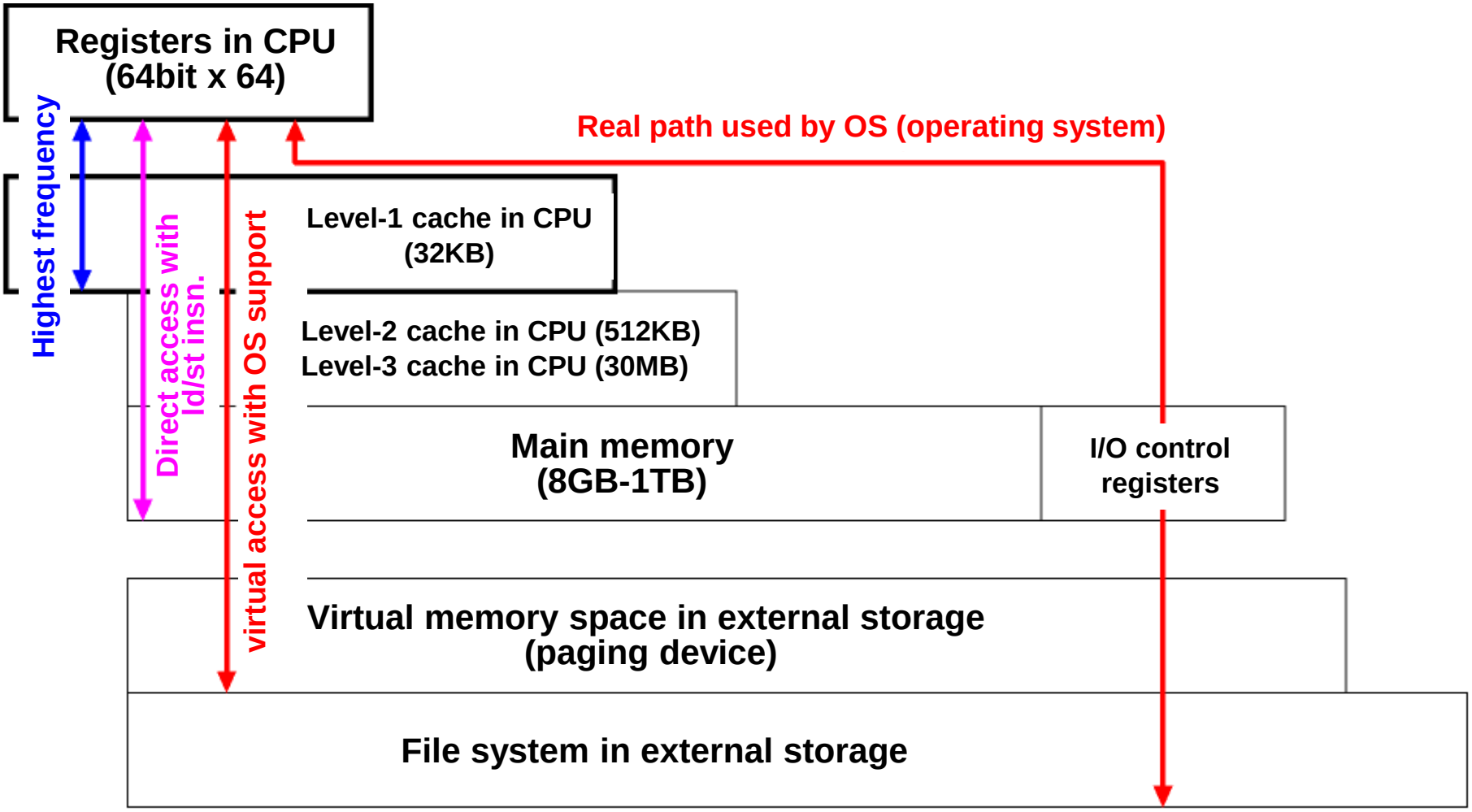
Computer System

Step4. CA0702:Cache memory and execution speed of programs

http://arch.naist.jp/Lectures/ARCH/ca0702/ca0702e.pdf

Memory Hierarchy

Too slow without cache memory



Cache is not almighty

Spatial locality:

Memory location near past accessed tend to be accessed again.

Temporal locality:

Memory location near recently accessed tend to be accessed for a while.

That means following programs run very slow speed

- **Working set > capacity of cache**
- **Sequential access with no-reuse**

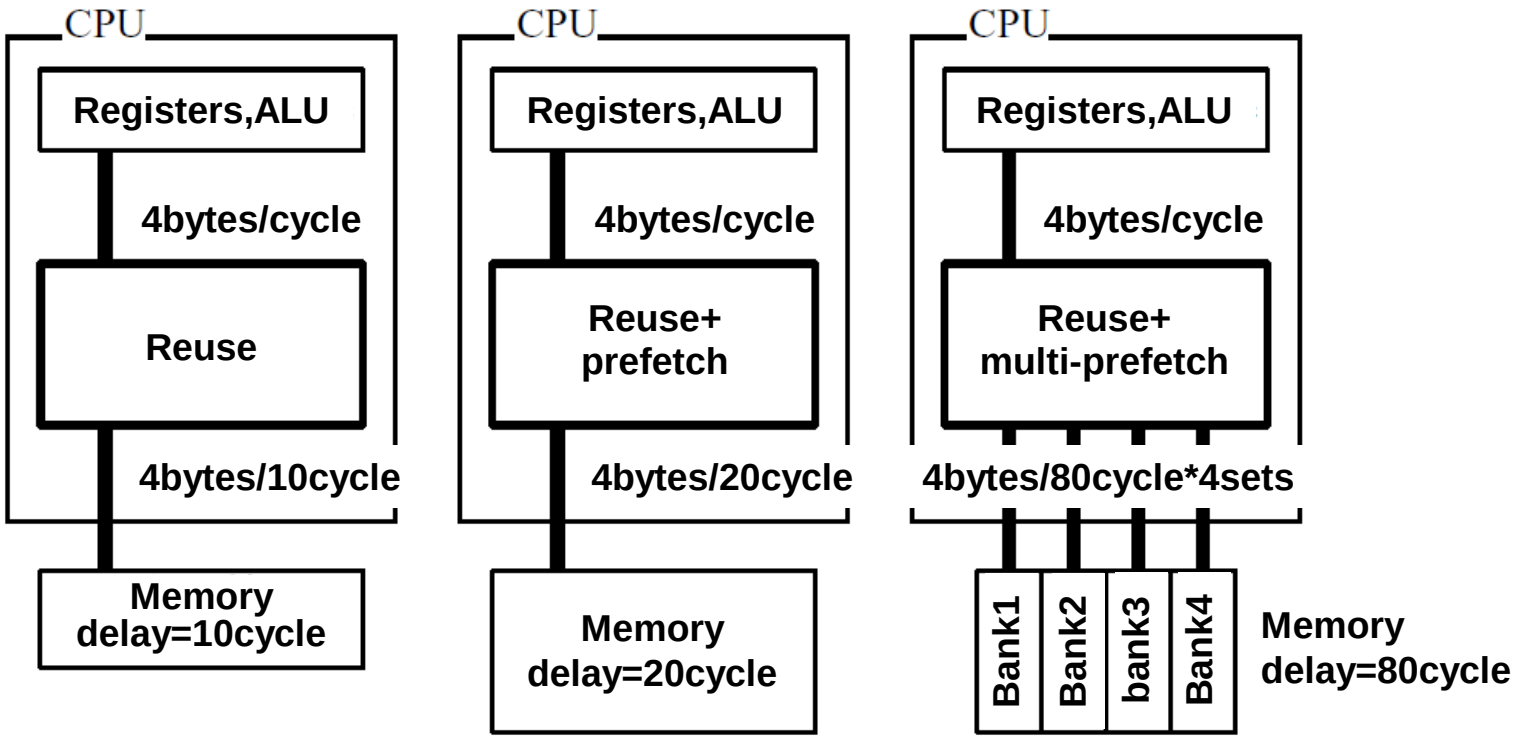
Which cache can execute your program in fastest speed?

- **Large but slow cache**
- **Small but fast cache**
- **Combination of multi-level cache**

Ans. No one can recommend.

It depends on the skill of the programmer.

Evolution of cache to cover slow memory

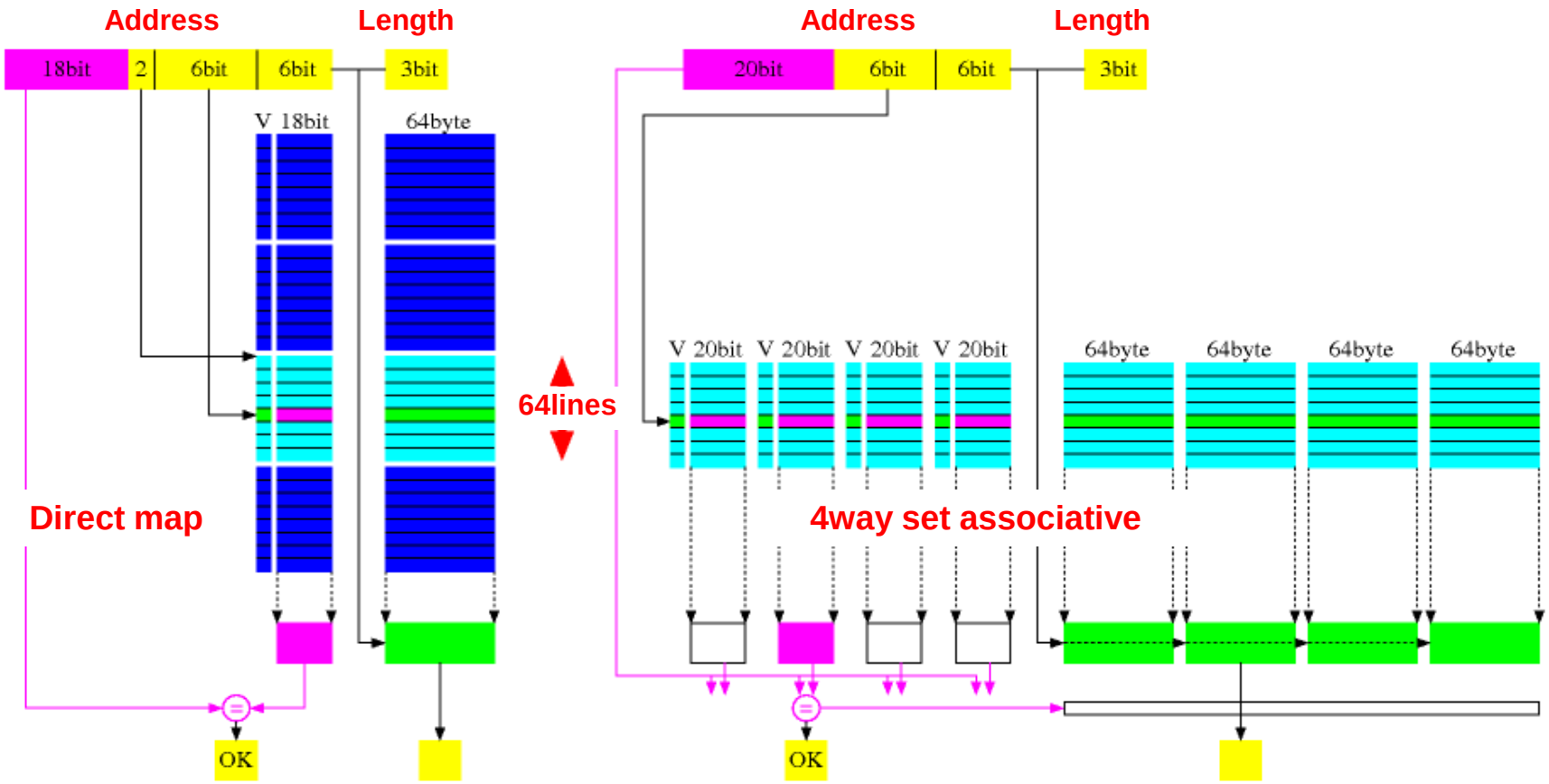


(a) Simple structure (b) Reuse+prefetch (c) Reuse+multiple-prefetch

Inside of cache

Memory address has three fields

- **High-bits for tag**
- **Mid-bits for line-number**
- **Low-bits for offset in each line**



NAIST

Type of cache (write-through)

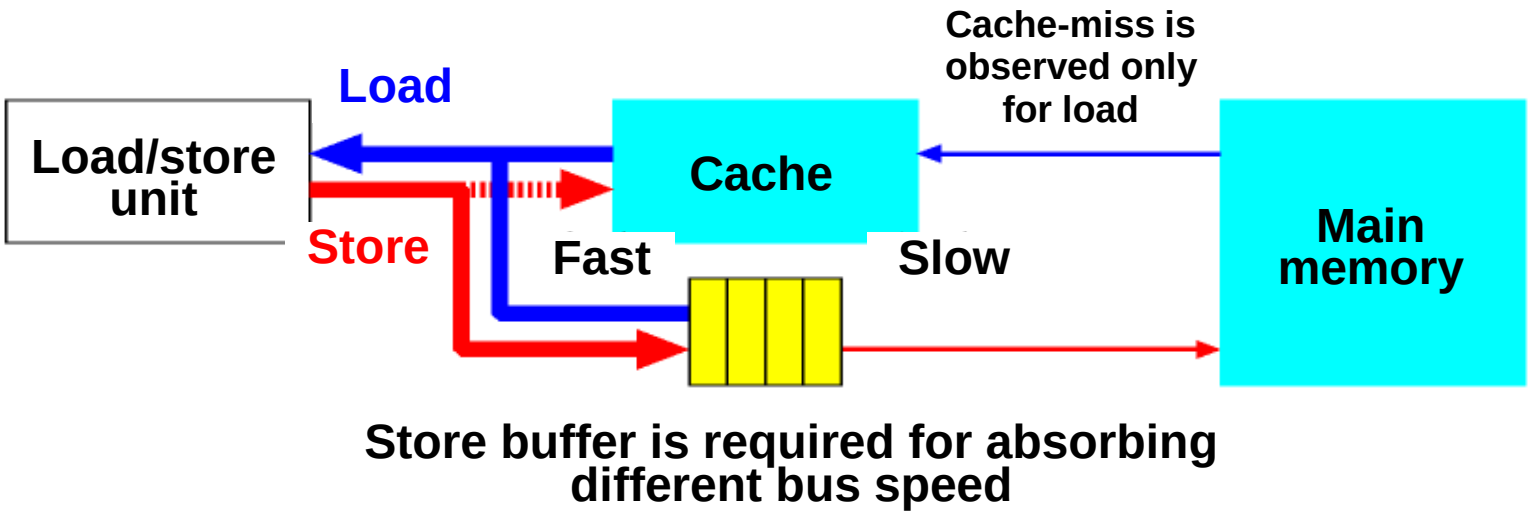
Write-through cache:

Store data to both of cache and memory simultaneously

- **× Store-buffer is required**
- **○ Replacement is simple**

Continuous store gives heavy pressure to store buffer.

- **If throughput from store buffer to memory is poor, the performance is significantly degraded.**



Type of cache (write-back)

Write-back cache:

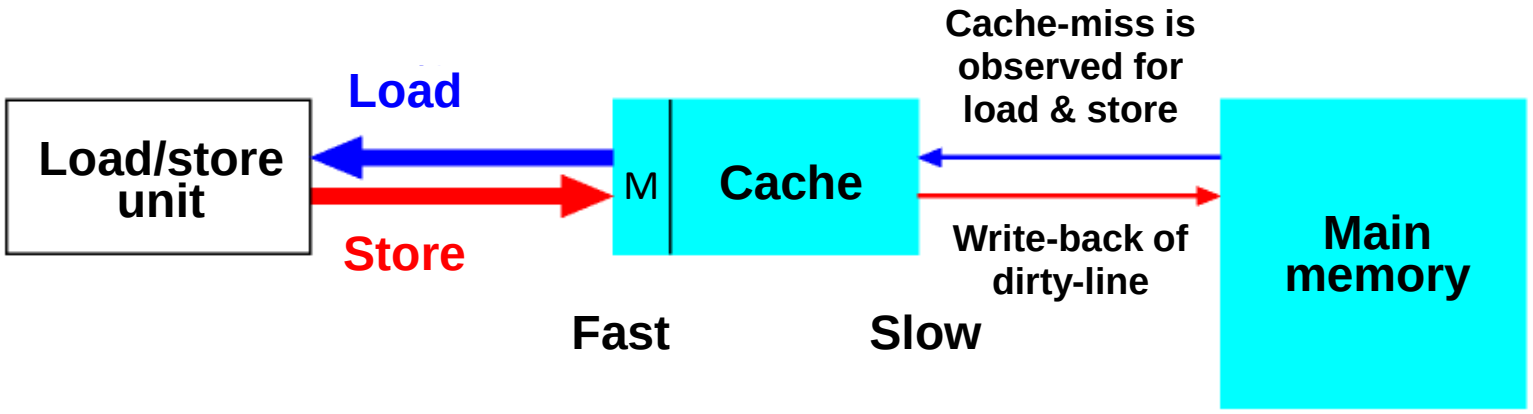
Store data only to cache, written back to memory later

○ high-speed for multiprocessor system

× Replacement is complicated

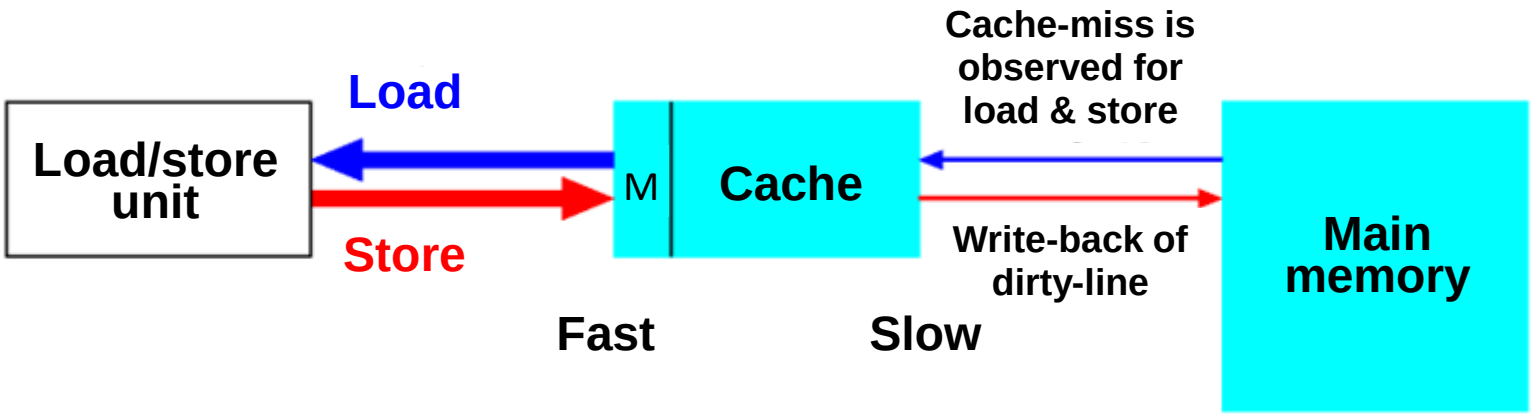
Can save power consumption of memory bus.

Can save traffic of memory bus.



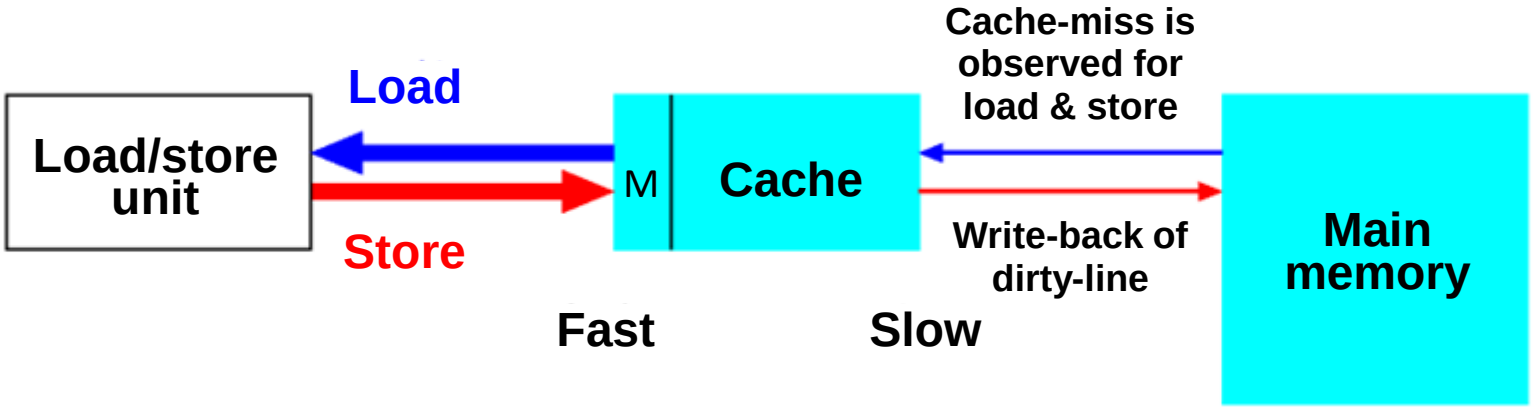
Formative assessment

- Describe the procedure when store misses cache. (consider both of the case M=0 and M=1)



Formative assessment

- Describe the procedure when store misses cache. (consider both of the case M=0 and M=1)



Let's assume "store addr-A" misses cache, and corresponding cache-line has valid data of addr-B.

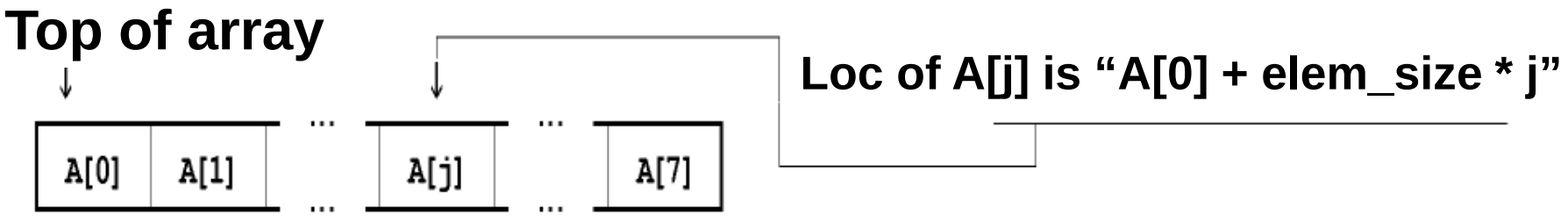
- If M=0, the cache-line has the same value as in addr-B.
  1. The line can be discarded immediately.
  2. Send read-request for addr-A to memory.
  3. Arrived data is stored in the line.
- If M=1, the cache-line has new value and memory has old value.
  1. Send write-request for addr-B to memory.
  2. Send read-request for addr-A to memory.
  3. Arrived data is stored in the line.

M is changed to 1.

Programming language and data cache

Alignment of data-structure and cache-structure (1D-array)

- Architectural important issue is layout of data-structure.



In case of C:

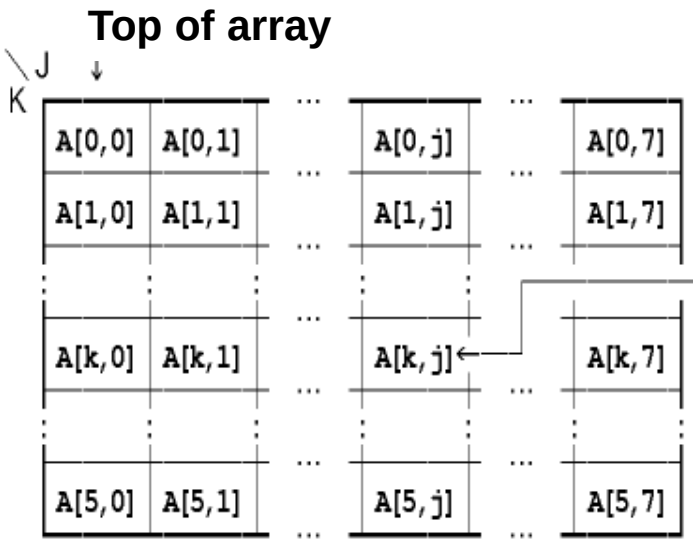
```
int i;
int A[1000];                    /* 0 .. 999
for (i=0; i<1000; i++) {        /* 0 .. 999
  A[i] = xxxx;
}
```

In case of FORTRAN:

```
INTEGER I
INTEGER A(1000)
DO 10 I=1,1000          ··· 1 .. 1000
  A(I) = xxxx           ··· loc of A[I] is “A[1] + elem_size * (I-1)”
10 CONTINUE
```

Alignment of data-structure and cache-structure (2D-array)

- **Location of data depends on programming language.**



Loc of A[k,j] is "A[0,0] + elem_size * (num_j * k + j)"
or "A[0,0] + elem_size * (num_k * j + k)"

In case of C:

```
for (i=0; i<1000; i++)   /* A[0[0] ... A[999][999]
  for (j=0; j<1000; j++)/* Inner-most loop should right-side index
    A[i][j] = xxxx;  /*  A[0][0] A[0][1] … A[999][998] A[999][999]
```

In case of FORTRAN:

```
DO 10 I=1,1000
  DO 10 J=1,1000     ⋯ Inner-most loop should left-side index
    A(J,I) = xxxx    ⋯ A[1][1] A[2][1] … A[999][1000] A[1000][1000]
10 CONTINUE
```

Estimation of cache capacity

```
double d0[ 512][1024/8]; /* 0.5MB */
double d1[1024][1024/8]; /* 1.0MB */
double d2[2048][1024/8]; /* 2.0MB */
double d3[4096][1024/8]; /* 4.0MB */
main() { int i, j;
    restme();
    for (i=0; i<4096*N; i++)
        for (j=0; j<512; j++) d0[j][0] += 1.0;
    printf(" .5M:%d", gettme());

    restme();
    for (i=0; i<2048*N; i++)
        for (j=0; j<1024; j++) d1[j][0] += 1.0;
    printf(" 1M:%d", gettme());

    restme();
    for (i=0; i<1024*N; i++)
    for (j=0; j<2048; j++) d2[j][0] += 1.0;
    printf(" 2M:%d", gettme());

    restme();
    for (i=0; i<512*N; i++)
    for (j=0; j<4096; j++) d3[j][0] += 1.0;
    printf(" 4M:%d\n", gettme());
}
```

Performance degradation by cache-miss

CPU	frequency-L2capacity	execution time (seconds)
SPARC	650M-0.5M	.5M:67 1M:616 2M:733 4M:742
Pentium-M	1.5G-1.0M	.5M:29 1M:34 2M:265 4M:301
XEON	2.8G-0.5M	.5M:25 1M:312 2M:305 4M:311

real app. (cache-hit-ratio=high) : 26sec

real app. (cache-hit-ratio=low) : 70sec

Pentium-4	3.2G-2.0M	.5M:22 1M:23 2M:25 4M:233

real app. (high): 25sec

real app. (low) : 51sec

Pentium-4	3.4G-1.0M	.5M:18 1M:22 2M:122 4M:121

real app. (high): 24sec

real app. (low) : 48sec

XEON	3.6G-1.0M	.5M:19 1M:19 2M:207 4M:205

real app. (high): 22sec

real app. (low) : 45sec

That’s all for today