Why do we have caches?

Our goal is to have a large memory system with a short effective access time.

The basic principles of locality:

1. Spatial locality.
   - Sequential access.

2. Temporal locality.

Cache and processor views

The cache side:

- The access time in the case of a hit.
- The access time in the case of a miss.
- The virtual to real address translation.

The processor side:

- The number of memory references per cycle.
- The size of the "word".

---

Address

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
<th>B/word</th>
</tr>
</thead>
</table>

| Byte address | Word address | Line address |

---
There are three types of misses in the cache:

1. compulsory misses,
2. capacity misses, and
3. conflict misses.

Which of the following techniques help in each case:

- more associativity,
- longer cache lines, and
- bigger cache sizes?
The trend for miss rates

Unfortunately, it tends to increase.

- As memory capacities increase, the program sizes increase. This makes it harder for a cache to hold the working set of the program.

- Programs become more complex over time with greater functionalities.

- The level of multiprogramming and complexity in the O/S increases as well. As the computing environment becomes more complex, the miss rate deteriorates.

Write policies

The cache write policies are

- Write through
  - Write allocate
  - No-write allocate

- Copyback
  - Write allocate
  - No-write allocate

Replacement policies

- LRU
- FIFO
- RAND

How do we design the caching system?

We want the caching to be transparent to the processor and to cause the least overall delay.

The cache is affected by many things from the processor as well as the memory sides.

We want to reduce the miss rate and the miss penalty.

⇒ What is the effect of all the various aspects on the miss rate?
Cache Environment

Modern multiprogramming operating systems present a hard problem for caches.

- In a multiprogrammed environment, the control is passed from program to program, in round-robin fashion, and eventually returns to the first program.

- In transaction processing, the system is resident in memory together with a number of support programs. Very short applications are brought in on demand and run through to completion.

It is possible to have a relatively low number of instructions between task switches.

Why does this low number cause a higher miss rate? Can a larger cache size reduce that miss rate?

Warm and cold caches

A warm cache: is a big enough cache that it may still contain a portion of the working set of a newly reactivated task.

A cold cache: is one that has no history from prior executions.

What happens in the cases of a very large number of tasks and of once-only transactions?

⇒ It is a waste of resources to design a large cache for a transaction environment.

I/O devices affect the cache

For high performance systems, the memory transfers directly to the I/O devices with no intervention from the processor.

- Upon reading from an input device to the memory, update the cache.

- When the memory writes to an output device, we must check that it has the most recent copy. Do we need this in a write-through cache?

We may regard I/O activities as "an additional" process in the system.

Split caches

Pro: Separate instruction and data caches offer the designer the possibility of significantly increased cache bandwidth.

Con: In the unified cache, the ratio of instruction to data working set elements changes during the execution of the program. A unified cache with the same size as the sum of a split data and instruction caches gives a lower effective miss rate.

The issue is “when do you split at design time or at run time?”

Instruction caches have a better spatial locality and usually use larger lines.

⇒ The ‘expected’ miss rates are adjusted differently for instruction and data caches.
Code density

- Denser instruction sets allow the cache to capture the working set of a program more easily.

- Compiler optimizations and good allocation of registers help to improve the “basic” code density of a certain architecture.

⇒ The miss rate is adjusted for the type of instruction set L/S, R/M, and R+M due to the varying code densities.

On-chip caches

For a fixed area, we want to minimize the overhead of the directory.

→ Use larger lines.

What is the effect of a large line on the miss time?

Sectored caches attempt to strike a balance between a low overhead and a short miss time by bringing only the needed part of the line on a miss.

Two caches: miss rates

Local miss rate: is the number of misses experienced by the cache divided by the number of incoming references.

Global miss rate: is the number of L2 misses divided by the number of references made by the processor.

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.

Two caches: inclusion

Does L2 cache include everything that exist in L1 cache? Let us have an example:

Consider a 2-line direct mapped L1 cache with a 4-line fully associative L2 cache and the following reference sequence: $R_{21}, R_{11}, R_{12}, R_{13}$.

Now a reference $R_{14}$ will replace $R_{21}$ in L2 and $R_{13}$ in L1.

Do we want logical inclusion or statistical inclusion?
Write assembly cache

This is useful to buffer memory writes in the case of write through caches. It is an “improved” write buffer.

• It is possible to reduce the write traffic (to the memory) to about one fourth of the value without such a cache.

• For write traffic, temporal is more important than spatial locality.

→ Use many short lines instead of a few large ones.

Traffic

The processor references the cache for instructions and data.

• If the instruction has \( I \) bytes and we fetch \( P \) bytes from the cache to the I-buffer every reference, then (ideally) each instruction causes a traffic of \( I/P \) references.

   \[ \text{In addition to that ideal traffic for the fetch of the following instruction, when does the processor ask for other instructions?} \]

• For data, we need to accommodate for both the read and write traffic.

   – Due to the difference in the code density in various architectures, the same amount of reads and writes in an algorithm translates to different average references per instruction in the different architectures.

   – If the instruction set has an instruction to load or store multiple registers (Why do we need those?), we must accommodate them as well.

Technology

The effect of caching is measured in two ways.

1. The effective memory access time.

   \[ T_{\text{eff mem}} = T_{\text{cache hit}} + \text{miss rate} (T_{\text{access}} + (\text{words per line} - 1)T_{\text{bus}}) \]

2. The CPU time per instruction for programs.

   \[ t_{\text{cycle}} \times \left[ \frac{\text{CPI}}{\text{instruction}} + \frac{\text{references}}{\text{instruction}} \times (\text{miss rate}) \times (\text{miss penalty}) \right] \]

Both measures are affected by the technology used.

The TLB and the cache

• The TLB holds the most recent translations of virtual page numbers to physical page numbers.

• In a simple scheme, we translate the address first then we access the cache.

The best case is to hit in both the TLB and the cache. A worse case is to

1. miss in the TLB,
2. go to the page table in the memory and find out the translation,
3. miss in the cache, and
4. get the data from the memory.

\[ \text{Do I hear someone saying something about the disk?} \]
Parallel access to TLB and cache

If the cache (set) size is less than the page size, we can work in parallel.

- For a page size of 4 kB we need 12 bits for the page offset.
- These offset bits remain the same in both the virtual and real address, hence we forward them to the cache.
- If the cache is direct-mapped and less than 4 kB, those bits are sufficient to index the line.
- In parallel, the TLB translates the virtual page number and the resultant physical page number is compared to the tag coming from the cache.

If we wanted a larger cache size, we can increase the associativity. For this example, a 16 kB with 4 ways associativity works well.

Virtual cache

We have two problems

- two different programs may use the same virtual address (each in its own address space) to access different real locations.
- two processes who want to communicate to a single real address use two different virtual addresses.

To solve this aliasing, we may

- purge the cache when a process switch occurs or
- add a process identifier to the tag bits in the cache.

What are the advantages and disadvantages of each solution?

Colored pages

Virtual page number | Virtual page offset | Virtual address
↓ ↓
Physical page number | Physical page offset | Physical address

<table>
<thead>
<tr>
<th>Tag</th>
<th>Color</th>
<th>bits</th>
<th>line</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>index</td>
<td>→</td>
<td>index</td>
<td></td>
</tr>
</tbody>
</table>

The OS chooses the real location of the virtual address so that the "colors" match. This technique

- allows for a parallel access of the TLB and the cache but
- causes a higher swapping rate as the memory nears full occupation.

Design rules

The miss rate is affected by many things:

- the size of the cache, its organization, and its line size;
- the write and replacement policies;
- the environment (multiprogramming, short transactions);
- the traffic characteristics (depends on the type of instruction set and code density);
- the type of the cache (unified, I-cache, D-cache, Write-assembly); and
- the level of the cache.

Excess CPI:

1. Find the cache miss rate, the references per instruction (instructions and data), and the miss penalty in cycles.
2. Multiply the three to get the excess CPI due to cache.
3. Similarly, get the additional delay due to the TLB.

The goal is to reduce the effect on the processor, not just miss rate or miss penalty.
Conclusions

• Caches operate on spatial and temporal locality.

• Caches attempt to provide the processor with the illusion of a large and fast memory.

• Many factors affect the miss rate of a cache.