Inside out of your computer memories

Hung-Wei Tseng
The memory gap problem
Stored-program computer

Processor

PC

instruction memory

120007a30: 0f00bb27 ldah gp,15(t12)
120007a34: 509cbd23 lda gp,-25520(gp)
120007a38: 00005d24 ldah t1,0(gp)
120007a3c: 0000bd24 ldah t4,0(gp)
120007a40: 2ca422a0 ld1 t0,-23508(t1)
120007a44: 130020e4 beq t0,120007a94
120007a48: 00003d24 ldah t0,0(gp)
120007a4c: 2ca4e2b3 stl zero,-23508(t1)
120007a50: 0004ff47 clr v0
120007a54: 28a4e5b3 stl zero,-23512(t4)
120007a58: 20a421a4 ldq t0,-23520(t0)
120007a5c: 0e0020e4 beq t0,120007a98
120007a60: 0204e147 mov t0,t1
120007a64: 0304ff47 clr t2
120007a68: 0500e0c3 br 120007a80
Why memory hierarchy?

The access time of DDR3-1600 DRAM is around 50ns, 100x to the cycle time of a 2GHz processor! SRAM is as fast as the processor, but $$$
The memory hierarchy in “inside out”

Islands (long-term memory)

Core Memory

Short-term Memory
Memory hierarchy

Fastest, Most Expensive

<table>
<thead>
<tr>
<th>Level</th>
<th>Access Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>&lt; 1ns</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>&lt; 1ns ~ 20 ns</td>
<td></td>
</tr>
<tr>
<td>Main Memory</td>
<td>50-60ns</td>
<td></td>
</tr>
<tr>
<td>Secondary Storage</td>
<td>10,000,000ns</td>
<td></td>
</tr>
</tbody>
</table>
Memory hierarchy

- **CPU**
- **Main Memory**
- **Secondary Storage**

Access time:
- Fastest, Most Expensive: <1ns
- $<1ns$ ~ 20 ns
- 50-60 ns
- 10,000,000 ns
Locality

• Temporal Locality
  • Referenced item tends to be referenced again soon.

• Spatial Locality
  • Items close by referenced item tends to be referenced soon.
    • example: consecutive instructions, arrays

• Let’s see how to build a memory hierarchy with “cache” that exploits “both” locality
## Demo revisited

```c
for(i = 0; i < ARRAY_SIZE; i++)
{
    for(j = 0; j < ARRAY_SIZE; j++)
    {
        c[i][j] = a[i][j] + b[i][j];
    }
}
```

```c
for(j = 0; j < ARRAY_SIZE; j++)
{
    for(i = 0; i < ARRAY_SIZE; i++)
    {
        c[i][j] = a[i][j] + b[i][j];
    }
}
```

<table>
<thead>
<tr>
<th>Array_size = 1024, 0.048s</th>
<th>Array_size = 1024, 0.252s</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5.25X faster)</td>
<td></td>
</tr>
</tbody>
</table>
Cache organization
Cache

• Like a cheat-sheet for the processor
• For a cheat-sheet, you may need to put
  • Most frequently asked concepts (temporal locality)
  • Problems, key points related to the frequently asked topics (spatial locality)
How do you make a cheatsheet?

- Go through your homework
- Write down the topic and content
- If running out of space: kick out the least recently used content

1. Performance equation
2. Amdahl’s law
3. MIPS
4. Power consumption
5. Performance equation 😊
6. Amdahl’s law 😊
7. MFLOPS

<table>
<thead>
<tr>
<th>Tag: the address prefix of data in the cacheline/block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance equation</td>
</tr>
<tr>
<td>Amdahl’s law</td>
</tr>
<tr>
<td>MFLOPS</td>
</tr>
<tr>
<td>Power consumption</td>
</tr>
</tbody>
</table>

Cacheline/block: data with the same prefix in their addresses
Let’s make memory great again!

- **Spatial locality**
  - Each hash entry contains a block of data
  - We bring a “block” of data each time
  - Cache blocks are a power of 2 in size.
  - Usually between 16B-128Bs
  - Tag: help us identify what’s in the block

- **Temporal locality**
  - LRU-like polices keeps the most frequently used data
A simple cache: a block can go anywhere

- Assume each block contains 16B data
- A total of 4 blocks
- LRU

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x4</td>
<td>0b000000100</td>
<td>0b000000000 - 0b00001111</td>
</tr>
<tr>
<td>2</td>
<td>0x48</td>
<td>0b01001000</td>
<td>0b010000000 - 0b01001111</td>
</tr>
<tr>
<td>3</td>
<td>0xC4</td>
<td>0b11001000</td>
<td>0b111100000 - 0b11111111</td>
</tr>
<tr>
<td>4</td>
<td>0xFC</td>
<td>0b11111000</td>
<td>0b011100000 - 0b01111111</td>
</tr>
<tr>
<td>5</td>
<td>0x12</td>
<td>0b00001100</td>
<td>0b010000000 - 0b01001111</td>
</tr>
<tr>
<td>6</td>
<td>0x44</td>
<td>0b01000100</td>
<td>0b111100000 - 0b11111111</td>
</tr>
<tr>
<td>7</td>
<td>0x68</td>
<td>0b01100100</td>
<td>0b011100000 - 0b01111111</td>
</tr>
</tbody>
</table>

- Too slow if the number of entries/blocks/cachelines is huge
Let’s make memory great again!

• Spatial locality
  • Each hash entry contains a block of data
  • We bring a “block” of data each time
  • Cache blocks are a power of 2 in size.
  • Usually between 16B-128Bs
  • Tag: help us identify what’s in the block

• Temporal locality
  • LRU-like polices keeps the most frequently used data

• Performance needs to be better than linear search
  • Make cache a hardware hash table!
  • The hash function takes memory addresses as inputs
The structure of a cache

**Set:** cache blocks/lines sharing the same index. A cache is called N-way set associative cache if N blocks share the same set/index (this one is a 2-way set cache)

<table>
<thead>
<tr>
<th>Valid</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
<th>Valid</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 0</td>
<td>1000 0001 0000 1000 0000</td>
<td></td>
<td></td>
<td>1 1</td>
<td>1000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

**Tag:**
the high order address bits stored along with the data in a block to identify the actual address of the cache line.

**valid:** if the data is meaningful
**dirty:** if the block is modified

**Block / Cacheline:** The basic unit of data storage in cache. Contains all data with the same tag/prefix and index in their memory addresses.
Accessing the cache

Hit: The data was found in the cache
Miss: The data was not found in the cache

Offset: The position of the requesting word in a cache block

Hit? miss?

memory address: 0x8 0 0 0 0 0 1 5 8

tag
index offset

The position of the requesting word in a cache block

Hit: The data was found in the cache
Miss: The data was not found in the cache

Offset: The position of the requesting word in a cache block

memory address: 0x8 0 0 0 0 0 1 5 8

tag
index offset

The position of the requesting word in a cache block
How many bits in each field?

- \( \lg(\text{number of sets}) \)
- \( \lg(\text{block size}) \)

Block / cacheline: ?

Diagram showing the relationship between valid, dirty, tag, index, offset, data, and hit?
C = ABS

- **C**: Capacity in data arrays
- **A**: Way-Associativity
  - N-way: N blocks in a set, $A = N$
  - 1 for direct-mapped cache
- **B**: Block Size (Cacheline)
  - How many bytes in a block
- **S**: Number of Sets:
  - A set contains blocks sharing the same index
  - 1 for fully associative cache
Corollary of $C = \text{ABS}$

- offset bits: $\log_2(B)$
- index bits: $\log_2(S)$
- tag bits: $\text{address\_length} - \log_2(S) - \log_2(B)$
  - $\text{address\_length}$ is 32 bits for 32-bit machine
- $(\text{address} / \text{block\_size}) \mod S = \text{set index}$
Multi-layer caches

- Speed of L1 matches the processor
- Caches data/code as many as possible in L2/L3 to avoid DRAM accesses

```
CPU

Main Memory

Secondary Storage
```

```
L1 $

L2 $

L3 $
```
### Array of structures or structure of arrays

<table>
<thead>
<tr>
<th></th>
<th>Array of objects</th>
<th>object of arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct grades</td>
<td>struct grades</td>
<td>struct grades</td>
</tr>
<tr>
<td></td>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>int id;</td>
<td>int *id;</td>
</tr>
<tr>
<td></td>
<td>double *homework;</td>
<td>double **homework;</td>
</tr>
<tr>
<td></td>
<td>double average;</td>
<td>double *average;</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

**average of each student**

**average of each homework**
How cache interacts with CPU
What happens on a read?

- Read hit
- Hit time
- Read miss?
  - Select victim block
    - LRU, random, FIFO, ...
    - Write back if dirty
  - Fetch Data from Lower Memory Hierarchy
    - As a unit of a cache block
      - Data with the same “block address” will be fetch
    - Miss penalty
What happens on a write? (Write Allocate, write back)

- Write hit?
  - Update in-place
  - Set dirty bit (Write-Back Policy)
- Write miss?
  - Select victim block
    - LRU, random, FIFO, ...
    - Write back to lower memory hierarchy if dirty
  - Fetch Data from Lower Memory Hierarchy
    - As a unit of a cache block
    - Miss penalty
Simulate a 2-way cache

- Consider a 2-way cache with 16 blocks (8 sets), a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000
  - \(8 = 2^3\) : 3 bits are used for the index
  - \(16 = 2^4\) : 4 bits are used for the byte offset
  - The tag is 32 - (3 + 4) = 25 bits
  - For example: 0b1000 0000 0000 0000 0000 0000 0000 0001 0000

- 34
- 0b10000000000000000000000000000000

- tag

- index

- offset
Simulate a 2-way cache

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0b100</td>
<td>1</td>
<td>0b100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0b100</td>
<td>1</td>
<td>0b110</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10 0000 0000</td>
<td>miss</td>
</tr>
<tr>
<td>0b10 0000 1000</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0001 0000</td>
<td>miss</td>
</tr>
<tr>
<td>0b10 0001 0100</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0001 0100</td>
<td>hit!</td>
</tr>
<tr>
<td>0b11 0001 0000</td>
<td>miss</td>
</tr>
<tr>
<td>0b10 0000 0000</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0000 1000</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0001 0000</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0001 0100</td>
<td>hit!</td>
</tr>
</tbody>
</table>
Special case: a direct-mapped cache

Block (cacheline): The basic unit of data storage in cache. Contains all data with the same tag and index in their address

memory address:

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 0000 0000 0000 0000</td>
<td>0001 01</td>
<td>01 1000</td>
</tr>
</tbody>
</table>

Tag: the high order address bits stored along with the data to identify the actual address of the cache line.

Hit: The data was found in the cache
Miss: The data was not found in the cache
Simulate a direct-mapped cache

- Consider a direct mapped (1-way) cache with 16 blocks, a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000

- $16 = 2^4$: 4 bits are used for the index
- $16 = 2^4$: 4 bits are used for the byte offset
- The tag is $32 - (4 + 4) = 24$ bits
- For example: 0b1000 0000 0000 0000 0000 0000 1000 0000
Simulate a direct-mapped cache

<table>
<thead>
<tr>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0b10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0b10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0000</td>
</tr>
<tr>
<td>0b10</td>
<td>1000</td>
</tr>
<tr>
<td>0b10</td>
<td>0000</td>
</tr>
<tr>
<td>0b10</td>
<td>0100</td>
</tr>
<tr>
<td>0b11</td>
<td>0000</td>
</tr>
<tr>
<td>0b10</td>
<td>0000</td>
</tr>
<tr>
<td>0b10</td>
<td>0000</td>
</tr>
<tr>
<td>0b10</td>
<td>0010</td>
</tr>
</tbody>
</table>

- miss
- hit!
Conflict in direct-mapped cache

If we have two frequently used cache blocks:

If they are usually used back-to-back, one will kick out the other all the time.
Way associativity and cache performance

The graph illustrates the relationship between associativity and miss rate for cache sizes ranging from 1 KB to 128 KB. The x-axis represents different levels of associativity (One-way, Two-way, Four-way, Eight-way), while the y-axis represents the miss rate. Each line corresponds to a specific cache size, showing a decrease in miss rate as associativity increases.
Pros & cons of way-associate caches

• Help alleviating the hash collision by having more blocks associating with each different index.
  • N-way associative: the block can be in N blocks of the cache
• Fully associative
  • The requested block can be anywhere in the cache
  • Or say N = the total number of cache blocks in the cache
• Slower
  • Increasing associativity requires multiple tag checks
  • N-Way associativity requires N parallel comparators
  • This is expensive in hardware and potentially slow.
  • This limits associativity L1 caches to 2-8.
  • Larger, slower caches can be more associative
Performance evaluation considering cache
Performance evaluation considering cache

• If the load/store instruction hits in L1 cache where the hit time is usually the same as a CPU cycle
  • The CPI of this instruction is the base CPI

• If the load/store instruction misses in L1, we need to access L2
  • The CPI of this instruction needs to include the cycles of accessing L2

• If the load/store instruction misses in both L1 and L2, we need to go to lower memory hierarchy (L3 or DRAM)
  • The CPI of this instruction needs to include the cycles of accessing L2, L3, DRAM
How to evaluate cache performance

- **CPI\textsubscript{Average}**: the average CPI of a memory instruction

  \[ \text{CPI}_{\text{Average}} = \text{CPI}_{\text{base}} \times \text{miss\_rate}_{L1} \times \text{miss\_penalty}_{L1} \]

- \text{miss\_penalty}_{L1} = \text{CPI}_{\text{accessing\_L2}} \times \text{miss\_rate}_{L2} \times \text{miss\_penalty}_{L2} \]

- \text{miss\_penalty}_{L2} = \text{CPI}_{\text{accessing\_L3}} \times \text{miss\_rate}_{L3} \times \text{miss\_penalty}_{L3} \]

- \text{miss\_penalty}_{L3} = \text{CPI}_{\text{accessing\_DRAM}} \times \text{miss\_rate}_{\text{DRAM}} \times \text{miss\_penalty}_{\text{DRAM}} \]

- If the problem is asking for **average memory access time**, transform the CPI values into/from time by multiplying with CPU cycle time!
Average memory access time

- Average Memory Access Time (AMAT) = Hit Time + Miss rate \times Miss penalty
- Miss penalty = AMAT of the lower memory hierarchy
- AMAT = \text{hit\_time}_{L1} + \text{miss\_rate}_{L1} \times AMAT_{L2}
  - AMAT_{L2} = \text{hit\_time}_{L2} + \text{miss\_rate}_{L2} \times AMAT_{DRAM}
Cache & Performance

- Application: 80% ALU, 20% Loads
- L1 I-cache miss rate: 5%, hit time: 1 cycle
- L1 D-cache miss rate: 10%, hit time: 1 cycle
- L2 U-Cache miss rate: 20%, hit time: 10 cycles
- Main memory hit time: 100 cycles
- What’s the average CPI?

CPI_{Average} = CPI_{base} + \text{miss}_\text{rate} \times \text{miss}_\text{penalty}

= 1 + 100\% \times (5\% \times (10 + 20\% \times (1 \times 100)))

+ 20\% \times (10\% \times (1) \times (10 + 20\% \times ((1) \times 100)))

= 3.1
Cache & Performance

- Application: 80% ALU, 20% Load/Store
- L1 I-cache miss rate: 5%, hit time: 1 cycle
- L1 D-cache miss rate: 10%, hit time: 1 cycle, 20% dirty
- L2 U-Cache miss rate: 20%, hit time: 10 cycles, 10% dirty
- Main memory hit time: 100 cycles
- What’s the average CPI?

\[
\text{CPI}_{\text{Average}} = \text{CPI}_{\text{base}} + \text{miss} \_\text{rate} \times \text{miss} \_\text{penalty}
\]

\[
= 1 + 100\% \times (5\% \times (10 + 20\% \times ((1 + 10\%) \times 100))
\]

\[
+ 20\% \times (10\% \times (1 + 20\%) \times (1 + 20\% \times ((1 + 10\%) \times 100)))
\]

\[
= 3.368
\]
Cause of cache misses
3Cs of misses

• Compulsory miss
  • Cold start miss. First-time access to a block

• Capacity miss
  • The working set size of an application is bigger than cache size

• Conflict miss
  • Required data replaced by block(s) mapping to the same set
  • Similar collision in hash
Simulate a 2-way cache

- Consider a 2-way cache with 16 blocks (8 sets), a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000
  - \( 8 = 2^3 \) : 3 bits are used for the index
  - \( 16 = 2^4 \) : 4 bits are used for the byte offset
  - The tag is \( 32 - (3 + 4) = 25 \) bits
  - For example: 0b1000 0000 0000 0000 0000 0000 0000 0001 0000
Simulate a 2-way cache

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0b100</td>
<td></td>
<td>1</td>
<td>0b110</td>
<td></td>
</tr>
</tbody>
</table>

```
0b10 0000 0000 0b10 0000 1000
0b10 0000 0000 0b10 0001 0000
0b10 0001 0100 0b10 0001 0100
0b11 0001 0000 0b10 0000 0000
0b10 0000 1000 0b10 0000 0000
0b10 0000 1000 0b10 0000 0000
0b10 0001 0000 0b10 0001 0000
0b10 0001 0100 0b10 0001 0100
```
Simulate a direct-mapped cache

• Consider a direct mapped (1-way) cache with 16 blocks, a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  • 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000
  • 16 = 2^4 : 4 bits are used for the index
  • 16 = 2^4 : 4 bits are used for the byte offset
  • The tag is 32 - (4 + 4) = 24 bits
  • For example: 0b1000 0000 0000 0000 0000 0000 0000 1000 0000
Simulate a direct-mapped cache

<table>
<thead>
<tr>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
<th>compulsory miss</th>
<th>hit!</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0000</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0000</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>0b11</td>
<td>0001</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0000</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0000</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
<td>0100</td>
<td></td>
</tr>
</tbody>
</table>

Tag index: 0b10
Improving 3Cs
Improvement of 3Cs

• 3Cs and A, B, C of caches
  • Compulsory miss
    • Increase B: increase miss penalty (more data must be fetched from lower hierarchy)
  • Capacity miss
    • Increase C: increase cost, access time, power
  • Conflict miss
    • Increase A: increase access time and power

• Or modify the memory access pattern of your program!
Live demo: Matrix Multiplication

- Matrix Multiplication

```c
for(i = 0; i < ARRAY_SIZE; i++) {
    for(j = 0; j < ARRAY_SIZE; j++) {
        for(k = 0; k < ARRAY_SIZE; k++) {
            c[i][j] += a[i][k]*b[k][j];
        }
    }
}
```

CSE101 tells you it’s $O(n^3)$
If $n=512$, it takes about 1 sec
How long is it take when $n=1024$?
Matrix Multiplication

- Matrix Multiplication

\[
\text{for}(i = 0; \ i < \ \text{ARRAY\_SIZE}; \ i++) \ { \\
\text{\quad for}(j = 0; \ j < \ \text{ARRAY\_SIZE}; \ j++) \ { \\
\text{\quad \quad for}(k = 0; \ k < \ \text{ARRAY\_SIZE}; \ k++) \ { \\
\quad \text{c}[i][j] \ += \ a[i][k]*b[k][j]; \\
\text{\quad } \} \\
\text{\quad } \} \\
\text{\quad } \}
\]

- If each dimension of your matrix is 1024
  - Each row takes 1024*8 bytes = 8KB
  - The L1 $ of intel Core i7 is 32KB, 8-way, 64-byte blocked
  - You can only hold at most 4 rows/columns of each matrix!
  - You need the same row when j increase!

Very likely a miss if array is large
Block algorithm for matrix multiplication

- Discover the cache miss rate
  - `valgrind --tool=cachegrind cmd`
    - cachegrind is a tool profiling the cache performance
- Performance counter
  - Intel® Performance Counter Monitor http://www.intel.com/software/pcm/
Block algorithm for matrix multiplication

```c
for(i = 0; i < ARRAY_SIZE; i++) {
    for(j = 0; j < ARRAY_SIZE; j++) {
        for(k = 0; k < ARRAY_SIZE; k++) {
            c[i][j] += a[i][k]*b[k][j];
        }
    }
}
```

```c
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
        }
    }
}
```

You only need to hold these sub-matrixes in your cache
Other cache optimizations
Split Data & Instruction caches

- Different area of memory
- Different access patterns
  - instruction accesses have lots of spatial locality
  - instruction accesses are predictable to the extent that branches are predictable
  - data accesses are less predictable
- Instruction accesses may interfere with data accesses
- Avoiding structural hazards in the pipeline
- Writes to I-cache are rare
Victim cache

- A small cache that captures the evicted blocks
- Can be built as fully associative since it’s small
- Consult when there is a miss
- Athlon has an 8-entry victim cache
- Reduce the **miss penalty** of conflict misses
for(i = 0; i < 1000000; i++) {
    D[i] = rand();
}

Characteristic of memory accesses

L2 access for D[0] - D[7]

Prefetching

for(i = 0; i < 1000000; i++) {
    D[i] = rand();
    // prefetch D[i+8] if i % 8 == 0
}

```c
```
Prefetching

- Identify the access pattern and proactively fetch data/instruction before the application asks for the data/instruction
  - Trigger the cache miss earlier to eliminate the miss when the application needs the data/instruction

- Hardware prefetch:
  - The processor can keep track the distance between misses. If there is a pattern, fetch miss_data_address+distance for a miss

- Software prefetching
  - Load data into $zero
  - Using prefetch instructions
Write buffer

• Every write to lower memory will first write to a small SRAM buffer.
  • sw does not incur data hazards, but the pipeline has to stall if the write misses
  • The write buffer will continue writing data to lower-level memory
  • The processor/higher-level memory can response as soon as the data is written to write buffer.

• Help reduce miss penalty

• Write merge
  • Since application has locality, it’s highly possible the evicted data have neighboring addresses. Write buffer delays the writes and allows these neighboring data to be grouped together.