Cache fetch and replacement policies

The *fetch policy* determines when information should be brought into the cache.

- Demand fetching—
- Prefetching—

Prefetching policies:

- Always prefetch. Prefetch block *i* +1 when a reference is made to block *i* for the first time.
  What is a disadvantage of this technique?

  Effectiveness: Reduces miss ratio by up to 75-80%; increases transfer ratio by 20-80%.

- Prefetch on miss. If a reference to block *i* misses, then fetch block *i* +1. (I.e., fetch two blocks at a time.)

  Effectiveness: Reduces miss ratio by up to 40%; increases transfer ratio by 10-20%.

To minimize the processor's waiting time on a miss, the *read-through* policy is often used:

- Forward requested word to processor.
- Fetch rest of block in wraparound fashion.

Write policies

There are two cases for a write policy to consider.[1]

Write-hit policies: What happens when there is a write hit.

- *Write-through* (also called store-through). Write to main memory whenever a write is performed to the cache.
- *Write-back* (also called store-in or copy-back). Write to main memory only when a block is purged from the cache.

Write-miss policies: What happens when there is a write miss. These policies can be characterized by three semi-dependent parameters.

- Write-allocate vs. no-write-allocate. If a write misses, do/do not allocate a line in the cache for the data written.
- Fetch-on-write vs. no-fetch-on-write. A write that misses in the cache causes/does not cause the block to be fetched from a lower level in the memory hierarchy.

Note that these two parameters are *not* the same. It is possible to have write-allocate without having fetch-on-write. What happens then?

- Write-before-hit vs. no-write-before-hit. Data is written into the cache before/only after checking the tags to make sure they match. Hence a write-before-hit policy will _____ _____ in case of a miss.

Combinations of these parameter settings give four useful strategies.

	Fetch-on-write?	
	Yes	No

[1]This discussion is adapted from "Cache write policies and performance," Norman Jouppi, *Proc. 20th International Symposium on Computer Architecture* (*ACM Computer Architecture News 21:2*), May 1993, pp. 191–201.

Write-allocate?				*Write-before-hit?*
Yes	Fetch-on-write	Write-validate	No	
	Fetch-on-write	Write-validate	Yes	
No		Write-around	No	
		Write-invalidate	Yes	

The shaded area in the diagram represents a policy that is not useful. Why? Because if data is going to be fetched on a write, a cache line better be allocated, or otherwise there is no place to put the data.

If data is going to be fetched on a write, it doesn't matter whether or not write-before-hit is used.

If data is not fetched on a write, three distinct strategies are possible.

- If write-allocate is used, the cache line is invalidated except for the data word that is written.

  (Again, it doesn't matter whether write-before-hit is used, because the same data winds up in the cache in either case.) This is called *write-validate.*

- If write-allocate is *not* used, then it does matter whether write-before-hit is used.
  - Without write-before-hit, write misses do not go into the cache at all, but rather go "around" it into the next lower level of the memory hierarchy. The old contents of the cache line are undisturbed. This is called *write-around.*

    (Note that in case of a write *hit*, writes are still directed into the cache; only misses go "around" it.)

  - If write-before-hit is used, then the cache line is corrupted (as data from the "wrong" block has just been written into it). The line must therefore be invalidated. This is called *write-invalidate.*

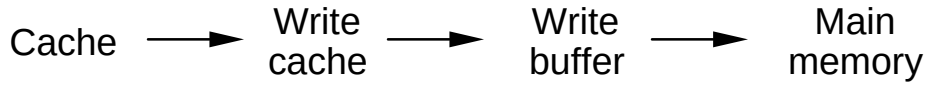
Jouppi found that fetch-on-write was the least effective of the four policies. In simulation studies, for systems with caches in the range of 8 KB to 128 KB and 16-byte lines,

- write-validate reduced the total number of misses (wrt. fetch-on-write) by 30–35%,
- write-around reduced the total number of misses by 15–25%, and
- write-invalidate reduced the total number of misses by 10–20%.

Typically, lines to be written are directed first to a write buffer (fast register-like storage), then later to main memory.

This avoids stalling the processor while the write is completed.

Jouppi found that write traffic could be decreased by adding a small fully-associative *write cache* in front of the write buffer:



Traffic from cache to main memory goes this way →

A write cache of just five entries could remove 40% of the write traffic from a 4 KB write-through cache, on average.

As caches get larger, however, a write-through design with a write cache can't compete a write-back cache, however.

Replacement policies

LRU is a good strategy for cache replacement.

In a set-associative cache, LRU is reasonably cheap to implement. Why?

With the LRU algorithm, the lines can be arranged in an *LRU stack*, in order of recency of reference. Suppose a string of references is—

a b c d a b e a b c d e

and there are 4 lines. Then the LRU stacks after each reference are—

a	*b*	*c*	*d*	*a*	*b*	*e*	*a*	*b*	*c*	*d*	*e*
	a	*b*	*c*	*d*	*a*	*b*	*e*	*a*	*b*	*c*	*d*
		a	*b*	*c*	*d*	*a*	*b*	*e*	*a*	*b*	*c*
			a	*b*	*c*	*d*	*d*	*d*	*e*	*a*	*b*
*	*	*	*			*			*	*	*

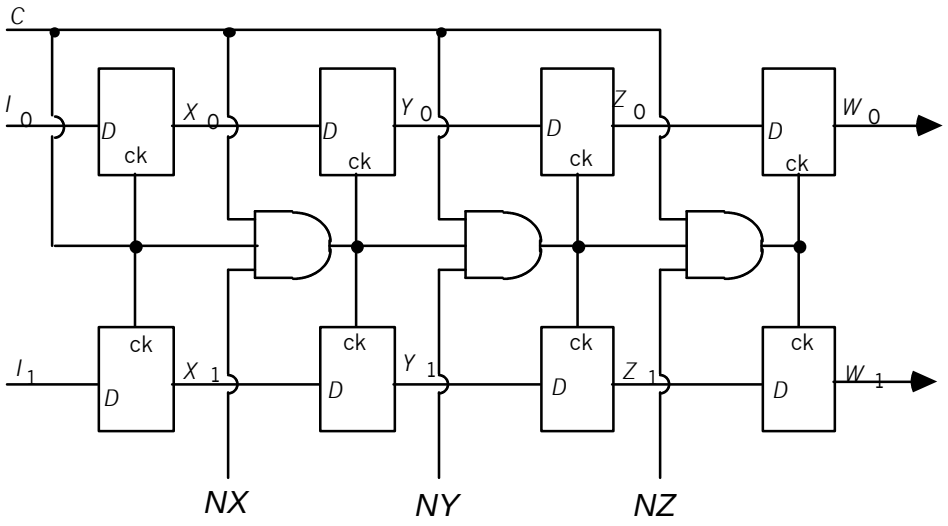
Notice that at each step:

- The line that is referenced moves to the top of the LRU stack.
- All lines below that line keep their same position.
- All lines above that line move down by one position.

Implementation — status flip-flops: Build a piece of hardware based on four 2-bit registers (shown in columns in the diagram below), called *X*, *Y*, *Z*, and *W*.

We will give the algorithm for updating when a hit occurs (when a miss occurs, there is plenty of time to update (the *X* register) with the number of the line that has been purged).

Number of bits of storage required: $E \log_2 E$ bits (where E is the number of elements in the set).



- *I* is the number of the line that has just been accessed.
- The number of the most recently used line is in *X* .
- The number of the least recently used line is in *W* .
- *C* is asserted whenever a cache hit (to this set) occurs.
- *NX* stands for “not *X* ”; it is 1 iff the line that is hit is not line *X.*
- *NY* and *NZ* are defined similarly.

What is the Boolean expression for *NX* ?

Whenever *C* is asserted, it causes the registers to be shifted to the right, until a 0 in *NX*, *NY*, or *NZ* is encountered.

Thus, it implements the LRU algorithm.

For large set sizes (8 or more), this hardware is too expensive to build. So,

- the set of lines is partitioned into several groups.
- the LRU group is determined.
- the LRU element in this LRU group is replaced.

Split caches

Multiple caches can be incorporated into a system design.

Many processors have separate instruction and data caches (I- and D-caches).

Since a combined cache of the same size would have a lower miss ratio, why would anyone want a split cache?

-

•

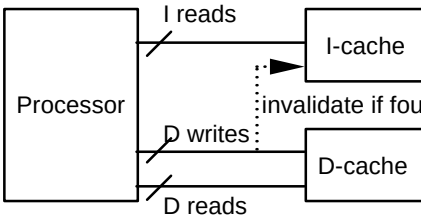
Another complication arises when instructions and data are located in the same cache block.

This is more frequent in old programming environments, where data is sometimes placed in the instruction stream. This tends to happen in Fortran programs.

Duplicate lines must either be supported or prohibited.

If duplicate lines are supported—

- When an I-reference misses, the block is fetched into the _____
- When a D-reference misses, the block is fetched into the _____
- When the processor executes a store, check both directories.
  - If the line is in the D-cache, follow the cache's write policy.
  - If the line is in the I-cache,



If duplicate lines are not supported—

- When an I-reference misses,
  - the block is fetched into the I-cache.
  - if the
- When a D-reference misses,
  - the block is fetched into the D-cache.
  - if the block is present in the I-cache, it is invalidated.

- When the processor executes a store,

Two-level caches

In recent years, the processor cycle time has been decreasing much faster than memory access times.

Cache access times are now often 20–40 times faster than main-memory access times.

What does this imply about cache misses?

What must be done to keep performance from suffering?

But this means that caches must be bigger.
Caches must also become faster to keep up with the processor.

- This means they must be on chip.
- But on-chip caches cannot be very big. (In CMOS they can be reasonably big, but not in GaAs or bipolar.)

The only way out of this dilemma is to build—

- a first-level (L1) cache, which is fast, and
- a second-level (L2) cache, which is big.

A miss in the L1 cache is serviced in the L2 cache, at the cost of a few (2–5) cycles.

To analyze a second-level cache, we use the *principle of inclusion*—a large second-level cache includes everything in the first-level cache.

We can then do the analysis by assuming the first-level cache did not exist, and measuring the hit ratio of the second-level cache alone.

How should the line length in the second-level cache relate to the line length in the first-level cache?

When we measure a two-level cache system, three miss ratios are of interest:

- The *local miss rate* for a cache is the

$$\frac{\text{\# misses experienced by the cache}}{\text{number of incoming references}}$$

  To compute this ratio for the L2 cache, we need to know the number of misses in the L1 cache.

- The *global miss rate* of the cache is

$$\frac{\text{\# L2 misses}}{\text{\# of references made by processor}}$$

  This is the primary measure of the L2 cache.

- The *solo miss rate* is the miss rate the cache would have if it were the only cache in the system.

  It is the miss rate defined by the principle of inclusion.

  If L2 contains *everything* in L1, then we can find the number of misses from analyzing a trace ignoring the presence of the L1 cache.

  If inclusion does not hold, we cannot do this.

The local miss rate is usually higher than the solo or global miss rates. This is no surprise.

What conditions need to be satisfied in order for inclusion to hold?

- The number of L2 sets has to be ≥ the number of L1 sets, irrespective of L2 associativity.

  (Assume that the L2 line size is ≥ L1 line size.)

If this were not true, multiple L1 sets would depend on a single L2 set for backing store. So references to one L1 set could affect the backing store for another L1 set.

- L2 associativity must be ≥ L1 associativity, irrespective of the number of sets.

  Otherwise, more entries in a particular set could fit into the L1 cache than the L2 cache, which means the L2 cache couldn't hold everything in the L1 cache.

- All reference information from L1 is passed to L2 so that it can update its LRU bits.

Even if all of these conditions hold, we still won't have logical inclusion if L1 is write-back. (However, we will still have *statistical inclusion*—L2 *usually* contains L1 data.)

Currently, the fastest processors have L1 and L2 caches on chip, and an L3 cache off-chip.

CPI as a measure of cache performance

(Borg et al., ISCA 1990): The hit ratio is useful for comparing the performance of caches.

But it does not give us a clear picture of the performance impact of the cache on the system.

CPI stands for *average cycles per instruction.*

It can be broken down into components contributed by each part of the memory hierarchy.

Let—

i be the number of instructions in the interval we are measuring,

$m_{cache\text{-}type}$ be the number of misses during the interval, and

$c_{cache\text{-}type}$ be the cost (in cycles) of a miss.

Then a cache of a particular type contributes the following amount to the overall CPI:

$$CPI_{cache\text{-}type} = \frac{m_{cache\text{-}type} \times c_{cache\text{-}type}}{i}$$

In the absence of memory delays, one instruction is executed per cycle.

Delays from each of the caches add to this time.

Supposing a machine has a data cache, an instruction cache, a second-level cache, and a write buffer, the CPI is given by—

$$CPI = 1 + CPI_{data} + CPI_{inst} + CPI_{L2} + CPI_{wtbuf}$$

Example: Ignore the instruction cache and the write buffer.

Assume that

- the data cache has a hit ratio of 90%,
- and the second-level cache has a hit ratio of 99% overall.

It takes

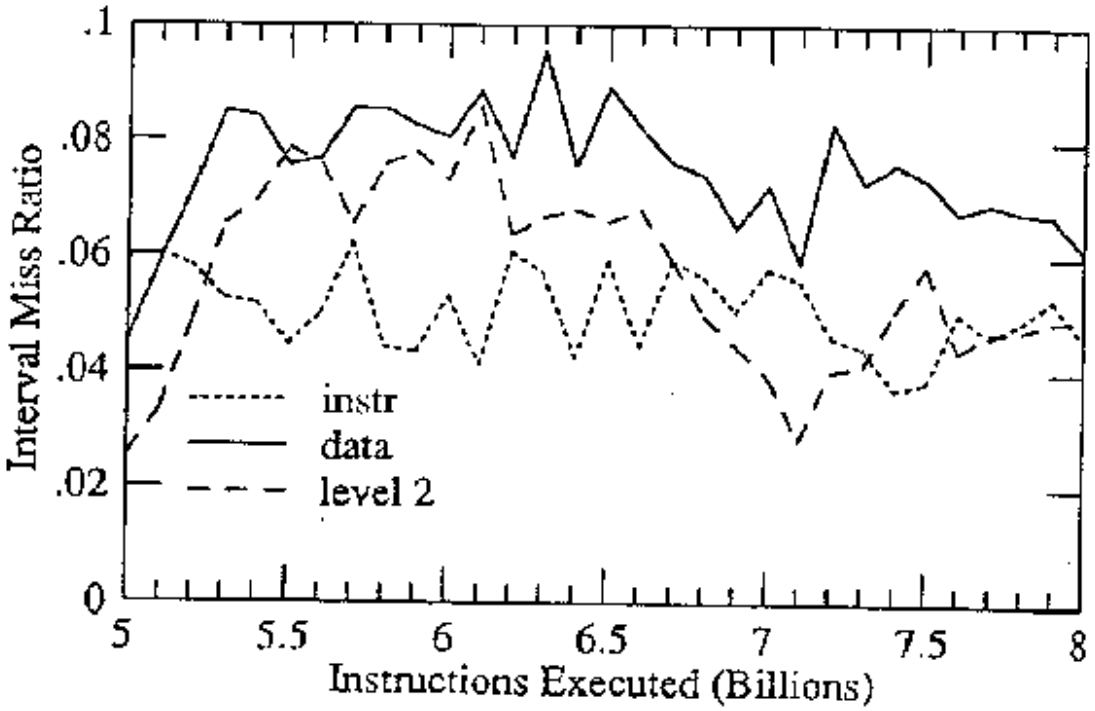
- 1 cycle to fetch data from the data cache,
- 5 cycles to fetch it from the second-level cache, and
- 20 cycles to fetch it from memory.

What is the CPI?

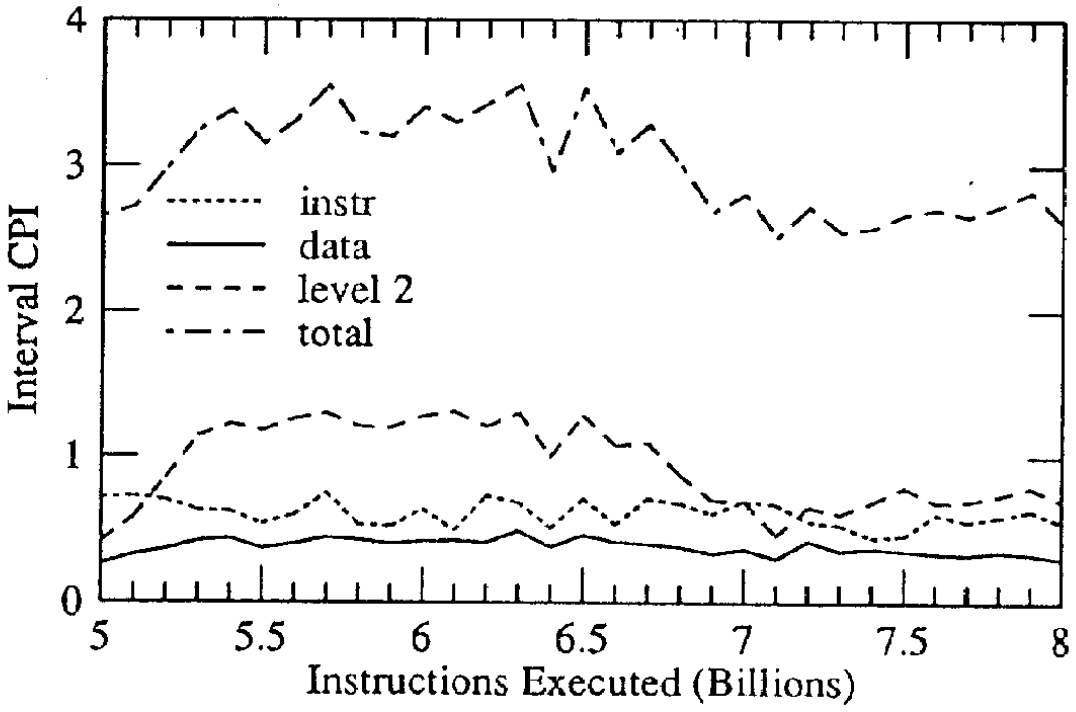
What would it be without the second-level cache?

Borg et al. simulated the degradation from missing the various caches.

The miss-ratio graph (below) shows that the miss ratio for the data cache usually exceeds that for the instruction and second-level caches.



But the second-level cache often contributes twice as much to the overall performance degradation:



Effects of associativity

Cache misses can be divided into four categories (Hill):

- *Compulsory misses* occur the first time that a block is referenced.
- *Conflict misses* are misses that would not occur if the cache were fully associative with LRU replacement.
- *Capacity misses* occur when the cache size is not sufficient to hold data between references.
- *Coherence misses* are misses caused by invalidations to preserve cache consistency in a multiprocessor (we will consider these later).

Conflict misses typically account for 20–40% of misses in a direct-mapped cache.

Clearly, associativity can remove no more misses than this.

However, associativity imposes costs of its own. What are these?

The simulations of Borg et al. showed that if an associative reference is more than 12–28% slower than a direct-mapped reference, a direct-mapped cache is faster.