國立清華大學資訊工程學系

Computer Architecture

103 學年度下學期

Final Project

(Due: 23:59 PM, June 25, 2015)

Topic

(Programming Project)

The indexing algorithm design for processor cache

Goal

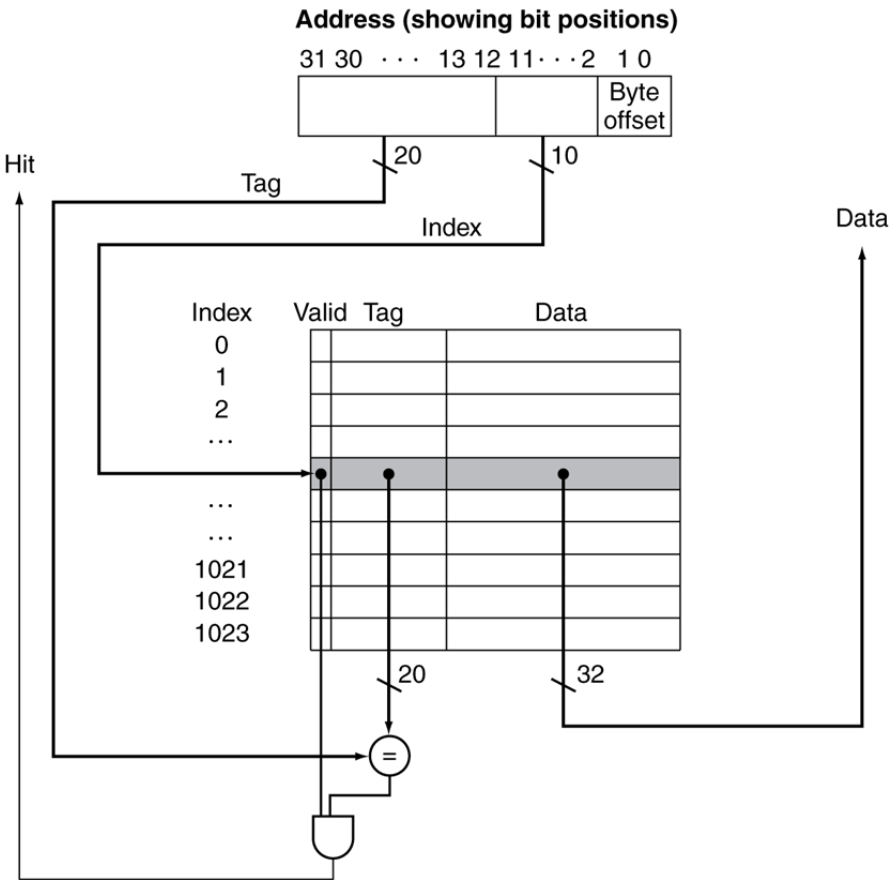
i. Study and implement least recently used (LRU) replacement policy.

ii. Design a cache indexing scheme to minimize cache conflict miss.

Introduction

In hierarchy memory system, the small memory, cache, is used to keep data temporarily for increasing the system performance. If the data is in cache, processor can get the data with high accessing speed. If the data is not in the cache, called cache miss, processor will read data from main memory. Accessing data form main memory is slower than from cache.

In set-associative or direct mapping scheme, when multiple frequently used data blocks compete for a same cache location, those data blocks will keep kick others out from cache, and results in lots of cache miss, this kind of cache misses is called the conflict misses.



The above figure shows the direct mapping scheme with 1024 cache sets. In direct mapping scheme, each set has one cache block. Byte offset takes two bits. The cache indexing needs 10 bits of 32 address bits for indexing 1024 sets. The other 20 bits are cache tag.

The cache indexing scheme decides the cache set in which a data to be stored. If different data accessed frequently are mapped into a same location, the system performance will be degraded due to cache conflict misses. That is, the design target of the cache indexing scheme is to reduce cache conflict misses.

Moreover, in some SOC design, the application software running on the embedded processor is always dedicated, for example, the Fourier transforms in digital signal processing. It is possible to analyze the memory access behavior of the software to optimize the cache indexing to reduce the cache miss. The objective of this project is to design the cache indexing scheme with the minimal cache miss count from the given addressing sequence.

Problem Definition

Give a cache with *E* sets and *A*-way set associativity. And, the addressing bus is *M*-bit. Offset is *F*-bit (Note that offset bits are always in the rightest side of address bits). We need to select $K = \log_2 E$ bits among all address bits except offset bits for indexing the cache.

There are totally $\binom{M-F}{K}$ possible valid solutions. Your job is to find a valid solution with minimal cache misses under LRU replacement policy.

Example 1 – (1-way associative)

If the addressing bus is 6 bits ($a_5\ a_4\ a_3\ a_2\ a_1\ a_0$) and offset is 0 and the cache has 8 Sets and uses direct map scheme, we need $\log_2 8 = 3$ bits for indexing 8 sets (#0~#7). There are $\binom{6}{3} = 20$ possible solutions.

Access Example:

Reference Address	Indexing by ($a_2\ a_1\ a_0$)	Cache Location
000100	000100	#4
001011	001011	#3

The reference sequence is as follows:

Reference sequence ($a_5\ a_4\ a_3\ a_2\ a_1\ a_0$)	Indexing by ($a_5\ a_4\ a_3$)		Indexing by ($a_2\ a_1\ a_0$)	
	set	status	set	status
001011	#3	miss	#1	miss
001000	#0	miss	#1	miss
001011	#3	hit	#1	miss
001000	#0	hit	#1	miss
Cache miss	2		4	

Example 2 – (2-way associative)

If the addressing bus is 6 bits (a_5 a_4 a_3 a_2 a_1 a_0) and offset is 2 and the cache has 4 sets and uses 2-way associative scheme, we need $\log_2 4 = 2$ bits for indexing 4 sets (#0~#3). There are $\binom{6}{2} = 15$ possible solutions.

The cache organization will be:

Sets	Block 1		Block 2	
#0	tag	data	tag	data
#1	tag	data	tag	data
#2	tag	data	tag	data
#3	tag	data	tag	data

The reference sequence is as follows:

Reference sequence (a_5 a_4 a_3 a_2 a_1 a_0)	Indexing by (a_5 a_4)		Indexing by (a_3 a_2)	
	set (#block)	status	set (#block)	status
000000	#0 (b1)	miss	#0 (b1)	miss
000100	#0 (b2)	miss	#1 (b1)	miss
001000	#0 (b1)	miss	#2 (b1)	miss
000000	#0 (b2)	miss	#0 (b1)	hit
111011	#3 (b1)	miss	#2 (b2)	miss
000000	#0 (b2)	hit	#0 (b1)	hit
111011	#3 (b1)	hit	#2 (b2)	hit
Cache miss	5		4	

Related Work

A previous work, “Zero Cost Indexing for Improved Processor Cache Performance “ is proposed by Tony Givargis [1]. To refer to this algorithm in your project is welcome, or your new idea is encouraged and will be given additional bonus.

Requirements

- Using C/C++ language. (Your program will be recompiled by g++ or code::block to check the correctness, so please make sure your program is compliable to g++ or code::block++.)
- Your program should pass the argument from command, like :
  "./arch_final cache.org reference.lst"
  (If you don't know what it is, see more in the last page.)
- Final cache indexing and cache miss count (index.rpt).
- Report; including algorithm description, flow-chart of algorithm, summary of result, and discussion.
- Demonstration. The time will be shown on class website.
- Your program should finish in one minute.

Input File Format

cache.org

```
Addressing_Bus : 6
Sets : 4
Associativity : 2
Offset : 2
```

reference.lst

```
.benchmark testcase1
000000
000100
001000
000000
001011
000000
001011
.end
```

Notice that, the index of the LSB is always 0. Meanwhile, the index of the MSB always has the biggest indexing number. For example, the last address in above "reference.lst" is 001011, which means bit index 3, 1, 0 (a_3, a_1, a_0) are 1, and bit index 5, 4, 2 (a_5, a_4, a_2) are 0.

Output File Format

index.rpt

```
Student ID: 991234
Addressing Bus: 6
Sets: 4
Associativity: 2
Offset: 2
Indexing bits count: 2
Indexing bits: 3 2
Total cache miss: 3

.benchmark testcase1
000000 miss
000100 miss
001000  miss
000000  hit
001011 hit
000000  hit
001011 hit
.end
```

Please note that, indexing bits should be printed from MSB to LSB (from big to small).

Grading

- Oral report – You should describe the flow of your algorithm clearly.
- 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.
- Run Time – Your program should not run over a minute.
- Demonstration – You have to daemon your code to TAs.
- Creativity – If your new idea is good, than you get higher grade.

References

[1] Tony Givargis "Zero Cost Indexing for Improved Processor Cache Performance",*ACM Transaction on Design Automation of Electronic Systems, Vol. 11, No. 1, pp.3-25, 2006.*

[2] Code::Blocks http://www.codeblocks.org/

[3] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., Joel Emer. "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)". *ISCA 2010.*

Appendix: argc, argv

In order to pass the arguments from command, your program would be like this:

```
int main ( int argc, char *argv[] ) {

     // code

     return 0;

}
```

The arguments `argc` and `argv` of `main` is used as a way to send arguments to a program. The program receives the number of arguments in `argc` (argument count) and the vector of arguments in `argv` (argument vector).
For example, the executable file is named `”myproject”`. If we type `./arch_final in1 in2` in terminal, the program `arch_final` will executed with some command line parameters. This would result in :

`argc = 3 , argv[] ={ ”arch_final”, ”in1”, ”in2” }.`