

Caching

CS351 : Saelee

von Neuman architecture

memory = **working** store

SRAM, DRAM, NVRAM, HDD

SRAM

- **S**tatic **R**andom **A**ccess **M**emory
 - One bit stored per flip-flop
 - Data stable as long as charge applied

DRAM

- **D**ynamic **R**andom **A**ccess **M**emory
 - One bit per capacitor — high density
 - Volatile and unstable; requires “refresh”

NVRAM

- **N**on-**V**olatile **R**andom **A**ccess **M**emory
 - Data maintained without power
 - e.g., compact flash, SSD
 - Relatively slow access times

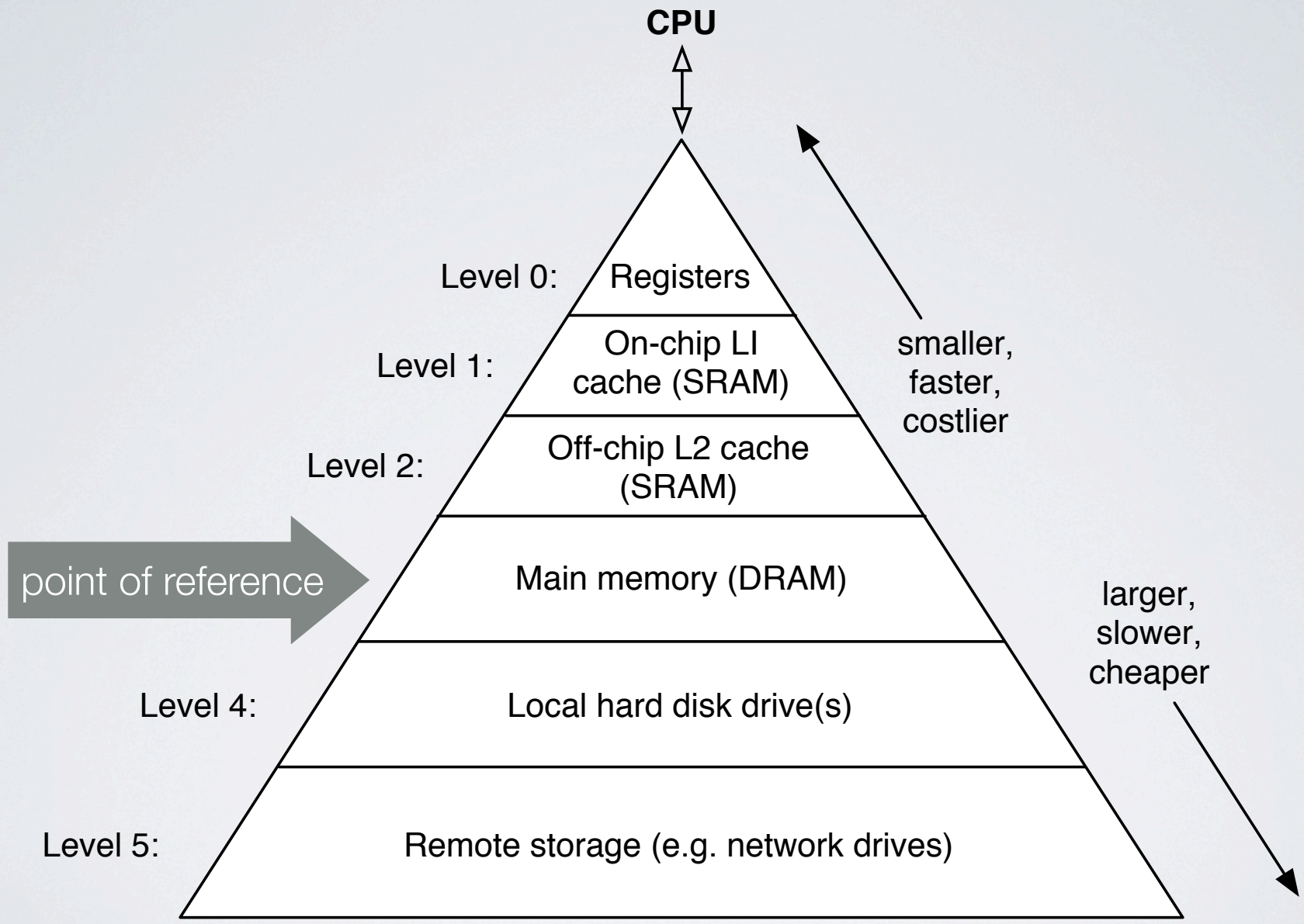
HDD

- **H**ard **D**isk **D**rive
- Spinning magnetic platters with read/write heads
- Seek (tracking) time + rotational latency
- Slow!

Relative Speeds

Type	Size	Speed
Registers	8 - 32 words	0 - 1 cycles
On-board SRAM	32 - 256 KB	1 - 3 cycles
Off-board SRAM	256 KB - 16 MB	~10 cycles
DRAM	128 MB - 16 GB	~100 cycles
HDD	1 GB - 4 TB	~10,000,000 cycles

The Memory **Hierarchy**



CPU



Level 0: Registers

Level 1: On-chip L1 cache (SRAM)

Level 2: Off-chip L2 cache (SRAM)

Main memory (DRAM)

Level 4: Local hard disk drive(s)

Level 5: Remote storage (e.g. network drives)

smaller,
faster,
costlier

point of reference

larger,
slower,
cheaper

maximize address space
minimize access latency

Cache

what to cache?

all programs exhibit
locality of reference

temporal locality

e.g.

repeated function calls
loop counter access

spatial locality

array access (stride- k)
sequential control flow

Cache size \ll Address space

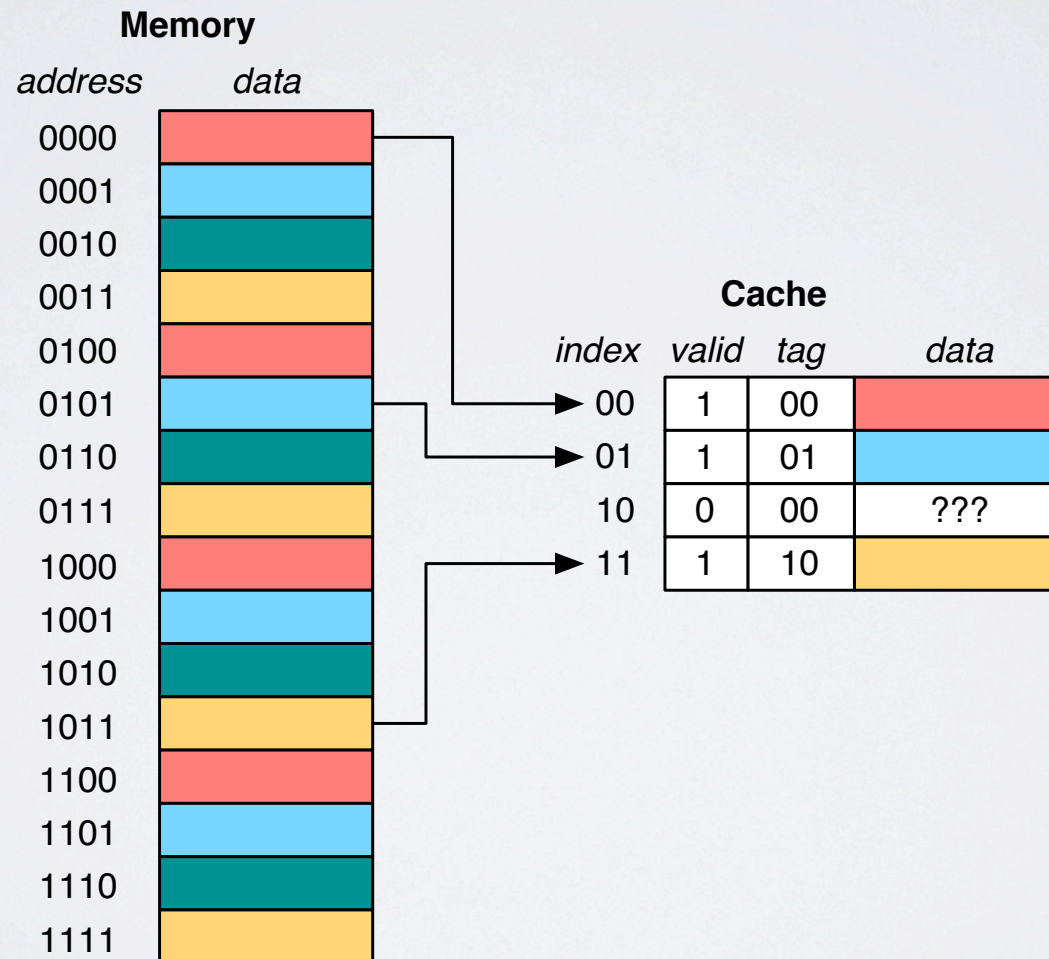
memory → cache
mapping

Direct-Mapped Cache



cache index = (address) mod (total # cache entries)

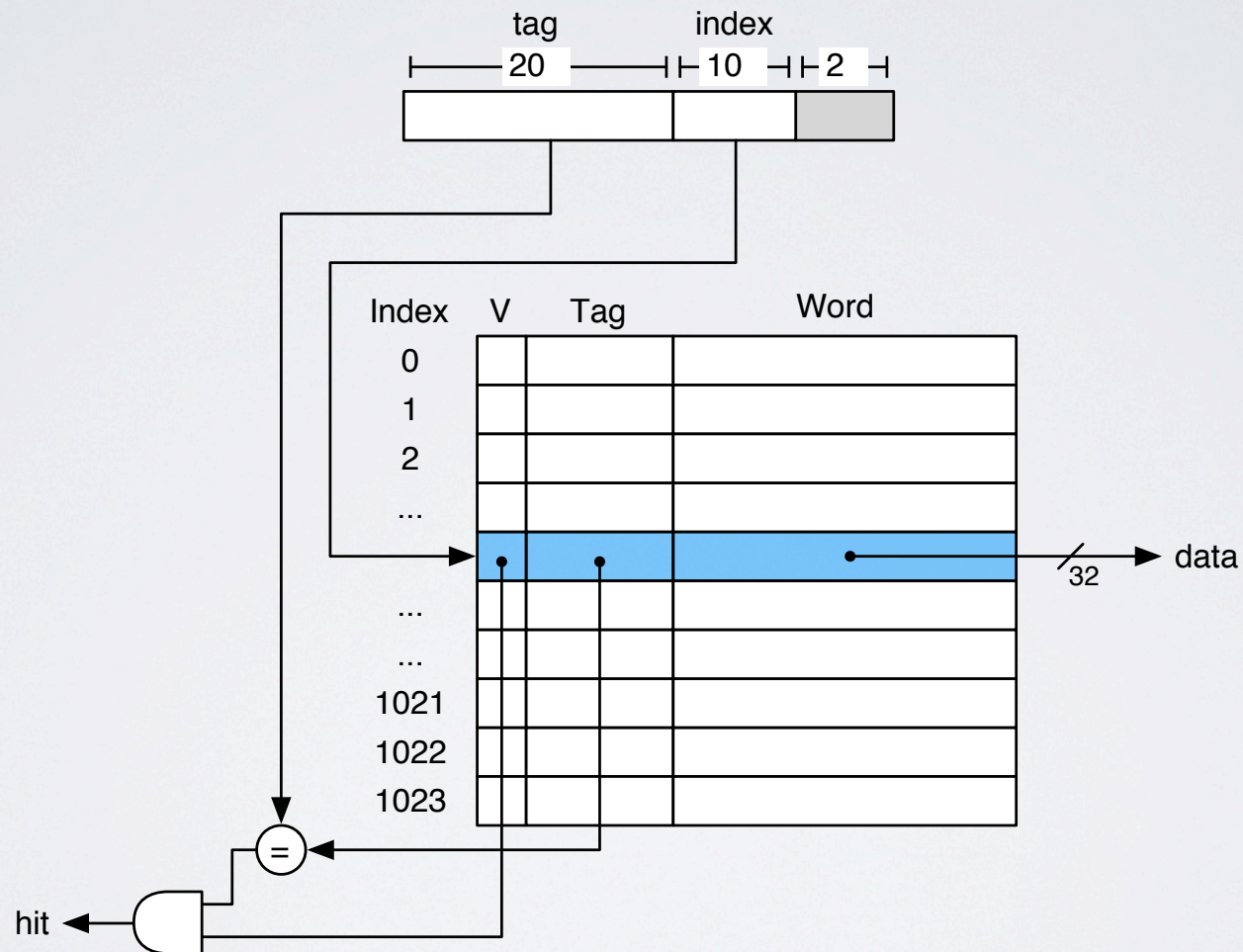
Valid & Tag Fields



cache block $>$ 1 byte

word “alignment”

Consider a cache with 2^{10} entries for 4-byte words

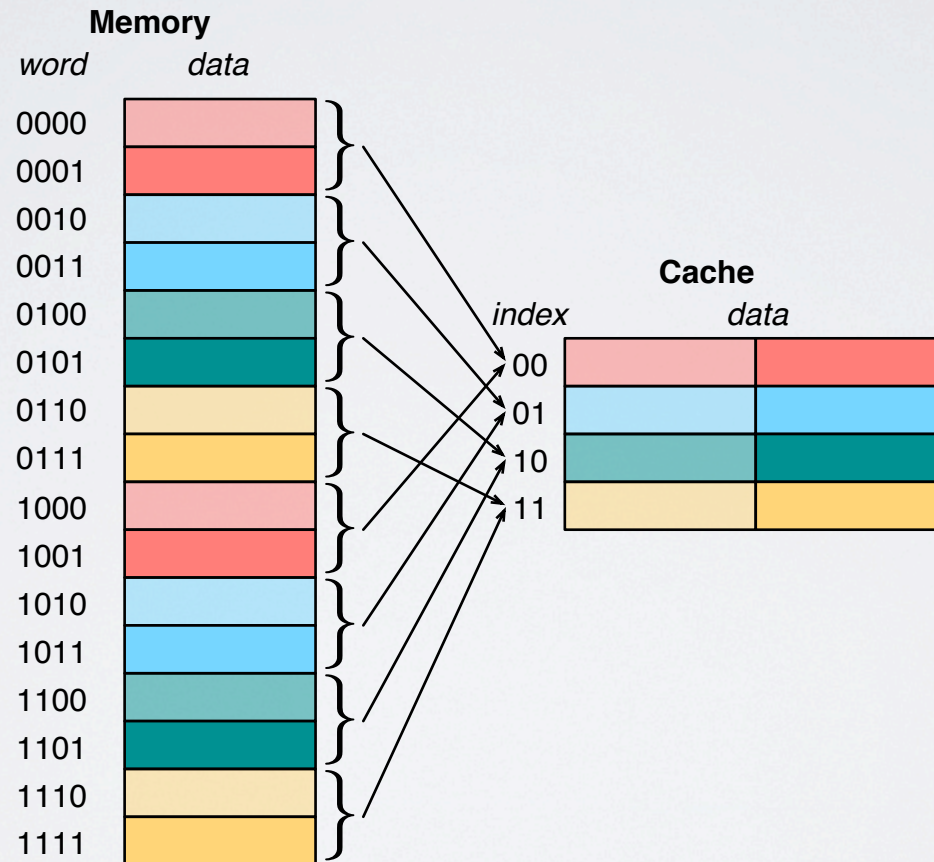


```
sum = 0;  
for (i = 0; i < n; i++)  
    sum += arr[i];  
return sum;
```

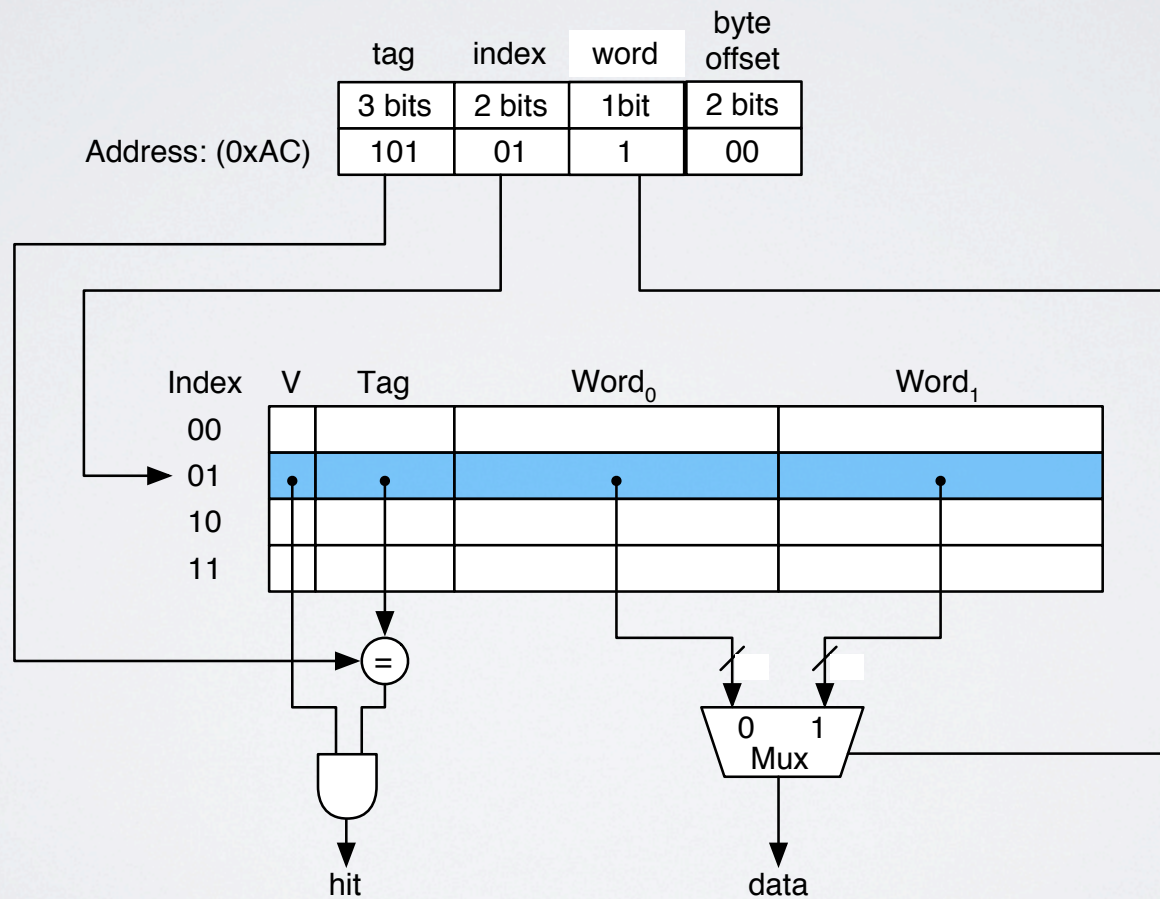
100% miss rate for **arr**!

requires “pre-caching”

Multi-word Blocks



Consider an 8-bit address and a cache with 4 blocks, with a block size of two 4-byte words



not scaleable

```
int dotprod (int *v1, int *v2, int n) {  
    int i, sum = 0;  
    for (i=0; i<n; i++)  
        sum += v1[i] * v2[i];  
    return sum;  
}
```

working set of data

collisions \Rightarrow cache misses

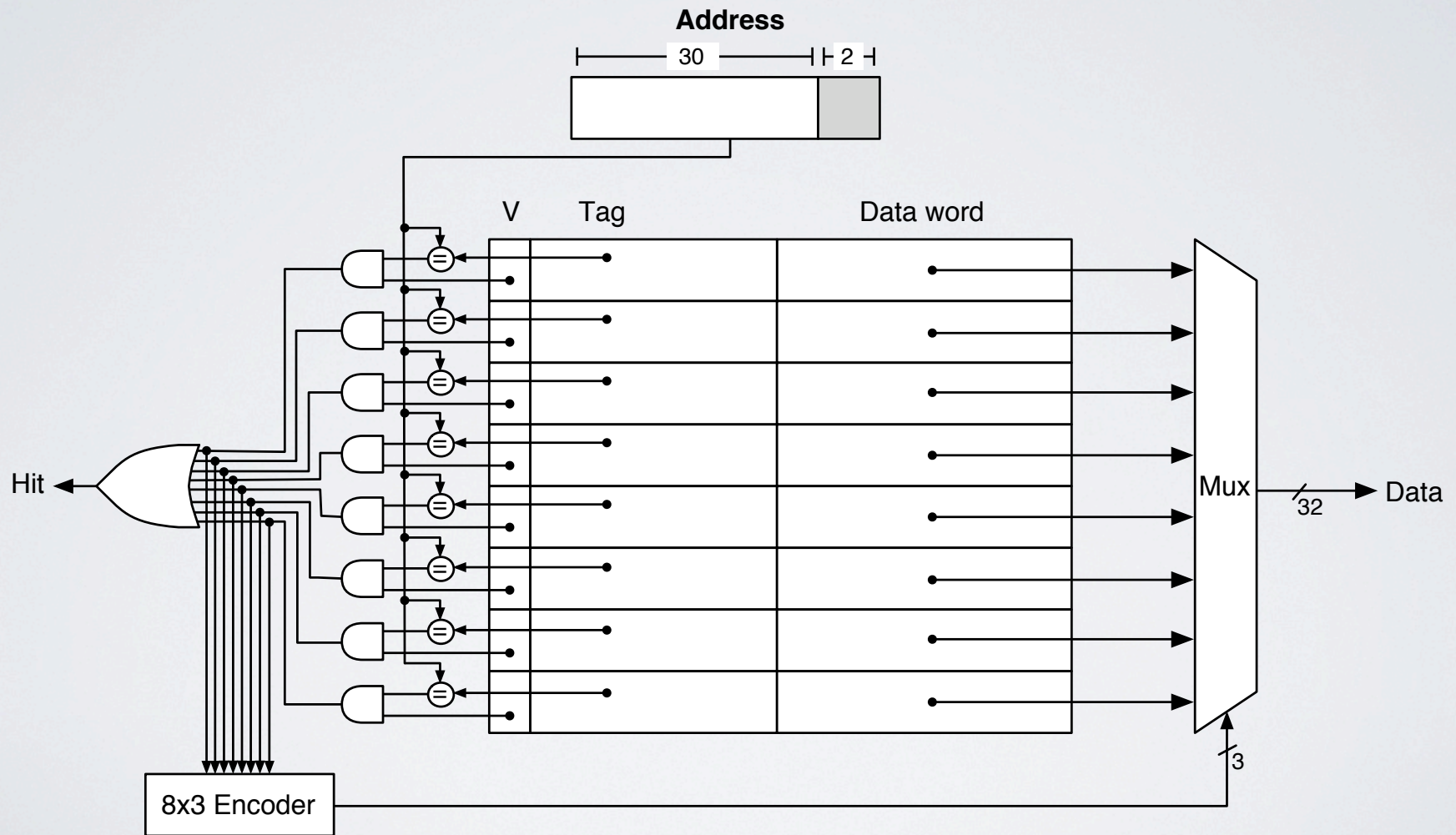
classify cache misses

compulsory / cold misses

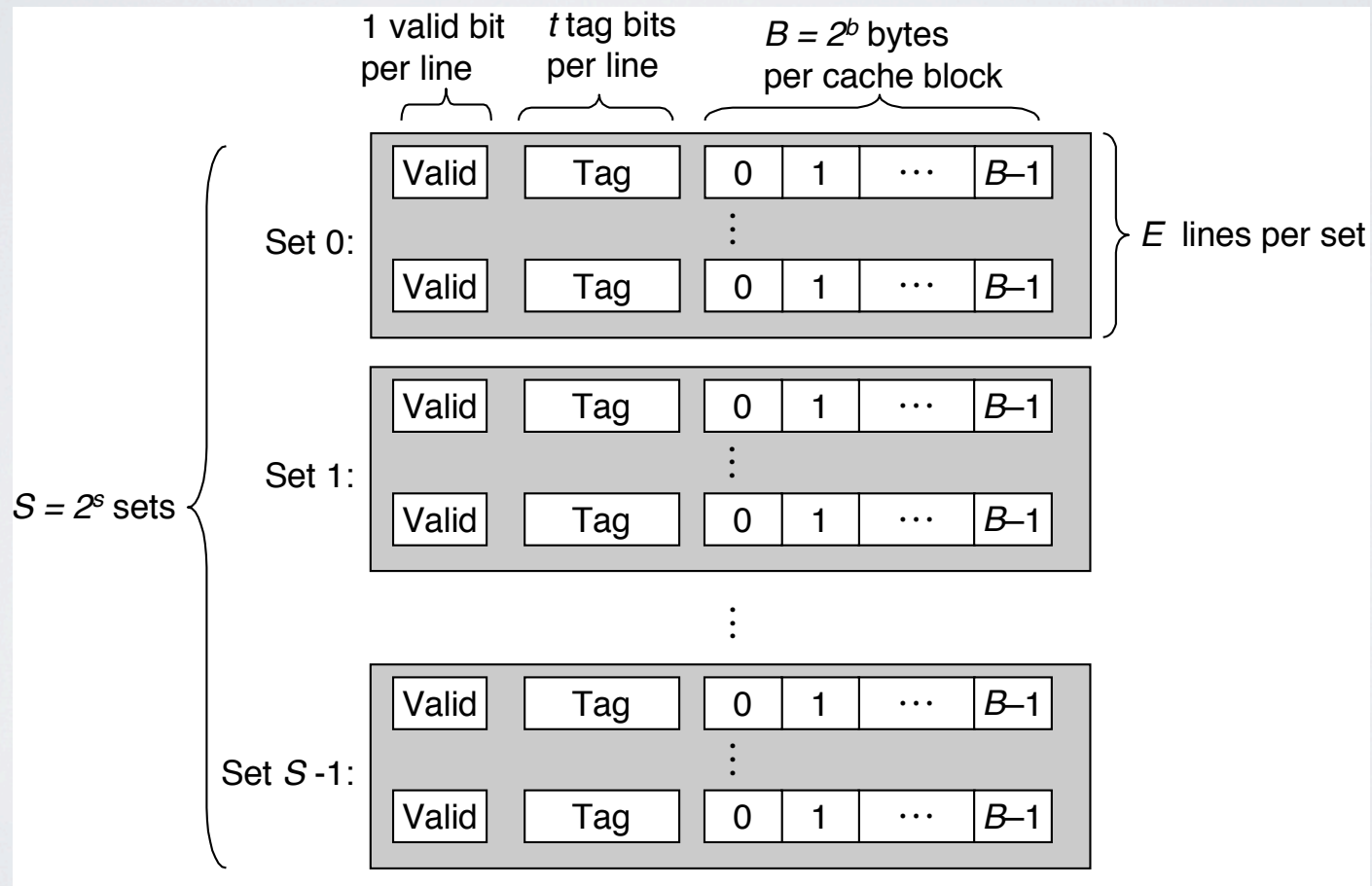
capacity misses

conflict misses

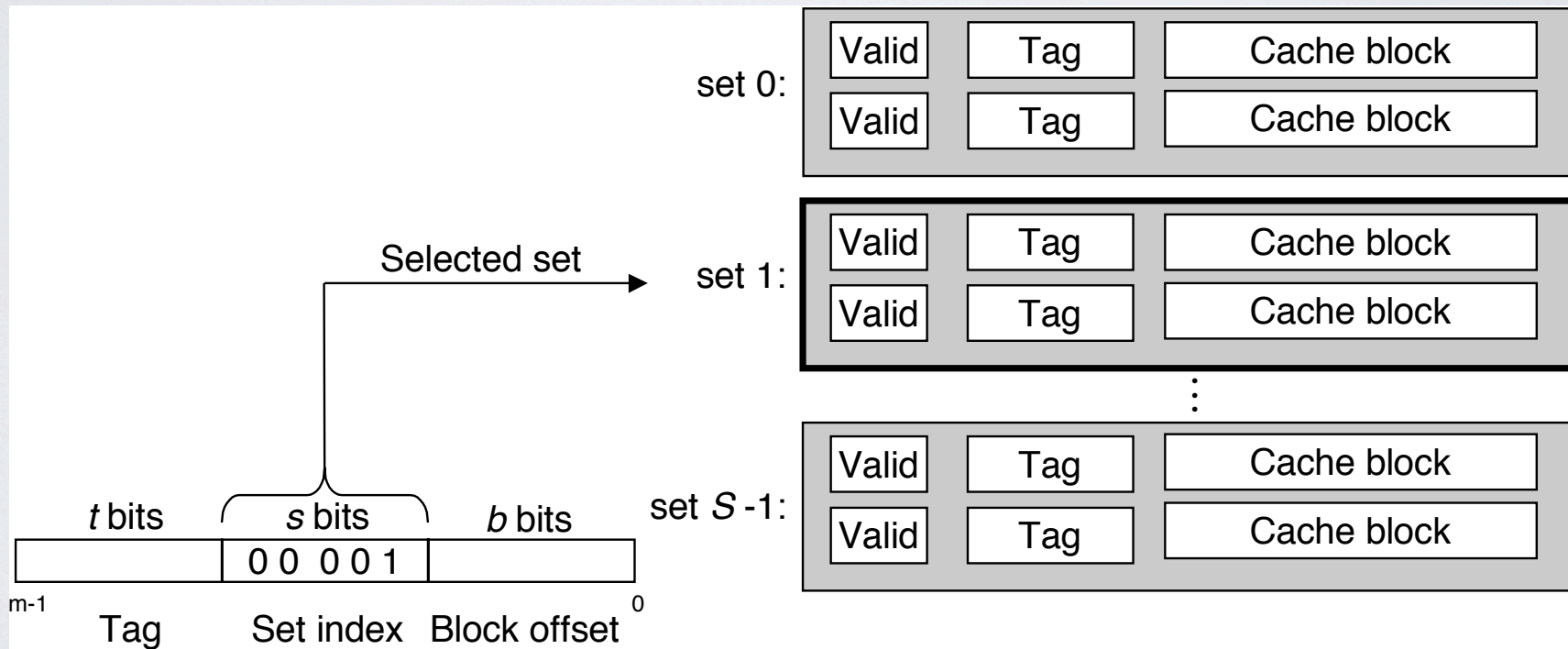
Fully-Associative Cache



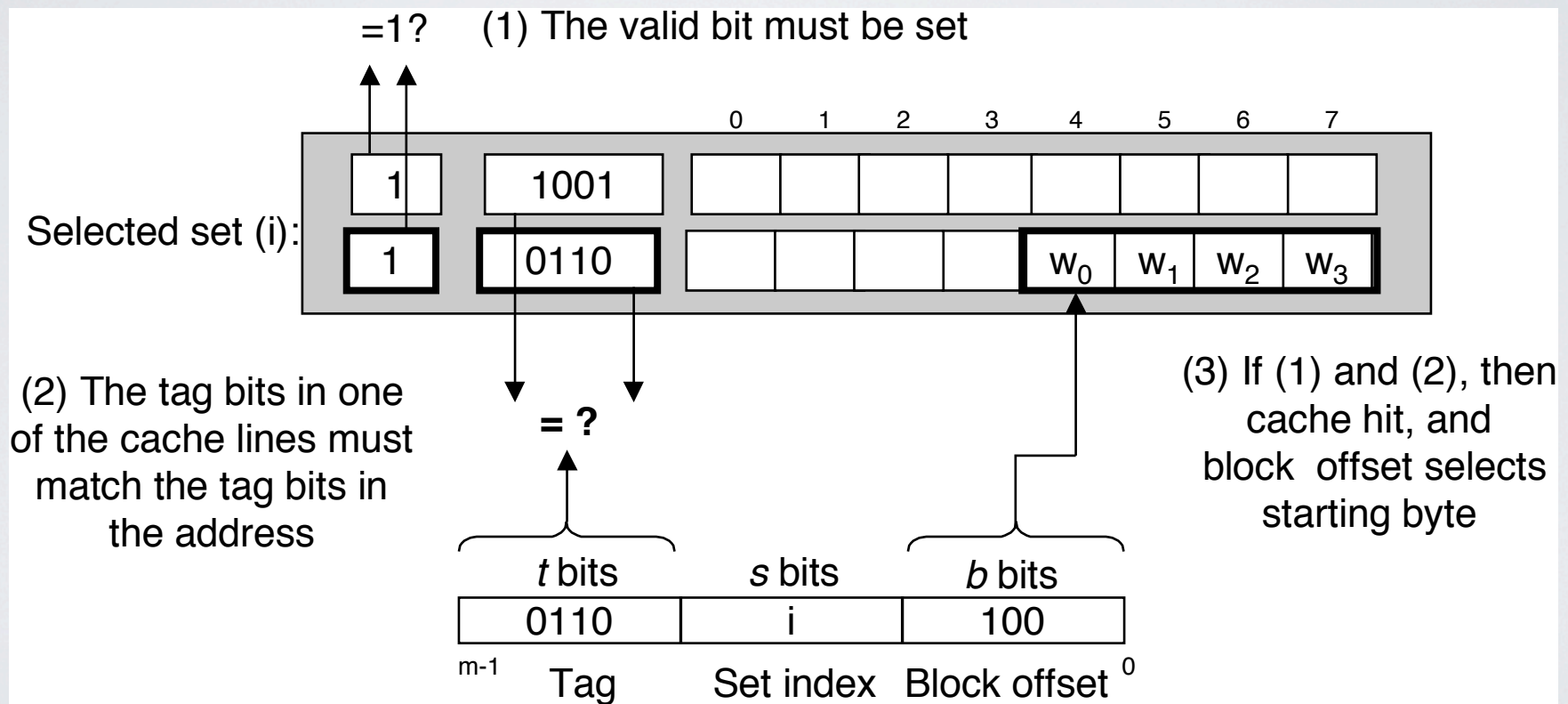
Set-Associative Cache



Set Selection



Tag matching & Block offset



previously: direct mapping

always replace
“old” mapping

old mapping ambiguous
(multiple mappings)

replacement policy?

FIFO

Least Frequently Used

Least Recently Used

updating the cache: when?

read hit \Rightarrow return cache data

read miss \Rightarrow update cache

writes more complex

write hit?

update both cache and
memory

write-through

inefficient

update cache only
update memory later

write-back

“dirty” bit

write miss?

only write to memory

write-no-allocate

temporal locality of writes?

“allocate” mapping in cache
update cache

write-allocate

write-through
+
no-write-allocate

write-back
+
write-allocate

higher level \Rightarrow write-through

lower level \Rightarrow write-back

cache metrics

hit rate (1 - miss rate)

hit time

miss penalty

minimize hit time

vs.

maximize hit rate

simple mapping

vs.

high associativity

multi-level caching

Miss Rate

- Typical numbers:
 - 3-10% for L1
 - $< 1\%$ for L2

Hit Time

- Typical numbers:
 - 1 clock cycle for L1
 - 3-8 clock cycles for L2

Miss Penalty

- Typically 25-100 cycles for main memory

Cache-Friendly Code

good temporal locality

small array stride-patterns

small working set

“blocking” data

“Memory Mountain”

```
/* The test function */
void test (int elems, int stride) {
    int i, result = 0;
    for (i = 0; i < elems; i += stride)
        result += data[i];
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run (int size, int stride, double Mhz) {
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride);           /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0); /* call test(elems, stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

```

/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10) /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23) /* ... up to 8 MB */
#define MAXSTRIDE 16 /* Strides range from 1 to 16 */
#define MAXElems MAXBYTES/sizeof(int)

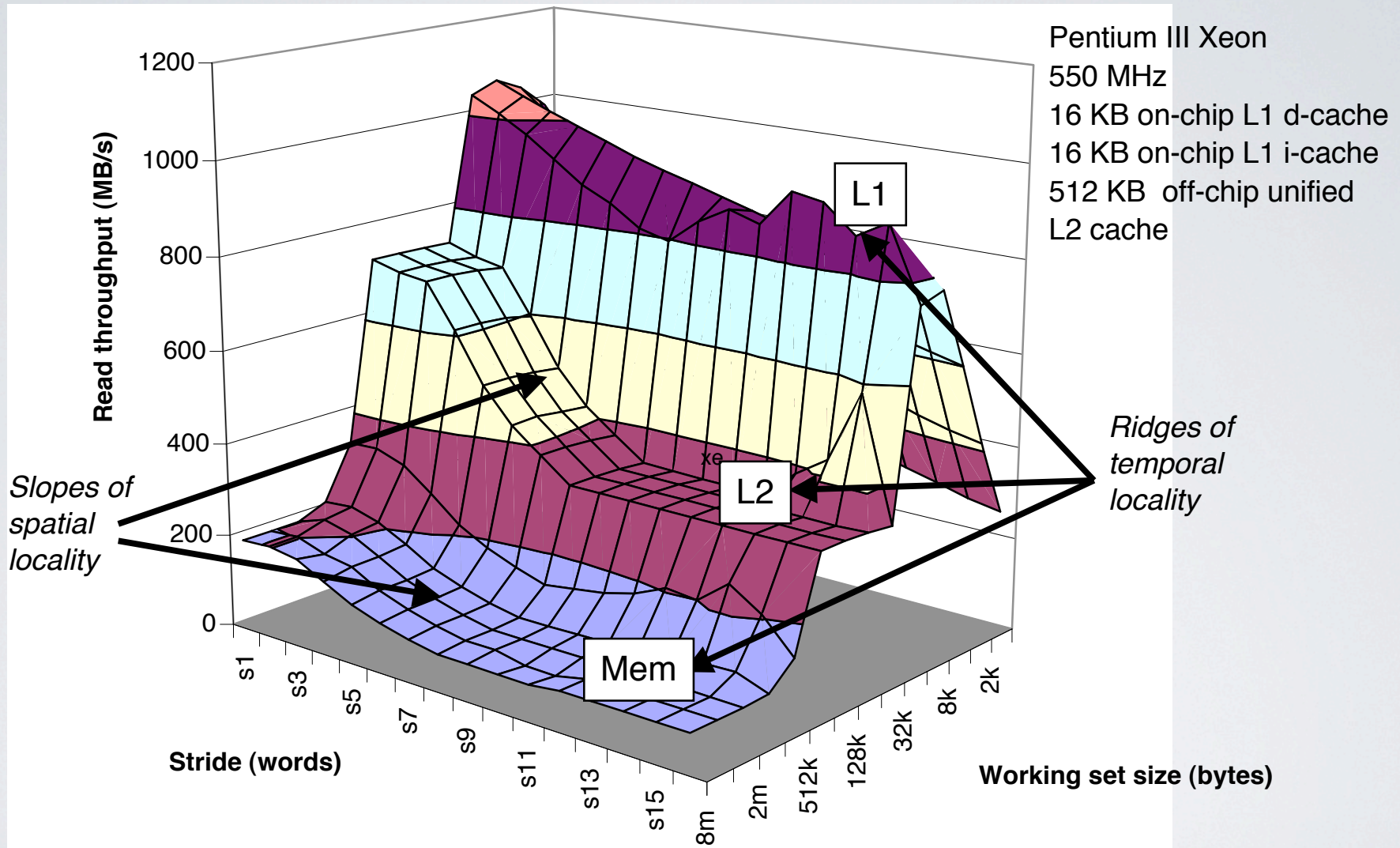
int data[MAXElems]; /* The array we'll be traversing */

int main () {
    int size; /* Working set size (in bytes) */
    int stride; /* Stride (in array elements) */
    double Mhz; /* Clock frequency */

    init_data(data, MAXElems); /* Initialize each element in data to 1 */
    Mhz = mhz(0); /* Estimate the clock frequency */

    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride, Mhz));
        printf("\n");
    }
    exit(0);
}

```



1200
1000
800
600
400
200
0

Slopes of spatial locality

Ridges of temporal locality

Stride (words)

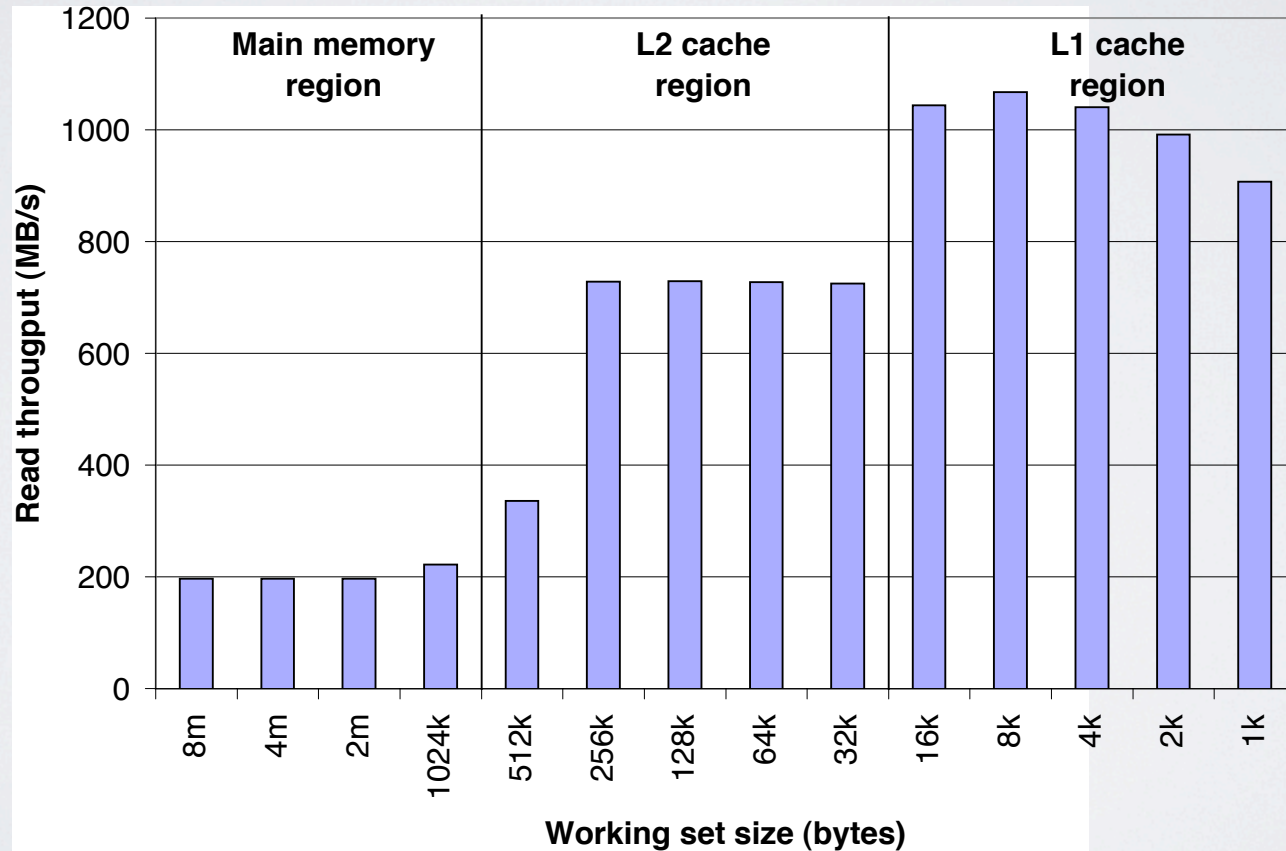
Working set size (bytes)

Mem

L2

L1

Slice through the memory mountain with stride=1



Slice through memory mountain with size=256KB

