Average Memory Access time =

Hit time +
Miss Rate \times Miss penalty

To reduce it, we can target each of these factors, and study how these can be reduced.
Improving Average Memory Access Time: Reducing Hit Time

**Method 1.** Use direct-mapped cache.

**Method 2.**
To improve the hit time for reads,
- Overlap tag check with data access.
- Discard data if tag does not match.

**Method 3.**
To improve the hit time for writes,
Pipeline write hit stages

- **Write 1**
  - TC
  - W

- **Write 2**
  - TC
  - W

- **Write 3**
  - TC
  - W

\[\text{time}\]
**Method 4.**

Write buffers speed up write-through caches.

![Diagram of write-through cache with processor, cache, write buffer, and main memory connections.]

**Processor:** Cache \(\rightarrow\) data

**Write buffer:** \(\rightarrow\) data

**Buffer controller:** Main Memory \(\rightarrow\) write buffer

Processor is free after updating the write buffer.
Optimization with write buffer

**Possibility of write merge.** If an \((x:=5)\) is issued after an \((x:=10)\) that is still in the write buffer, then the second write could merge with (i.e. overwrite) the first write, before M is updated.

**Possibility of early reading**

Data can be read unfinished previous writes still pending in the write buffer, although this can potentially complicate life.
Method 5. **Virtual Cache**

*Only for* systems supporting **virtual memory**

- **Virtual address** is translated to **physical address**.
- **Physical address** is used to check **cache tags**.

*Why not store the **virtual addresses** as tags?*

After context switching
- Either the cache needs to be flushed, since the new process may try to use the old page numbers.
- Or the tags must contain the process ids that can be used to distinguish between processes.
Reducing Miss Penalty

**Method 1:** *Give priority to read miss over write.*

Consider a direct mapped cache using write-through. Assume that addresses 512 and 1024 map to the same cache block.

\[
\begin{align*}
\text{M}[512] & \rightarrow R3; \quad \text{*value of R3 in write buffer*} \\
R1 & \rightarrow \text{M}[1024]; \quad \text{*read miss, fetch M[1024]*} \\
R2 & \rightarrow \text{M}[512]; \quad \text{*read miss, fetch M[512]*} \\
& \quad \text{*value of R3 not yet written*} \\
& \quad \text{*R2 } \neq \text{ R3*}
\end{align*}
\]

*Read miss must wait until the write buffer is empty.*

To reduce the wait, let read miss check the write buffer. If there is no conflict, read M to get the data. Else, read from the write buffer.
**Method 2**

*Early restart and Critical word first*

Do not wait for the whole block to be loaded into the cache. As soon as the requested word arrives, send it to the CPU.

Request to transfer the *missing word* first. Let CPU continue while the rest of the cache block is being filled.

**Method 3.**

*Use additional levels of cache (L2, L3 etc)*
Method 4. *Nonblocking cache*

**Instruction 1**  Cache Miss

**Instruction 2**  This is a hit But should it wait for 1?

In *dynamic instruction scheduling*, a stalled instruction does not necessarily block the subsequent instructions. So, instruction 2 can pass instruction 1.

*Example of Hit under Miss*

With a *packet switched bus (i.e. split-transaction bus)*, it is possible to implement *hit under miss under miss*.
Support for Nonblocking Cache

Extensively used in high performance computer systems. Miss buffer stores the missing cache lines from M until these are transferred to the cache by the cache controller.
What is cache coherence problem?

A first look

Consider the scenario:

Initially, $x1 = x2 = X = 5$.

P1 writes $X := 10$ using write-through.

P2 now reads $X$ and uses its local copy $x2$, but finds that $X$ is still 5.

P2 does not know that P1 modified $X$. 
Impact of cache on I/O

**Configuration 1**
No cache coherence problem, but there is a risk of data overrun, when the controller is incapable of handling the data traffic.

**Configuration 2**
Coherence problem exists with write-back cache.
A solution to the cache coherence problem

When the processor (or the I/O device) writes into a shared block, every other copy of it is considered dirty. The memory controller maintains a list of dirty blocks.

When the I/O device (or the processor) wants to read a dirty block, the memory controller supplies a clean copy of the dirty block from the main memory.