache miss count from the given addressing sequence.**

Problem Definition

In this project, we consider byte addresses and byte addressable cache systems. Give a cache with the M-bit address bits, B-byte block size, E cache sets, and using A-way set associativity. There are $O = \log_2 B$ bits offset, and we need to select $K = \log_2 E$ bits among upper (M-O) address bits for indexing the cache. There are totally $\binom{M-O}{K}$ possible valid indexing schemes. Your job is to (i) implement the NRU replacement policy and the LSB indexing scheme and (ii) find a valid indexing scheme with minimal cache misses.

Example 1 – (direct-map or 1-way associative)

Consider the address is 8 bits ($a_7a_6a_5a_4a_3a_2a_1a_0$), and the cache has 4-byte block size, 8 cache sets, and using direct map scheme. Then, the lower $\log_2 4 = 2$ bits are offset bits that cannot be selected as the indexing bits, and we need $\log_2 8 = 3$ bits for indexing 8 sets (#0~#7).. Thus, in this case, there are $\binom{8-2}{3} = 20$ possible indexing schemes.

Access Example:

Reference Address	Indexing by ($a_4a_3a_2$) (LSB)	Allocation
00010000	000**10**000	#4
00101100	001**011**00	#3

The reference sequence is as follows:

Reference sequence	Indexing by ($a_4a_3a_2$) (LSB)		Indexing by ($a_7a_6a_5$)	
($a_7a_6a_5a_4a_3a_2a_1a_0$)	Allocation	Cache hit/miss	Allocation	Cache hit/miss
00101100	#3	miss	#1	miss
00100000	#0	miss	#1	miss
00101100	#3	hit	#1	miss
00100000	#0	hit	#1	miss
Total cache miss count	2		4	

Example 2 – (2-way associative)

Assume the address is 8 bits ($a_7a_6a_5a_4a_3a_2a_1a_0$), the cache has 4-byte block size, 4 cache sets with, and using 2-way associative policy. There are $\log_2 4 = 2$ bits for offset, and we need $\log_2 4 = 2$ bits for indexing 4 sets (#0~#3). Hence, there are $\binom{8-2}{2} = 15$ possible indexing schemes in this case.

The cache organization will be:

Set number		way0			way1
#0	tag	data		tag	data
#1	tag	data		tag	data
#2	tag	data		tag	data
#3	tag	data		tag	data

Access Example:

Reference Address	Indexing by (a_3a_2) (LSB)	Allocation
00010000	00010000	#0
00101100	00101100	#3

The reference sequence is as follows:

Reference sequence ($a_7a_6a_5a_4a_3a_2a_1a_0$)	Indexing by (a_3a_2) (LSB)		Indexing by (a_5a_4)	
	Allocation	Cache hti/miss	Allocation	Cache hit/miss
00000000	#0 (way0)	miss	#0 (way0)	miss
00010000	#0 (way1)	miss	#1 (way0)	miss
00100000	#0 (way0)	miss	#2 (way0)	miss
00000000	#0 (way1)	miss	#0 (way0)	hit
00101100	#3 (way0)	miss	#2 (way1)	miss
00000000	#0 (way1)	hit	#0 (way0)	hit
00101100	#3 (way0)	hit	#2 (way1)	hit
Total cache miss count	5		4	

Input/output file example

Following input/output file format is a requirement in this project. You get no point without any excuse for any format violation.

In this project, the index of the first bit always has the biggest indexing number. Meanwhile, the index of the last bit is always 0. For example, the last address in the following file, reference2.lst, is 00101100, which indicates that the bit indexes 7, 6, 4, 1, 0 (a_7, a_6, a_4, a_1, a_0) are 0 and the bit indexes 5, 3, 2 (a_5, a_3, a_2) are 1.

Input file example: cache2.org

```
Address_bits: 8
Block_size: 4
Cache_sets: 4
Associativity: 2
```

Input file example: reference2.lst

```
.benchmark testcase1
00000000
00010000
00100000
00000000
00101100
00000000
00101100
.end
```

As shown previously (the first part of this project), the indexing bits can simply use the LSB, i.e. the lower bits. In the following case, the least significant bits (LSBs) are used as the indexing bits, i.e. 3 and 2. The final miss count is 5.

Output file example: index.rpt

```
Address bits: 8

Block size: 4

Cache sets: 4

Associativity: 2


Offset bit count: 2

Indexing bit count: 2

Indexing bits: 3 2


.benchmark testcase1

00000000 miss

00010000 miss

00100000 miss

00000000 miss

00101100 miss

00000000 hit

00101100 hit

.end


Total cache miss count: 5
```

Also (the second part of this project), the indexing bits can be carefully selected to achieve a lower miss count. In this case, the bits 5 and 4 are used as the indexing bits. The final miss count is reduced to 4.

Output file example: index.rpt

```
Address bits: 8
Block size: 4
Cache sets: 4
Associativity: 2

Offset bit count: 2
Indexing bit count: 2
Indexing bits: 5 4

.benchmark testcase1
00000000 miss
00010000 miss
00100000 miss
00000000 hit
00101100 miss
00000000 hit
00101100 hit
.end

Total cache miss count: 4
```

Requirements

- Your program should follow the requirements.

  - Implemented in C/C++ programming language and compliable to g++ compiler. A binary executable, named **project,** should be generated.

  - **Pass the I/O filenames from command**, as follows:
        ```
        $ ./project cache1.org reference1.lst index.rpt
        ```
    where project is your binary code, cache1.org, reference1.lst are input files, index.rpt is the output file name. (The input files are ASCII format.)
    Notice that you will get no point if not follow the IO format requirement!!

  - Output the total cache miss count into the output file, i.e. index.rpt.

  - Terminate within a minute.

- Report; including algorithm flow-chart, description of data structure, etc.

- Demonstration. The nthucad server will be used as the demonstration platform. (ssh://nthucad.cs.nthu.edu.tw port 22)

Grading

- Output file format – The output file should be generated in output file format.

- Correctness – The number of indexing bits and the count of cache miss should be correct.

- Cache miss count – The count of cache miss is the main performance criterion of the cache system. Over 70% grading benchmarks has the best solution on LSB indexing scheme.

- Run Time – Your program should terminate within a minute.

- Demonstration – You have to daemon your code and the flow of your algorithm to TAs clearly.

Notice that this project plays an important role on the final score.
You are not going to pass the course if not doing this project!

Related Work

For the second part, a previous work, "Zero Cost Indexing for Improved Processor Cache Performance ", is proposed by Tony Givargis [2].

References

[1] David A. Patterson and John L. Hennessy. 2008. Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design) (4th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[2] Tony Givargis. 2006. Zero cost indexing for improved processor cache performance. ACM Trans. Des. Autom. Electron. Syst. 11, 1 (January 2006), 3-25. DOI=http://dx.doi.org/10.1145/1124713.1124715

[3] Code::Blocks, http://www.codeblocks.org/

[4] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In Proceedings of the 37th annual international symposium on Computer architecture (ISCA '10). ACM, New York, NY, USA, 60-71. DOI: https://doi.org/10.1145/1815961.1815971

[5] Wikipedia, cache replacement policies – least recently used (LRU) https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used_ (LRU)