Memory Hierarchy

- Why?
  - Memory hierarchies take advantage of "memory locality".
  - i.e. future memory accesses are near past memory accesses.

- Types of locality:
  - Temporal: we access the same data often.
  - Spatial: our next mem access is very close to our last accesses.
• So... Why is this important?

  - We exploit locality by "caching" data that is likely to be used again by the CPU.
  - Think of cache as a place to store/keep data close to the processor.

• Our memory hierarchy will look like:

```
+-----------------+                  +-----------------+
| Size /          |                  | Cost             |
| Access Time     |                  |                  |
+-----------------+                  +-----------------+
    | DGS             |                  |                  |
    | on-chip cache   |                  |                  |
    | off-chip cache  |                  |                  |
    | main memory     |                  |                  |
    | virtual memory / disk |          |                  |
```
A memory access will look like:

\[ \text{CPU} \rightarrow \text{L1} \rightarrow \text{L2} \rightarrow \text{MM} \]

- **Hit**: desired data found at desired level.
- **Miss**: desired data not found at desired level.

**Hit time**: time to access, if found, data.
(usually a particular level)

**Miss penalty**: time to bring desired data to current level after a miss + deliver data to CPU.

\[ i.e. \, \text{Miss time} = \text{Access time} + \text{Transfer time} \]

where:
- **Access time**: memory latency, or access to next level.
- **Transfer time**: multiple transfer for larger data sizes.

**Hit ratio** = \% time data is in cache

**Miss ratio** = ?
EXAMPLE 1

Given the following architecture:

\[ \text{MP} \leftrightarrow \text{Cache} \leftrightarrow \text{RAM} \]

Assume we have a program with 100 mem references. With the following distribution:

- 75 cache hits
- 25 cache misses

\[ T_{\text{total}} = (\# \text{hits}) \cdot (t_{\text{hit}}) + (\# \text{misses}) \cdot (t_{\text{miss penalty}}) \]

\[ T_{\text{total}} = (75) \cdot (1) + (25) \cdot (10 + 1) = 350 \text{ cycles} \]

\[ t_{\text{average}} = \frac{T_{\text{total}}}{100} = \frac{350}{100} = 3.5 \text{ cycles} \]
* Yeah that's interesting, so what?

- Main problem: Cache is small, and MM is big!
  we must define a function that maps many elements of the main memory, to any one element in the cache.

- Cache unit: \underline{Block or line}

- Usually a Block = \( x \) words \((x \gg 1)\), why?

- Block is usually the atomic unit of transfer from MM.

- The MM->cache position mapping policy = \underline{Associativity}

- TYPES: (assume a cache with \( C \) blocks, or \( S \) sets of \( N \) blocks)

  1. \underline{Fully Associative}: Block \( Y \) from MM can go to any block in cache.
  2. \underline{Direct Mapped}: Block \( Y \) from MM can only go to block \((Y \mod \underline{C})\) in cache.

  3. \underline{S-Set Associative}: Block \( Y \) from MM can go to anywhere in set \((Y \mod \underline{S})\) in the cache.
SOME EXAMPLES

DIRECT MAPPED:

ASSUME 64KB cache w. 32-byte block size
2-WAY SET ASSOCIATIVE

32kB cache, 16-Byte block size

---

use 4/12 signals to decide to which block in the set to look.

* ASSOCIATIVE vs. DIRECT MAPPED?
Food for Thought:

- Large blocks useful to exploit spatial locality.
- If the block size is too large? We may waste space!
- However the longer the block, the smaller the tag space.

Modus Operandi:

1. Tag + Index → Access cache & determine Hit/Miss.
2. If Hit then return requested data.
3. If Miss then:
   3.1. Select a cache block to be replaced
   3.2. Access next Memory level (still?)
   3.3. Load entire missed line into cache
   3.4. Return data (Everyone is happy now!)
4. If next memory level is also a cache, then go to step 1. for that cache.
Hum... but what about stores?

- Stores may not require stalls
- Consistency between cache/memory must be enforced!

  1. **Write Through**: all writes go to cache + memory at the same time.

  2. **Write Back**: writes go 1st to cache, lines are sent back to memory only when they're about to be replaced.

- Allocate space in cache if store-miss

  1. **Write Allocate**: bring written line into cache
  2. **Write Around**: ignore cache
REPLACEMENT ISSUES

• What happens to a line in cache when we bring new data to it?

@ DIRECT MAPPED:
  - No choice, we can only replace 1 specific line
  - Write data back to memory, (again no choice)

@ ASSOCIATIVE:
  - Several choices for the location of the data
  - Must establish a replacement policy.
    (i.e. FIFO, LIFO, LRU, Random, etc.)

• Write Data back
  - Write data back always (safest).
  - Write only "dirty" data back.

• Note: Replacement policy is only required if cache is associative!


**CACHE PERFORMANCE**

\[ \text{CPI}_{\text{total}} = \text{CPI}_{\text{base}} + \text{CPI}_{\text{memory}} \]

\[ \text{CPI}_{\text{memory}} = \# \text{ of stall cycles due to memory accesses} \]

\[ \text{CPI}_{\text{memory}} = \left( \frac{\text{accesses}}{\text{instruction}} \right) \times (\text{miss-rate}) \times (\text{miss-penalty}) \]

*assuming:
  - pipeline stalls on reads and write misses
  - miss penalty for read/write misses is the same
  - cache hits have no stalls*
VIRTUAL MEMORY!

- It's just like caching, I promise!
- Simply use Main Memory as a cache for a larger memory, i.e., the hard disk!
- Of course to make it more complex "they" introduced new terminology:

\[
\begin{array}{c|c}
\text{cache} & \text{Virtual Memory} \\
\hline
\text{block} & \rightarrow \text{Page} \\
\text{cache miss} & \rightarrow \text{Page fault} \\
\text{address} & \rightarrow \text{Virtual Address} \\
\text{index} & \rightarrow \text{physical Address}
\end{array}
\]

- note: the atomic unit of transfer is the page. It's usually determined by the OS.
  Normally Page size \(\geq\) Cache block size.
VIRTUAL MEMORY vs. CACHING

- VM is a **functionality enhancement** not a **performance enhancement**
- Pages are at least one order of magnitude larger than cache/memory blocks.
- Usually the mapping of pages to Main Memory is **fully associative**.
- Only write back is allowed, why?
- Miss penalty are way larger.
- Misses are handled by software (but not hits).

---

[Diagram showing virtual address mapping to physical memory and disk]

Valid bit → V.Addr → Physical Address

Virtual Address → Disk
VIRTUAL ADDRESS TRANSLATION USING PAGE TABLES:

Virtual Address: \[ \text{VIRTUAL PAGE NUMBER} \quad \text{PAGE OFFSET} \]

Page Table:

Valid?

Physical Page Number

\[ \text{PHYSICAL PAGE \#} \quad \text{PAGE OFFSET} \quad : \text{Physical Address} \]

- All page mappings must be in the table
- No tag needed
- Hit/Miss? Just look at the \text{Valid} Bit
- Fully \text{ASSOCIATIVE}?
Making the Common Case Fast:

Can we speed up address translation?

- Yes! Use TLB!!
- TLB = Translation Lookaside Buffer
- How does the TLB interact with cache?

1. CPU sends virtual address
2. TLB translates to physical (we may have a page fault, thus we must reload!)
3. If hit in TLB, then just pass addr. to cache
4. Cache returns data (again we may have a cache fault/miss)
- TLB is really an assoc. cache.

- How does it interact with cache?

**TLB**: Send to TLB and check w. TAG

- Physical Page # | Page Offset

  - now it becomes a physical Addr for the cache!

- Add: TAG, INDEX, OFFSET

  - SEND TO CACHE!
Data Flow for this Mess

Virtual Address

TLB

TLB Miss Exception

TLB hit?

Yes

Write?

No

Cache miss!

Stall

Cache hit?

No

write access bit?

No

write protection exception

Yes

write data to cache, update tag, put data and address into the write buffer

Yes

return data to CPU

No

try to read data from cache

return data to CPU
EXAMPLE: Given the following machine

<table>
<thead>
<tr>
<th></th>
<th>TLB</th>
<th>L1</th>
<th>L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>16</td>
<td>8kB</td>
<td>128kB</td>
</tr>
<tr>
<td>Block Size</td>
<td>8kB</td>
<td>32B</td>
<td>32B</td>
</tr>
<tr>
<td>Associativity</td>
<td>16</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Miss Penalty</td>
<td>200</td>
<td>60</td>
<td>150</td>
</tr>
<tr>
<td>Hit time</td>
<td>0</td>
<td>2</td>
<td>60</td>
</tr>
</tbody>
</table>

CODE 1

```c
EIGHTBYTEDOUBLE M[64][64][64];
int i, j, k;
for (i=0; i<64; i++)
    for (j=0; j<64; j++)
        for (k=0; k<64; k++)
            M[i][j][k] += 1.0;
```

CODE 2

```c
EIGHTBYTEDOUBLE M[64][64][64];
int i, j, k;
for (k=0; k<64; k++)
    for (j=0; j<64; j++)
        for (i=0; i<64; i++)
            M[i][j][k] += 1.0;
```
Q. **TLB ADDRESS:**

\[
\begin{array}{c|c}
31 & 13 \ 12 \ 0 \\
\hline
\text{TAG} & \text{PAGE OFFSET} \\
\end{array}
\]

Q. **L1 ADDRESS:**

\[
\begin{array}{c|c|c}
31 & 12 & 11 \ 54 \ 0 \\
\hline
\text{TAG} & \text{INDEX} & \text{OFFSET} \\
\end{array}
\]

Q. **L2 ADDRESS:**

\[
\begin{array}{c|c|c|c}
31 & 16 & 15 \ 54 \ 0 \\
\hline
\text{TAG} & \text{INDEX} & \text{OFFSET} \\
\end{array}
\]

Q. **Average access time for data accesses?**

\[\text{Average Data}: \, MP_{\text{TLB}} \cdot M_{\text{TLB}} + 2 + MP_{\text{HitL1}} \left( 60 + MP_{\text{HitL2}} \cdot 150 \right)\]
Q: What is the total data memory access time for code 1?

A: Note that the memory accesses are purely sequential!

Thus we will have a miss once per page & once per cache line.

# of memory accesses = \(2 \times 2^6 \times 2^6 \times 2 = 2^{19}\) total mem accesses

Assume: each cache line holds 4 doubles.

Each page holds \(2^{10}\) doubles.

\[
\text{total time} = 2^{19} \times 2\text{ns} + \frac{2^{18}}{4} \left(\frac{60 + 150}{2^{10}}\right) + \frac{2^{18}}{2^{10}} \left(\frac{200}{2^{10}}\right) \text{TLB penalty}
\]

\[
= 2^{19} \times 2\text{ns} + \frac{2^{18}}{4} \left(\frac{210}{2^{10}}\right) + \frac{2^{18}}{2^{10}} \left(\frac{200}{2^{10}}\right) \text{TLB penalty}
\]

easy!
A: What is the total data access time for code 2?

A: Note now data accesses are not sequential!

How much memory are we using?

$2^{18}$ elements in the array, each $2^8$ bytes

thus we are using $2^{24}$ bytes

How much does the TLB fit?

$2^4, 2^{13} = 2^{17}$

However by looking at the access pattern, we see that we have a TLB miss + cache miss every mem access cread at least 1024 byte strides!

so:

$T_{total} = 2^{18} \left( 2 + 60 + 150 + 200 \right)$