Memory Hierarchy --- Caching

*CSCI 2021: Machine Architecture and Organization*

Antonia Zhai
Department Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~zhai

With Slides from Bryant and O’Hallaron

An Example Memory Hierarchy

- **L0:** Registers
- **L1:** L1 cache (SRAM)
- **L2:** L2 cache (SRAM)
- **L3:** Main memory (DRAM)
- **L4:** Local secondary storage (local disks)
- **L5:** Remote secondary storage (tapes, distributed file systems, Web servers)

<table>
<thead>
<tr>
<th>Cache Level</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>Smaller, faster, costlier per byte</td>
</tr>
<tr>
<td>L2</td>
<td>Larger, slower, cheaper per byte</td>
</tr>
</tbody>
</table>

**Registers**
- CPU registers hold words retrieved from L1 cache

**L1 cache**
- L1 cache holds cache lines retrieved from L2 cache

**L2 cache**
- L2 cache holds cache lines retrieved from main memory

**Main memory**
- Main memory holds disk blocks retrieved from local disks

**Local disks**
- Local disks hold files retrieved from disks on remote network servers

**Remote secondary storage**
- Remote secondary storage (tapes, distributed file systems, Web servers)
Caches

- **Cache**: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy:
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- Why do memory hierarchies work?
  - Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
- **Big Idea**: The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

---

General Caching Concepts

Program needs object d, which is stored in some block b.

**Cache hit**
- Program finds b in the cache at level k. E.g., block 14.

**Cache miss**
- b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
- If level k cache is full, then some current block must be replaced (evicted). Which one is the "victim"?
  - **Placement policy**: where can the new block go? E.g., b mod 4
  - **Replacement policy**: which block should be evicted? E.g., LRU
General Caching Concepts:
Types of Cache Misses

- **Cold (compulsory) miss**
  - Cold misses occur because the cache is empty.

- **Conflict miss**
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.

- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.

Locality

Principle of Locality:

- **Temporal locality**: Recently referenced items are likely to be referenced in the near future.
- **Spatial locality**: Items with nearby addresses tend to be referenced close together in time.

Locality Example:

- **Data**
  - Reference array elements in succession **Spatial locality**
  - Reference sum each iteration: **Temporal locality**

- **Instructions**
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**
Storage Units Organization

- Rule-of-Thumb: The larger the structure, the slower the access time
- Solution: Hierarchical design

There is locality in data access
- Put things that you are likely to use in the near future close to you

Caching the Memory

- Cache uses **SRAM**: Static Random Access Memory
  - No refresh
- Main Memory is **DRAM**: Dynamic Random Access Memory
  - Dynamic since needs to be refreshed periodically
The 1,2,3,4 of Caching

1. Where can a block be placed in the upper level?
2. How is a block found if it is in the upper level?
3. Which block should be replaced on a miss?
4. What happens on a write?

General Cache Organization (S, E, B)

\[ E = 2^e \text{ lines per set} \]

\[ S = 2^s \text{ sets} \]

\[ C = S \times E \times B \text{ data bytes} \]

valid bit

B = \(2^b\) bytes per cache block (the data)
Cache Read

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

E = 2^e lines per set

S = 2^s sets

B = 2^b bytes per cache block (the data)

Address of word:

<table>
<thead>
<tr>
<th>t bits</th>
<th>s bits</th>
<th>b bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>set</td>
<td>block</td>
</tr>
<tr>
<td>index</td>
<td></td>
<td>offset</td>
</tr>
</tbody>
</table>

data begins at this offset

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

Address of int:

<table>
<thead>
<tr>
<th>t bits</th>
<th>0_..1</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>find set</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

Valid? + match: assume yes = hit

Address of int:

Block offset

Int (4 Bytes) is here

No match: old line is evicted and replaced
**Direct-Mapped Cache Simulation**

M=16 byte addresses, B=2 bytes/block,
S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

- 0 [0000],
- 1 [0001], miss
- 7 [0111], hit
- 8 [1000], miss
- 0 [0000], miss

<table>
<thead>
<tr>
<th>Set 0</th>
<th>1</th>
<th>0</th>
<th>M[0-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td>1</td>
<td>0</td>
<td>M[6-7]</td>
</tr>
</tbody>
</table>

**Example**

Cache block: 2 bytes
Associativity: Directly Mapped
Cache size: 16 bytes

<table>
<thead>
<tr>
<th>100</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>001</td>
<td>19</td>
<td>21</td>
<td>23</td>
<td>25</td>
<td>27</td>
<td>29</td>
<td>31</td>
</tr>
<tr>
<td>010</td>
<td>35</td>
<td>37</td>
<td>39</td>
<td>41</td>
<td>43</td>
<td>45</td>
<td>47</td>
</tr>
<tr>
<td>011</td>
<td>51</td>
<td>53</td>
<td>55</td>
<td>57</td>
<td>59</td>
<td>61</td>
<td>63</td>
</tr>
<tr>
<td>100</td>
<td>67</td>
<td>69</td>
<td>71</td>
<td>73</td>
<td>75</td>
<td>77</td>
<td>79</td>
</tr>
<tr>
<td>101</td>
<td>83</td>
<td>85</td>
<td>87</td>
<td>89</td>
<td>91</td>
<td>93</td>
<td>95</td>
</tr>
<tr>
<td>110</td>
<td>99</td>
<td>101</td>
<td>103</td>
<td>105</td>
<td>107</td>
<td>109</td>
<td>111</td>
</tr>
<tr>
<td>111</td>
<td>115</td>
<td>117</td>
<td>119</td>
<td>121</td>
<td>123</td>
<td>125</td>
<td>127</td>
</tr>
</tbody>
</table>
for(I = 0; I < 100; i++) {
    ... = a[i];
}

Compulsory miss
Capacity miss
Example

```
for(I = 0; I < 100; i++)
{
    ... = a[i];
    ... = b[i];
}
```

Load the following:

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>001</td>
<td>19</td>
<td>21</td>
<td>23</td>
<td>25</td>
<td>27</td>
<td>29</td>
<td>31</td>
<td>33</td>
</tr>
<tr>
<td>010</td>
<td>35</td>
<td>37</td>
<td>39</td>
<td>41</td>
<td>43</td>
<td>45</td>
<td>47</td>
<td>49</td>
</tr>
<tr>
<td>011</td>
<td>51</td>
<td>53</td>
<td>55</td>
<td>57</td>
<td>59</td>
<td>61</td>
<td>63</td>
<td>65</td>
</tr>
<tr>
<td>100</td>
<td>67</td>
<td>69</td>
<td>71</td>
<td>73</td>
<td>75</td>
<td>77</td>
<td>79</td>
<td>81</td>
</tr>
<tr>
<td>101</td>
<td>83</td>
<td>85</td>
<td>87</td>
<td>89</td>
<td>91</td>
<td>93</td>
<td>95</td>
<td>97</td>
</tr>
<tr>
<td>110</td>
<td>99</td>
<td>101</td>
<td>103</td>
<td>105</td>
<td>107</td>
<td>109</td>
<td>111</td>
<td>113</td>
</tr>
<tr>
<td>111</td>
<td>115</td>
<td>117</td>
<td>119</td>
<td>121</td>
<td>123</td>
<td>125</td>
<td>127</td>
<td>129</td>
</tr>
</tbody>
</table>

Example

```
for(I = 0; I < 100; i++)
{
    ... = a[i];
}
```

Load the following:

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>001</td>
<td>19</td>
<td>21</td>
<td>23</td>
<td>25</td>
<td>27</td>
<td>29</td>
<td>31</td>
<td>33</td>
</tr>
<tr>
<td>010</td>
<td>35</td>
<td>37</td>
<td>39</td>
<td>41</td>
<td>43</td>
<td>45</td>
<td>47</td>
<td>49</td>
</tr>
<tr>
<td>011</td>
<td>51</td>
<td>53</td>
<td>55</td>
<td>57</td>
<td>59</td>
<td>61</td>
<td>63</td>
<td>65</td>
</tr>
<tr>
<td>100</td>
<td>67</td>
<td>69</td>
<td>71</td>
<td>73</td>
<td>75</td>
<td>77</td>
<td>79</td>
<td>81</td>
</tr>
<tr>
<td>101</td>
<td>83</td>
<td>85</td>
<td>87</td>
<td>89</td>
<td>91</td>
<td>93</td>
<td>95</td>
<td>97</td>
</tr>
<tr>
<td>110</td>
<td>99</td>
<td>101</td>
<td>103</td>
<td>105</td>
<td>107</td>
<td>109</td>
<td>111</td>
<td>113</td>
</tr>
<tr>
<td>111</td>
<td>115</td>
<td>117</td>
<td>119</td>
<td>121</td>
<td>123</td>
<td>125</td>
<td>127</td>
<td>129</td>
</tr>
</tbody>
</table>

Compulsory miss
Capacity miss
Conflict miss
### Example

```java
for(I = 0; I < 100; i++) {
    ... = a[i];
    ... = b[i];
    ... = c[i];
}
```

### Load the following

```
010000
110000
010001
110001
010010
110010
```

### Data Tag Valid

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>001</th>
<th>001</th>
<th>001</th>
<th>001</th>
<th>001</th>
<th>001</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
</tbody>
</table>

### Conflict miss

```
010000
110000
010001
110001
010010
110010
```

### Data Tag Valid

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>001</th>
<th>001</th>
<th>001</th>
<th>001</th>
<th>001</th>
<th>001</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
</tbody>
</table>

```
010000
110000
010001
110001
010010
110010
```

### Conflict miss
A Higher Level Example

int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}

int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}

assume: cold (empty) cache,
a[0][0] goes here

32 B = 4 doubles

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

find set
E-way Set Associative Cache (Here: \( E = 2 \))

\( E = 2 \): Two lines per set
Assume: cache block size 8 bytes

Address of short int:

<table>
<thead>
<tr>
<th>t bits</th>
<th>0...01</th>
<th>100</th>
</tr>
</thead>
</table>

valid? + match: yes = hit

\[ \begin{array}{l}
\text{tag} \quad 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7 \\
\text{v} \end{array} \]

\[ \begin{array}{l}
\text{tag} \quad 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7 \\
\text{v} \end{array} \]

block offset

No match:
- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

short int (2 Bytes) is here
2-Way Set Associative Cache Simulation

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):
0  [0000], miss
1  [0001], hit
7  [0111], miss
8  [1000], miss
0  [0000], hit

\[
\begin{array}{ccc}
\text{Set 0} & 0 & 1 \\
\text{Set 1} & 0 & 1 \\
\end{array}
\]

Example Cache Block: 2 bytes
Associativity: 2
Cache size: 16 bytes

\[
\begin{array}{cccccccc}
\text{000} & 3 & 5 & 7 & 9 & 11 & 13 & 15 & 17 \\
\text{001} & 19 & 21 & 23 & 25 & 27 & 29 & 31 & 33 \\
\text{010} & 35 & 37 & 39 & 41 & 43 & 45 & 47 & 49 \\
\text{011} & 51 & 53 & 55 & 57 & 59 & 61 & 63 & 65 \\
\text{100} & 67 & 69 & 71 & 73 & 75 & 77 & 79 & 81 \\
\text{101} & 83 & 85 & 87 & 89 & 91 & 93 & 95 & 97 \\
\text{110} & 99 & 101 & 103 & 105 & 107 & 109 & 111 & 113 \\
\text{111} & 115 & 117 & 119 & 121 & 123 & 125 & 127 & 129 \\
\end{array}
\]
Example

for(I = 0; I < 100; i++) {
    ... = a[i];
}

Load the following
010000
010001
010010
010011
010100
010101
010110
010111
011000
011001
011010
011011
011100
011101
011110
011111
011111

Compulsory miss
Capacity miss
### Example

**Load the following**

<table>
<thead>
<tr>
<th>Index</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>001</td>
<td>19</td>
<td>21</td>
<td>23</td>
<td>25</td>
<td>27</td>
<td>29</td>
<td>31</td>
<td>33</td>
</tr>
<tr>
<td>010</td>
<td>35</td>
<td>37</td>
<td>39</td>
<td>41</td>
<td>43</td>
<td>45</td>
<td>47</td>
<td>49</td>
</tr>
<tr>
<td>011</td>
<td>51</td>
<td>53</td>
<td>55</td>
<td>57</td>
<td>59</td>
<td>61</td>
<td>63</td>
<td>65</td>
</tr>
<tr>
<td>100</td>
<td>67</td>
<td>69</td>
<td>71</td>
<td>73</td>
<td>75</td>
<td>77</td>
<td>79</td>
<td>81</td>
</tr>
<tr>
<td>101</td>
<td>83</td>
<td>85</td>
<td>87</td>
<td>89</td>
<td>91</td>
<td>93</td>
<td>95</td>
<td>97</td>
</tr>
<tr>
<td>110</td>
<td>99</td>
<td>101</td>
<td>103</td>
<td>105</td>
<td>107</td>
<td>109</td>
<td>111</td>
<td>113</td>
</tr>
<tr>
<td>111</td>
<td>119</td>
<td>121</td>
<td>123</td>
<td>125</td>
<td>127</td>
<td>129</td>
<td>131</td>
<td>133</td>
</tr>
</tbody>
</table>

**For (I = 0; I < 100; i++) {**

```c
  ... = a[i];
  ... = b[i];
  ... = c[i];
```

} **Conflict miss**

---

### Example

**Load the following**

<table>
<thead>
<tr>
<th>Index</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>001</td>
<td>19</td>
<td>21</td>
<td>23</td>
<td>25</td>
<td>27</td>
<td>29</td>
<td>31</td>
<td>33</td>
</tr>
<tr>
<td>010</td>
<td>35</td>
<td>37</td>
<td>39</td>
<td>41</td>
<td>43</td>
<td>45</td>
<td>47</td>
<td>49</td>
</tr>
<tr>
<td>011</td>
<td>51</td>
<td>53</td>
<td>55</td>
<td>57</td>
<td>59</td>
<td>61</td>
<td>63</td>
<td>65</td>
</tr>
<tr>
<td>100</td>
<td>67</td>
<td>69</td>
<td>71</td>
<td>73</td>
<td>75</td>
<td>77</td>
<td>79</td>
<td>81</td>
</tr>
<tr>
<td>101</td>
<td>83</td>
<td>85</td>
<td>87</td>
<td>89</td>
<td>91</td>
<td>93</td>
<td>95</td>
<td>97</td>
</tr>
<tr>
<td>110</td>
<td>99</td>
<td>101</td>
<td>103</td>
<td>105</td>
<td>107</td>
<td>109</td>
<td>111</td>
<td>113</td>
</tr>
<tr>
<td>111</td>
<td>119</td>
<td>121</td>
<td>123</td>
<td>125</td>
<td>127</td>
<td>129</td>
<td>131</td>
<td>133</td>
</tr>
</tbody>
</table>

**For (I = 0; I < 100; i++) {**

```c
  ... = a[i];
  ... = b[i];
  ... = c[i];
```

} **Conflict miss**

---
A Higher Level Example

```c
int sum_array_rows(double a[16][16]) {
    int i, j;
    double sum = 0;
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```c
int sum_array_col(double a[16][16]) {
    int i, j;
    double sum = 0;
    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

What about writes?

- Multiple copies of data exist:
  - L1, L2, Main Memory, Disk
- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes immediately to memory)
- Typical
  - Write-through + No-write-allocate
  - Write-back + Write-allocate
**Intel Core i7 Cache Hierarchy**

- **Processor package**
  - **Core 0**
    - Registers
    - L1 d-cache
    - L1 i-cache
    - L2 unified cache
  - **Core 3**
    - Registers
    - L1 d-cache
    - L1 i-cache
    - L2 unified cache
  - **...**
  - L3 unified cache (shared by all cores)

- **Main memory**

---

**Cache Performance**

- L1 i-cache and d-cache: 32 KB, 8-way, Access: 4 cycles
- L2 unified cache: 256 KB, 8-way, Access: 11 cycles
- L3 unified cache: 8 MB, 16-way, Access: 30-40 cycles

Block size: 64 bytes for all caches.
Cache Performance Metrics

Miss Rate
- Fraction of memory references not found in cache (misses/references)
- Typical numbers:
  - 3-10% for L1; can be quite small for L2, depending on size, etc.

Hit Time
- Time to deliver a line in the cache to the processor
  - includes time to determine whether the line is in the cache
- Typical numbers:
  - 1 clock cycle for L1; 3-8 clock cycles for L2

Miss Penalty
- Additional time required because of a miss
  - Typically 25-100 cycles for accessing the main memory

Average Memory Access Time

\[ AMAT = \text{Average Memory Access Time} \]

\[ AMAT = \text{HitTime} + \text{MissRate} \times \text{MissPenalty} \]

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
**Let’s think about those numbers**

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory

- Would you believe 99% hits is twice as good as 97%?
  - Consider:
    - cache hit time of 1 cycle
    - miss penalty of 100 cycles

  - Average access time:
    - 97% hits: $1 \text{ cycle} + 0.03 \times 100 \text{ cycles} = 4 \text{ cycles}$
    - 99% hits: $1 \text{ cycle} + 0.01 \times 100 \text{ cycles} = 2 \text{ cycles}$

- This is why "miss rate" is used instead of "hit rate"

**Writing Cache Friendly Code**

- Make the common case go fast
  - Focus on the inner loops of the core functions

- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories.
Writing Cache Friendly Code

16-bit address space

for (i = 0; i < 997; i++)
    for (j = 0; j < 997; j++)
        sum += a[i][j];
    return sum;

Cache

Miss rate = 1/4 = 25%

Miss rate = 100%
How to Improve Cache Performance?

Hit Time = 1 Cycle \hspace{1cm} Miss Penalty = 100 Cycle

**Average memory access time =**

\[
\text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}
\]

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.

What if we have a bigger cache?

Another Example: Matrix Multiplication

Description:
- Multiply N x N matrices
- \(O(N^3)\) total operations

Accesses
- \(N\) reads per source element
- \(N\) values summed per destination

```c
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Variable sum held in register
Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = 32B (big enough for 4 8-byte words)
- Matrix dimension (N) is very large
- Cache is not even big enough to hold multiple rows

Analysis Method:

- Look at access pattern of inner loop

Matrix Multiplication (ijk)

```c
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

- Misses per Inner Loop Iteration:
  
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Inner loop:
Matrix Multiplication (jik)

```c
/* jik */
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum
    }
}
```

**Inner loop:**
- (i,*)
- (*,j)
- (i,j)

**Row-wise**
**Column-wise**
**Fixed**

**Misses per Inner Loop Iteration:**

\[
\begin{array}{ccc}
A & B & C \\
0.25 & 1.0 & 0.0
\end{array}
\]

4/8/15
CSCI 2021
47

Matrix Multiplication (kij)

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

**Inner loop:**
- (i,k)
- (k,*)
- (*,i)

**Fixed**
**Row-wise**
**Row-wise**

**Misses per Inner Loop Iteration:**

\[
\begin{array}{ccc}
A & B & C \\
0.0 & 0.25 & 0.25
\end{array}
\]

4/8/15
CSCI 2021
48
Matrix Multiplication (ikj)

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

* Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Matrix Multiplication (jki)

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

* Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

4/8/15  CSCI 2021  49

4/8/15  CSCI 2021  50
Matrix Multiplication (kji)

/* kji */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}

- Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Summary of Matrix Multiplication

- Cache hit = 1 cycle; Cache miss penalty = 100 cycle

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = 1.25

kij (& ikj):
- 2 loads, 1 store
- misses/iter = 0.5

jki (& kji):
- 2 loads, 1 store
- misses/iter = 2.0

for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}

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

for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
Example: Matrix Multiplication

```c
double *c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n+k]*b[k*n + j];
}
```

Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)

- First iteration:
  - $n/8 + n = 9n/8$ misses

- Afterwards in cache:
  (schematic)
Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size $C \ll n$ (much smaller than $n$)

- Second iteration:
  - Again:
    - $n/8 + n = 9n/8$ misses

- Total misses:
  - $9n/8 \times n^2 = (9/8) \times n^3$

Blocked Matrix Multiplication

```c
C = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i += B)
        for (j = 0; j < n; j += B)
            for (k = 0; k < n; k += B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

Block size $B \times B$
Cache Miss Analysis

• Assume:
  • Cache block = 8 doubles
  • Cache size \( C \ll n \) (much smaller than \( n \))
  • Three blocks fit into cache: \( 3B^2 < C \)

• First (block) iteration:
  • \( B^2/8 \) misses for each block
  • \( 2n/B * B^2/8 = nB/4 \)
    (omitting matrix \( c \))

• Afterwards in cache
  (schematic)

\[
\begin{array}{ccc}
\text{Block size } B \times B & \text{=} & n/B \text{ blocks}
\end{array}
\]

• Second (block) iteration:
  • Same as first iteration
  • \( 2n/B * B^2/8 = nB/4 \)

• Total misses:
  • \( nB/4 * (n/B)^2 = n^3/(4B) \)
Summary

• No blocking: \((9/8) \times n^3\)
• Blocking: \((1/(4B)) \times n^3\)

• Suggest largest possible block size \(B\), but limit \(3B^2 < C\)

• Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    • Input data: \(3n^2\), computation \(2n^3\)
    • Every array element used \(O(n)\) times!
  • But program has to be written properly

Concluding Observations

Programmer can optimize for cache performance
• How data structures are organized
• How data are accessed
  • Nested loop structure
All systems favor “cache friendly code”
• Getting absolute optimum performance is very platform specific
  • Cache sizes, line sizes, associativities, etc.
• Can get most of the advantage with generic code
  • Keep working set reasonably small (temporal locality)
  • Use small strides (spatial locality)