Caches

CS61C Fall 2017

Contents

1 Why we want caches

1.1 Locality & Partitioning memory for caching

Accessing memory is costly relative to accessing data from registers, with the former taking over 10x longer than the latter. However, as you've seen in MIPS programs, we can't fit all of the data we need for a program in just the registers, so we need some way of making memory accesses efficient. As it turns out, the data consumed by programs exhibits both temporal and spacial locality—after accessing a piece of data at memory address M, programs will often re-use that same data in a given time period and often use data at addresses adjacent to M. The main idea behind caching is then to take advantage of these localities by 1) saving data that has just been accessed from memory 2) prefetching data adjacent to recently accessed memory. In the example below, it would be useful to cache the value of k as we use it in every iteration of the loop, as well as to prefetch elements of A as we will eventually need to access all elements of A.

```
int bar(int* A, int k) {
  int i = 0;
  for (i = 0; i < A_LEN; i++)
    A[i] += k;
```

Figure 1: A program with locality.

In other words, when a program takes the long walk to memory to access data, it will not only bring back the data we wanted, but also data around it, and store both into our cache. This way, instead of going all the way back to memory, the CPU can check the cache for the desired data first, and hopefully save precious picoseconds (which builds up over thousands of memory accesses)! Physically, a cache is a hardware component larger/slower than a register but smaller/faster than memory.

Ideally, computers would be able to see into the future, saving only the data we'll actually reuse later and prefetching just the right amount of adjacent data. Unfortunately, neither humans nor computers can see into the future, so each computer agrees on a number of bytes (i.e. unit of size) that it will take from memory whenever memory is accessed. This number of bytes is the **cache block size**, or the number of bytes we can fit into one block of our cache. Memory is then divided up into chunks, of the same size as the cache block size, such that, when accessing any memory address, the computer can quickly find what other bytes of data to also take.

	...				
chunk starting at 0xABCDEF20	'C'	'A'	'S'	'H'	0xABCDEF2C
	'$	'$'	'$'	'\0'	0xABCDEF28
	BF	FF	FF	CE	0xABCDEF24
	BF	FF	FF	CC	0xABCDEF20
chunk starting at 0xABCDEF10	'C	'A'	'C'	'H'	0xABCDEF1C
	'E	'S'	'!'	'\0'	0xABCDEF18
	00	00	00	EE	0xABCDEF14
	00	00	00	FF	0xABCDEF10
chunk starting at 0xABCDEF00	'6	'1'	'C'	'\0'	0xABCDEF0C
	'I'	'L'	'U'	'V'	0xABCDEF08
	00	00	00	02	0xABCDEF04
	00	00	00	01	0xABCDEF00
	...				

Figure 2: Dividing memory into cache block sized chunks.

For example, say we had a 32 bit machine and a cache block size of 16 B (as shown above). Our 2^{32} B of memory would be divided up into 16 B chunks, such that if we were to access address 0xABCDEF00, then we would take data at 0xABCDEF00, 0xABCDEF01, 0xABCDEF02, ..., 0xABCDEF0F. Accessing any address in the range 0xABCDEF00, 0xABCDEF01, 0xABCDEF02, ..., 0xABCDEF0F, would then take *all* of the data in the range 0xABCDEF00, 0xABCDEF01, 0xABCDEF02, ..., 0xABCDEF0F.

2 Cache Specifications & Organization

In general, a cache is specified by its total size, T, its cache block size, B, and its associativity N. The number of entries or rows in a cache, E, can then be computed as:

$$\frac{T \text{ bytes}}{B \text{ bytes / block}} \frac{1}{N \text{ blocks / row}} = \frac{T}{BN} = E \; rows$$

.

Caches are then organized into rows of cache blocks, where each cache block of data is paired with bits of metadata identifying and describing the block of data. Visually, you can imagine a cache as just a table, depicted below (this cache has associativity N = 1).

		+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15
	Tag	Offset															
Index = 0	0xABCDEF	00	00	00	01	00	00	00	02	'I'	'L'	'U'	'V'	'6'	'1'	'C'	\0
Index = 1																	
Index = 2																	
Index = 3																	
	...	...															
Index = 14																	
Index = 15																	

Figure 3: An example cache.

Each row has an **index**, from 0 to $E - 1$. Within a cache block of data, each byte of data has an **offset** indicating which byte corresponds to which address within the block (e.g. if a cache block contained the data in the range 0xABCDEF00, 0xABCDEF01, 0xABCDEF02, ..., 0xABCDEF0F, then the byte with offset = 0 corresponds to the data at 0xABCDEF00, the byte with offset = 1 corresponds to the data at 0xABCDEF01, and so on and so forth). Each block (in this case each row, since associativity is 1), has a **tag**, as indicated by the tag column below. The tag can be thought of as a unique identifier for a block of data, so that you can tell which memory addresses that block of data corresponds to.

To facilitate breaking down where data for a memory address should be stored in the cache, memory addresses then have corresponding **tag fields**, **index fields**, and **offset fields**, with the number of bits for each field being calculated as:

$$I = \#\text{ of index field bits} = \log_2(E)$$

$$O = \#\text{ of offset field bits} = \log_2(B)$$

$$T = \#\text{ of tag field bits} = \#\text{ of address bits} - I - O$$

These computations should make sense as finding the number of bits needed to count E rows and each of B bytes. This might seem like a lot of rote number crunching, but the uses of these computations/fields should become clearer in the next section.

As an example, let's break the memory address, 0xABCDEF84 into its tag, index, and offset fields, where our cache is 4 KiB in total size with cache blocks of 16B and associativity, N = 1:

$$E = \frac{2^{12}}{(2^4)(1)} = 2^8,\ I = \log_2(E) = \log_2(2^8) = 8,$$

$$O = \log_2(B) = \log_2(2^4) = 4,\ T = \log_2(2^{32}) - 8 - 4 = 20$$

31 ... 12	11 ... 4	3 ... 0
Tag	Index	Offset
ABCDE	F8	4

Just a note on terminology, when $N = 1$, we say that the cache is a *direct mapped* cache. When $N = \#$ of cache blocks $= T/B$, we say that the cache is *fully associative*, as there is only 1 row in the entire cache.

3 The Caching Scheme

Now that we know what a cache looks like, we can understand the scheme used to figure out where data from a given memory address is stored in our cache. We're going to gradually walk through the rationale of the scheme, and hopefully motivate the organization discussed in the previous section.

As of now, we could think of cache entries as key-value pairs, with a memory address mapping to one cache block of values in corresponding memory (for now you can think of the tag column to contain the entire memory address, instead of the tag field). To make our cache scheme efficient, we want to be able to quickly add and retrieve data from the cache (put/get), and quickly check if our cache contains the data for a certain address (contains). For the rest of this section, consider a cache with 16B cache blocks, associativity $N = 1$, and a total size of 256B. This is the cache depicted in figure 2.(Quick exercise, how many rows, E, does this cache have and how many index bits?)

key = addr.	value = cache block of data															
0xABCDEF00	00	00	00	01	00	00	00	02	'I'	'L'	'U'	'V'	'6'	'1'	'C'	\0

Figure 4: Our current cache element.

Before discussing the scheme itself, observe that our current address-cache block pair scheme is slightly redundant, as we don't need to store the entire memory address. Since the least significant bits (i.e. the offset field bits) of the memory address can always be determined by the offset of the byte within its block, we don't need to store those bits of the address in our tag/key/address.

Concretely, we could store 0xABCDEF00 as the tag for our cache block of data, but instead we could store only 0xABCDEF0 and then concatenate a byte's offset within the block to this truncated address in order to derive the full memory address that the byte corresponds to. For example, the byte with offset 11 in this cache block of data would correspond to memory address {0xABCDEF0, 0xB} = 0xABCDEF0B. Accordingly, so that we save space and store less bits for the tag column in our cache, we could store in our tag column just the tag and index sections of the memory address:

key = addr.	value = cache block of data															
0xABCDEF0	00	00	00	01	00	00	00	02	'I'	'L'	'U'	'V'	'6'	'1'	'C'	\0

Figure 5: Saving 4 bits in our key.

3.1 Cache rhymes with Hash (Table)

Naively, we could treat our cache as just a list of these address-cache block pairs, and add pairs into the next empty row. Recalling 61B, in the worst case, it would take linear time (with respect to the number of possible cache blocks) to find an empty row to add data, and linear time to compare keys/memory addresses in the cache to find the desired data corresponding to an address. This isn't very efficient, especially for caches with many blocks. Instead, let's use a data structure that has constant time puts/gets/contains: hash tables/maps.

As a reminder, hash tables are fixed-length arrays that will hash the keys of its elements in order to determine at which index in the array, we should store the element's key-value pair. In the case of our cache, our keys are memory addresses (i.e. nonnegative integers), and each row of our cache is an index. Also, note that we are going to hash our truncated address that do not have the offset bits.The simplest way (probably) to hash integers into E possible rows is to compute: $index = (addr.) \mod E$, which is equivalent to taking the index field bits of our address and interpreting the corresponding number as the index!

	Tag	Offset															
Index = 0	0xABCDEF**0**	00	00	00	01	00	00	00	02	'I'	'L'	'U'	'V'	'6'	'1'	'C'	\0
Index = 1	0xBFFCEC**1**																
Index = 2	0xBFFCEC**2**																
	...	...															
Index = 13	0x123456**D**																
Index = 14	0xABCDEF**E**																
Index = 15	0xABCDEF**F**																

Figure 6: Index fields mapping to cache rows.

For example, with our example cache of 16 rows, 0xABCDEF0 would have an index field of 0x0, so we would store that address and its corresponding block of data in the 0th row of our cache. Similarly, 0xABCDEF7 and 0x123456D, would be stored in the 7th and 13th rows, respectively. Note that it's coincidental that the number of index and offset bits is the same. They always depend on the parameter of the cache, and were chosen to be 4 for simpler parsing in this note (most of the time you'll have to convert addresses to binary first).

Observe that with this conception of the index, we now again have redundant bits in the "address" we store in the tag column. Since, in order to determine the index field bits of the full memory address corresponding to a byte of data in our cache, we can always just take the index of the row that the byte is stored in and convert it to a I-bit number, we don't need to store the index field bits either. Concretely, for the previous examples 0xABCDEF0, 0xABCDEF7, and 0x123456D, we could instead store only 0xABCDEF, 0xABCDEF, and 0x123456, and then recover the index bits by observing that they are stored in the 0th, 7th, and 13th rows of the cache respectively, and then converting 0, 7, and 13 to binary/hex.

Accordingly, so that we save space and store less bits for the tag column in our cache, we could store in our tag column just the tag section of the memory address. And that's why the tag field of the memory address is what it is.

key = addr.	value = cache block of data															
0xABCDEF	00	00	00	01	00	00	00	02	'I'	'L'	'U'	'V'	'6'	'1'	'C'	\0

Figure 7: Saving another 4 bits in our key.

Now, with our hash table like scheme, we can put data in our cache in constant time (with respect to the number of blocks in our cache), by computing the tag field and index fields of the address, and then storing the tag field along with the block of data at the computed index/row in our cache. We can also check if an

address' data is in our cache/retrieve that data in constant time, by taking our desired address, splitting it into the tag and index fields of the address, and then going to that index/row in our cache and comparing if the tag field of our desired address matches the value in the tag column of that index/row.

3.2 Collisions & Associativity

You may have noticed that it is possible (and probably likely) for two memory addresses to have the same index field, such as 0xABCDEF07 and 0x1234560D. Since their tags are different, we can clearly tell that they correspond to different memory. What happens if 0xABCDEF07 and its data are in the cache, and then we access and try to put 0x1234560D and its data in the cache? Just as in hash tables, a collision! Since both addresses hash to the same index but have different tags, we must evict 0xABCDEF07 and its data from row 0, as our cache has associativity $N = 1$ and can only have 1 block per row/index.

Depending on your program's memory access pattern, this eviction could be problematic (what if you access 0xABCDEF07 again right after 0x1234560D?). Associativity attempts to solve this problem by allowing for more than one cache block to be stored at the same row/index, where n-way associativity allows for n blocks to be stored per row. This means that each row of the cache would now have n tag columns for each of n blocks of data.

	Tag	Data				Tag	Data			
Index = 0	0xABCDEF	1	2	'ILUV'	'61C'	0x123456	0	4	8	12
Index = 1										
	...	...				...	...			

Figure 8: 2-Way Associative Cache

In the previous example, if our cache was instead 2-way associative, then we would not have needed to evict 0xABCDEF07 from row 0, and this is depicted above (note: each data box is now one word instead of one byte). Associativity solves our eviction problem, but it does have two tradeoffs. Space wise, in order to increase our associativity, but keep the cache block size and number of rows of our cache the same, we would need to increase the total size of our cache. It is expensive to do so (faster storage like caches are more expensive than slower storage like memory), so design choices to increase associativity usually result in a decrease in the number of rows of our cache. (Review the formula for E when N increases). Time wise, when checking if our cache contains a certain memory address, we must now check the n different tags in a row instead of a single tag, which is decidedly slower as we must now use multiple comparators and muxes to do so.

4 An example program & Types of Misses

Now, let's walkthrough an example program and see how our cache affects our memory accesses. If we are able to find the data for a memory address in our cache, instead of having to go to memory, then that access is a **hit**. If we are unable to, and must go to memory, then the access is a **miss**. The hit rate and miss rate of a program is then calculated as: $\frac{\text{\# of hits or misses}}{\text{\# of total accesses}}$.

Each read or write to memory is considered a memory access, so something like `A[0] = 4;` (a write of A) and `int x = A[0];` (a read of A) are both accesses. A read and a write, such as `A[i] += 2 <==> A[i] = A[i] + 2` is then two accesses in a single line, and it is possible for the read of A[i] to first miss but then the following write to A[i] to hit, after reading A[i] brings A[i]'s data into the cache.

For the program below, consider the same cache as in the previous section, and an array of 256 integers, A, that begins at memory address 0xBFFFEC00. Our cache starts empty.

4.1 Compulsory Misses

Beginning with section I of our program, at $i = 0$, we want to access the 4 bytes of data starting at memory address 0xBFFFEC00. The tag, index, and offset fields of this address break down as: 0xBFFFEC, 0x0, and

```
int foo(int* A) {
  int i = 0;

  // section I
  int sum = 0;
  for (i = 0; i < 256; i++)
    sum += A[i];

  // section II
  int negSum = 0;
  for (i = 255; i >=0; i--)
    negSum -= A[i];

  /* cache is flushed,
  and is again empty */
  // section III
  int x = A[0];
  int y = A[64];
  A[0] = 4;

  /* cache is flushed,
  and is again empty */
  // section IV
  int quarter_sum = 0;
  for (i = 0; i < 64; i++)
      quarter_sum += A[i];
  int mystery = quarter_sum + A[64];

}
```

Figure 9: An example program.

0xBFFFEFFC	00	00	00	FF	A[255]
0xBFFFEFF8	00	00	00	FE	A[254]
0xBFFFEFF4	00	00	00	FD	A[253]
0xBFFFEFF0	00	00	00	FC	A[252]
0xBFFFEFEC	00	00	00	FB	A[251]
	...				
0xBFFFED00	00	00	00	40	A[64]
	...				
0xBFFFEC1C	00	00	00	07	A[7]
0xBFFFEC18	00	00	00	06	A[6]
0xBFFFEC14	00	00	00	05	A[5]
0xBFFFEC10	00	00	00	04	A[4]
0xBFFFEC0C	00	00	00	03	A[3]
0xBFFFEC08	00	00	00	02	A[2]
0xBFFFEC04	00	00	00	01	A[1]
0xBFFFEC00	00	00	00	00	A[0]

Figure 10: Memory contents for array A.

0x0. Therefore, we can check if there is a tag of 0xBFFFEC at index 0x0 = 0, and since our cache is empty, we will not find such and miss. This is a **compulsory miss**, as the data we wanted to access was never in our cache, so there was no way we could have had a hit and the miss was necessary. However, afterwards, we will then add the bytes of data from the range 0xBFFFEC00 to 0xBFFFEC0F into our cache, at index 12, with tag 0xBFFFEC. The contents of our cache after $i = 0$ are shown below.

	Tag	Offset															
Index = 0	0xBFFFEC	00	00	00	00	00	00	00	01	00	00	00	02	00	00	00	03
Index = 1																	
Index = 2																	
	...	...															
Index = 13																	
Index = 14																	
Index = 15																	

Figure 11: Following i = 0.

Next, let's look at $i = 1$. A[1] is be stored at address 0xBFFFEC04 (4 bytes after A[0]). Is the data at the address in our cache? If we break the address down, then we find that its tag, index, and offset fields are 0xBFFFEC, 0x0, and 0x4. Now, looking in our cache, is row 0 occupied? Yes. Is the tag the same as the tag of our desired data address, 0xBFFFEC? Yes, so the data at the desired address is inside our cache and we can find it at row 0 in offset 4 of the block of data stored there. We have a hit! As an exercise, verify that $i = 2,\ 3$ also result in hits. Therefore, for every 4 values of i, we have 1 miss out of 4 total accesses, for a miss rate of 1/4 and a hit rate of 3/4. Now, does this miss/hit rate for every four values of i change as i continues to increment? Think about it, and see if you can conclude the miss rate for the entirety of section I. Our cache up to $i = 12$ is shown below (see if you can fill in the cache for up to $i = 64$).

	Tag	Offset															
Index = 0	0xBFFFEC	00	00	00	00	00	00	00	01	00	00	00	02	00	00	00	03
Index = 1	0xBFFFEC	00	00	00	04	00	00	00	05	00	00	00	06	00	00	00	07
Index = 2	0xBFFFEC	A[8] = 8				A[9] = 9				A[10] = 10				A[11] = 11			
	...	...															
Index = 13																	
Index = 14																	
Index = 15																	

Figure 12: Following i = 12.

Let's now look at section II. First, note that our cache is no longer empty. At the conclusion of section I, our cache should contain the last quarter of our array, A ($\frac{\text{256B cache}}{\text{256*4B array}} = \frac{1}{4}$). The state of our cache is depicted below (only the last four rows for brevity). For the first access in section II, is A[255] in our cache? One way to check would be to see if it's address is in the cache: 0xBFFFEC00 + 255 * 4 = 0xBFFFEC00 + 0xFF << 2 = 0xBFFFEC00 + 0b001111111100 = 0xBFFFEC00 + 0x3FC = 0xBFFFEFFC. Another is to observe that the last quarter of A is in our cache, so A[255] should also be in the cache.

Either way we have a hit, and for the first quarter of the loop (from $i = 255..192$), we should have a hit rate of 1. However, once we are past the first quarter of the loop, our cache no longer already contains the

contents of A that we need, and our hit/miss rate will resemble that of section I for the remaining three quarters of the loop. Therefore, we can calculate our total hit rate as $(1/4)(1) + (3/4)(3/4) = 13/16$.

	Tag	Offset			
	...	...			
Index = 12	0xBFFFEF	A[240] = 240	A[241] = 241	A[242] = 242	A[243] = 243
Index = 13	0xBFFFEF	A[244] = 244	A[245] = 245	A[246] = 246	A[247] = 247
Index = 14	0xBFFFEF	A[248] = 248	A[249] = 249	A[250] = 250	A[251] = 251
Index = 15	0xBFFFEF	A[252] = 252	A[253] = 253	A[254] = 254	A[255] = 255

Figure 13: At the end of section I.

4.2 Conflict Misses

Let's look at section III. (Note that our cache is again empty). Accessing A[0] will result in a miss and bringing the cache block corresponding to 0xBFFFEC00 into row 0 of our cache. Then, accessing A[64] (at address 0xBFFFEC00 + 64 * 4 = 0xBFFFEC00 + 256 = 0xBFFFEC00 + 0x100 = 0xBFFFED00) will also miss and now evict A[0] from row 0 of our cache, as both share the same index of 0. Accessing A[0] on the next line will then again result in a miss! Specifically, this will be a **conflict miss**, as the data (A[0]) we desired had been brought into our cache before, but due to a conflict with another block of data (A[64]), the desired data was evicted.

	Tag	Offset														
Index = 0	0xBFFFED	00	00	00	40	00	00	00	41	00	00	00	42	00	00	00	43
Index = 1																	
Index = 2																	
	...	...															

Figure 14: Following int y = A[64].

In other words, a conflict miss is a miss that resulted due to a preceding data access evicting the desired data. Another way to understand conflict misses is to ask: if the cache's associativity had been greater, would the miss have been avoided? For example, if our cache had instead been 2 way associative (for simplicity, assume that our cache's total size doubles to accommodate such), we would have avoided the conflict miss on A[0] = 4, as depicted by the state of our cache below before A[0] = 4.

	Tag	Data				Tag	Data			
Index = 0	0xBFFFEC	A[0]	A[1]	A[2]	A[3]	0xBFFFED	A[64]	A[65]	A[66]	A[67]
Index = 1										
	...	...				...	...			

Figure 15: 2-Way Associative, following int y = A[64].