Cache Impact on Program Performance

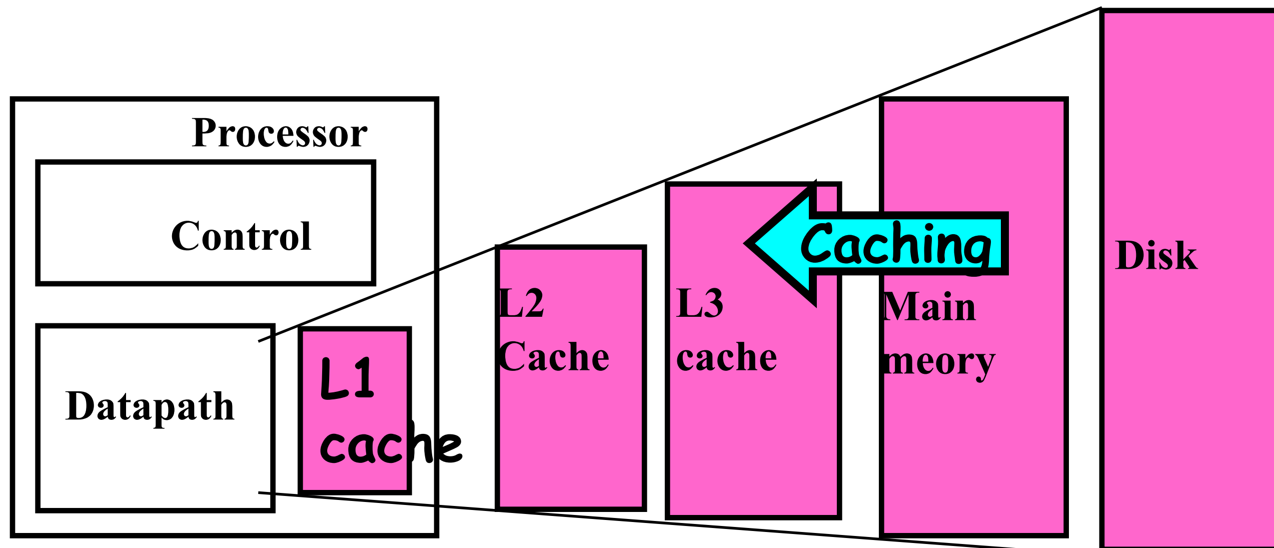
T. Yang. UCSB CS240A. 2017

UCSB

Multi-level cache in computer systems

Topics

- Performance analysis for multi-level cache
- Cache performance optimization through program transformation



Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KB each for instructions/data per core	64 KB each for instructions/data per core
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L1 hit time (load-use)	Not Available	3 clock cycles
L2 cache organization	Unified (instruction and data) per core	Unified (instruction and data) per core
L2 cache size	256 KB (0.25 MB)	512 KB (0.5 MB)
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	Not Available	9 clock cycles
L3 cache organization	Unified (instruction and data)	Unified (instruction and data)
L3 cache size	8192 KB (8 MB), shared	2048 KB (2 MB), shared
L3 block size	64 bytes	64 bytes
L3 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L3 hit time	Not Available	38 (?)clock cycles

UCSB

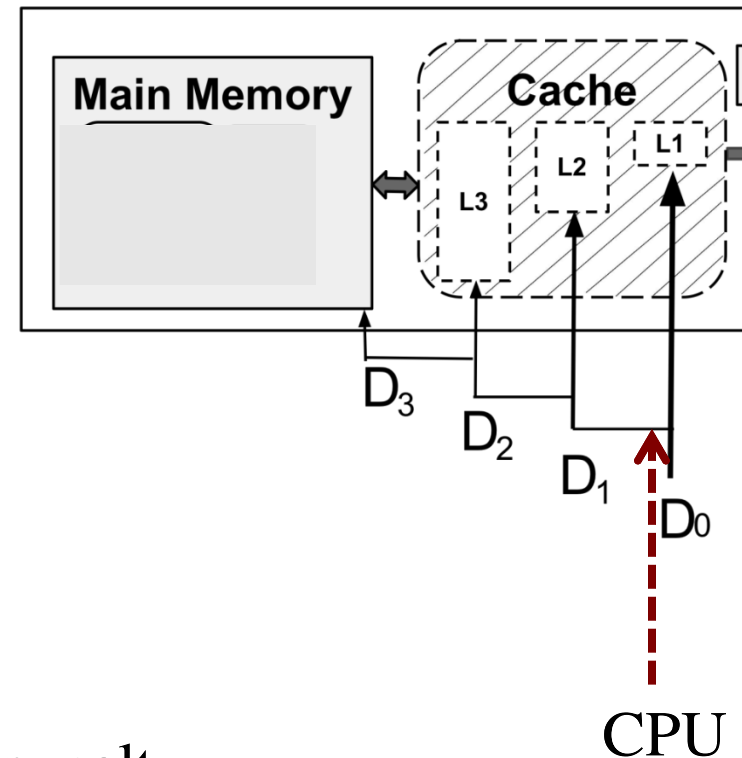
Cache misses and data access time

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: $m_3 = D_3/D_2$



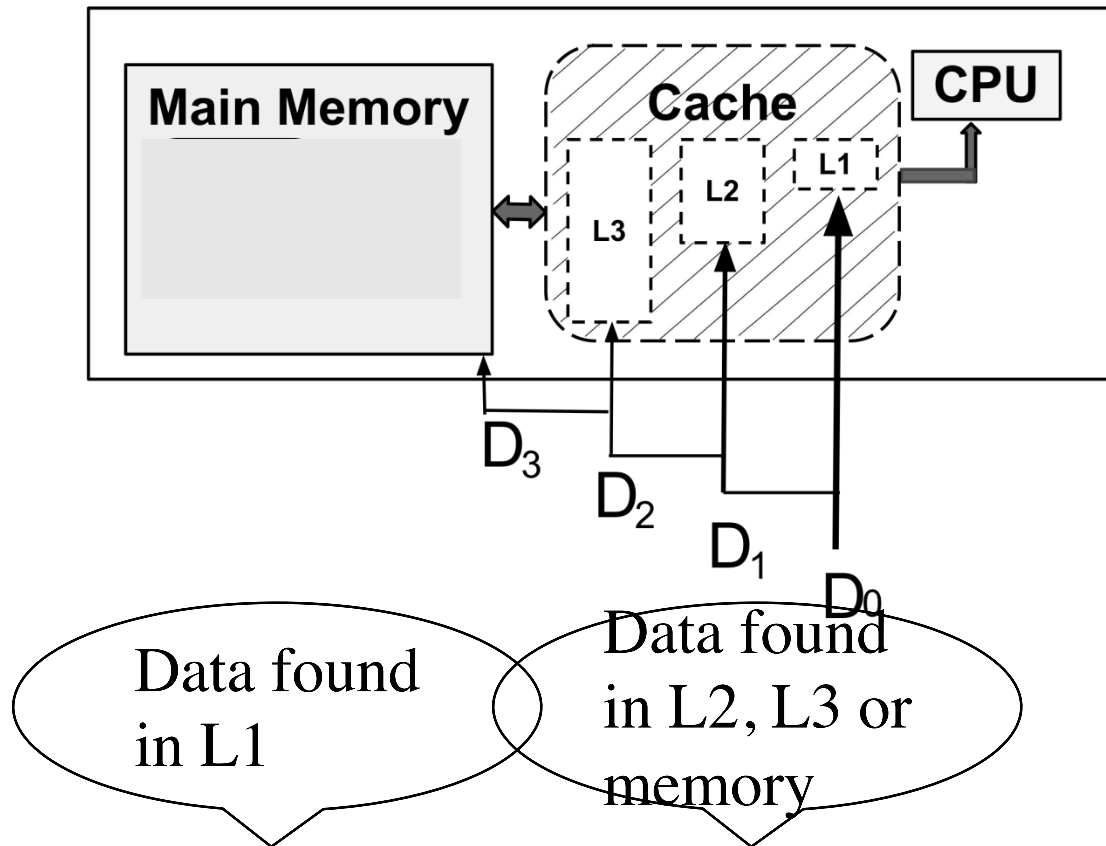
Memory and cache access time:

δ_i : access time at cache level i

δ_{mem} : access time in memory.

Average access time $= \text{total time}/D_0 = \delta_1 + m_1 * \text{penalty}$

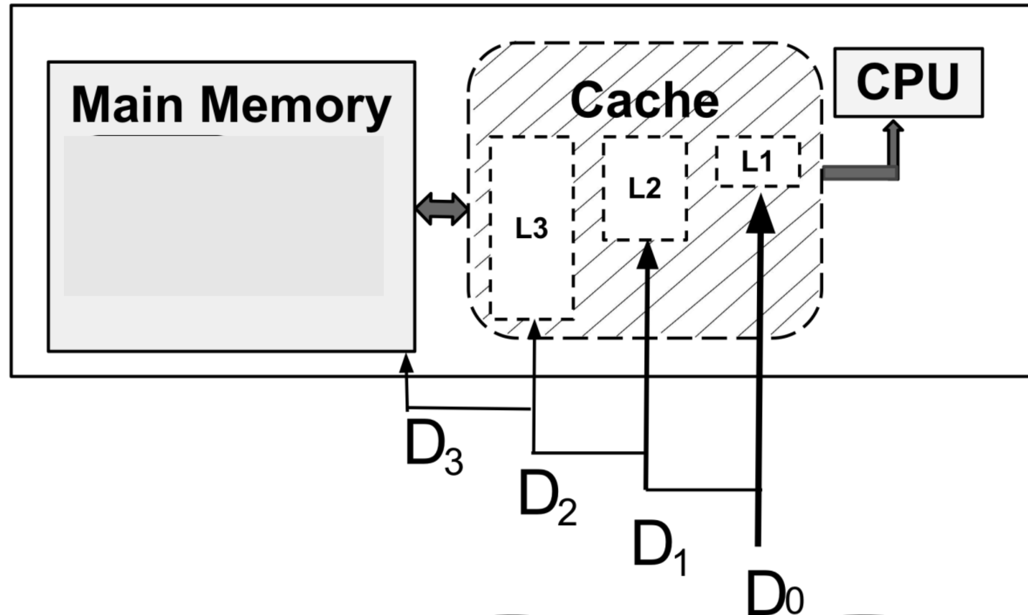
Average memory access time (AMAT)



AMAT= $\delta_1 + m_1 * \text{penalty}$

~2 cycles

Total data access time



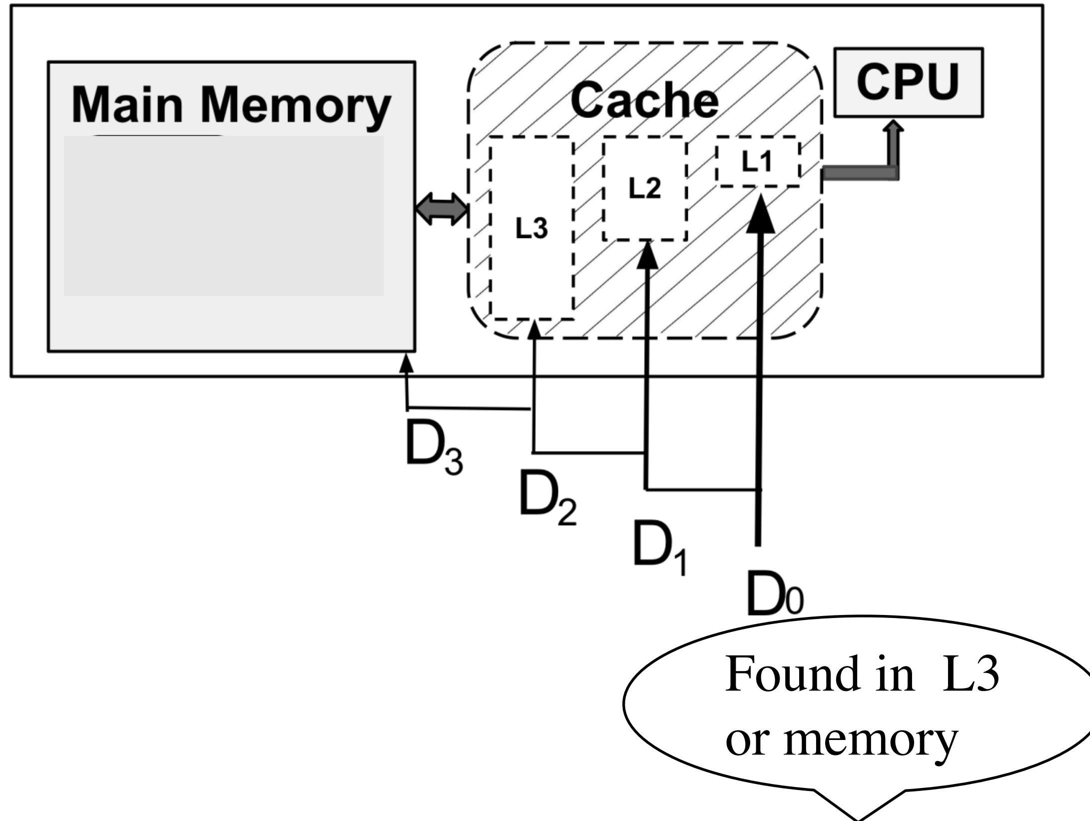
Data found in L2

Data found in L3 or memory

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

~2 cycles

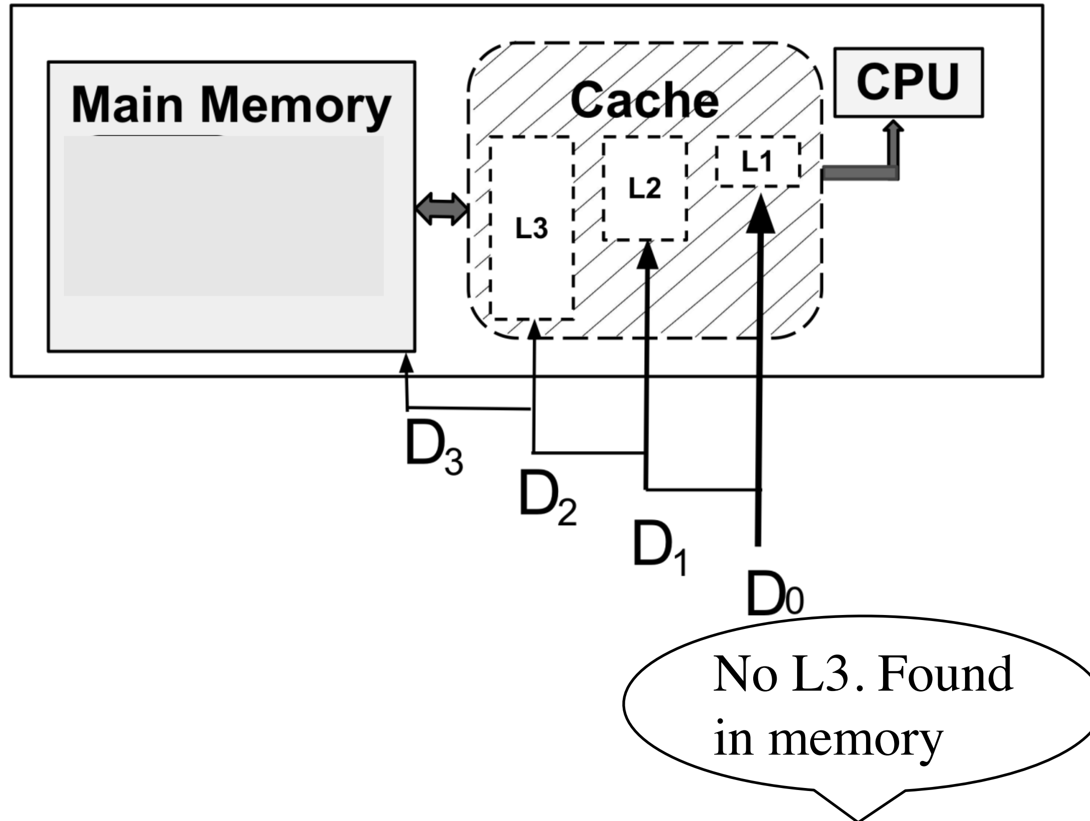
Total data access time



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

~2 cycles

Total data access time



Average time = $\delta_1 + m_1 [\delta_2 + m_2 \delta_{mem}]$

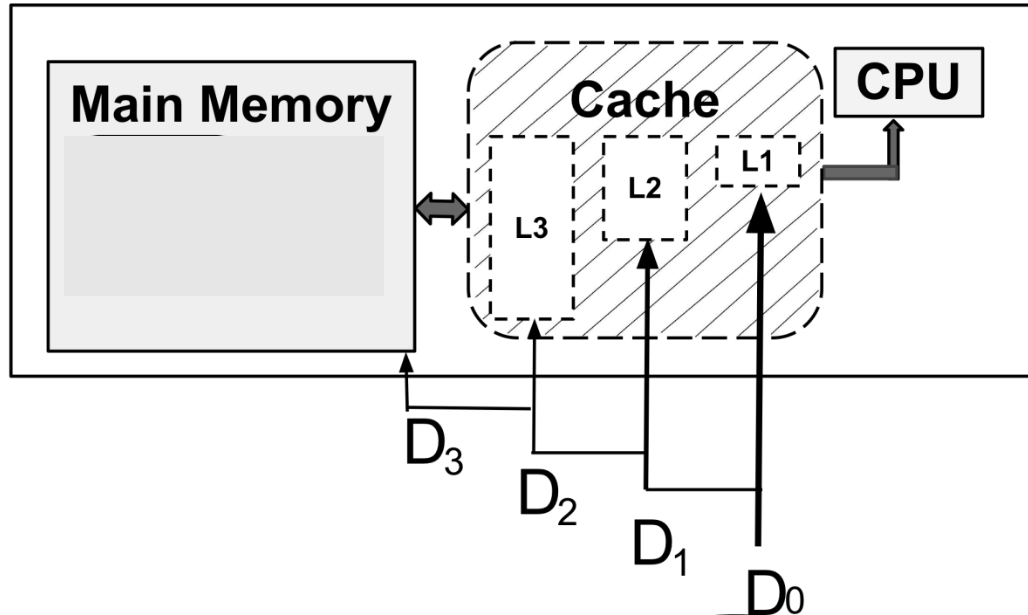
~2 cycles (δ_1)

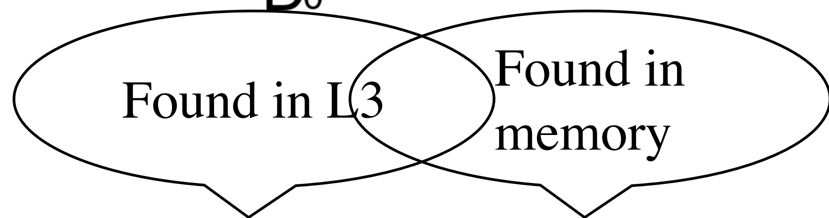
~10 cycles (δ_2)

~100-200 cycles (δ_{mem})

UCSB

Total data access time



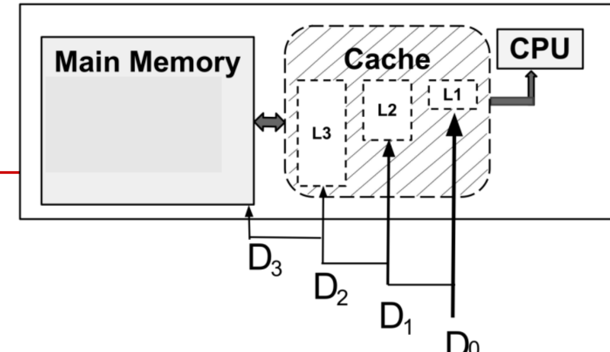


Average memory access time (AMAT)=

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

UCSB

Local vs. Global Miss Rates



- ***Local miss rate*** – the fraction of references to one level of a cache that miss. For example, $m_2 = D_2/D_1$

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 = D_2/D_0
  - L2$ local miss rate >> than the global miss rate

- Notice Global L2 miss rate = $D_2/D_0 = D_1/D_0 * D_2/D_1 = m_1 m_2$

UCSB

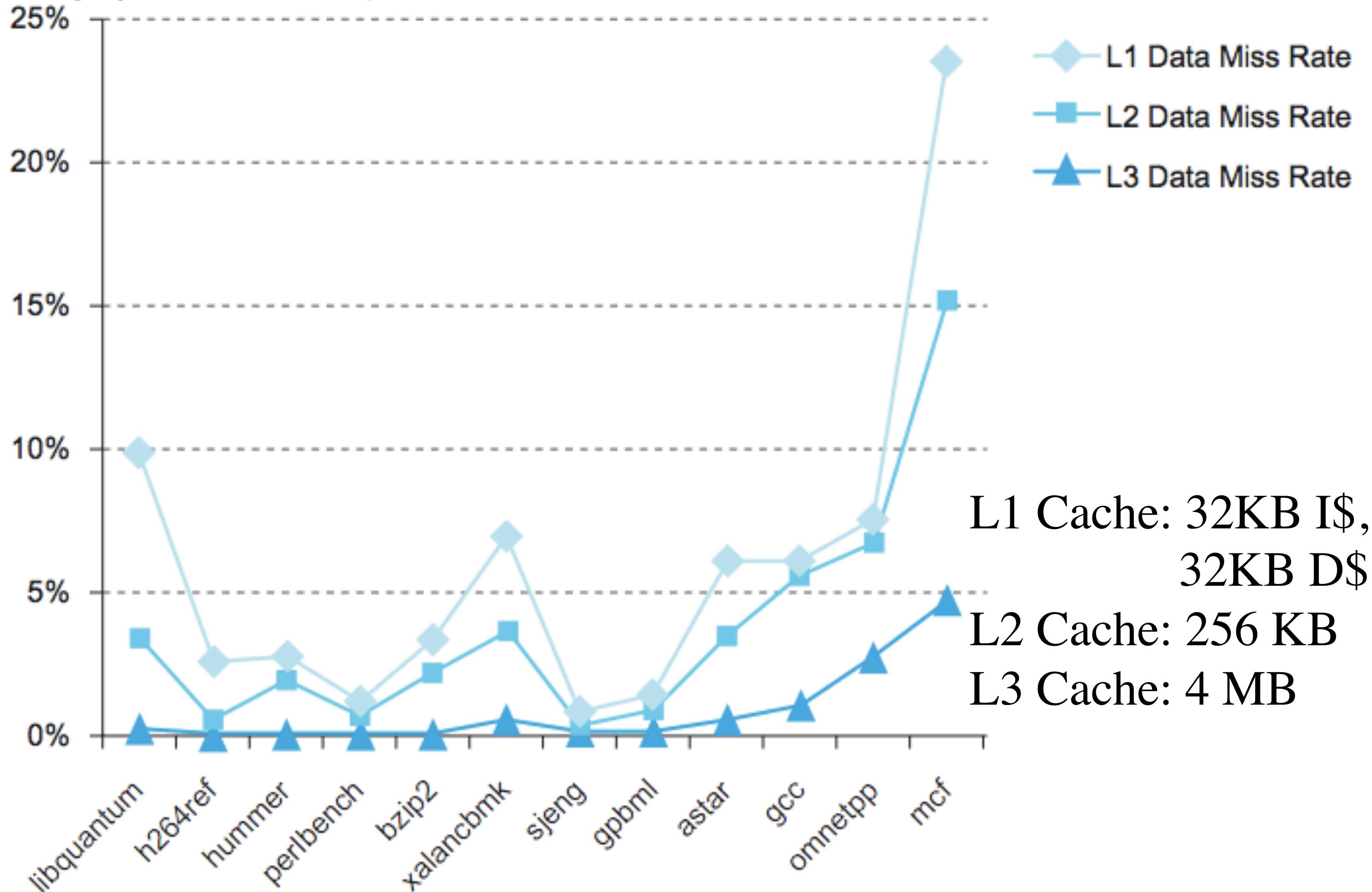
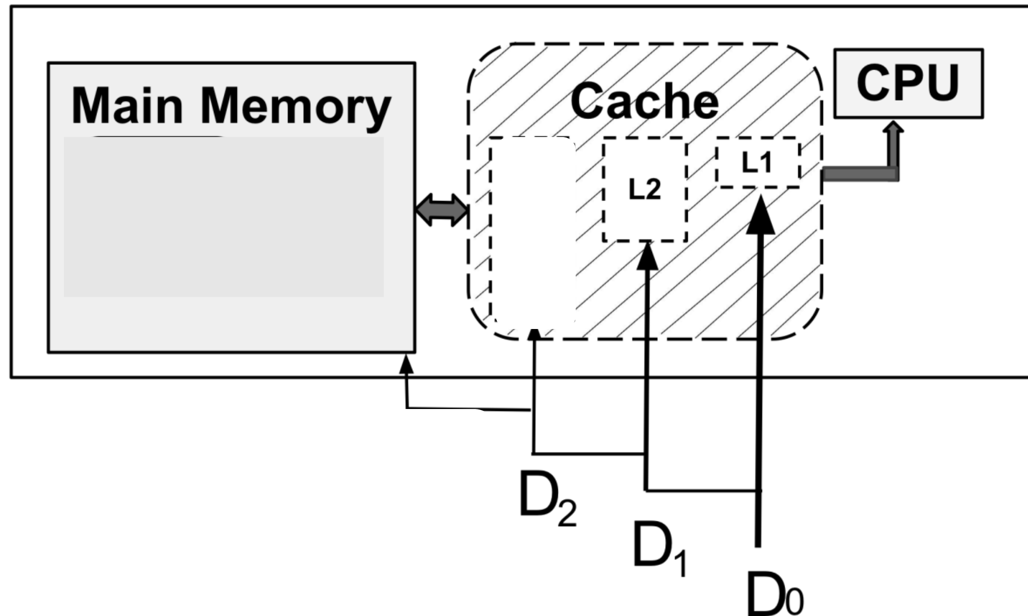


FIGURE 5.47 The L1, L2, and L3 data cache miss rates for the Intel Core i7 920 running the full integer SPECCPU2006 benchmarks.

Average memory access time with no L3 cache



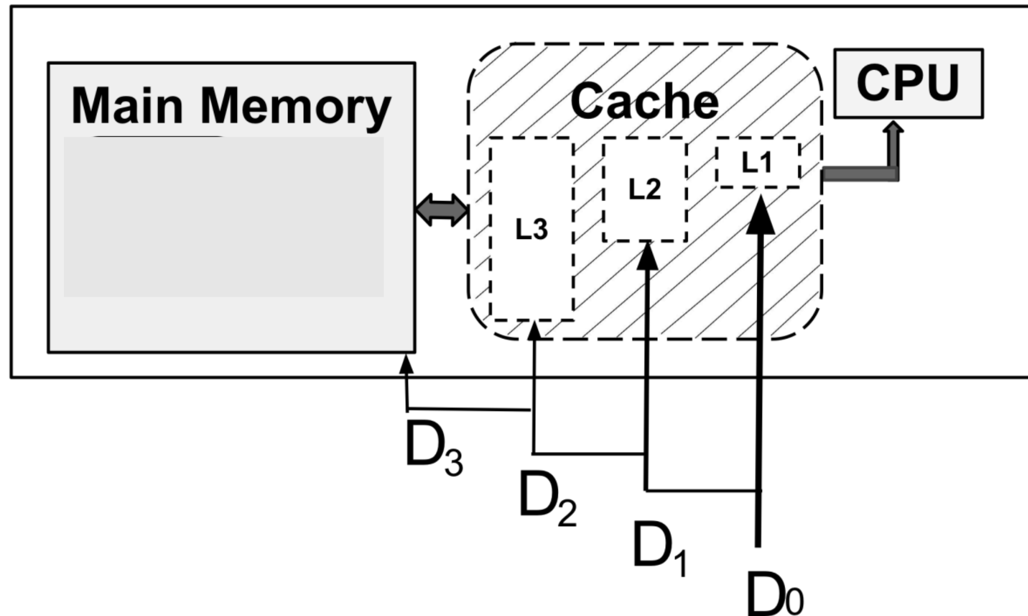
$$\delta_1 + m_1 [\delta_2 + m_2 \delta_{mem}]$$

AMAT =

$$= \delta_1 + m_1 \delta_2 + m_1 m_2 \delta_{mem}$$

$$= \delta_1 + m_1 \delta_2 + GMiss_2 \delta_{mem}$$

Average memory access time with L3 cache



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

$= \delta_1 + m_1 \delta_2 + m_1 m_2 \delta_3 + m_1 m_2 m_3 \delta_{mem}$

$= \delta_1 + m_1 \delta_2 + GMiss_2 \delta_3 + GMiss_3 \delta_{mem}$

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*(10 + 0.4*(50)) + 0.06*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

(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;

(i) What is the local miss rate in the L2$?

20/40 = 50%

(ii) What is the global miss rate in the L2$?

20/1000 = 2%

(iii) What is the local miss rate in the L3$?

10/20 = 50%

(iv) What is the global miss rate in the L3$?

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$?*

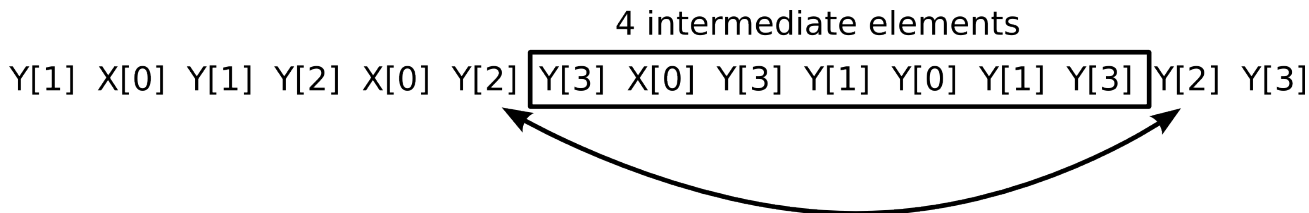
1 + 4%*10 + 2%*400 = 1 + 0.4 + 8 = 9.4

Cache-aware Programming

- **Reuse values in cache as much as possible**
  - **exploit temporal locality in program**
  - **Example 1:** Y[2] is revisited continously

```
For i=1 to n
   y[2]=y[2]+3
```

  - **Example 2 with access sequence:** Y[2] is revisited after a few instructions later

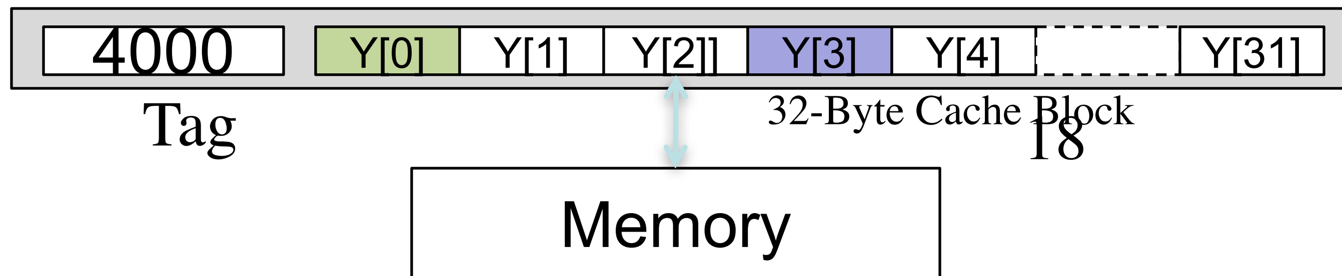


Cache-aware Programming

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

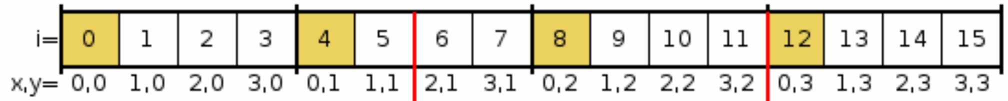
```
For i=1 to n
   y[i]=y[i]+3
```

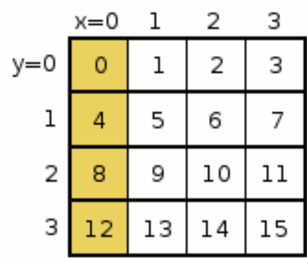
Visiting Y[1] benefits next access of Y[2]

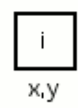


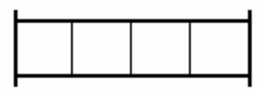
UCSB

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] …

UCSB

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

```
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

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

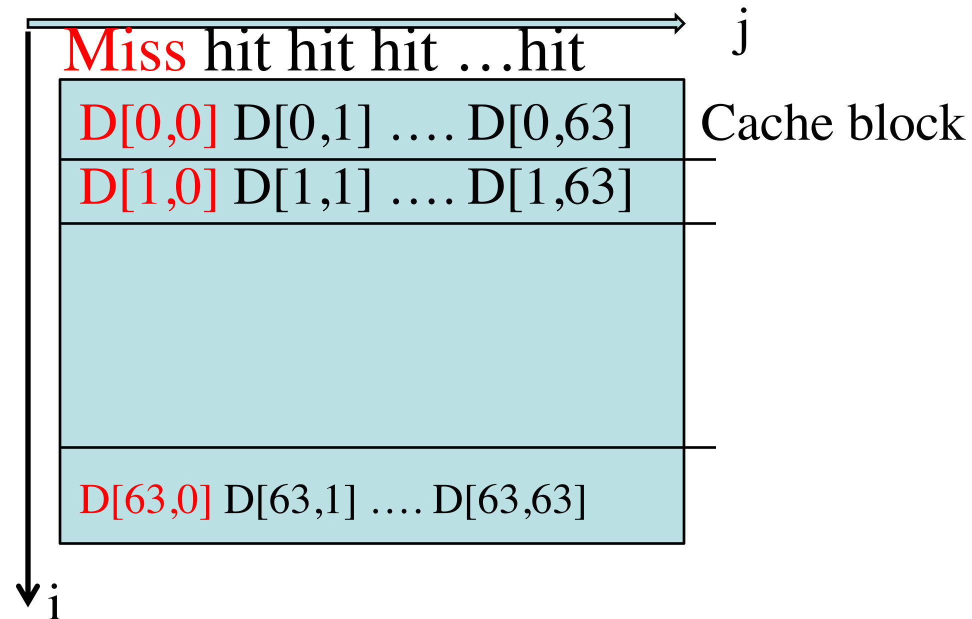
What is cache miss rate?

UCSB

Data Access Pattern and cache miss

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

1 cache miss
in one inner loop
iteration

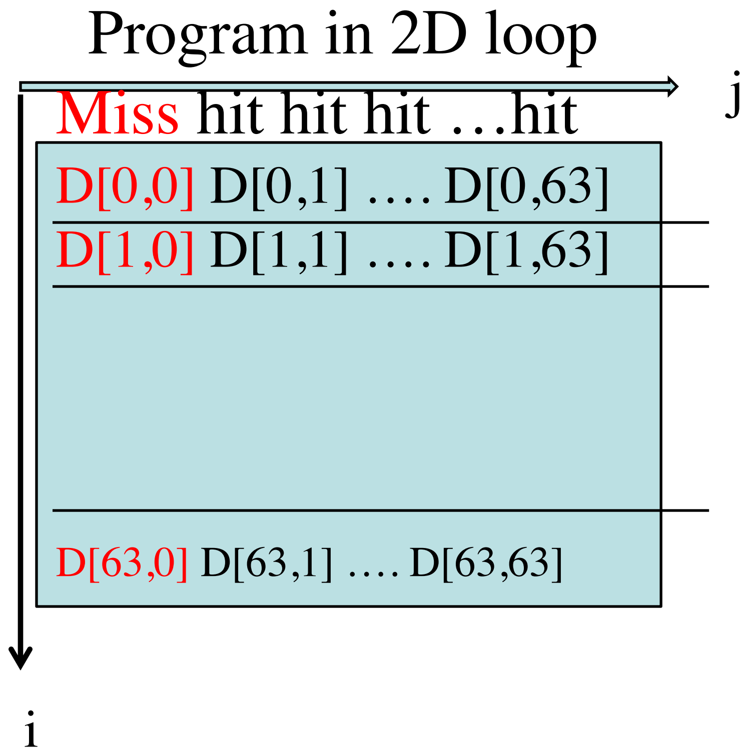


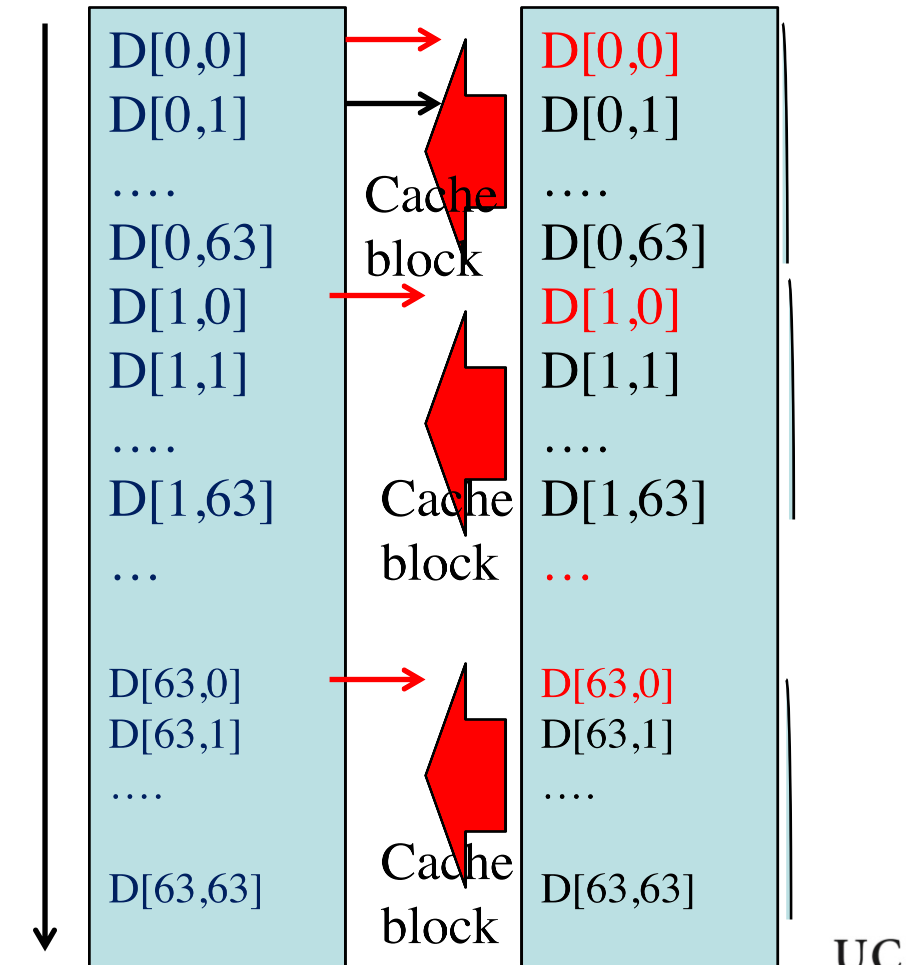
64 cache miss out of 64*64 access.
There is spatial locality. Fetched cache block is used 64 times before swapping out (consecutive data access within the inner loop

UCSB
  ```

Memory layout and data access by block

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





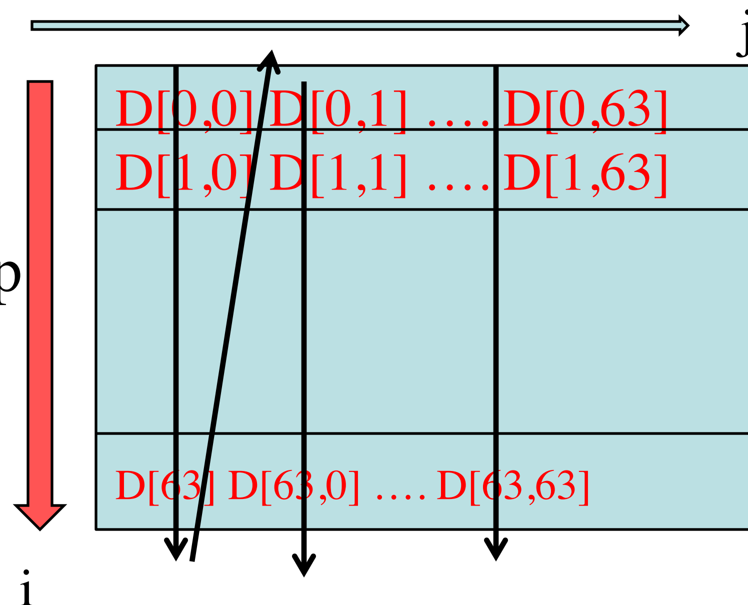
64 cache miss out of 64*64 access.

UCSB

Data Locality and Cache Miss

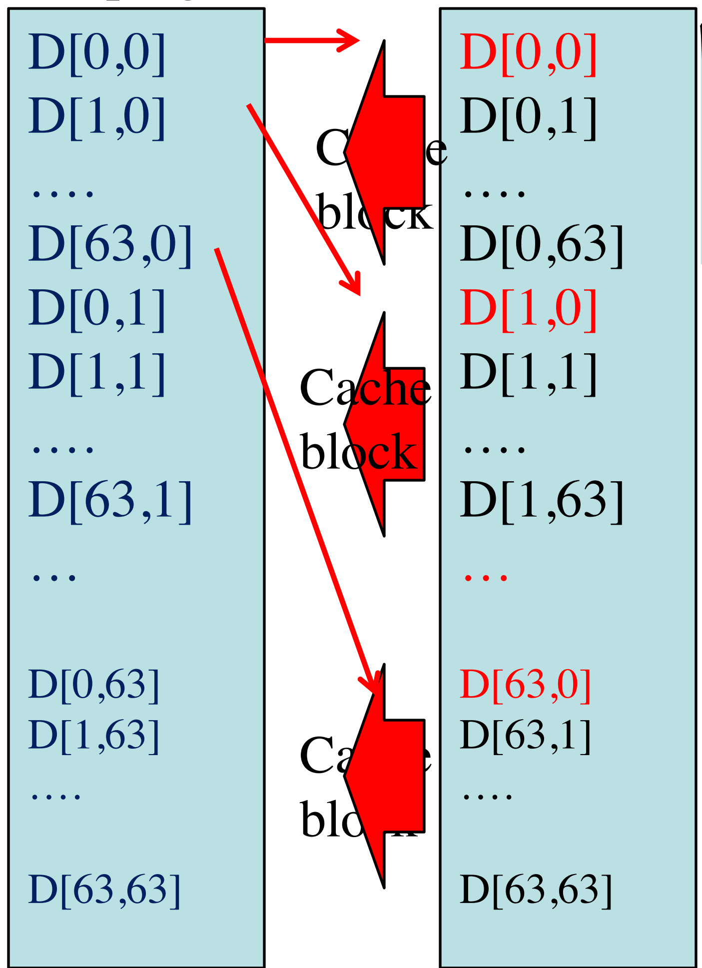
```
• for (j = 0; j <64; j++)
      for (i = 0; i < 64; i++)
          D[i][j] = 0;
```

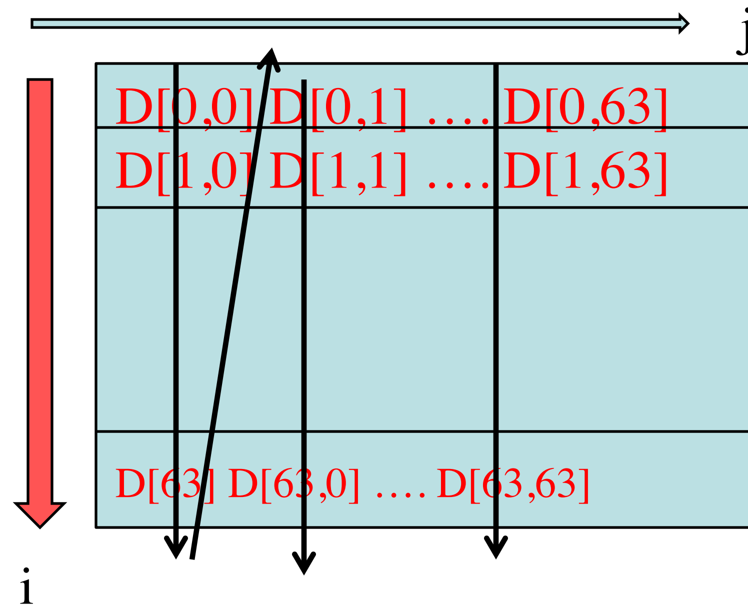
64 cache miss
in one inner loop
iteration



100% cache miss
There is no spatial locality. Fetched block is only used once before swapping out.

Memory layout and data access by block



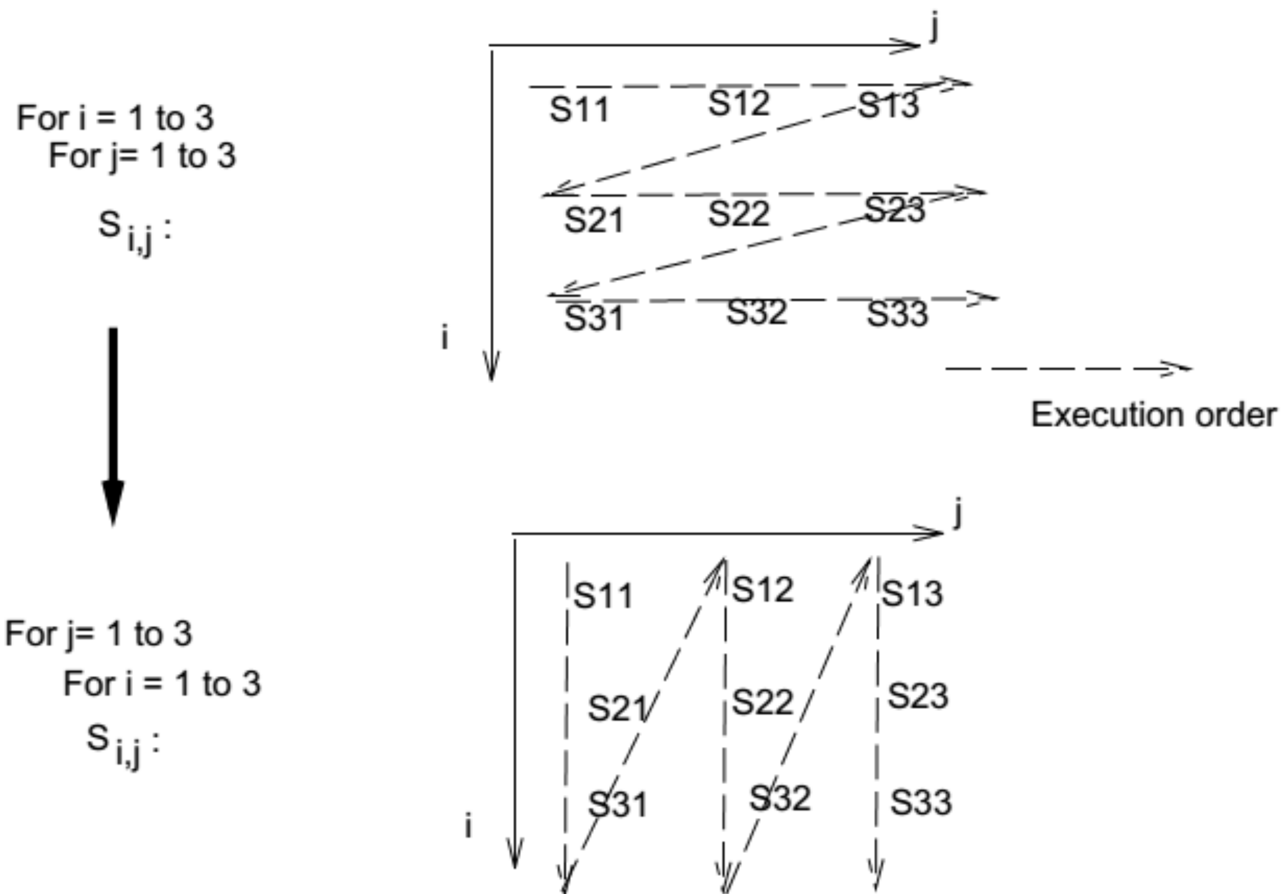


100% cache miss

UCSB

Summary of Example 1: Loop interchange alters execution order and data access patterns

- **Exploit more spatial locality in this case**



UCSB

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)
        A[i] =A[i]+1
```

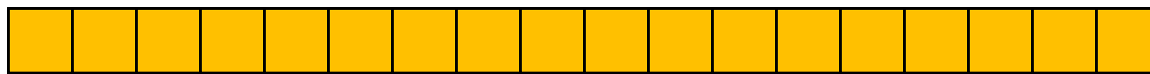
Analyze cache hit ratio when varying cache block size, or step size (stride distance)

UCSB

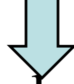
Example 2: Focus on inner loop

```
for (i = 0; i < 64; i +=stepsize)
            A[i] =A[i]+1
```

memory



Cache block



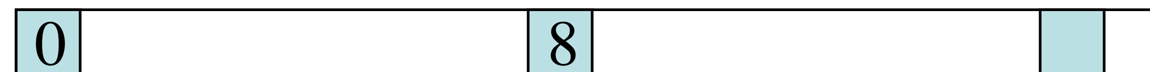
Data access order/index

Stepsize

S=1

S=2

S=4



S=8

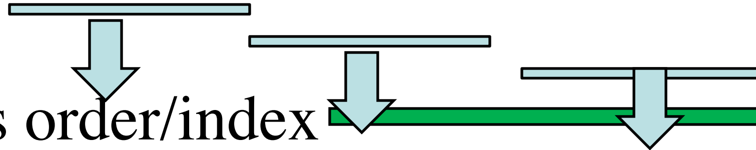
Step size or also called stride distance

Step size =2

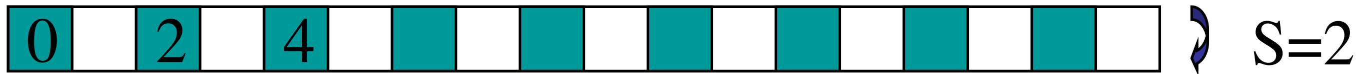
- ```
  for (i = 0; i < 64; I +=stepsize)
                 A[i] =A[i]+1    //read, write A[i]
  ```

Memory

Cache block



Data access order/index



M/H H/H

M/H H/H

M/H H/H

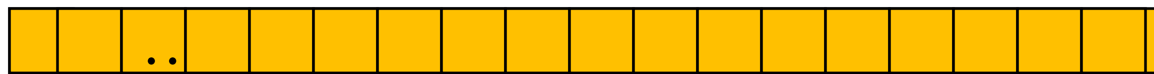
UCSB
  ```

Repeat many times

- `for (k = 0; k<repcount; k++)`

```
for (i = 0; i < 64; I +=stepsize)
        A[i] =A[i]+1    //read, write A[i]
```

Memory



Cache block

Data access order/index

S=2 integers

M/H H/H

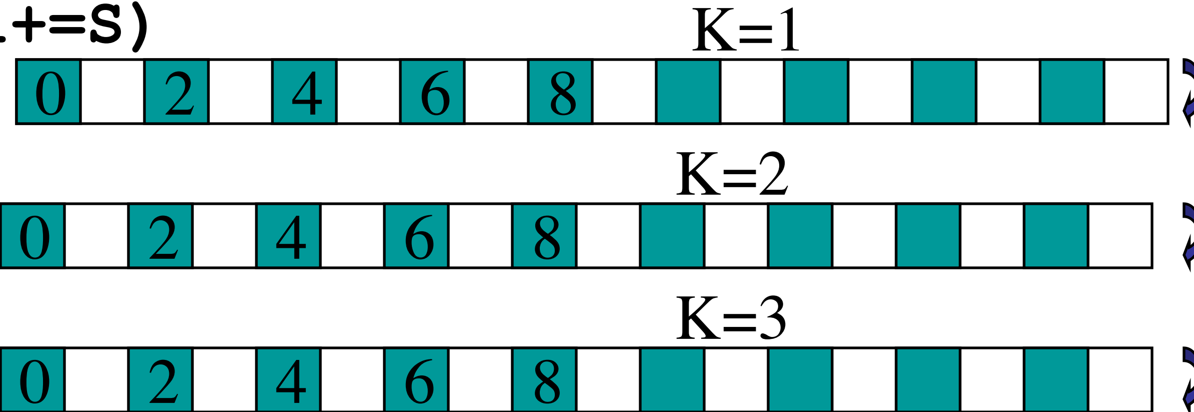
M/H H/H

M/H H/H

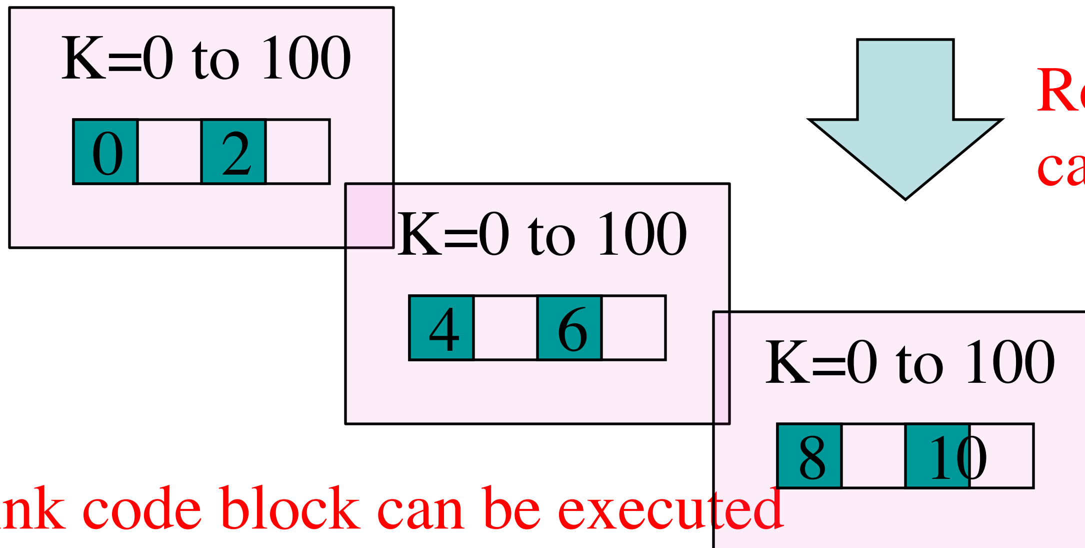
Array has 16 blocks. Inner loop accesses 32 elements, and fetches all 16 blocks. Each block is used as R/W/R/W. Cache size = 8 blocks and cannot hold all 16 blocks fetched.

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



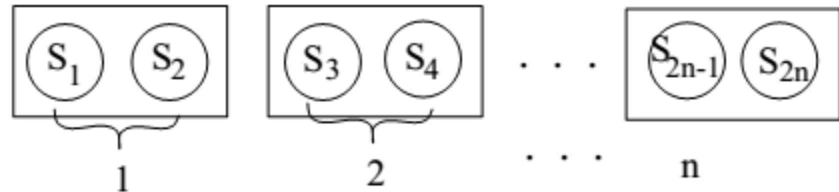
Pink code block can be executed fitting into cache

Rewrite a program loop for better cache usage

- Loop blocking (cache blocking)

For i=1 to 2n

$$S_i : a_i = b_i + c_i$$



with blocksize=2

Rewrite as

For $i = 1$ to n

do S_{2i-1}, S_{2i}

- **More general: Given** `for (i = 0; i < 64; i+=S)`
  `A[i] =f(A[i])`
- **Rewrite as:**

```
for (bi = 0; bi<64; bi=bi+blocksize)
    for (i = bi; i<bi+ blocksize;i+=S)
         A[i] =f(A[i])
```

Example 2: Cache blocking for better performance

- For (k=0; k=100; k++)

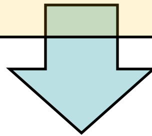
```
for (i = 0; i < 64; i=i+S)
          A[i] =f(A[i])
```

- **Rewrite as:**

  For (k=0; k=100; k++)

```
for (bi = 0; bi<64; bi=bi+blocksize)
    for (i = bi; i<bi+blocksize; i+=S)
          A[i] =f(A[i])
```

Look interchange



```
for (bi = 0; bi<64; bi=bi+blocksize)
```

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

Pink code block can be executed fitting into cache

UCSB

Example 3: Matrix multiplication C=A*B

Cij=Row Ai * Col Bj

```
For i= 0 to n-1
   For j= 0 to n-1
      For k=0 to n-1
         C[i][j] +=A[i][k]* B[k][j]
```

A_1 B_1 B_2 B_3 ... $=$ C_1

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} * \begin{pmatrix} 5 & 7 \\ 6 & 8 \end{pmatrix}$$

$$= \begin{pmatrix} 1*5+2*6 & 1*7+2*8 \\ 3*5+4*6 & 3*7+4*8 \end{pmatrix} = \begin{pmatrix} 17 & 23 \\ 39 & 53 \end{pmatrix}$$

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]
```

UCSB
```

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)

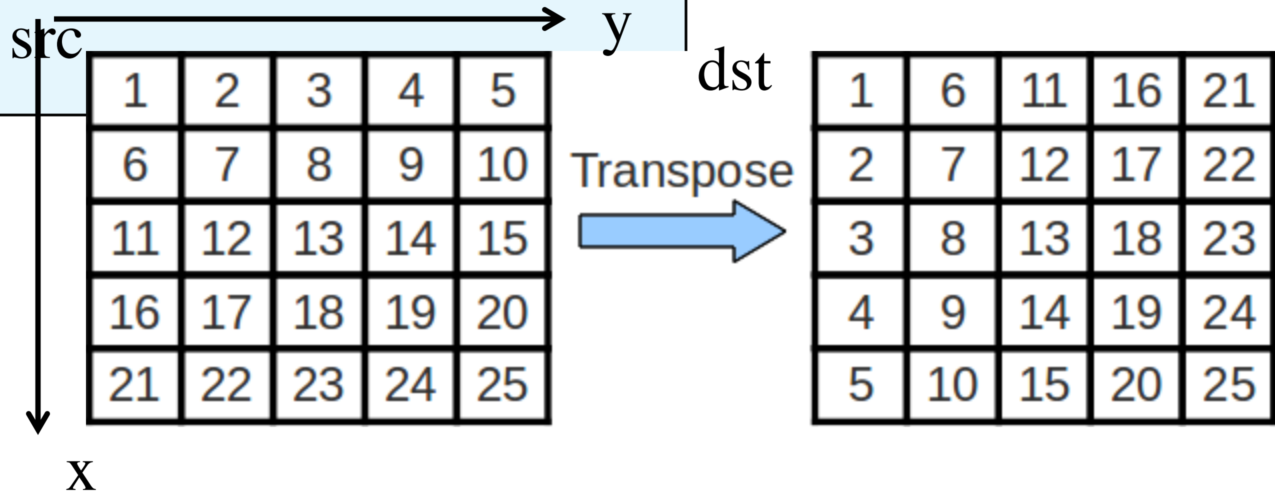
Which code has better cache performance (faster)?

-- Study impact of stride on inner most loop which does most computation

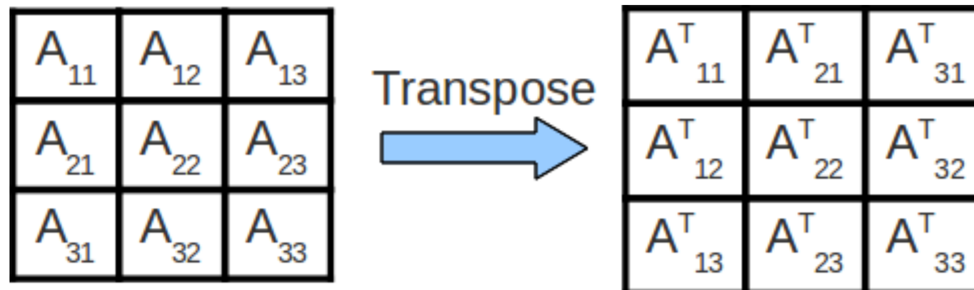
```
    for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
        for (i = 0; i < n; i++)
          C[i+j*n] += A[i+k*n]* B[k+j*n]
```

Example 4: Cache blocking for matrix transpose

```
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



UCSB

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

A_{11}	A_{12}	A_{13}
A_{21}	A_{22}	A_{23}
A_{31}	A_{32}	A_{33}

Transpose

A^T_{11}	A^T_{21}	A^T_{31}
A^T_{12}	A^T_{22}	A^T_{32}
A^T_{13}	A^T_{23}	A^T_{33}

```
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];
```