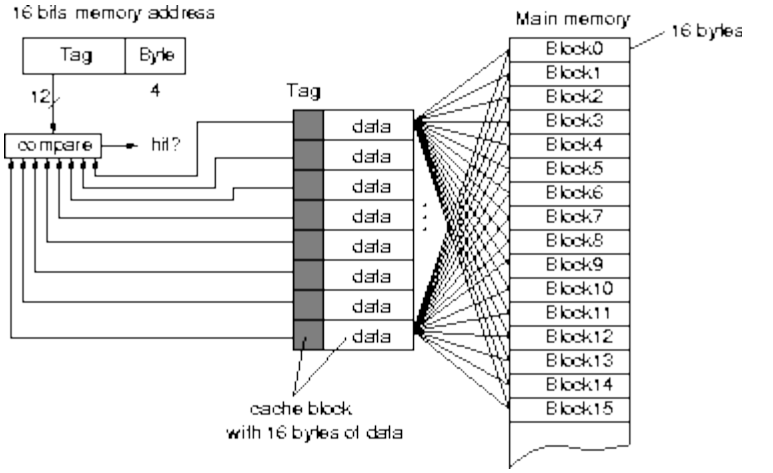
cse5441 - parallel computing

cache management

part two



https://staff.fnwi.uva.nl/a.d.pimentel/apr/img14.gif

thrashing

example	m	C	B	E	S	t	s	b
X1	32	1024	32	1	32	22	5	5

let @x[256] = AAAA0000
@y[256] = AAAA0400

sizeof(float) = 4 bytes

sum is in a register

```
float dotprod (float x[256], float y[256])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 256; i++)
    {
        sum  +=  x[ i ] * y[ i ];
    }
    return sum;
}
```

x[0] @ AAAA0000 = 1010 1010 1010 1010 0000 0000 0000 0000
set bits = 00 000

y[0] @ AAAA0400 = 1010 1010 1010 1010 0000 0100 0000 0000
set bits = 00 000

set-associative cache

example

example	m	C	B	E	S	t	s	b
X2	32	2048	32	2	32	22	5	5

let @x[256] = AAAA0000
@y[256] = AAAA0400

sizeof(float) = 4 bytes

sum is in a register

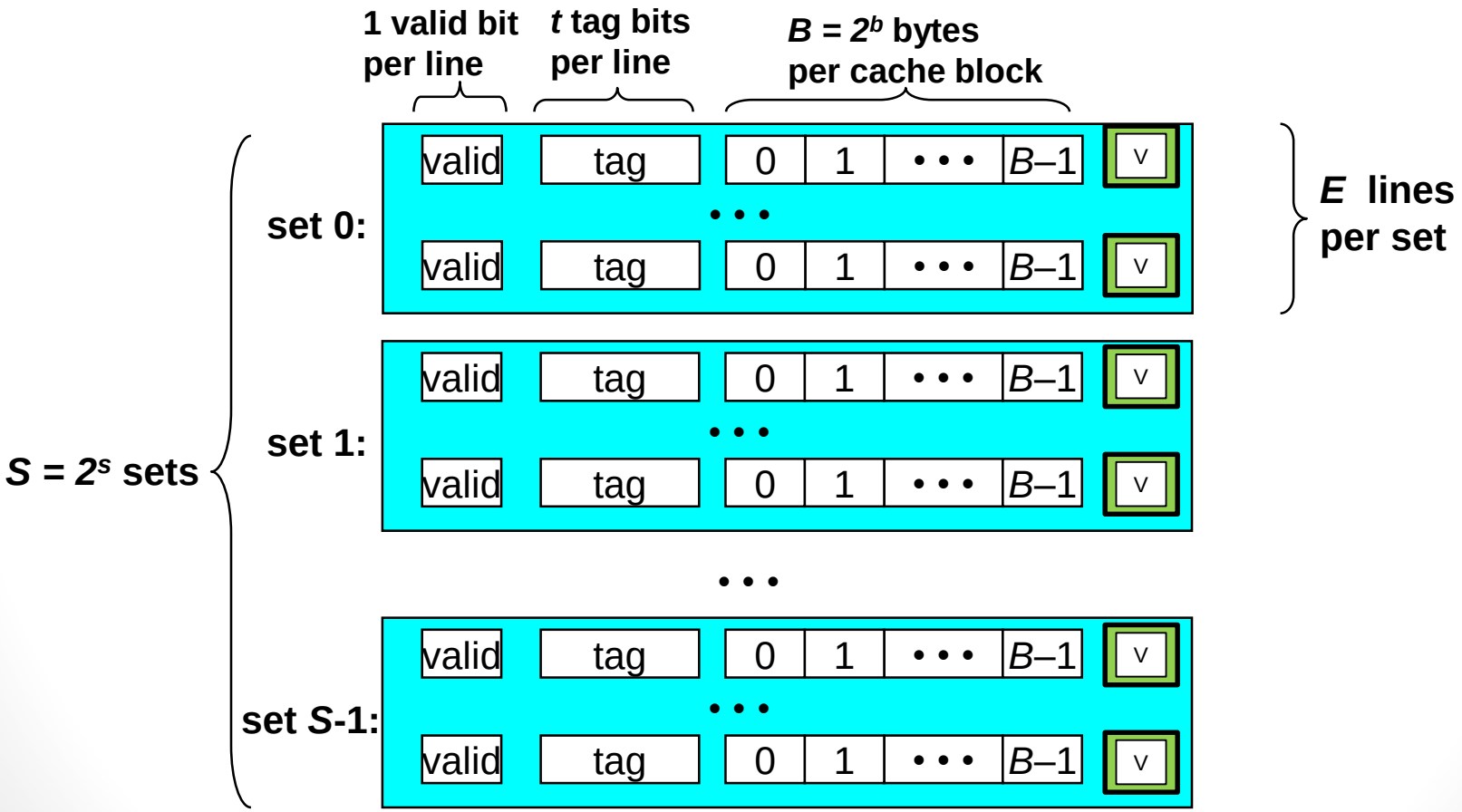
```
float dotprod (float x[256], float y[256])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 256; i++)
    {
        sum  +=  x[ i ] * y[ i ];
    }
    return sum;
}
```

x[0] @ AAAA0000 = 1010 1010 1010 1010 0000 0000 0000 0000
set bits = 00 000 fills 1st line in set

y[0] @ AAAA0400 = 1010 1010 1010 1010 0000 0100 0000 0000
set bits = 00 000 fills 2nd line in set

set-associative cache

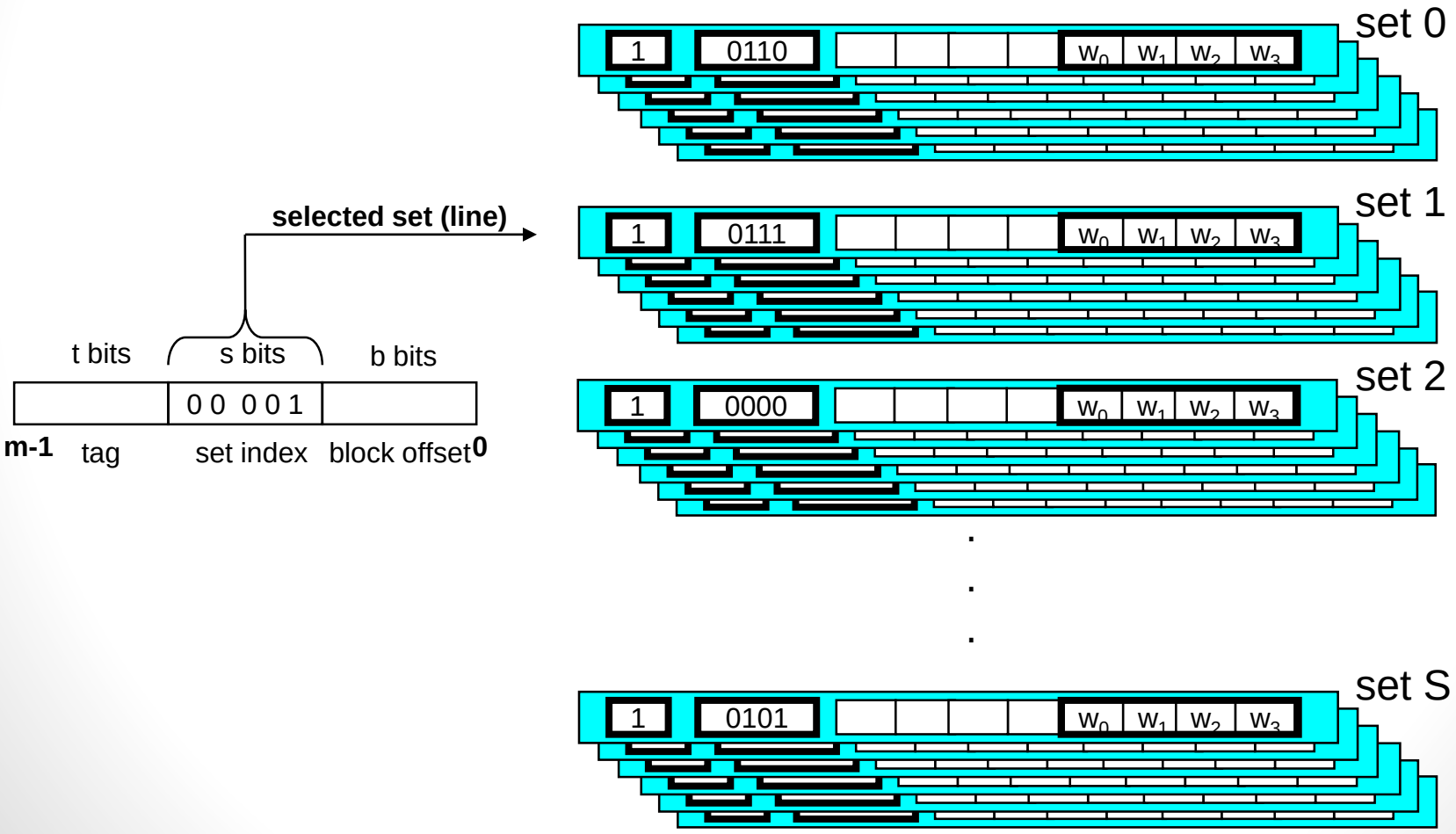
structure



Cache size: $C = B \times E \times S$ data bytes

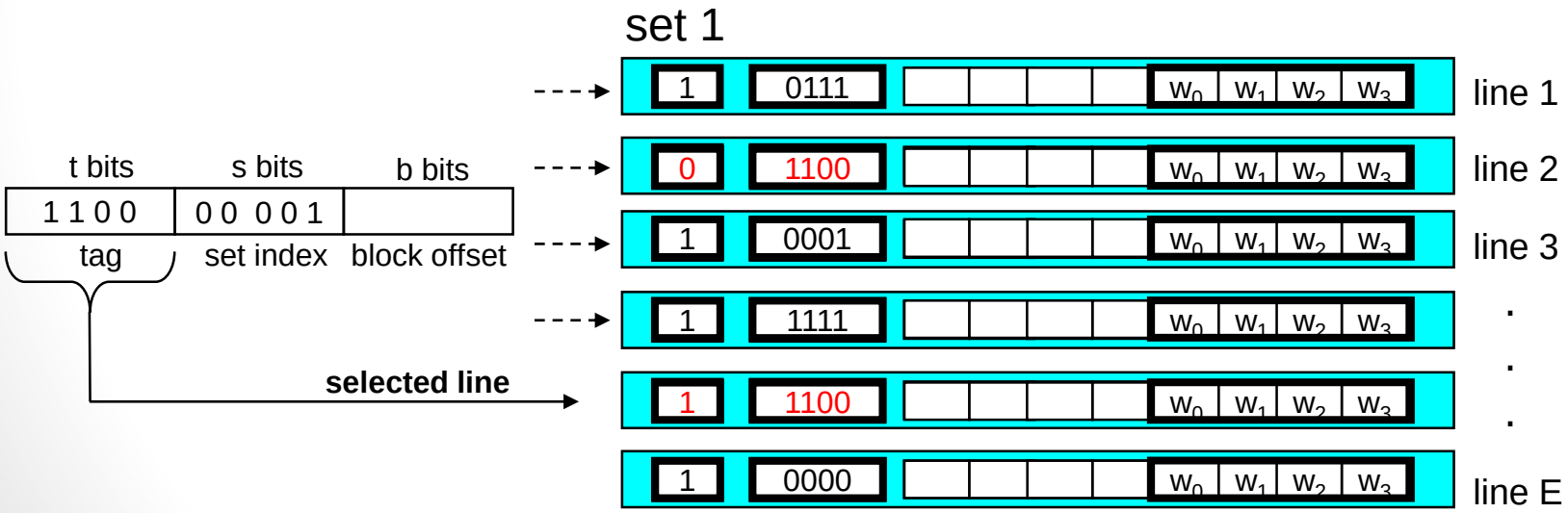
set - associative cache
set selection

- Use the set index bits to determine the set of interest.



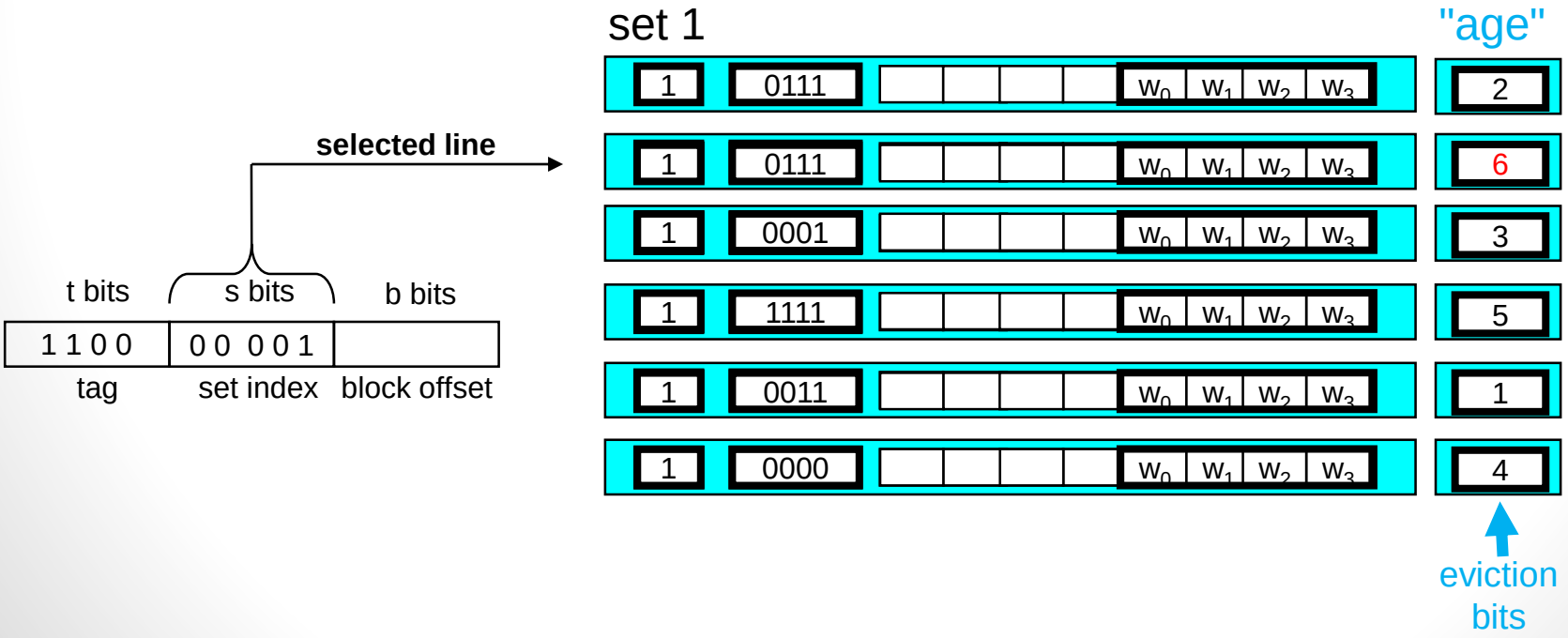
set - associative cache line selection on read hit

- scan each cache line in set
  - if valid bit is set
    - if tag matches
      - cache hit



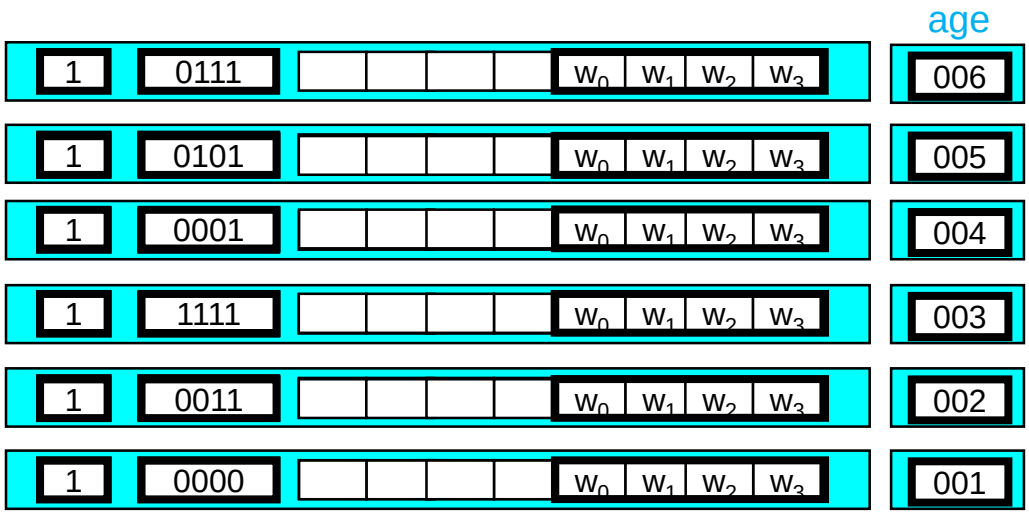
set - associative cache line selection on read miss

- scan each cache line in set, determine is a miss
- retrieve from slower memory
- scan for empty line
  - if full, pick a loser

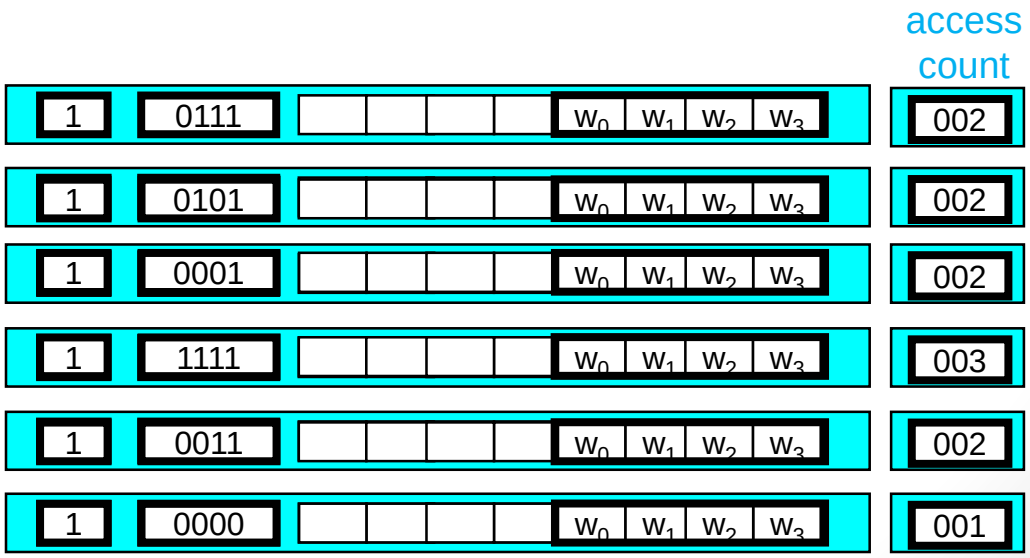


LRU / LFU

LRU
sequential
access



LFU
equivalent
access



it's your turn . . .

problem	m	C	B	E	S	t	s	b
IYT 3-09	64	4096	32	4	32			

let @x[1024] = 0...0AAAA0000
@y[1024] = 0...0AAAA1000

sizeof(float) = 4 bytes

sum is in a register
LRU eviction

```
float dotprod (float x[1024], float y[1024])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 1024; i++)
    {
        sum  +=  x[ i ] * y[ i ];
    }
    return sum;
}
```

what are remaining cache parameters?
what is the hit rate for this loop?
what is in the cache at the end of loop execution?

set-associative cache

drawbacks

- sum += x[i] * y[i] * z[i]
- cache size / expense
- time to determine a hit
- what if the cache is full?
  - eviction policy

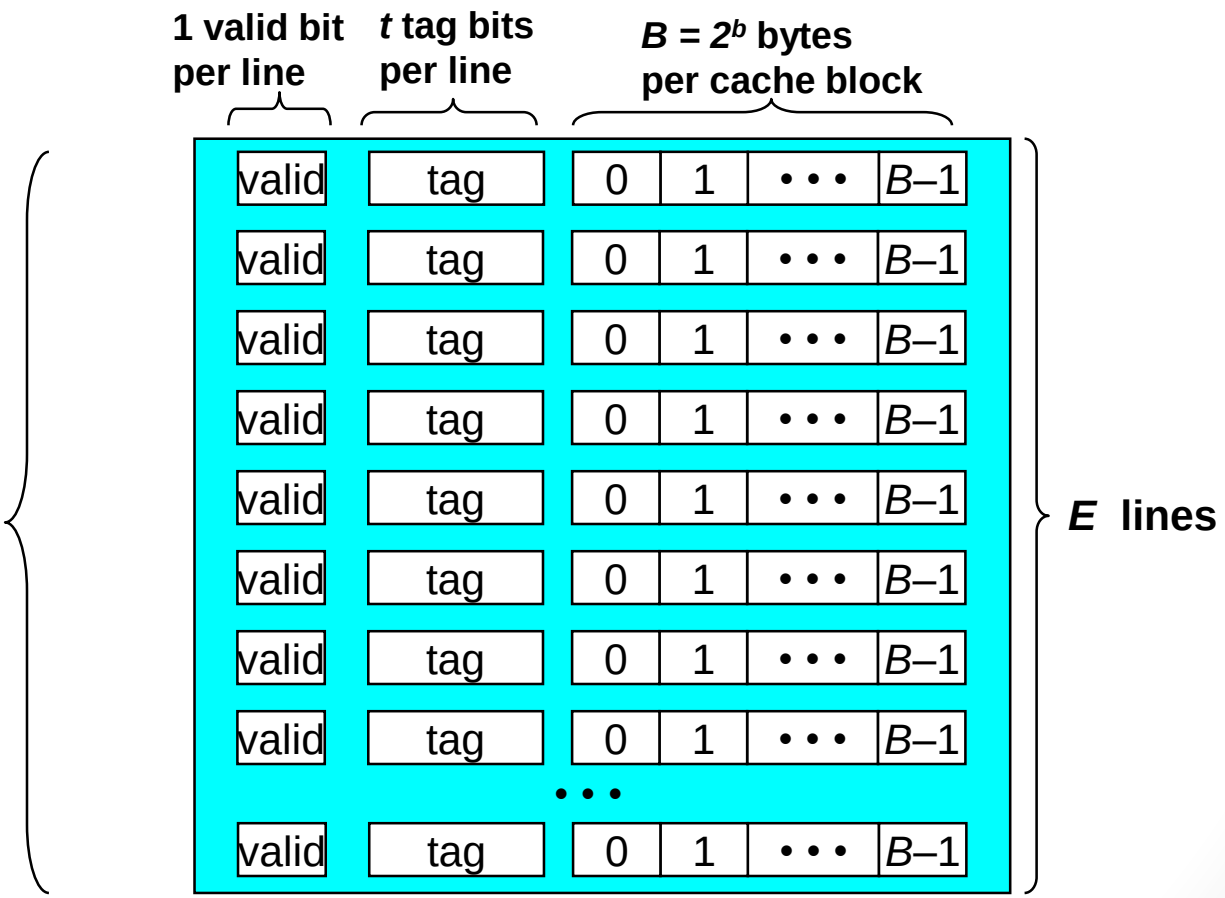
fully associative cache

structure

Cache is an array of sets.

Each set contains one or more lines.

Each line holds a block of data.



Cache size: $C = B \times E$ data bytes

fully associative cache

example	m	C	B	E	S	t	s	b
X3	32	2048	32	64	1	27	0	5

let @x[256] = AAAA0000
@y[256] = AAAA0400

sizeof(float) = 4 bytes

sum is in a register

```
float dotprod (float x[256], float y[256])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 256; i++)
    {
        sum  +=  x[ i ] * y[ i ];
    }
    return sum;
}
```

x[0] @ AAAA0000 = 1010 1010 1010 1010 0000 0000 0000 0000
set bits = fills 1st line in set

y[0] @ AAAA0400 = 1010 1010 1010 1010 0000 0100 0000 0000
set bits = fills 2nd line in set

... and so forth ...

it's your turn . . .

problem	m	C	B	E	S	t	s	b
IYT 3-13	32	2048	32	64	1	?	?	?

let @x[1024] = 0...0AAAA0000
@y[1024] = 0...0AAAA1000
@z[1024] = 0...0AAAA2000

sizeof(float) = 4 bytes

sum is in a register
LRU eviction

```
float dotplus (float x[1024], float y[1024],
                                    float z[1024])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 1024; i++)
    {
        sum  +=  x[ i ] * y[ i ] + z[ i ];
    }
    return sum;
}
```

what are remaining cache parameters?
what is the hit rate for this loop?
what is in the cache at the end of loop execution?

it's your turn . . .

problem	m	C	B	E	S	t	s	b
IYT 3-14	32	2048	32	2	?	?	?	?

let @x[1024] = 0...0AAAA0000
@y[1024] = 0...0AAAA1000
@z[1024] = 0...0AAAA2000

sizeof(float) = 4 bytes

sum is in a register
LRU eviction

```
float dotplus (float x[1024], float y[1024],
                                    float z[1024])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 1024; i++)
    {
        sum  +=  x[ i ] * y[ i ] + z[ i ];
    }
    return sum;
}
```

what are remaining cache parameters?
what is the hit rate for this loop?
what is in the cache at the end of loop execution?

it's your turn . . .

problem	m	C	B	E	S	t	s	b
IYT 3-15	32	2048	32	4	?	?	?	?

let @x[512] = 0...0AAAA0000
@y[512] = 0...0AAAA1000
@z[512] = 0...0AAAA2000

sizeof(float) = 4 bytes

sum is in a register
LRU eviction

```
float dotplus (float x[512], float y[512],
                                    float z[512])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 512; i++)
    {
        sum  +=  x[ i ] * y[ i ] + z[ i ];
    }
    return sum;
}
```

what are remaining cache parameters?
what is the hit rate for this loop?
what is in the cache at the end of loop execution?

set associative cache

example

problem	m	C	B	E	S	t	s	b
X4	64	4096	32	4	32	54	5	5

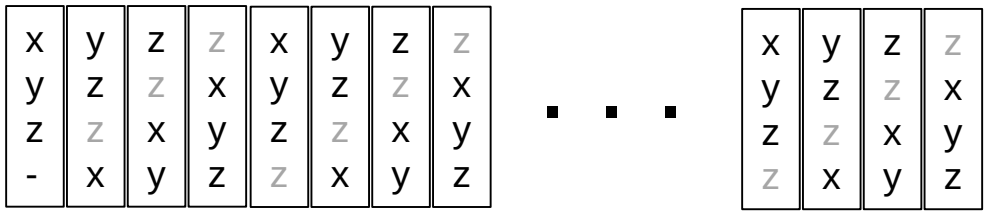
let @x[1024] = 0...0AAAA0000
@y[1024] = 0...0AAAA1000
@z[1024] = 0...0AAAA2000

sizeof(float) = 4 bytes

sum is in a register
LRU eviction

```
float dotplus(float x[1024], float y[1024],
                                    float z[1024])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 1024; i++)
    {
        sum  +=  x[ i ] * y[ i ] + z[ i ];
    }
    return sum;
}
```

for each set:
line 1
line 2
line 3
line 4



http://images.clipartpanda.com/under-construction-clipart-under-constr-01.jpg

fully associative cache

drawbacks

- more complex and expensive logic (compared to DMC)
- increased controller logic may slow low-level cache access times
- increased scan lengths may slow low-level cache access times
- preserving access times may decrease feasible cache size
- totally unnecessary for many common access patterns

cache associativity

generally speaking ...

direct mapped cache	n-way set associative mapped cache	fully associative mapped cache
high capacity cheap fast		less capacity expensive slower
thrashing / high conflict misses		no conflict misses

https://image.shutterstock.com/image-illustration/middle-ground-compromise-negotiation-venn-260nw-670579153.jpg

cache management - part two

cache properties

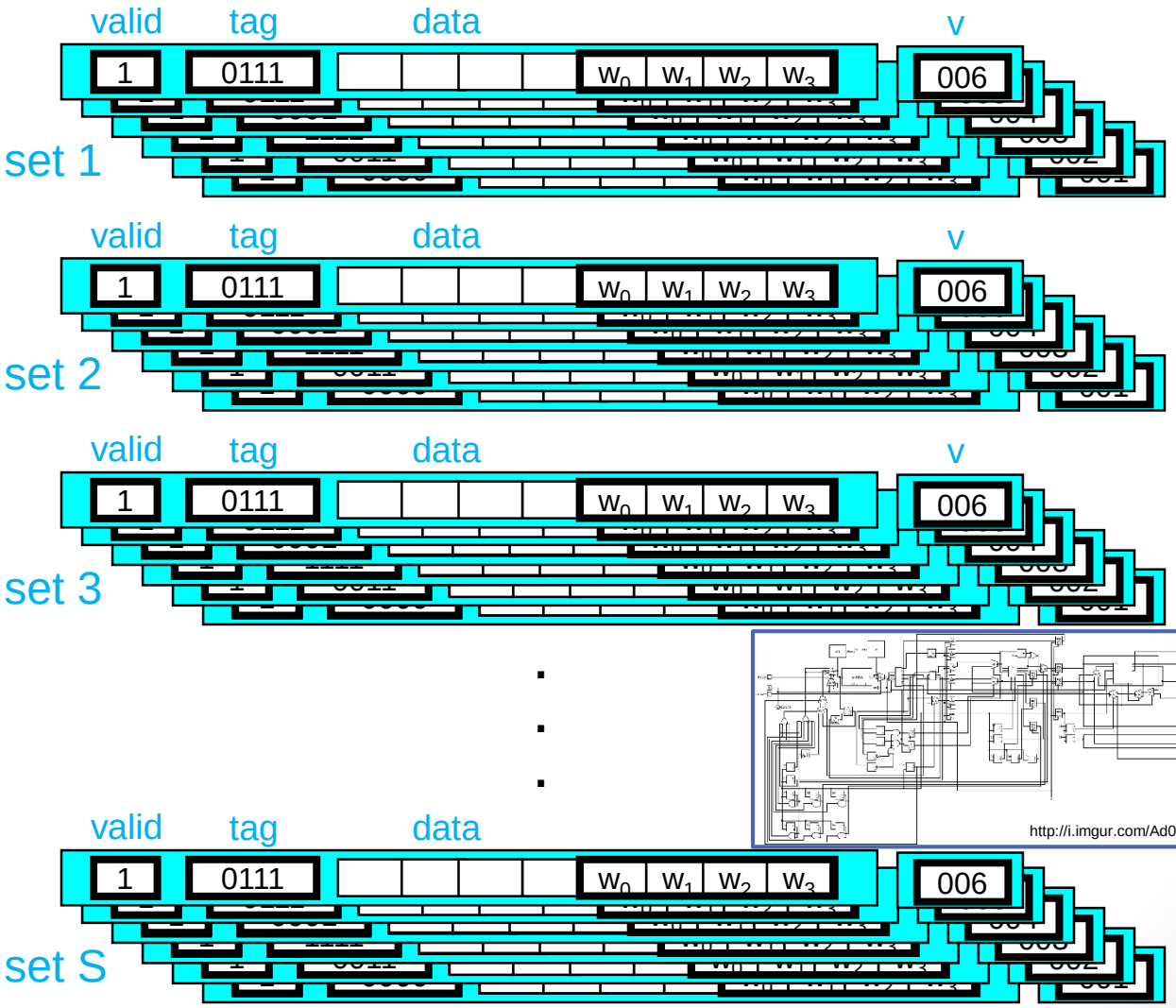
simplified cache implementation size

cache "size"
= S x E x B
bytes

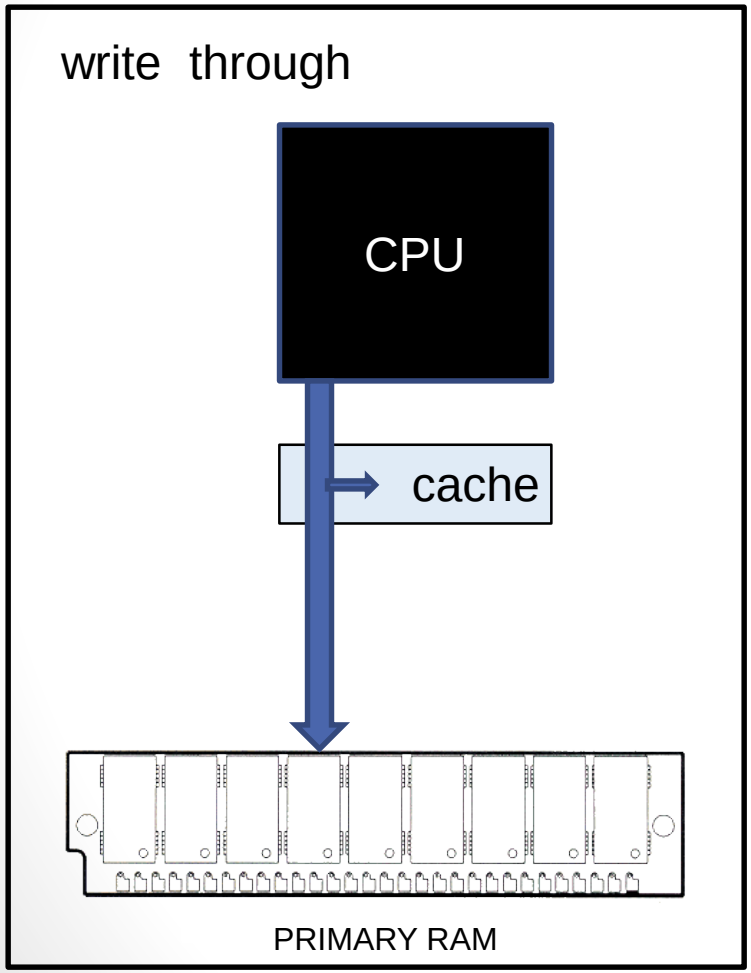
cache
"implementation size"

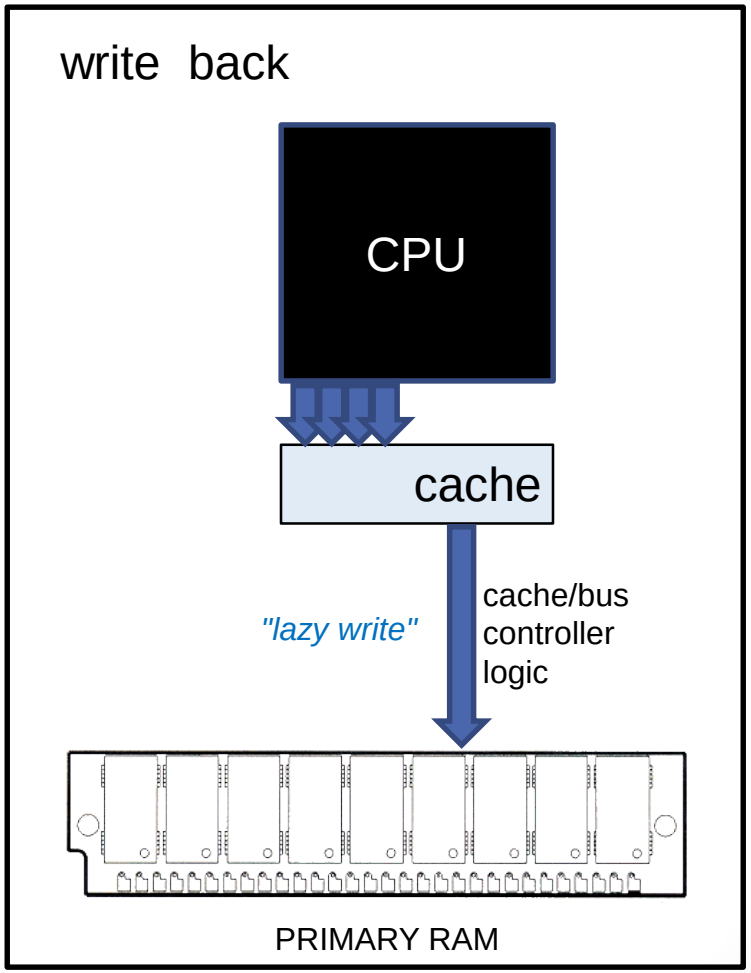
= S x E x
(1 +
t +
8B +
ε
)
bits

ε = |v|



cache writes





http://www.techadvice.com/tech/images/Simm30a.gif

unification and sharing

cache content

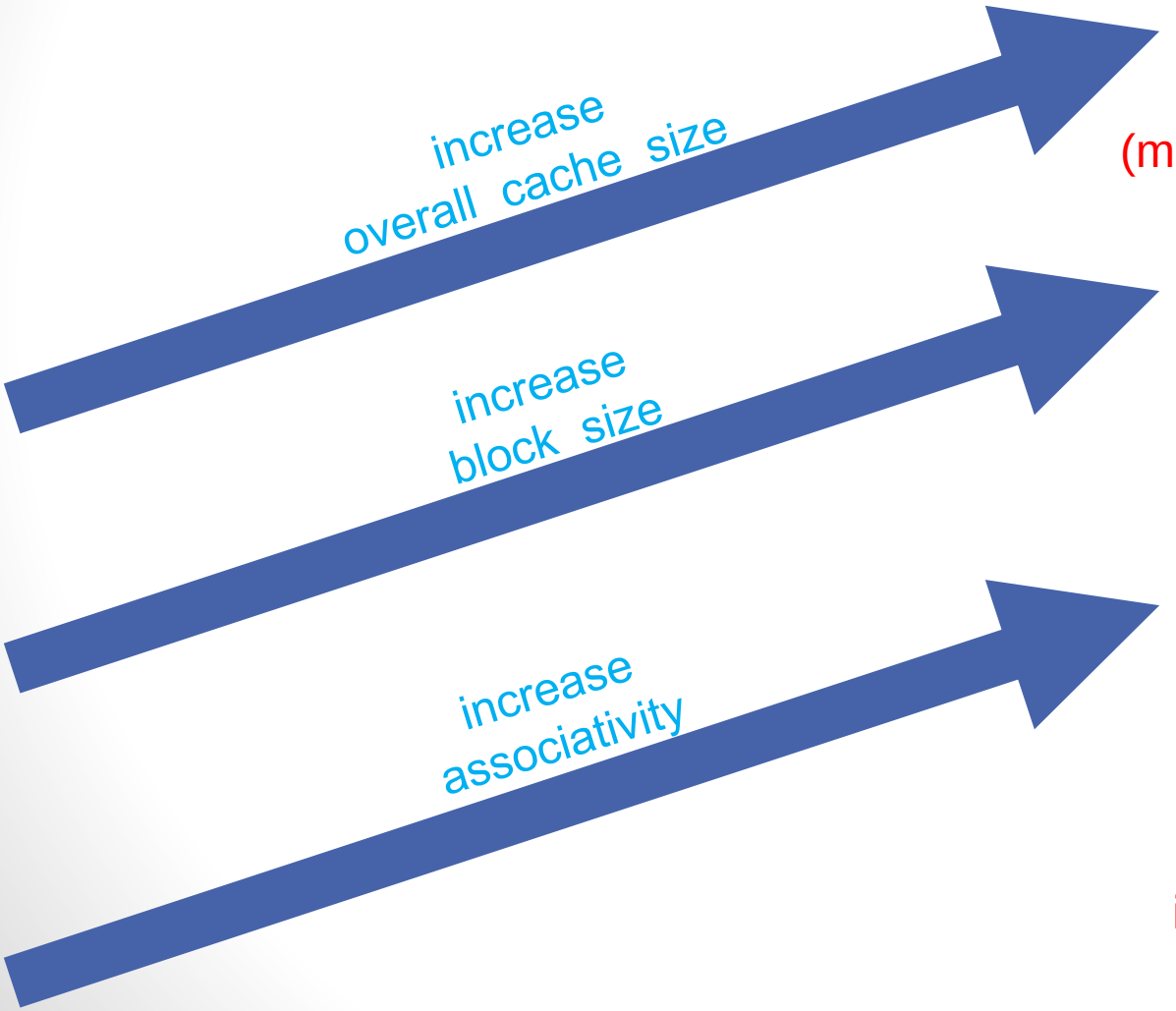
- split instruction and data caches
- unified cache

cache access

- private cache
- shared cache

affects of cache size parameters

generalization



improved
hit
rate

slower,
(meet one-cycle target?)

better
spatial
locality

fewer lines,
larger miss penalty

decreased
thrashing

expensive,
slow,
more complex,
increased penalites

cache management - part two

cache performance metrics

overall cache performance

Average Memory Access Time (AMAT)

t_hit	hit time	cycles to complete detection and transfer of a cache hit
prob_miss	miss probability	% time for cache miss
penalty_miss	miss penalty	additional cycles incurred to process a cache miss

overall cache performance

AMAT = t_hit + prob_miss * penalty_miss

(hit time + (miss rate * miss penalty)

- reduce hit time
- reduce miss penalty
- reduce miss rate

overall cache performance

single-level cache

AMAT = t_hit + prob_miss * penalty_miss

- L1$ hits in 1 cycle, with 80% hit rate
- main memory hits in 1000 additional cycles

AMAT = 1 + (1-.8)(1000)
= 201

- L1$ hits in 1 cycle, with 85% hit rate
- main memory hits in 1000 additional cycles

AMAT = 1 + (1-.85)(1000)
= 151

overall cache performance

multi-level cache

$AMAT_i = t\_hit_i + prob\_miss_i * penalty\_miss_i$

- L1$ hits in 1 cycle, with 50% hit rate
- L2$ hits in 10 cycles, with 75% hit rate
- L3$ hits in 100 cycles, with 90% hit rate
- main memory hits in 1000 cycles

$$
\begin{aligned}
AMAT_1 &= 1 + (1-.5)(AMAT_2) \\
&= 1 + (1-.5)(10 + (1-.75)(AMAT_3)) \\
&= 1 + (1-.5)(10 + (1-.75)(100 + (1-.9)(AMAT_m))) \\
&= 1 + (1-.5)(10 + (1-.75)(100 + (1-.9)(1000))) \\
&= 31
\end{aligned}
$$

what percentage of accesses hit main memory?

it's your turn . . .

overall cache performance

$AMAT_i = t\_hit_i + prob\_miss_i * penalty\_miss_i$

- L1$ hits in 1 cycle, with 25% hit rate
- L2$ hits in 6 cycles, with 80% hit rate
- L3$ hits in 60 cycles, with 95% hit rate
- main memory hits in 1000 cycles

$AMAT_1 = ??$

what percentage of accesses hit main memory?

it's your turn . . .

overall cache performance

$AMAT_i = t\_hit_i + prob\_miss_i * penalty\_miss_i$

- L1$ hits in 1 cycle, with 70% hit rate
- L2$ hits in 20 cycles, with 70% hit rate
- L3$ hits in 200 cycles, with 70% hit rate
- main memory hits in 1000 cycles

$AMAT_1 = ??$

what percentage of accesses hit main memory?

cache management - part two

cache friendly code

writing cache-friendly code

- maximize spatial locality in data organization
- maximize spatial locality in data access
- maximize temporal locality
- engineer access strides
- padding
- blocking / tiling
- computational structure (ordered fetching)

spatial locality in data organization

```
struct  part_type
{
    int         id;
    int         value;
    int         supplier_id[MAX_SUPPLIER];
    blob        image;
}  parts[MAX_PARTS];
```

spatial locality in data access

A

```
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)
    {
        sum[ j ]  += matrix[ j ][ i ]
    }
}
```

B

```
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)
    {
        sum[ i ]  += matrix[ i ][ j ]
    }
}
```

temporal locality

B

```
for (int i = 0; i < cols; i++)
{
    for (int j = 0; j < rows; j++)
    {
        sum[ i ]  += matrix[ i ][ j ]
    }
}
```

access strides

```
int a[size], b[size], c[size];

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

let size = 2048
sizeof(int) = 4
direct mapped cache,
m = 32
B = 32
S = 64
(E = 1)

padding

```
int a[pSize], b[pSize], c[pSize];

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

let size   = 2048
sizeof(int) = 4
direct mapped cache,
    m = 32
    B = 32
    S = 64
   (E =   1)

let pSize   = size + B/4 = 2056
sizeof(int) = 4
direct mapped cache,
    m = 32
    B = 32
    S = 64
   (E =   1)

blocking / tiling

sizeof(float) = 8
C = 64K
m = 32
B = 32
S = 64
E = 1

note border condition is finessed ...

```
float  A [1024] [1024],  R [1024],  C [1024]

for ( i = 0; i < 1024; i++ )

    for ( j = 0; j < 1024; j++ )


        R[ i ]  +=  A[ i ][ j ]
        C[ i ]  -=  A[ i ][ j ]

        many other computations ...
```

```
float  A [1024] [1024],  R [1024],  C [1024]

for ( i = 0; i < 1024; i++ )

    for ( jb = 0; jb < 1024; jb += 4 )

        for ( j = 0; j < 4; j++ )
            R[ i ]  +=  A[ i ][ jb+j ]
            C[ i ]  +=  A[ i ][ jb+j ]

        many other computations ...
```


http://images.clipartpanda.com/under-construction-clipart-under-constr-01.jpg

ordered fetching

approach A:

declare large data structure A
initialize large data structure A

declare large data structure B
initialize large data structure B

declare large data structure C
initialize large data structure C

.
.
.

compute A
compute B
compute C

approach B:

declare large data structure A
initialize large data structure A
compute A
.
.
.

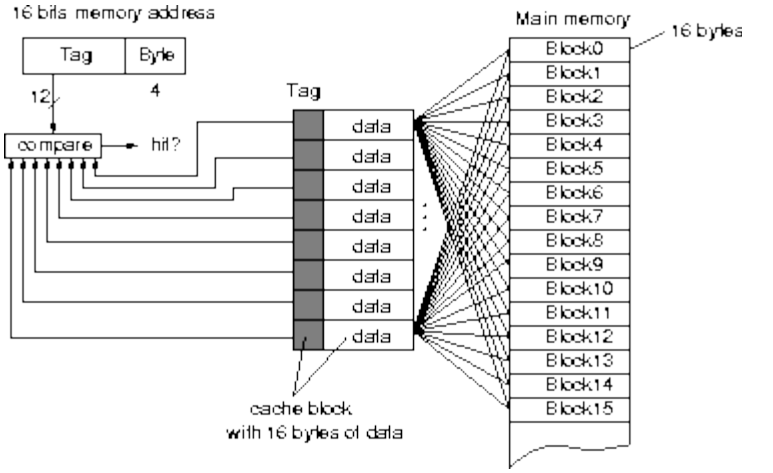
declare large data structure B
initialize large data structure B
compute B
.
.
.

declare large data structure C
initialize large data structure C
compute C

cse5441 - parallel computing

cache management

part two



https://staff.fnwi.uva.nl/a.d.pimentel/apr/img14.gif