Cache Impact on Program Performance

T. Yang. UCSB CS240A. 2017
Multi-level cache in computer systems

Topics
- Performance analysis for multi-level cache
- Cache performance optimization through program transformation
<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4 (Barcelona)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache organization</td>
<td>Split instruction and data caches</td>
<td>Split instruction and data caches</td>
</tr>
<tr>
<td>L1 cache size</td>
<td>32 KB each for instructions/data per core</td>
<td>64 KB each for instructions/data per core</td>
</tr>
<tr>
<td>L1 block size</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L1 write policy</td>
<td>Write-back, Write-allocate</td>
<td>Write-back, Write-allocate</td>
</tr>
<tr>
<td>L1 hit time (load-use)</td>
<td>Not Available</td>
<td>3 clock cycles</td>
</tr>
<tr>
<td>L2 cache organization</td>
<td>Unified (instruction and data) per core</td>
<td>Unified (instruction and data) per core</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>256 KB (0.25 MB)</td>
<td>512 KB (0.5 MB)</td>
</tr>
<tr>
<td>L2 block size</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L2 write policy</td>
<td>Write-back, Write-allocate</td>
<td>Write-back, Write-allocate</td>
</tr>
<tr>
<td>L2 hit time</td>
<td>Not Available</td>
<td>9 clock cycles</td>
</tr>
<tr>
<td>L3 cache organization</td>
<td>Unified (instruction and data)</td>
<td>Unified (instruction and data)</td>
</tr>
<tr>
<td>L3 cache size</td>
<td>8192 KB (8 MB), shared</td>
<td>2048 KB (2 MB), shared</td>
</tr>
<tr>
<td>L3 block size</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L3 write policy</td>
<td>Write-back, Write-allocate</td>
<td>Write-back, Write-allocate</td>
</tr>
<tr>
<td>L3 hit time</td>
<td>Not Available</td>
<td>38 (?)clock cycles</td>
</tr>
</tbody>
</table>
Cache misses and data access time

$D_0$: total memory data accesses.

$D_1$: missed access at L1. $m_1$ local miss ratio of L1: $m_1 = D_1/D_0$

$D_2$: missed access at L2. $m_2$ local miss ratio of L2: $m_2 = D_2/D_1$

$D_3$: missed access at L3

$m_3$ local miss ratio of L2: $m_3 = D_3/D_2$

Memory and cache access time:

$\delta_i$: access time at cache level $i$

$\delta_{\text{mem}}$: access time in memory.

Average access time

$$= \frac{\text{total time}}{D_0} = \delta_1 + m_1 \times \text{penalty}$$
Average memory access time (AMAT)

\[ AMAT = \delta_1 + m_1 \times \text{penalty} \]

\( \approx 2 \text{ cycles} \)
Total data access time

Average time =

\[ \delta_1 + m_1 [\delta_2 + m_2 \text{Penalty}] \]

\(~2 \text{ cycles}\)
Total data access time

Average time = $\delta_1 + m_1 \left[ \delta_2 + m_2 \text{Penalty} \right]$

Found in L3 or memory

~2 cycles
Total data access time

Average time = \[ \delta_1 + m_1 \left[ \delta_2 + m_2 \delta_{\text{mem}} \right] \]

- \( \sim 2 \) cycles
- \( \sim 10 \) cycles
- \( \sim 100-200 \) cycles

No L3. Found in memory
Total data access time

Average memory access time (AMAT) =

\[ \delta_1 + m_1 [\delta_2 + m_2 [\delta_3 + m_3 \delta_{\text{mem}}]] \]
Local vs. Global Miss Rates

- **Local miss rate** – the fraction of references to one level of a cache that miss. For example, \( m_2 = \frac{D_2}{D_1} \)

Notice total L2 accesses is L1 Misses

- **Global miss rate** – the fraction of references that miss in all levels of a multilevel cache
  - Global L2 miss rate = \( \frac{D_2}{D_0} \)
  - L2$ local miss rate >> than the global miss rate

- Notice Global L2 miss rate = \( \frac{D_2}{D_0} = \frac{D_1}{D_0} \times \frac{D_2}{D_1} = m_1 \times m_2 \)
L1 Cache: 32KB I$, 32KB D$
L2 Cache: 256 KB
L3 Cache: 4 MB

FIGURE 5.47 The L1, L2, and L3 data cache miss rates for the Intel Core i7 920 running the full integer SPECCPU2006 benchmarks.
Average memory access time with no L3 cache

\[
AMAT = \delta_1 + m_1 (\delta_2 + m_2 \delta_{\text{mem}})
\]

\[
= \delta_1 + m_1 \delta_2 + m_1 m_2 \delta_{\text{mem}}
\]

\[
= \delta_1 + m_1 \delta_2 + \text{GMiss}_2 \delta_{\text{mem}}
\]
Average memory access time with L3 cache

\[ \text{AMAT} = \delta_1 + m_1 [\delta_2 + m_2 [\delta_3 + m_3 \delta_{\text{mem}}]] \]

\[ = \delta_1 + m_1 \delta_2 + m_1 m_2 \delta_3 + m_1 m_2 m_3 \delta_{\text{mem}} \]

\[ = \delta_1 + m_1 \delta_2 + \text{GMiss}_2 \delta_3 + \text{GMiss}_3 \delta_{\text{mem}} \]
Example

1. Suppose that you have a cache system with the following properties. What is the AMAT?
   a) L1$ hits in 1 cycle (local miss rate 25%)
   b) L2$ hits in 10 cycles (local miss rate 40%)
   c) L3$ hits in 50 cycles (global miss rate 6%)
   d) Main memory hits in 100 cycles (always hits)

What is average memory access time?

The AMAT is \( 1 + 0.25 \times (10 + 0.4 \times (50)) + 0.06 \times 100 = 14.5 \) cycles.
Example

(b) Given the following specification:
For every 1000 CPU-to-memory references
40 will miss in L1$;
20 will miss in L2$;
10 will miss in L3$;
L1$ hits in 1 clock cycle;
L2$ hits in 10 clock cycles;
L3$ hits in 100 clock cycles;
Main memory access is 400 clock cycles;

What is the average memory access time with L1, L2, and L3?
Example

(i) What is the local miss rate in the L2$? 
\[
\frac{20}{40} = 50\% 
\]

(ii) What is the global miss rate in the L2$? 
\[
\frac{20}{1000} = 2\% 
\]

(iii) What is the local miss rate in the L3$? 
\[
\frac{10}{20} = 50\% 
\]

(iv) What is the global miss rate in the L3$? 
\[
\frac{10}{1000} = 1\% 
\]

(v) What is the AMAT with all three levels of cache? 
\[
1 + 4\%*(10+50\%*(100+50\%*400)) = 1 + 0.4 + 2\%*300 = 7.4 
\]

(vi) What is the AMAT for a two-level cache without L3$? 
\[
1 + 4\%*10 + 2\%*400 = 1 + 0.4 + 8 = 9.4 
\]
Cache-aware Programming

• Reuse values in cache as much as possible
  ▪ exploit temporal locality in program
  ▪ Example 1: \( Y[2] \) is revisited continuously

\[
\text{For } i=1 \text{ to } n \\
\]

• Example 2 with access sequence: \( Y[2] \) is revisited after a few instructions later

\[
\]
Cache-aware Programming

- Take advantage of better bandwidth by getting a chunk of memory to cache and use whole chunk
  - Exploit spatial locality in program

For i=1 to n
  \[ y[i] = y[i] + 3 \]

Visiting \( Y[1] \) benefits next access of \( Y[2] \)
2D array layout in memory (just like 1D array)

• for (x = 0; x < 3; x++) {
    for (y = 0; y < 3; y++) {
        a[y][x] = 0; // implemented as array[3*y+x] = 0
    }
}

→ access order a[0][0], a[1][0], a[2][0], a[3][0] …
Exploit spatial data locality via program rewriting: Example 1

- Each cache block has 64 bytes. Cache has 128 bytes
- Program structure (data access pattern)
  - char D[64][64];
  - Each row is stored in one cache line block
  - Program 1
    
    ```c
    for (j = 0; j < 64; j++)
        for (i = 0; i < 64; i++)
            D[i][j] = 0;
    ```

64*64 data byte access → What is cache miss rate?

- Program 2
  
  ```c
  for (i = 0; i < 64; i++)
      for (j = 0; j < 64; j++)
          D[i][j] = 0;
  ```

What is cache miss rate?
for (i = 0; i < 64; j++)
  for (j = 0; j < 64; i++)
    D[i][j] = 0;

1 cache miss in one inner loop iteration

64 cache miss out of 64*64 access.
There is spatial locality. Fetched cache block is used 64 times before swapping out (consecutive data access within the inner loop
Memory layout and data access by block

- **Memory layout of Char D[64][64] same as Char D[64*64]**

  **Data access order of a program**

  **Memory layout**

  **Program in 2D loop**

  Miss hit hit hit …hit

  D[0,0]  D[0,1]  ….  D[0,63]
  D[1,0]  D[1,1]  ….  D[1,63]

  D[63,0]  D[63,1]  ….  D[63,63]

  j

  i

  64 cache miss out of 64*64 access.
Data Locality and Cache Miss

- for \( (j = 0; j < 64; j++) \)
  - for \( (i = 0; i < 64; i++) \)
    - \( D[i][j] = 0; \)

64 cache miss in one inner loop iteration

100% cache miss
There is no spatial locality. Fetched block is only used once before swapping out.
Memory layout and data access by block

Data access order of a program

<table>
<thead>
<tr>
<th>D[0,0]</th>
<th>D[1,0]</th>
<th>....</th>
<th>D[63,0]</th>
<th>D[0,1]</th>
<th>D[1,1]</th>
<th>....</th>
<th>D[63,1]</th>
<th>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>D[0,63]</td>
<td>D[1,0]</td>
<td>D[1,1]</td>
<td>....</td>
<td>D[63,0]</td>
<td>D[63,1]</td>
<td>....</td>
<td>D[63,63]</td>
<td></td>
</tr>
</tbody>
</table>

Memory layout

<table>
<thead>
<tr>
<th>D[0,0]</th>
<th>D[0,1]</th>
<th>....</th>
<th>D[0,63]</th>
<th>D[1,0]</th>
<th>D[1,1]</th>
<th>....</th>
<th>D[1,63]</th>
<th>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>D[63]</td>
<td>D[63,0]</td>
<td>D[63,1]</td>
<td>....</td>
<td>D[63,63]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program in 2D loop

100% cache miss
Summary of Example 1: Loop interchange alters execution order and data access patterns

- Exploit more spatial locality in this case
Program rewriting example 2: cache blocking for better temporal locality

- Cache size = 8 blocks = 128 bytes
  - Cache block size = 16 bytes, hosting 4 integers
- Program structure
  ```
  int A[64]; // sizeof(int)=4 bytes
  for (k = 0; k < repcount; k++)
    for (i = 0; i < 64; i += stepsize)
  ```

Analyze cache hit ratio when varying cache block size, or step size (stride distance)
Example 2: Focus on inner loop

```
for (i = 0; i < 64; i += stepsize)
```

Step size or also called stride distance
Step size = 2

for (i = 0; i < 64; I += stepsize)  
Repeat many times

- for (k = 0; k < repcount; k++)
  
  for (i = 0; i < 64; I += stepsize)

Array has 16 blocks. Inner loop accesses 32 elements, and fetches all 16 blocks. Each block is used as R/W/R/W. Cache size = 8 blocks and cannot hold all 16 blocks fetched.
Cache blocking to exploit temporal locality

For (k=0; k=100; k++)
   for (i = 0; i < 64; i += S)
      A[i] = f(A[i])

Rewrite program with cache blocking

Pink code block can be executed fitting into cache
- Loop blocking (cache blocking)

For $i=1$ to $2n$

$$S_i : a_i = b_i + c_i$$

Rewrite as:

$$\text{for } i = 1 \text{ to } n$$

with blocksize=2

$$\text{do } S_{2i-1}, S_{2i}$$

- More general: Given

$$\text{for } (i = 0; i < 64; i+=S)$$

$$A[i] = f(A[i])$$

- Rewrite as:

$$\text{for } (bi = 0; bi < 64; bi = bi + \text{blocksize})$$

$$\text{for } (i = bi; i < bi + \text{blocksize}; i += S)$$

$$A[i] = f(A[i])$$
Example 2: Cache blocking for better performance

- For \( k = 0 \) to \( k = 100 \), \( k++ \)
  
  
  \[
  \begin{array}{c}
  \text{for } (i = 0; i < 64; i=i+S) \\
  A[i] = f(A[i])
  \end{array}
  \]

- Rewrite as:
  
  For \( k = 0 \) to \( k = 100 \), \( k++ \)

  \[
  \begin{array}{c}
  \text{for } (bi = 0; bi<64; bi=bi+blocksize) \\
  \text{for } (i = bi; i<bi+blocksize; i+=S) \\
  A[i] = f(A[i])
  \end{array}
  \]

Look interchange

\[
\text{for } (bi = 0; bi<64; bi=bi+blocksize)
\]

- For \( k = 0 \) to \( k = 100 \), \( k++ \)

  \[
  \begin{array}{c}
  \text{for } (i = bi; i<bi+ blocksize; i+=S) \\
  A[i] = f(A[i])
  \end{array}
  \]

Pink code block can be executed fitting into cache
Example 3: Matrix multiplication $C = A \times B$

For $i = 0$ to $n-1$

For $j = 0$ to $n-1$

For $k = 0$ to $n-1$

$C[i][j] += A[i][k] \times B[k][j]$

Example:

$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

$B = \begin{pmatrix} 5 & 7 \\ 6 & 8 \end{pmatrix}$

$C = A \times B$

$C = \begin{pmatrix} 1 \times 5 + 2 \times 6 & 1 \times 7 + 2 \times 8 \\ 3 \times 5 + 4 \times 6 & 3 \times 7 + 4 \times 8 \end{pmatrix} = \begin{pmatrix} 17 & 23 \\ 39 & 53 \end{pmatrix}$
Example 3: matrix multiplication code

2D array implemented using 1D layout

- for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      \[C[i+j*n] +\] += \[A[i+k*n]* B[k+j*n]\]

3 loop controls can interchange (C elements are modified independently with no dependence)

Which code has better cache performance (faster)?

for (j = 0; j < n; j++)
  for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
      \[C[i+j*n] +\] += \[A[i+k*n]* B[k+j*n]\]
Example 3: matrix multiplication code

2D array implemented using 1D layout

- for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      C[i+j*n] += A[i+k*n] * B[k+j*n]

3 loop controls can interchange (C elements are modified independently with no dependence)

Which code has better cache performance (faster)?

-- Study impact of stride on inner most loop which does most computation
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      for (i = 0; i < n; i++)
        C[i+j*n] += A[i+k*n] * B[k+j*n]
Example 4: Cache blocking for matrix transpose

```c
for (x = 0; x < n; x++) {
    for (y = 0; y < n; y++) {
        dst[y + x * n] = src[x + y * n];
    }
}
```

Rewrite code with cache blocking
Example 4: Cache blocking for matrix transpose

```
for (x = 0; x < n; x++) {
    for (y = 0; y < n; y++) {
        dst[y + x * n] = src[x + y * n];
    }
}
```

Rewrite code with cache blocking

```
for (i = 0; i < n; i += blocksize) {
    for (x = i; x < i+blocksize; ++x) {
        for (j = 0; j < n; j += blocksize) {
            for (y = j; y < j+blocksize; ++y) {
                dst[y + x * n] = src[x + y * n];
            }
        }
    }
}
```