Lecture 09
Caches – part 3

Euhyun Moon, Ph.D.

Machine Learning Systems (MLSys) Lab

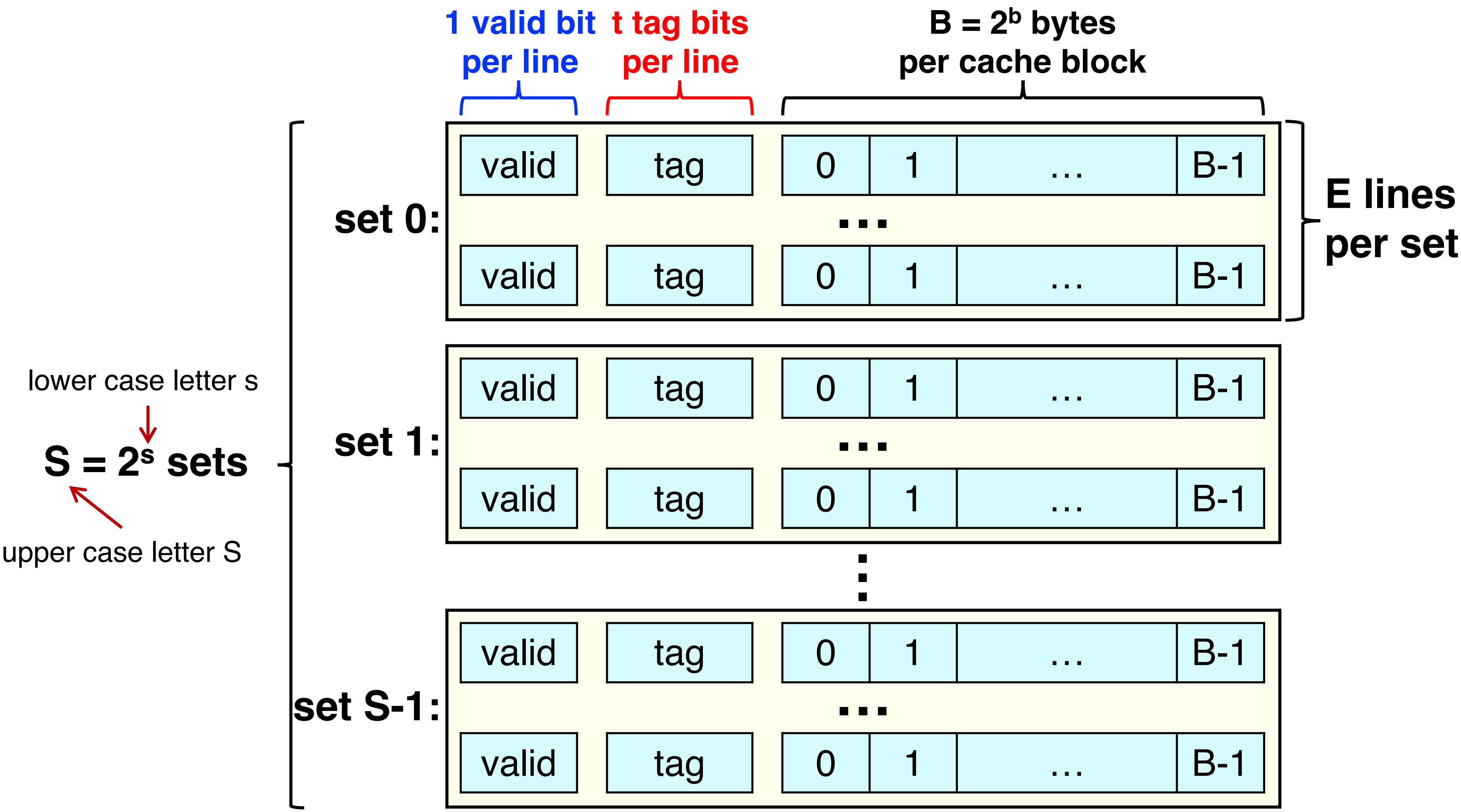
Computer Science and Engineering

Sogang University

Making Memory Accesses Fast!

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache structure
- **Cache mappings**
  - **Direct-mapped cache**
  - Set associative cache
  - Fully associative cache
- Cache performance metrics
- Cache-friendly code

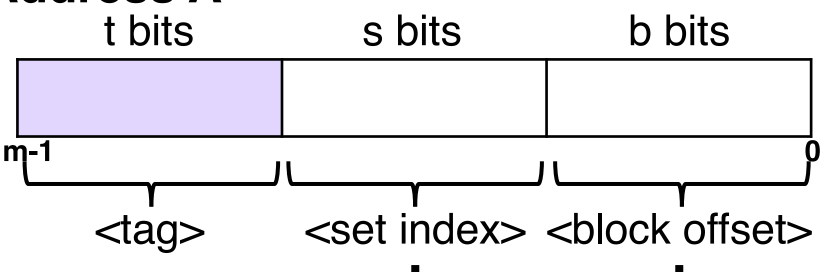
Review: Cache Structure

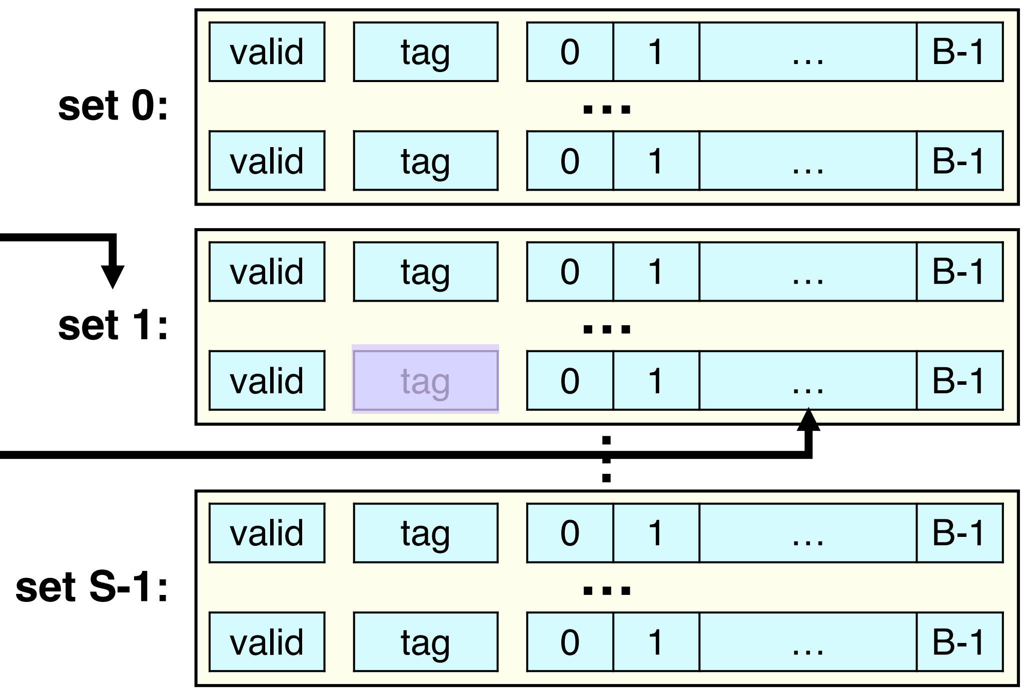


Cache size: C = B × E × S data bytes

Review: Cache Structure

Address A





M = size of addressable memory
m = memory address size (bits)
($M = 2^m$)

C = cache size (bytes) = B × E × S
B = cache block size = 2^b
E = # lines per set
S = # sets in the cache = 2^s

s = # bits for set offset
t = # tag bits
b = # bits for block offset

m = t + s + b

Set Bits

example	m	C	B	E	S	t	s	b
1	32	1024	4	1	256	22	8	2

```
for ( i = 0; i < 256; i++ )
{
      sum += my_array[ i ];
}

max = my_array[ 0 ]
for ( i = 1; i < 256; i++ )
{
      if ( my_array[ i ] > max )
      {
            max = my_array[ i ];
      }
}
```

Assume that

sizeof(element)	= 4 bytes
@my_array[256]	= AAAA0400
sum and max are in registers	

Access my_array[0]

AAAA0400	= 1010 1010 1010 1010 0000 0100 0000 0000
tag	= 1010 1010 1010 1010 0000 01
set	= 00 0000 00

Next access

AAAA0404	= 1010 1010 1010 1010 0000 0100 0000 0100
tag	= 1010 1010 1010 1010 0000 01
set	= 00 0000 01

⋮

If Set Bits were MSB (Most Significant Bit)

example	m	C	B	E	S	t	s	b
1	32	1024	4	1	256	22	8	2

```
for ( i = 0; i < 256; i++ )
{
      sum += my_array[ i ];
}

max = my_array[ 0 ]
for ( i = 1; i < 256; i++ )
{
      if ( my_array[ i ] > max )
      {
            max = my_array[ i ];
      }
}
```

Access my_array[0]

AAAA0400	= 1010 1010 1010 1010 0000 0100 0000 0000
tag	= 1010 1010 0000 0100 0000 00
set	= 1010 1010

Next access

AAAA0404	= 1010 1010 1010 1010 0000 0100 0000 0100
tag	= 1010 1010 0000 0100 0000 01
set	= 1010 1010

⋮

Direct-Mapped Cache

example	m	C	B	E	S	t	s	b
2	32	16384	32	1				

```
for ( i = 0; i < 4096; i++ )
{
    sum += my_array[ i ];
}
```

Assume that

sizeof(element)	= 8 bytes
@my_array[4096]	= AAAA0000
sum is in a register	

Direct-Mapped Cache

example	m	C	B	E	S	t	s	b
2	32	16384	32	1	512	18	9	5

```
for ( i = 0; i < 4096; i++ )
{
    sum += my_array[ i ];
}
```

Assume that

sizeof(element)	= 8 bytes
@my_array[4096]	= AAAA0000
sum is in a register	

AAAA0000	= 1010 1010 1010 1010 0000 0000 0000 0000
tag	=
set	=
offset	=

Direct-Mapped Cache

example	m	C	B	E	S	t	s	b
2	32	16384	32	1	512	18	9	5

```
for ( i = 0; i < 4096; i++ )
{
    sum += my_array[ i ];
}
```

Assume that

sizeof(element)	= 8 bytes
@my_array[4096]	= AAAA0000
sum is in a register	

AAAA0000	= 1010 1010 1010 1010 0000 0000 0000 0000
tag	= 1010 1010 1010 1010 00
set	= 00 0000 000
offset	= 0 0000

(cold) miss

AAAA0008	= 1010 1010 1010 1010 0000 0000 0000 1000
tag	=
set	=
offset	=

Direct-Mapped Cache

example	m	C	B	E	S	t	s	b
2	32	16384	32	1	512	18	9	5

```
for ( i = 0; i < 4096; i++ )
{
      sum += my_array[ i ];
}
```

Assume that

sizeof(element)	= 8 bytes
@my_array[4096]	= AAAA0000
sum is in a register	

AAAA0000	= 1010 1010 1010 1010 0000 0000 0000 0000	(cold) miss
tag	= 1010 1010 1010 1010 00	
set	= 00 0000 000	
offset	= 0 0000	

AAAA0008	= 1010 1010 1010 1010 0000 0000 0000 1000	hit
tag	= 1010 1010 1010 1010 00	
set	= 00 0000 000	
offset	= 0 1000	

Direct-Mapped Cache

example	m	C	B	E	S	t	s	b
2	32	16384	32	1	512	18	9	5

```
for ( i = 0; i < 4096; i++ )
{
    sum += my_array[ i ];
}
```

Assume that

sizeof(element)	= 8 bytes
@my_array[4096]	= AAAA0000
sum is in a register	

AAAA0008	= 1010 1010 1010 1010 0000 0000 0000 1000	hit
tag	= 1010 1010 1010 1010 00	
set	= 00 0000 000	
offset	= 0 1000	

AAAA0010	= 1010 1010 1010 1010 0000 0000 0001 0000
tag	=
set	=
offset	=

Direct-Mapped Cache

example	m	C	B	E	S	t	s	b
2	32	16384	32	1	512	18	9	5

```
for ( i = 0; i < 4096; i++ )
{
    sum += my_array[ i ];
}
```

Assume that

sizeof(element)	= 8 bytes
@my_array[4096]	= AAAA0000
sum is in a register	

AAAA0008	= 1010 1010 1010 1010 0000 0000 0000 1000	hit
tag	= 1010 1010 1010 1010 00	
set	= 00 0000 000	
offset	= 0 1000	

AAAA0010	= 1010 1010 1010 1010 0000 0000 0001 0000	hit
tag	= 1010 1010 1010 1010 00	
set	= 00 0000 000	
offset	= 1 0000	

Direct-Mapped Cache

example	m	C	B	E	S	t	s	b
2	32	16384	32	1	512	18	9	5

```
for ( i = 0; i < 4096; i++ )
{
      sum += my_array[ i ];
}
```

Assume that

sizeof(element)	= 8 bytes
@my_array[4096]	= AAAA0000
sum is in a register	

AAAA0010	= 1010 1010 1010 1010 0000 0000 0001 0000	hit
tag	= 1010 1010 1010 1010 00	
set	= 00 0000 000	
offset	= 1 0000	

AAAA0018	= 1010 1010 1010 1010 0000 0000 0001 1000
tag	=
set	=
offset	=

Direct-Mapped Cache

example	m	C	B	E	S	t	s	b
2	32	16384	32	1	512	18	9	5

```
for ( i = 0; i < 4096; i++ )
{
    sum += my_array[ i ];
}
```

Assume that

sizeof(element)	= 8 bytes
@my_array[4096]	= AAAA0000
sum is in a register	

AAAA0010	= 1010 1010 1010 1010 0000 0000 0001 0000	hit
tag	= 1010 1010 1010 1010 00	
set	= 00 0000 000	
offset	= 1 0000	

AAAA0018	= 1010 1010 1010 1010 0000 0000 0001 1000	hit
tag	= 1010 1010 1010 1010 00	
set	= 00 0000 000	
offset	= 1 1000	

Direct-Mapped Cache

example	m	C	B	E	S	t	s	b
2	32	16384	32	1	512	18	9	5

```
for ( i = 0; i < 4096; i++ )
{
      sum += my_array[ i ];
}
```

Assume that

sizeof(element)	= 8 bytes
@my_array[4096]	= AAAA0000
sum is in a register	

AAAA0018	= 1010 1010 1010 1010 0000 0000 0001 1000
tag	= 1010 1010 1010 1010 00
set	= 00 0000 000
offset	= 1 1000

hit

AAAA0020	= 1010 1010 1010 1010 0000 0000 0010 0000
tag	=
set	=
offset	=

Direct-Mapped Cache

example	m	C	B	E	S	t	s	b
2	32	16384	32	1	512	18	9	5

```
for ( i = 0; i < 4096; i++ )
{
      sum += my_array[ i ];
}
```

Assume that

sizeof(element)	= 8 bytes
@my_array[4096]	= AAAA0000
sum is in a register	

AAAA0018	= 1010 1010 1010 1010 0000 0000 0001 1000	hit
tag	= 1010 1010 1010 1010 00	
set	= 00 0000 000	
offset	= 1 1000	

AAAA0020	= 1010 1010 1010 1010 0000 0000 0010 0000	(cold) miss
tag	= 1010 1010 1010 1010 00	
set	= 00 0000 001	
offset	= 0 0000	

Class Activity – 9.7

Assume that

sizeof(element)	= 1 bytes
@my_vals1[4096]	= 0xAAAA0000
@my_vals2[4096]	= 0xAAAA1000
@results[4096]	= 0xAAAA2000

```
for ( i = 0; i < 4096; i++ )
{
        results[ i ] += my_vals1[ i ] – my_vals2[ i ];
}
```

Assume that declarations and code may not be modified

Specify a DM (Direct-Mapped) cache that will achieve a hit rate of ≥ 50%

(m, C, B, E, S, t, s, b = ?)

Class Activity – 9.8

Assume that

sizeof(element)	= 1 bytes
@my_vals1[4096]	= 0xAAAA0000
@my_vals2[4096]	= 0xAAAA1000
@results[4096]	= 0xAAAA2000

```
for ( i = 0; i < 4096; i++ )
{
        results[ i ] += my_vals1[ i ] – my_vals2[ i ];
}
```

Assume that declarations and code may not be modified

Specify a DM (Direct-Mapped) cache that will achieve a hit rate of $\geq$ 50%

(m, C, B, E, S, t, s, b = ?)

What is the smallest DM cache that could achieve a hit rate of $\geq$ 50%

Class Activity – 9.9

Assume that

sizeof(element)	= 1 bytes
@my_vals1[4096]	= 0xAAAA0000
@my_vals2[4096]	= 0xAAAA1000
@results[4096]	= 0xAAAA2000

```
for ( i = 0; i < 4096; i+=2 )
{
    results[ i ] += my_vals1[ i ] – my_vals2[ i ];
}
```

Specify a DM (Direct-Mapped) cache that will achieve a hit rate of ≥ 50%

(m, C, B, E, S, t, s, b = ?)

What is the smallest DM cache that could achieve a hit rate of ≥ 50%

Class Activity – 9.10

Assume that

sizeof(element)	= 4 bytes
@my_vals1[4096]	= 0xAAAA0000
@my_vals2[4096]	= 0xAAAA4000
@results[4096]	= 0xAAAA8000

```
for ( i = 0; i < 4096; i++ )
{
        results[ i ] += my_vals1[ i ] – my_vals2[ i ];
}
for ( i = 0; i < 4096; i++ )
{
        results[ i ]++;
}
```

Specify a DM (Direct-Mapped) cache that will achieve a hit rate of ≥ 50%

(m, C, B, E, S, t, s, b = ?)

What is the smallest DM cache that could achieve a hit rate of 100% for the second loop?

Thrashing

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

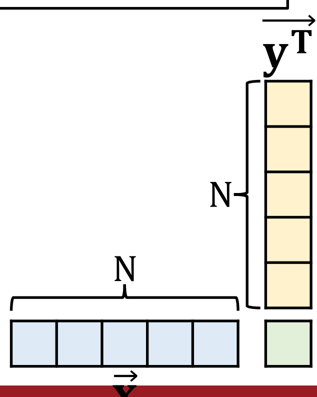
```
float dotproduct (float x[256], float y[256])
{
        float sum = 0.0;
        int i;

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

Assume that

sizeof(float)	= 4 bytes
@x[256]	= AAAA0000
@y[256]	= AAAA0400
sum and i are in registers	

$$\vec{\mathbf{x}} \cdot \vec{\mathbf{y}} = \sum_{i=0}^{\mathrm{N}-1} x_i y_i$$



Access x[0] @ AAAA0000

AAAA0000	= 1010 1010 1010 1010 0000 0000 0000 0000
set bits	= 00 000

Access y[0] @ AAAA0400

AAAA0000	= 1010 1010 1010 1010 0000 0100 0000 0000
set bits	= 00 000

Making Memory Accesses Fast!

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache structure
- **Cache mappings**
  - Direct-mapped cache
  - **Set associative cache**
  - Fully associative cache
- Cache performance metrics
- Cache-friendly code

Set Associative Cache

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

```
float dotproduct (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;
}
```

Assume that

sizeof(float)	= 4 bytes
@x[256]	= AAAA0000
@y[256]	= AAAA0400
sum and i are in registers	

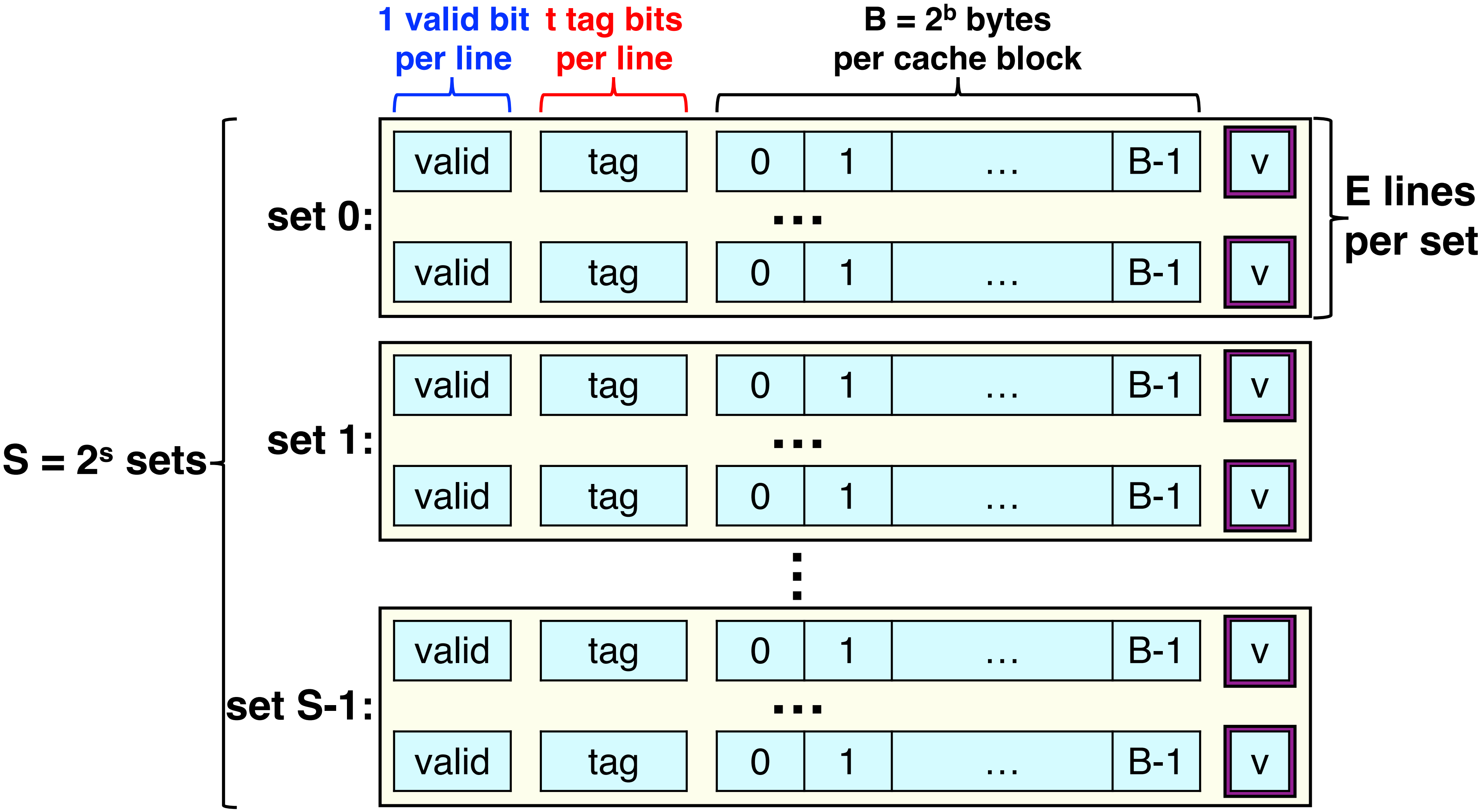
Access x[0] @ AAAA0000 **fills 1^{st} line in set 0**

AAAA0000	= 1010 1010 1010 1010 0000 0000 0000 0000
set bits	= 00 000

Access y[0] @ AAAA0400 **fills 2^{nd} line in set 0**

AAAA0000	= 1010 1010 1010 1010 0000 0100 0000 0000
set bits	= 00 000

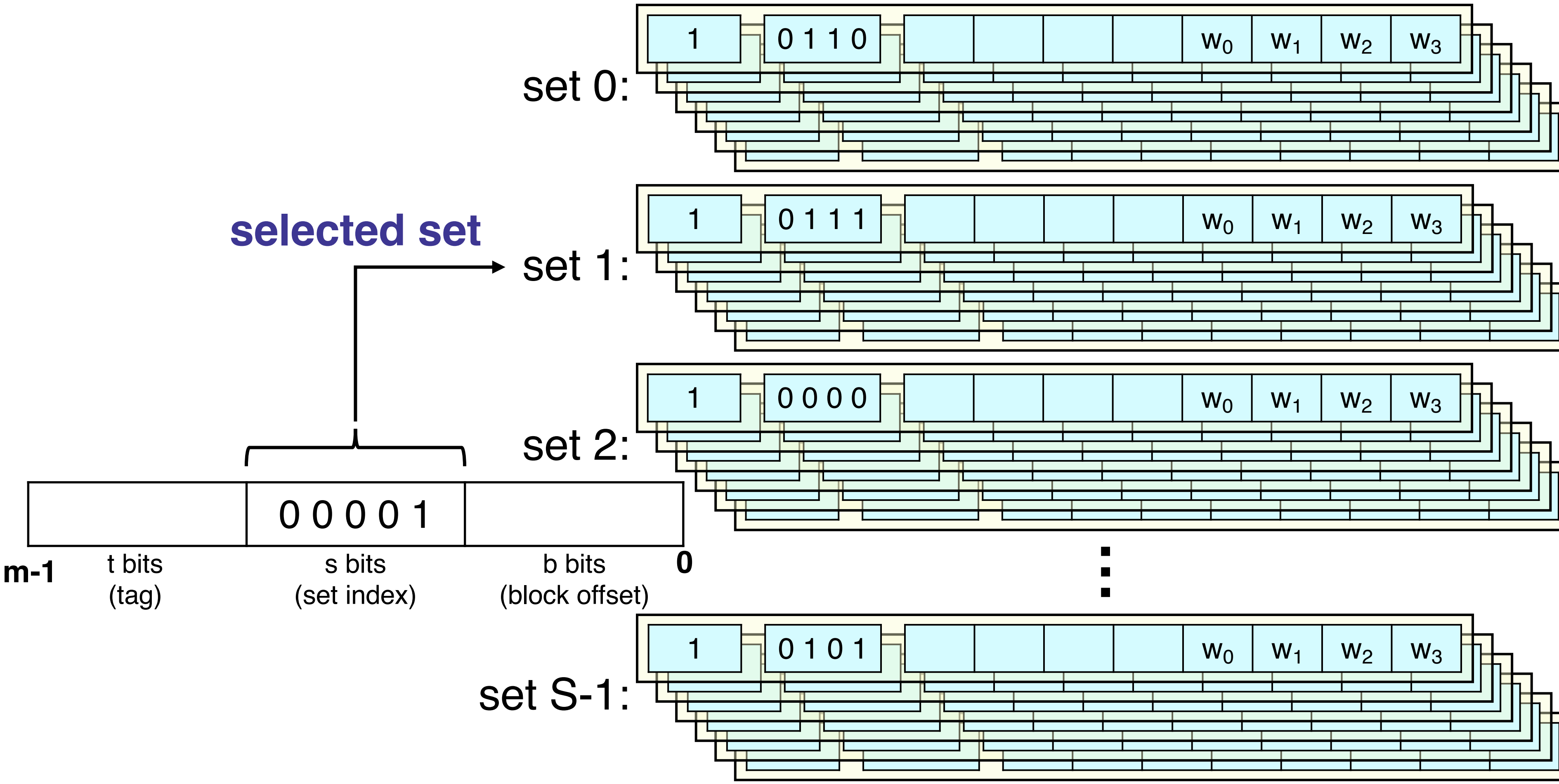
Set Associative Cache Structure



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

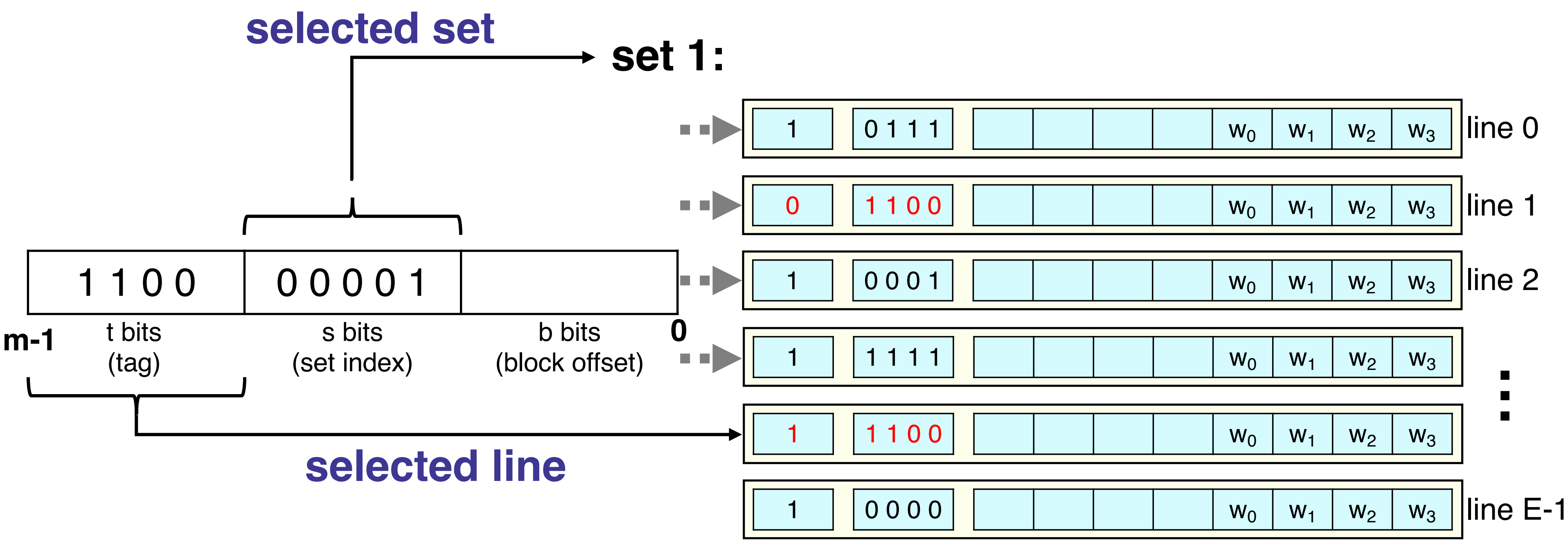
Set Associative Cache

- Set selection
  - Use the set index bits to determine the set of interest



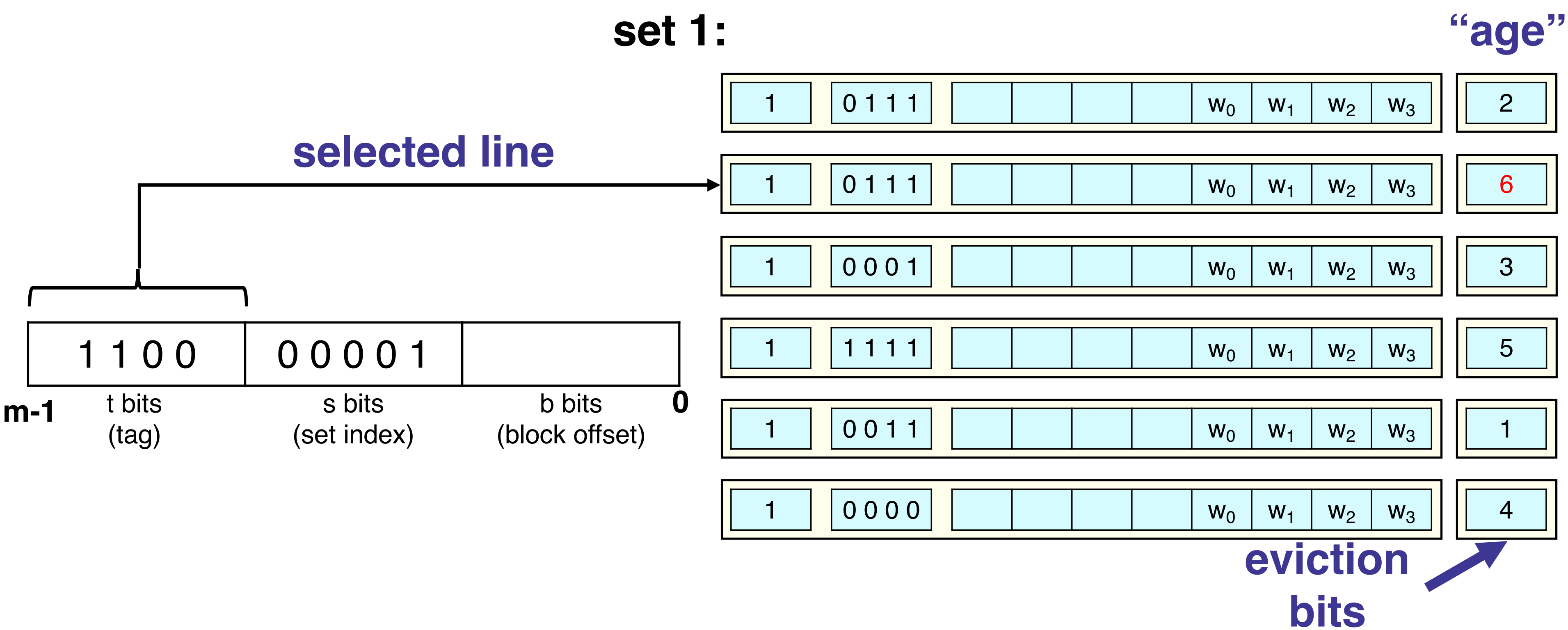
Set Associative Cache

- Line selection on read hit
  - Scan each cahce 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 a miss
  - Retrieve from slower memory
  - Scan for empty line
    - If full, pick a loser



LRU / LFU

LRU (Least Recently Used)

Sequential access

										age
1	0 1 1 1					w_0	w_1	w_2	w_3	006
1	0 1 0 1					w_0	w_1	w_2	w_3	005
1	0 0 0 1					w_0	w_1	w_2	w_3	004
1	1 1 1 1					w_0	w_1	w_2	w_3	003
1	0 0 1 1					w_0	w_1	w_2	w_3	002
1	0 0 0 0					w_0	w_1	w_2	w_3	001

LFU (Least Frequently Used)

Equivalent access

										access count
1	0 1 1 1					w_0	w_1	w_2	w_3	002
1	0 1 0 1					w_0	w_1	w_2	w_3	002
1	0 0 0 1					w_0	w_1	w_2	w_3	002
1	1 1 1 1					w_0	w_1	w_2	w_3	003
1	0 0 1 1					w_0	w_1	w_2	w_3	002
1	0 0 0 0					w_0	w_1	w_2	w_3	001

Class Activity – 9.11

problem	m	C	B	E	S	t	s	b
1	64	4096	32	4	32			

```
float dotproduct (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;
}
```

Assume that

sizeof(float)	= 4 bytes
@x[1024]	= 0…0AAAA0000
@y[1024]	= 0…0AAAA1000
sum and i are in registers	
LRU eviction	

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

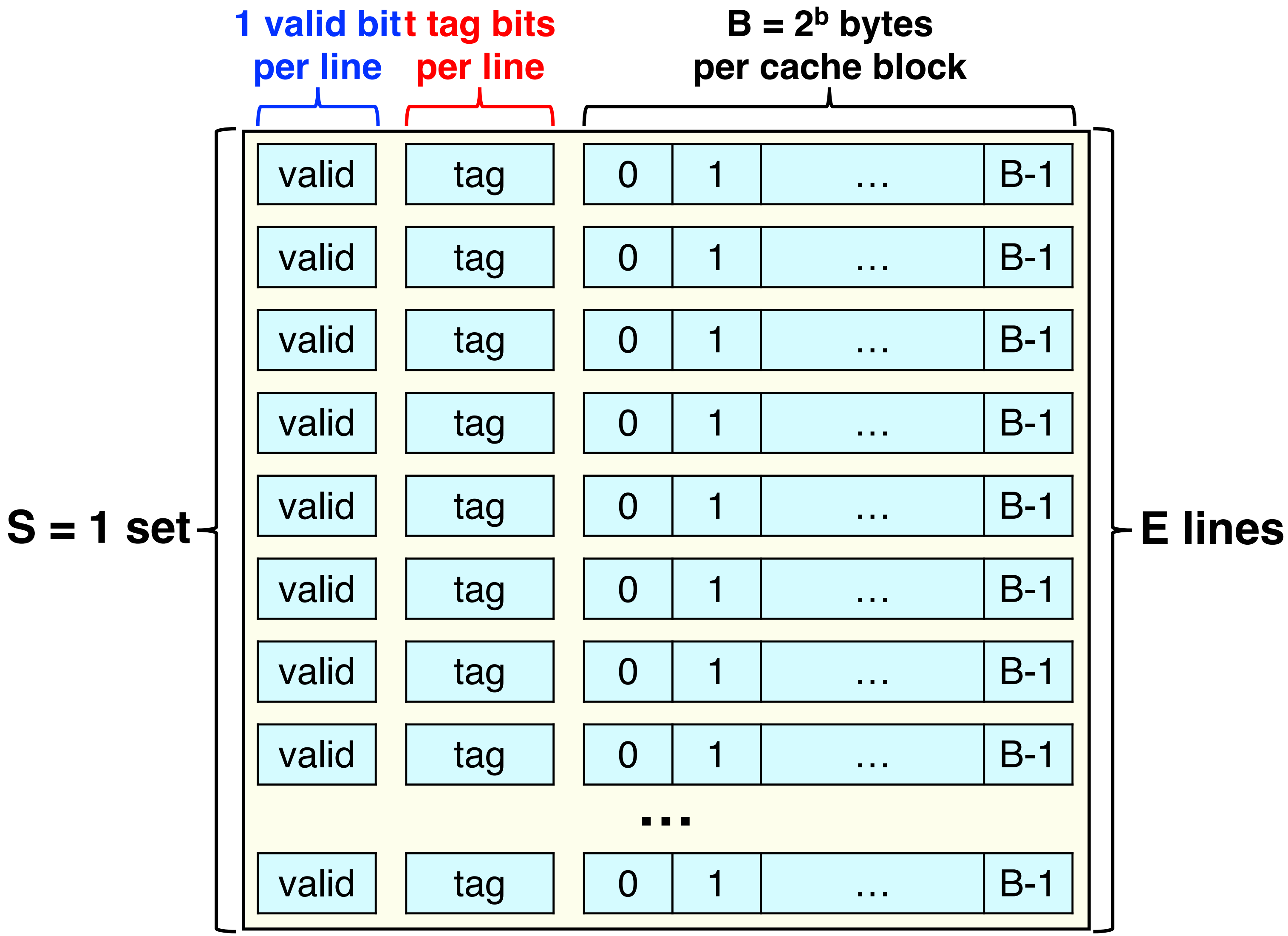
Set Associative Cache

- Drawbacks of set associative cache
  - `sum += x[i] * y[i] * z [i]`
  - Cache size / expense
  - Time to determine a hit
  - What if the cache is full?
    - Eviction policy

Making Memory Accesses Fast!

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache structure
- **Cache mappings**
  - Direct-mapped cache
  - Set associative cache
  - **Fully associative cache**
- Cache performance metrics
- Cache-friendly code

Fully Associative Cache Structure



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

Fully Associative Cache

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

```
float dotproduct (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;
}
```

Assume that

sizeof(float)	= 4 bytes
@x[256]	= AAAA0000
@y[256]	= AAAA0400
sum and i are in registers	

Access x[0] @ AAAA0000 **fills 1st line in set**

AAAA0000	= 1010 1010 1010 1010 0000 0000 0000 0000
set bits	=

Access y[0] @ AAAA0400 **fills 2nd line in set**

AAAA0000	= 1010 1010 1010 1010 0000 0100 0000 0000
set bits	=

⋮

Class Activity – 9.12

problem	m	C	B	E	S	t	s	b
1	32	2048	32	64	1			

```
float dotproduct (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;
}
```

Assume that

sizeof(float)	= 4 bytes
@x[1024]	= 0…0AAAA0000
@y[1024]	= 0…0AAAA1000
@z[1024]	= 0…0AAAA2000
sum and i are in registers	
LRU eviction	

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

Class Activity – 9.13

problem	m	C	B	E	S	t	s	b
2	32	2048	32	2				

```
float dotproduct (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;
}
```

Assume that

sizeof(float)	= 4 bytes
@x[1024]	= 0…0AAAA0000
@y[1024]	= 0…0AAAA1000
@z[1024]	= 0…0AAAA2000
sum and i are in registers	
LRU eviction	

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

Class Activity – 9.14

problem	m	C	B	E	S	t	s	b
3	32	2048	32	4				

```
float dotproduct (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;
}
```

Assume that

sizeof(float)	= 4 bytes
@x[512]	= 0…0AAAA0000
@y[512]	= 0…0AAAA1000
@z[512]	= 0…0AAAA2000
sum and i are in registers	
LRU eviction	

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

Fully Associative Cache

- Drawbacks of fully associative cache
  - More complex and expensive logic (compared to Direct-Mapped Cache)
  - 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

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