Notes on Caches
Andreas Moshovos
Spring 2007

Please refer to the book for a complete treatment of caches. Here we review a few examples that explain some of the underlying concepts.

Cache Indexing

A cache is organized as a two-dimensional array of blocks. The rows are called *sets*, and the columns are called *ways*. At the one extreme there is only one set. This is a fully-associative cache. In this case the number of ways equals the number of blocks. At the other extreme there is a single column. This is a direct-mapped cache. In this case, the number of sets equals the number of blocks. Any configuration in between is called an N-way set-associative cache where N is the number of ways. Notice that the number of sets is not specified. We can derived it given the total cache capacity.

Given an address from the CPU we need to index the cache. By this we mean, selecting the set in which the address may be cached. The common indexing scheme partitions the incoming address into three fields. Starting from the MSB these are the TAG, SET and OFFSET:

TAG	SET	OFFSET

<- ------------------------------ Address bits ---------------------------- ->

For a fully-associative cache the set field does not exist. This is because there is only one set. For the direct-mapped cache, if the set width is S bits it holds that 2^S = #Blocks.

Let’s go over a few examples.

Example #1

1MB cache, with 64B blocks, 4-way Set-Associative, write-through with LRU replacement. Assume a 36-bit byte-addressable address space.

In this case, there are 2^36 different byte addresses.

It’s good practice to try to write the parameters as a power of two where possible. So we have Capacity = 1MB = 2^20 bytes, Block Size = 64B = 2^6, associativity = 4 = 2^2. Notice that the capacity refers to the total number of data bytes that the cache can store. It does not include the overhead bits required by the tags, valid bits and LRU bits.

From the size of the blocks we can immediately derive the width of the offset field. It holds that offset_width = lg(Block Size), where lg is the logarithm base two. So, in this case OFFw = 6 bits.

To find the width of the SET field we need to determine the number of rows. To do that we need to determine the number of blocks. We are given that this is a 4-way set-associative cache, hence each set has four blocks.

It holds:

#Blocks = Capacity / BlockSize = 2^20 / 2^6 = 2^14. There are 2^14 blocks.

#Sets = #Blocks / #ways = 2^14 / 2^2 = 2^12. There are 2^12 sets, hence the SET field is 12 bits wide.

So the indexing is done as follows:

<- --- 36 - 12 - 6 --- -> <- ------- 12 -------- -> <- -------- 6 -------- ->

TAG	SET	OFFSET

To access the cache we first use the 12 SET bits to select one of the 2^12 sets. Then we compare the TAG field with the TAG fields of the four blocks that belong to the set. If a match is found, we finally use the OFFSET field to select the appropriate byte out of the 64 stored in the block.

Example #2

36KB, 9-way set-associative cache, with 8-byte blocks, write-back with LRU replacement and a 2^24 address space (byte addressable).

We immediately know that OFF width = 3 bits (from the block size).

#blocks = Capacity / block size = 9 * 4K / 2^3 = 9 * 2^12 / 2^3 = 9 x 2^9.

#sets = #blocks / #ways = 9 x 2^9 / 9 = 2^9.

So the SET field is 9 bits wide.

It follows that that TAG field is 24 - 9 - 3 bits wide.

Example #3

If we have 7KB of data cache and assuming 128B blocks what can the associativity be?

Let's first see how many blocks are there:

#blocks = Capacity / block size = 7 x 2^10 / 2^7 = 7 x 2^3.

We can organize the blocks into one set. This would be a Fully-Associative cache.

Can we organize the blocks into two sets? Each set would have 7 x 2^3/2 = 7x2^2 blocks. Yes we can. This is a 28-way set-associative cache.

Can we organize the blocks into four sets? Notice that the number of sets *has* to be a power of two since we assume that we use a portion of the address to index the sets directly (note that there are advanced designs that use different functions but we will not discuss those in this course). So, if we had four sets, each set would have 7x2^3/2^2 = 7x2 blocks. Yes, we can. This is a 14-way set-associative cache.

How about 8 sets? Each set would have 7 blocks. This is a 7-way set-associative cache.

How about 16 sets? Each set would have 4.5 blocks. This is not possible.

How about 32? Each set would have have 1 3/4 blocks. This is not possible.

How about 64? No, there are only 56 blocks.

Storage Requirements

How many storage bits are required to implement a 256KB cache, with 16B blocks, that is 4-way set-associative, uses write-back, LRU replacement and assuming a 2^36 byte-addressable address space?

Bits are required for:

1. The data
2. The tags
3. The valid bits
4. The dirty bits
5. The LRU bits

Data bits = 256K x 8 = 2^18 x 2^3 = 2^21 bits = 2Mbits

To calculate the rest we need to figure out how the cache is indexed.

#blocks = 2^18 / 2^4 = 2^14
#sets = 2^14 / 2^2 = 2^12

So, the TAG field is 36-12-4 = 18 bits wide.

Tag bits = #blocks x TAG field width = 2^14 * 18 bits

There is one valid bit and one dirty bit per blocks

Valid Bits = 2^14 and Dirty Bits = 2^14

The LRU bits are associated with the set. They tell us in which order the blocks within the set were accessed. As we explained we need at least lg(ways!) bits to encode this information. So:

LRUbits = 2^12 x lg(4!)

Determining Hits and Misses

Let us assume that a program accesses the following addresses:

$200, $204, $208, $20C, $2F4, $2F0, $200, $204, $218, $21C, $24C, $2F4, repeated two times

Given a cache with 8 four byte blocks determine how many hits and misses will occur.

#1 Direct-Mapped Cache

Since the block size is four bytes the lower two bits are the offset within the block. These are show in italics.
Since there are 8 blocks, and this is a direct-mapped cache, there are 8 sets. Hence, the next 3 bits are the set. These are underlined.
The rest are the TAG show in normal font.

ADDRESS IN HEX	ADDRESS IN BINARY	HIT/MISS	Action
$200	0010 000<u>0</u> <u>00</u>*00*	Miss	Bring Block 0010 0000 00XX from memory into set 000
$204	0010 000<u>0</u> <u>01</u>*00*	Miss	Bring block 0010 0000 01XX from memory into set 001
$208	0010 000<u>0</u> <u>10</u>*00*	Miss	Bring block 0010 0000 10XX from memory into set 010
$20C	0010 000<u>0</u> <u>11</u>*00*	Miss	Bring block 0010 0000 11XX from memory into set 011
$2F4	0010 111<u>1</u> <u>01</u>*00*	Miss	Bring block 0010 1111 01XX from memory into set 101
$2F0	0010 111<u>1</u> <u>00</u>*00*	Miss	Bring block 0010 1111 00XX from memory into set 100
$200	0010 000<u>0</u> <u>00</u>*00*	Hit	Set 000
$204	0010 000<u>0</u> <u>01</u>*00*	Hit	Set 001
$218	0010 000<u>1</u> <u>10</u>*00*	Miss	Bring block 0010 0001 10XX from memory into set 110
$21C	0010 000<u>1</u> <u>11</u>*00*	Miss	Bring block 0010 0001 11XX from memory into set 111
$24C	0010 010<u>0</u> <u>11</u>*00*	Miss	Bring block 0010 0100 11XX from memory into set 011 Replace block 0010 0000 11XX
$2F4	0010 111<u>1</u> <u>01</u>*00*	Hit	Set 101
$200	0010 000<u>0</u> <u>00</u>*00*	Hit	Set 000
$204	0010 000<u>0</u> <u>01</u>*00*	Hit	Set 001

$208	0010 0000 1000	Hit	Set 010
$20C	0010 0000 1100	Miss	Bring block 0010 0000 11XX from memory into set 011 Replace block 0010 0100 11XX
$2F4	0010 1111 0100	Hit	Set 101
$2F0	0010 1111 0000	Hit	Set 100
$200	0010 0000 0000	Hit	Set 000
$204	0010 0000 0100	Hit	Set 001
$218	0010 0001 1000	Hit	Set 110
$21C	0010 0001 1100	Hit	Set 111
$24C	0010 0100 1100	Miss	Bring block 0010 0100 11XX from memory into set 011 Replace block 0010 0000 11XX
$2F4	0010 1111 0100	Hit	Set 101

#2 Fully-Associative Cache

In this case there is only one set with 8 blocks.

ADDRESS IN HEX	ADDRESS IN BINARY	LRU Order	HIT/MISS	Action
$200	0010 0000 0000	$200	Miss	Bring Block 0010 0000 00XX from memory
$204	0010 0000 0100	$204, $200	Miss	Bring block 0010 0000 01XX from memory
$208	0010 0000 1000	$208, $204, $200	Miss	Bring block 0010 0000 10XX from memory
$20C	0010 0000 1100	$20C, $208, $204, $200	Miss	Bring block 0010 0000 11XX from memory
$2F4	0010 1111 0100	$2F4, $20C, $208, $204, $200	Miss	Bring block 0010 1111 01XX from memory
$2F0	0010 1111 0000	$2F0, $2F4, $20C, $208, $204, $200	Miss	Bring block 0010 1111 00XX from memory
$200	0010 0000 0000	$200, $2F0, $2F4, $20C, $208, $204	Hit	
$204	0010 0000 0100	$204, $200, $2F0, $2F4, $20C, $208	Hit	
$218	0010 0001 1000	$218, $204, $200, $2F0, $2F4, $20C, $208	Miss	Bring block 0010 0001 10XX from memory
$21C	0010 0001 1100	$21C, $218, $204, $200, $2F0, $2F4, $20C, $208	Miss	Bring block 0010 0001 11XX from memory
$24C	0010 0100 1100	$24C, $21C, $218, $204, $200, $2F0, $2F4, $20C	Miss	Bring block 0010 0100 11XX from memory Replace block containing $208
$2F4	0010 1111 0100	$2F4, $24C, $21C, $218, $204, $200, $2F0, $20C	Hit	
$200	0010 0000 0000	$200, $2F4, $24C, $21C, $218, $204, $2F0, $20C	Hit	
$204	0010 0000 0100	$204, $200, $2F4, $24C, $21C, $218, $2F0, $20C	Hit	
$208	0010 0000 1000	$208, $204, $200, $2F4, $24C, $21C, $218, $2F0	Miss	Bring block 0010 0000 10XX from memory Replace block containing $20C
$20C	0010 0000 1100	$20C, $208, $204, $200, $2F4, $24C, $21C, $218	Miss	Bring block 0010 0000 11XX from memory Replace block containing $2F0
$2F4			Hit	

	0010 1111 01*00*	$2F4, $20C, $208, $204, $200, $24C, $21C, $218		
$2F0	0010 1111 00*00*	$2F0, $2F4, $20C, $208, $204, $200, $24C, $21C	Miss	Bring block 0010 1111 01XX from memory Replace block containing $218
$200	0010 0000 00*00*	$200, $2F0, $2F4, $20C, $208, $204, $24C, $21C	Hit	
$204	0010 0000 01*00*	$204, $200, $2F0, $2F4, $20C, $208, $24C, $21C	Hit	
$218	0010 0001 10*00*	$218, $204, $200, $2F0, $2F4, $20C, $208, $24C	Miss	Bring block 0010 0001 10XX from memory Replace block containing $21C
$21C	0010 0001 11*00*	$21C, $218, $204, $200, $2F0, $2F4, $20C, $208	Miss	Bring block 0010 0001 11XX from memory Replace block containing $24C
$24C	0010 0100 11*00*	$24C, $21C, $218, $204, $200, $2F0, $2F4, $20C	Miss	Bring block 0010 0100 11XX from memory Replace block containing $208
$2F4	0010 1111 01*00*	$2F4, $24C, $21C, $218, $204, $200, $2F0, $2F4	Miss	Bring block 0010 0100 11XX from memory Replace block contaning $20C