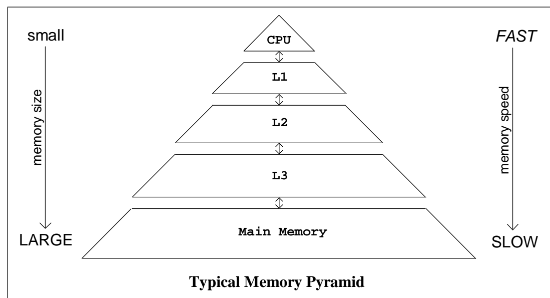
By Joshua Cantrell
jjc@cory.berkeley.edu

Caches

Caches At Many Levels

Caches not only exist between a computer's main memory and microprocessor, but between any different types of storage devices. The microprocessor can be thought of as memory device because it must first load data into registers (it's own local memory) before it can do an operation. Because most microprocessors lacks enough registers to store an entire program, they need to read the data from an outside memory source. These outside memory sources are typically slower, but can hold more information (and for the most part are less expensive per bit of data). The lower levels of cache memory try to imitate the highest level of memory, so they act as abstract interfaces between the higher and lower memory levels.[1]



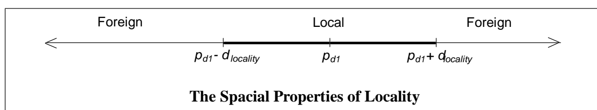
Typical Memory Pyramid

Direct Mapped Cache

The Principal of Locality

Direct mapped caches depend entirely on the concept of *the principal of locality*. When a program is being run, most of it's instructions and data fall into little clumps of memory which are all close to each other. The closer they reside in memory, the more local they are to each other. The further away they become, the more foreign they are to each other.

$$p_{d2} \text{ is local to } p_{d1} \text{ if } p_{d1} + \Delta p_{d12} = p_{d2} \text{ and } |\Delta p_{d12}| < d_{locality}$$



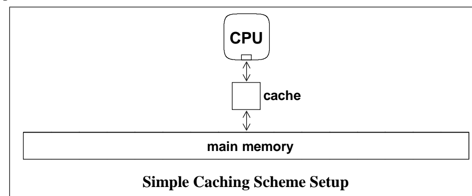
The Spacial Properties of Locality

[1] Pyramid description of memory borrowed from Professor Patterson's Spring 1999 lecture notes.

By Joshua Cantrell
jjc@cory.berkeley.edu

Fitting the Cache to Memory

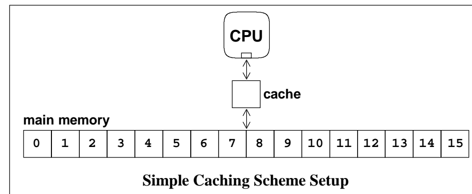
The cache memory is usually smaller than the memory it is trying to imitate. This means it cannot have all of the higher level memory in itself at one time. Even though it has to look like the main memory to the CPU (or previous level of cache), it cannot promise that it will have the data needed immediately upon request (of course it doesn't want the CPU to know that, it lets the CPU think it has the data). Let's assume our cache is $\frac{1}{16}$th the size of the main memory and is placed as provided:



Simple Caching Scheme Setup

Notice how the CPU cannot talk to the main memory directly, it must go through the cache to get the data. Our cache is supposed to save commonly used data in itself so the CPU doesn't have to wait for a long period of time to get the required data from the slow main memory.

We can divide up the main memory into chunks the size of the cache. Each chunk will be formed of consecutive bytes stringed together so they can all be more or less *local* to each other. Obviously, data in neighboring chunks can also be local to data in a chunk if it is close relative to the size of the cache.



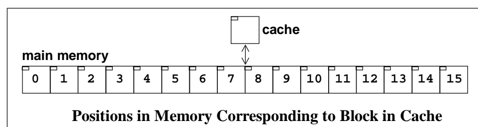
Simple Caching Scheme Setup

A direct mapped cache tries to mirror data from these chunks into the cache. Since we know *everything* in a program will not just be in one of the chunks, a direct mapped cache divides itself into blocks of data which contain several bytes of data each. This way the cache can hold data from several chunks at one time.

By Joshua Cantrell
jjc@cory.berkeley.edu

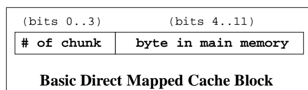
Mirroring Data to the Cache

The data is put into the cache in the same location it is taken from one of the chunks in memory. For example the first block in the cache corresponds to the first bit of data in each chunk.



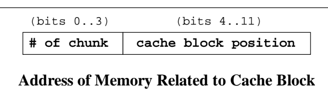
Positions in Memory Corresponding to Block in Cache

Let's say the cache has $2^8 = 256$ bytes of storage capability. That would mean the memory could store a total of $2^{8+4} = 4096$ bytes. Assuming that each block of the cache stored just 1 byte, what would a block look like? Well, the block would need to know what data was in the location in memory, so it would need 8 bits to store a byte. But what if the CPU asked the cache for the memory, how would the cache know if the data belonged to the correct chunk in memory? So, it'll need a number so that it knows which block of data the data belongs to. This should take 4 bits to hold a value from 0 to 15.



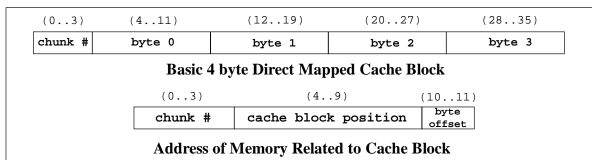
Basic Direct Mapped Cache Block

Since the position of the block in the cache is the same as the position of the data in one of the chunks of memory, we know which block in the cache to check to find data in the memory. Blocks containing single bytes are the easiest case, we just take the address into memory and use the 8 least significant bits (LSBs).



Address of Memory Related to Cache Block

But what about the case where a block contains many bytes? How can we then relate the memory address to the cache block position? Well, then the most least significant of the LSBs are used as a byte offset into the cache block's data.



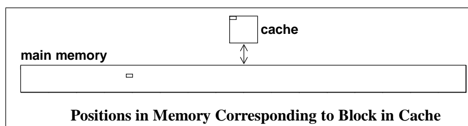
Address of Memory Related to Cache Block

By Joshua Cantrell
jjc@cory.berkeley.edu

Fully Associative Cache

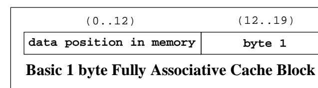
Putting Cached Data Anywhere

What data goes into direct mapped cache blocks depends on the position of the data being cached in memory. In fully associative caches, the blocks in the cache can be used to store data found anywhere in the memory. This means the position in the cache has no direct correlation with the position in memory!



Positions in Memory Corresponding to Block in Cache

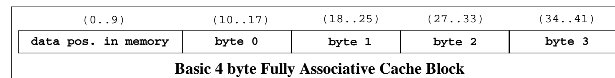
How then do we know what memory address a particular block in the cache is supposed to represent? If each block is a single byte and there are $2^{12} = 4096$ bytes in the main memory, that would mean each cache block would need 12 bits to distinguish it from any other! This is because the data stored in the block could be from any byte in memory!



Basic 1 byte Fully Associative Cache Block

Notice that the address isn't partitioned into parts like in direct mapped caches because the position of a block in a fully associative cache has nothing to do with the address provided to the cache. This means that the cache has to check every element in the cache to find out if data for a particular memory location is present (or not present). This can take a sizable number of transistors, so it isn't generally desirable to make large fully associative caches.

Associative caches can also store more than just one byte. As you add the number of bytes stored in a cache block, it no longer needs the full address of memory because each element is aligned to the block's data size. For example, if our block contained four bytes, then we would only need $2^{12-2} = 2^{10} = 1024$ values, or 10 bits, to determine what data in memory the cache represented.



Basic 4 byte Fully Associative Cache Block

Least Recently Used (LRU)

Because any data in the memory can be stored in a fully associative cache, there must be a way to know what should and shouldn't be in the cache. The best method is to replace the least recently used data before more recently used data. This can be done by giving each block more information, like the time it was used last, or through sorting in order of most recently used to least recently used. Both of which can use many logic transistors.

By Joshua Cantrell
jjc@cory.berkeley.edu

Random

A simpler, yet effective method for choosing a block to be replaced in the cache is through a random number. This can work given that the value is sufficiently random enough. This isn't as good as using LRU, but takes less logic and does work reasonably well.

Set-Associative Cache

Hits and Misses

If a memory address is given to a cache, and the data is present inside the cache, a *hit* is said to have occurred (you've sunk my battleship!). If a memory address is given to a cache, and the data is not present inside the cache, a *miss* is said to have occurred.

In the case of a hit while reading data, the cache retrieves the data from the cache. In the case of a miss while reading data, the cache must ask the upper level of memory for the data it doesn't have available (in the case of a *write-back* cache it must also write the data in the cache to the upper level of memory if it is *dirty*[2]). Because the upper level of memory is slower, a miss increases the amount of time needed for data to be retrieved from memory.

In the case of a hit while writing data, the cache must write the data to the cache, and if it is *write-through*, it must also write the data to a *write buffer*. If the write buffer becomes full, this may cause a time penalty. In the case of a miss while writing data, if the cache is *write-through*, it will just write the data into the cache and write buffer, otherwise if the cache is *write-back*, it will first check to see if the data in the cache is *dirty*, write the data to memory if it is changed from memory, read the data in the associated address from memory[3], and finally write the data to the cache.

Direct Mapped Advantages & Disadvantages

Direct mapped caches have the advantage that they have relatively quick access time with very little logic, keeping the chip size and power consumption down. Their disadvantage is that there may be times where you must access data constantly in the same location of multiple memory chunks causing many consecutive misses.

Fully Associative Advantages & Disadvantages

Fully associative caches have the advantage that they have extremely low miss rates because things used the most (given that you use LRU) will always be available. Their disadvantage is that it requires more logic and complexity making the chip size larger and increasing power consumption.

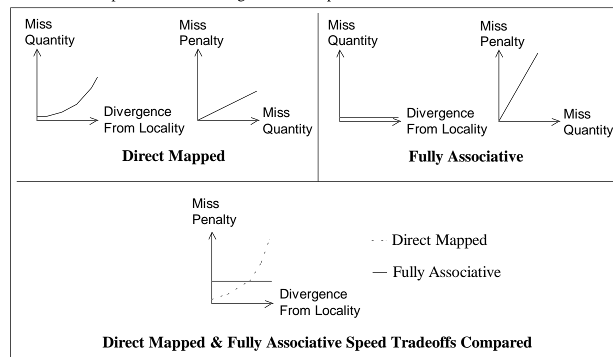
[2] changed from what resides in memory

[3] I don't know why this step exists. Maybe I have it wrong, so I'll have to find out for certain.

By Joshua Cantrell
jjc@cory.berkeley.edu

Combining Fully Associative and Direct Mapped

What if we could combine the two types of caches? Could we get the best of both worlds, or at least a compromise that would give us better performance? Yes we can![4]



Direct Mapped & Fully Associative Speed Tradeoffs Compared

In many cases, there isn't that much divergence from locality so Direct Mapped would work well, however as your cache becomes much smaller than the quantity of memory being imitated, even what might seem local within the great expanse of your main memory isn't that local to the cache. It makes sense in that case to use a combination of these two techniques.

A cache entry in this case would be indexed in the same way as a block in direct mapping, except that the data would be fully associative cached in respect to the memory chunk number.

(bits 0..3)	(bits 4..11)
# of chunk	byte in main memory
# of chunk	byte in main memory
# of chunk	byte in main memory
# of chunk	byte in main memory

4-Way Set Associative Cache Entry

This type of cache is termed *n-way set associative*. The 'n' stands for the number of associative blocks inside the main direct mapped cache entry.

[4] I made up the data in the graphs as an example. I bet there may be some divergence from reality, but it might be helpful qualitatively.