Eleventh class

\_\_\_\_\_\_\_\_\_\_\_\_\_\_

A "Cache" Picture

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Caches work by partitioning a memory address into three fields: an offset field, an index field, and a tag field.

+-----------+-------------+--------------+

| | | |

| tag field | index field | offset field |

| | | |

+-----------+-------------+--------------+

[ line number ]

These are related by the four fundamental cache laws. What is interesting is that since each field is some number of bits, cache quantities become powers of 2. Let a memory address be 'n' bits. Let a line contain B = 2^b bytes.

Then, according to the first equation below, we see that the \_offset field\_ is 'b' bits. A computer determines the byte offset by masking the lower-order 'b' bits, and then interpreting these bits as a natural number. Computers use masking and shifting because they don't have pocket calculators.

0) byte number = memory address `mod` the number of bytes

in a line

But the byte offset is of interest to relatively few people. The cache cares far more about the index field and the tag field. The first step therefore is to \_discard\_ the offset field and focus on the remaining bits. This too is accomplished by masking and shifting. The remaining bits make up the \_line number\_. The equation is:

1) line number = memory address `div` the number of bytes

in a line

We now partition the line number into a (cache) index and a tag value. To do this, we need to know the size of the cache array. Let the cache contain M = 2^m elements. Then the index field is the lower-order 'm' bits of the line number, which itself has 'n - b' bits. More masking and shifting.

2) element index = line number `mod` the number of elements

in the cache

At this point, we have 't = n - b - m' bits left over, which means that there are

T = 2^t possible tag values. Still more masking and shifting.

3) tag value = line number `div` the number of elements

in the cache

In short, what we have described with the `mod` and `div` operators is just a mathematical way of describing these mask and shift operations.

You remember that, in a direct-mapped cache, the array elements are \_frames\_, whereas, in an m-way set-associative cache, the array elements are \_sets\_ of frames. Both frames and sets have cache indices.

When a line is copied from memory into the cache, it is placed in the cache elements with the calculated index.

Because many lines are mapped to the same array element, a tag value is used to identify which line got copied. This is required to be able to look up lines in caches that a program has need for.

Tags are space overhead in that a cache must store one t-bit tag value for every line they contain. In an m-way set-associative cache, tags are also time overhead during lookup, in that 't' bits from the line being looked up must be compared with every tag value stored in the set.

You can imagine that the cache equations give rise to exercises that students must solve. These exercises may be easy or hard. Here is one from a previous tutorial.

Consider a computer with 4,096 Bs of main memory. It has a direct-mapped cache with 32 frames. Moreover, cache lines contain 16 Bs. Show the partition of a memory address and calculate the cache index for byte address [a53].

Solution: A memory address has 12 bits, since 4,096 = 2^12. The offset field has 4 bits, since 16 = 2^4. The index field has 5 bits, since 32 = 2^5. The partition is therefore: <3,5,4>, which is 12 bits.

Now, let's place the line containing byte [a53]. The line number is [a5] = [a53] `div` [10]. The lower-order 5 bits of [a5] are '00101' = [05].

The cache index is 5 decimals, and the 3-bit tag field is [5], which is '101' in binary (note the ambiguity in hex representations).

Note that sometimes it is less work just to write out the bit patterns in full, while in larger problems this may not be the case.

I steal one more example from the same tutorial.

Consider a computer with 4,294,967,296 = 2^32 Bs of main memory. It has a 2-way set associative cache with some number of sets. The cache has 1,024 = 2^10 frames. So, there must be 512 = 2^9 sets. A line is 32 = 2^5 bytes.

Partition the memory address, and then find the cache index of byte [b46d5300].

By inspection, the partition is <18,9,5>, which is 32 bits.

Byte [b46d5300] is in line number [5a36a98] = [b46d5300] `div` [20]. The cache index is [098] = [5a36a98] `mod` [200], which is 152. And the tag value is [5a36]101, which is [2d1b5] (or 184,757, in decimal).

Here, I used various calculational tricks.

The goal of a memory hierarchy is to keep all of the code and data that is needed, presently and in the near future, close to the processor. Ideally, this information should be reachable in a single cycle, and this dictates putting the cache on the processor chip. However, the capacity of this on-chip SRAM memory must not be too large, because the time to access it is roughly proportional to its size. For this reason, modern computers have multilevel caches, with the cache size increasing as we go down the cache hierarchy.

Alas, cache copies die young.

Caches are much smaller than main memory so, at any moment, they contain only a tiny fraction of the executing program's code and data. When the processor issues a memory reference, or the fetch-execute cycle initiates a memory access, a cache lookup is performed to determine whether the requested code or data item is present in the cache. If so, we have a \_cache hit\_. If not, we have a \_cache miss\_. Cache misses cause us to recursively query the remaining levels of the cache hierarchy, and, in the worst case, force us to perform a full memory access. In this worst case, the time spent querying must be \_added\_ to the memory latency.

For simplicity, in the following paragraphs, we focus on single-level D-caches only.

Caches are high-speed buffers between main memory and the CPU. Because their storage capacity is much less than that of main memory, there are five basic cache-design questions (we will add a sixth later):

1) When a memory reference (load or store) leads to a cache miss, do we \_automatically\_ make a copy? (the copy-decision problem)

2) When a copy is made, how many bytes do we copy? (the cache-line-size problem)

3) When we do make a copy, where do we put it? (the placement problem)

4) How do we determine whether a copy of a given data item exists? (the lookup problem)

5) What do we do if the cache is "full" (this has two meanings), and there is no room for a new data item we wish to bring in? (the replacement problem)

Later, we will consider:

6) What happens on a write?

---

A simplified set of answers, for conventional cache design, is as follows:

1) [Automatically?] Yes, we make a copy of the data item---indeed, a copy of the whole line---automatically whenever a memory reference for the item results in a cache miss (this is \_demand\_ copying).

Potential problems: No program information is used by the cache about whether, or how soon, or how many times, the data item will be used again. Also, no program information is used by the cache about whether the program has no, some, or a lot of interest in nearby memory locations. The second comment concerns the \_granularity\_ of copies, i.e., the cache-line length.

2) [Granularity?] Say that the memory reference is for one memory word. Then, we might copy just that word, or we might copy a \_sequence\_ of contiguous words that contain the referenced word. The amount that is copied is called a \_cache line\_. If the cache line contains few words (say, 1 or 2), the cache line is \_short\_. If the cache line contains many words (say, 16 or 32), the cache line is \_long\_. Cache designers who believe that all programs have spatial locality argue in favor of long cache lines. Wiser computer architects argue in favor of short cache lines as part of a general

policy to avoid mis-speculation in all its forms.

3) [Placement?] The cache is divided into a number of \_cache frames\_. We must have some policy to map memory locations into cache frames. Some policies are tight and don't give us much choice; other policies are loose and give us more freedom. We will discuss these policies in detail below.

4) [Lookup?] In a real-world cache, each frame, which has some index position, contains a \_tag field\_ in addition to the data contents. Whether a memory reference leads to a hit or a miss is determined by i) finding the right cache frame, and ii) checking the tag. A variant of this lookup strategy is i) finding the right \_set\_ of frames, and ii) checking the tag of each frame in the set.

5) [Replacement?] If there is a cache miss, and the new data item \_collides\_ with a data item already in the cache, then one cache line (the \_victim\_) must be ejected to make room for the new data item. Victim selection is determined by a replacement algorithm. If the placement policy is tight, the victim is already determined. If the placement policy is loose, we have some choice as to which cache line should be evicted. With a tight policy, you collide with some item that maps to the same frame you do. This is a matter of \_conflict\_. With a loose policy, you collide with a set of cache frames that is completely filled. This is a matter of \_capacity\_.

A cache is an array. With the exception of the (somewhat theoretical) loosest placement policy, caches are addressed by indices. When there is indexing, each cache entry contains a tag field consisting of some high-order bits from the memory address of the copied data item. We will see how the address is divided in just a moment. The cache capacity is the product of the number of cache frames and the size of a cache line. The tags are extra (in fact, they are overhead).

Placement Policy

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

There are a range of mapping (placement) policies. If any cache line can be placed into any cache frame, then we have a \_fully associative\_ cache. If a cache line can only be placed in precisely one cache frame, then we have a \_direct-mapped\_ cache. If a cache line can only be placed in precisely one \_set\_ of cache frames, then we have a \_set-associative\_ cache.

Large fully associative caches are impractical. To avoid a lengthy linear search in an m-way set-associative cache, which would be too slow, we must use either i) some number of comparators, or ii) an associative memory, for copy lookup. An associative memory is an \_alternative\_ to an indexed array.

Array elements are located, and retrieved, by indexing. In contrast, elements of an associative memory are located, and retrieved, by simultaneously matching a searched-for key against \_every\_ key in the associative memory, which should be thought of as a sequence of <key,value> pairs.

Direct mapped and set associative

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1) A direct-mapped cache consists of a number of frames, each with its own index. Say the cache-line number is 'n' bits long, and there are 2^k cache frames. Then the low-order 'k' bits of the cache-line number determine the cache frame. Obviously, many cache lines map into the same cache frame. They are distinguished by the remaining 'n-k' bits, which are stored as a tag field in the cache frame. To look up a cache line, we

i) compute the cache index, and ii) compare the tag field (the 'n-k' bits) with the high-order bits of the address of the requested cache line. We also need a valid bit which indicates whether the cache frame contains genuine data or garbage.

2) An m-way set-associative cache consists of a number of frame sets, each of which consists of 'm' frames. Example: A 32-way set-associative cache with 4,096 cache frames would give rise to 2^7 sets each containing 2^5 frames. During lookup, we would match 32 tags in parallel (this would require an associative memory). Each cache line maps to precisely one of the 2^7 sets. Instead of indexing frames, we now index \_sets\_ of frames.

Say the cache-line number is 'n' bits long, and there are 2^k cache sets. Then the low-order 'k' bits of the cache-line number determine the cache set. Obviously, many cache lines map into the same cache set. They are distinguished by the remaining 'n-k' bits, which are stored as a tag field in each \_occupied\_ cache frame in the cache set.

To look up a cache line, we

i) compute the cache index, and

ii) compare all 'm' tag fields ('m' times 'n-k' bits) with the high-order bits of the address of the cache line. We also need valid bits which indicate whether each cache

frame in the set of 'm' frames contains genuine data or garbage.

3) In a fully associative cache, there is effectively only one set. This means that there are \_no indices\_. To look up a cache line in a fully associative cache with 2^k cache frames, each tag field is the whole n-bit line number, there is no index field, and we must match the line number we seek, in parallel, against all 2^k stored tags. We also need valid bits which indicate whether each cache frame contains genuine data or garbage.

Example: A computer has a byte-addressable memory and 32-bit memory addresses.

A cache line contains 64 bytes. A cache-line number is therefore 26 bits long.

Example: A computer has a 40-bit line number and a direct-mapped cache with 4K frames. The frame number is therefore 12 bits long and the tag field is 28 bits long.

Example: A computer has a 40-bit line number and a 4-way set-associative cache

with 4K frames. The set-index number is therefore 10 bits long and the tag field is 30 bits long. In general, increasing the wayness by a factor of 2 decreases the size of the index by 1 bit and increases the size of the tag by 1 bit.

Replacement Policy

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In an m-way set-associative cache, when there is a cache miss, the indexed set of 'm' frames may be full. To make room for a new cache line, a victim line must be selected from the set. The best scheme is \_least recently used\_ (LRU):

Replace that cache line that has been unused for the longest time. If you

don't want to pay for the bookkeeping, use \_random\_ replacement. I have heard a suggestion that a cache twice as large with random replacement has a miss rate equal to the smaller cache with LRU.

Write Policy

\_\_\_\_\_\_\_\_\_\_\_\_

Consider a store instruction. If we wrote only into the D-cache, the cached copy would differ from the memory original. We could use \_write through\_ and write the data into both the memory and the cache. This typically has poor performance. The alternative to write through is \_write back\_. When a write occurs, the new value is written only to the line in the cache. The modified line is written to the next lower level of the hierarchy, which we will always take to be the main memory, when it is replaced. Unmodified lines are simply evicted from the cache when they are replaced.

Three C's

\_\_\_\_\_\_\_\_\_

It is useful to classify cache misses into three categories:

1) \_Compulsory\_ misses occur the first time a cache line is referenced.

2) \_Conflict\_ misses occur when more than 'm' cache lines compete for the same cache frames in an m-way set-associative cache.

3) \_Capacity\_ misses occur when the working set of the program is too large for the cache, even if i) the cache is fully associative, and

ii) optimal cache-line replacement is used.

Example: Consider a 1-way set-associative cache (direct-mapped cache). If a cache line maps to cache frame 'f', but frame 'f' is occupied by another line, then that is a conflict miss. More precisely, it is a conflict miss unless it is a compulsory miss (we check membership in the three categories in order).

Example: Consider a 4-way set-associative cache. If a cache line maps to cache set 's', but set 's' is occupied by 4 lines different from the line we are looking up, then that is a conflict miss. The remark about checking in order still applies.

Example: Consider a 4-way set-associative cache with 4K cache frames. This is 1K sets with 4 frames per set. Now, double the cache size to 8K frames. This is 2K sets with 4 frames per set. A cache line that missed in the smaller cache but hit in the larger cache \_appears\_ to count as a capacity miss. But order still matters. That is, a cache line that missed in the smaller cache and was categorized as a conflict miss would not count as a capacity miss. Technically, a capacity miss requires a fully associative cache, but doubling the cache size did make conflict misses half as likely.

Cache-Performance Equations

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

A. Expected/average/effective memory-access time:

tbar = cache-hit time + miss rate \* memory latency

B. Effective memory bandwidth:

effective memory bandwidth = actual delivered operand bandwidth/

miss rate

The first formula calculates the average time to access memory.

The second formula shows the amount of \_bandwidth amplification\_ as a function of miss rate.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Little's law reminder

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The actual delivered operand bandwidth is the minimum of:

i) the memory-reference concurrency divided by the memory latency, and

ii) the hardware bandwidth (the physical upper limit on bandwidth).

Since Little's law is applicable only in steady-state conditions, we may observe that concurrency/latency also equals the memory-request bandwidth.

Little's law quantifies the performance benefit of all forms of pipelining, from 'fdxmw' pipelines to memory pipelines.

Conclusion

\_\_\_\_\_\_\_\_\_\_

If a program with a high degree of data reuse runs on a computer with a cache, then the cache will be used \_temporally\_. If a program with a high degree of contiguous, sequential memory accessing runs on a computer with long cache lines, then the cache will be used \_spatially\_. (A cache with single-word cache lines cannot be used spatially).

The principle of \_value locality\_ is simple:

Hierarchically, using any mechanism you can think of, minimize the wire distances that values travel to reach the arithmetic operations that they are operands of.

Equivalently, avoid long-range communication whenever possible.

Caches used temporally are one mechanism that can minimize the wire distances that values travel to reach the arithmetic operations that need them.

Loading a value into the processor from remote memory incurs an enormous wire-distance cost, but reusing the value amortizes this cost since cache accesses have small wire-distance cost. Similarly, writing a value into the cache and then using the value has a low wire-distance cost, viz., two cache accesses. These are the two ways caches are used temporally. Note that registers can \_only\_ be used temporally.

Caches are used spatially when a cache miss for a word automatically causes the speculative prefetching of a multiword cache line. A cache used spatially does not minimize the wire distances that values travel (the 'n' words loaded into the cache travel the same distances over the same wires as they would if they were loaded separately). More precisely, the memory pipelining in cache-line prefetching has some positive effect because it tolerates (some) memory latency, but it reduces neither the bandwidth requirements nor the power consumption.

In simple English, long cache lines are a parallelism (latency-tolerating) trick, not a locality (latency-avoiding) trick.

Speculative prefetching of multiword cache lines is a form of parallelism in which the entire line is obtained by memory pipelining. The amount of parallelism is the length of the cache line. This parallelism tolerates some of the latency of accessing local memory. But so little---the concurrency is 16 or 32, not 400 or 500---that it does not really deserve the name of "memory pipelining".