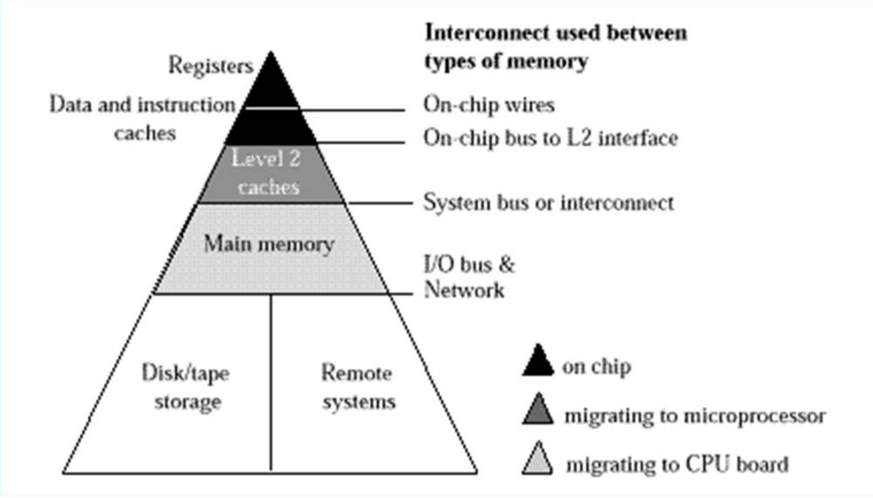
SE-292 High Performance Computing
Memory Hierarchy

R. Govindarajan
govind@serc

Memory Hierarchy



Memory Organization

Memory hierarchy

- CPU registers
  - few in number (typically 16/32/128)
  - subcycle access time (nsec)
- Cache memory
  - on-chip memory
  - 10's of KBytes (to a few MBytes) of locations.
  - access time of a few cycles
- Main memory
  - 100's of MBytes storage
  - access time several 10's of cycles
- Secondary storage (like disk)
  - 100's of GBytes storage
  - access time msec

3

Cache Memory; Memory Hierarchy

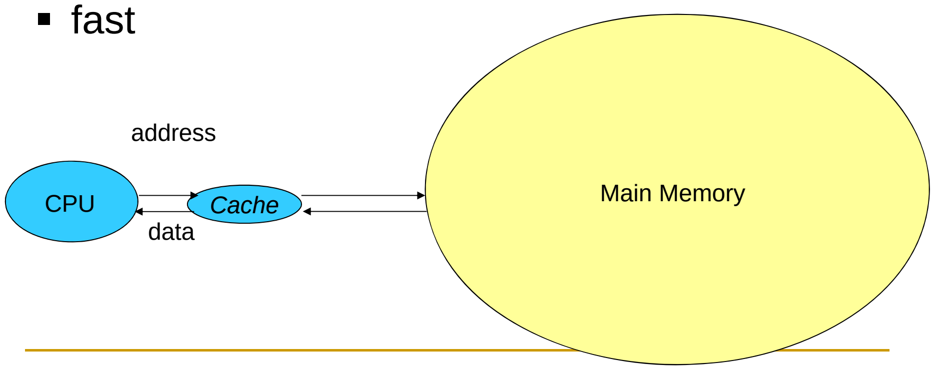
- Recall: In discussing pipeline, we assumed that memory latency will be hidden so that it appears to operate at processor speed
- Cache Memory: HW that makes this happen
  - Design principle: Locality of Reference
  - Temporal locality: least recently used objects are least likely to be referenced in the near future
  - Spatial locality: neighbours of recently reference locations are likely to be referenced in the near future

4

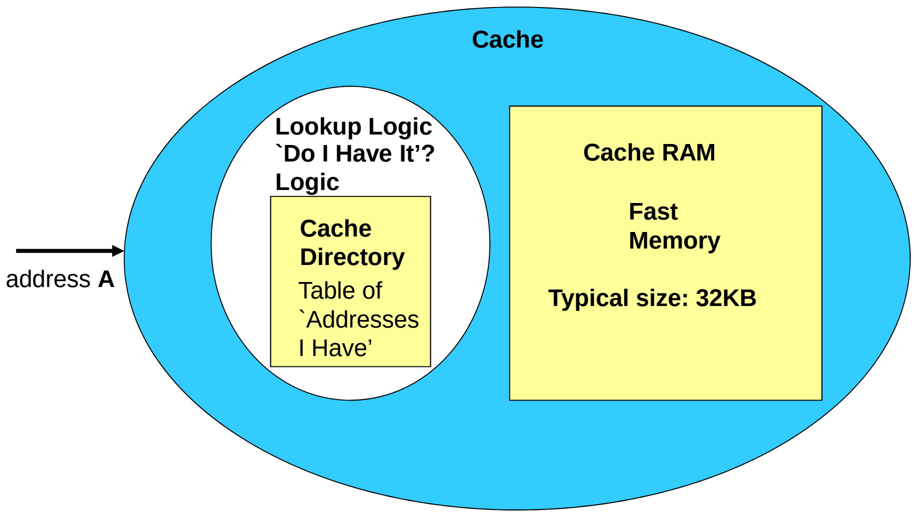
Cache Memory Exploits This

Cache: Hardware structure that provides memory contents the processor references

- directly (most of the time)
- fast

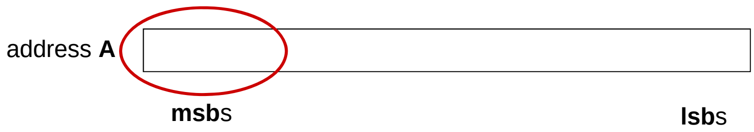


Cache Design



How to do Fast Lookup?

- Search Algorithms
- Hashing: Hash table, indexed into using a hash function
- Hash function on address A. Which bits?



For a small program, everything would index into the same place (collision)
A and neighbours possibly differ only in these bits; should be treated as one

7

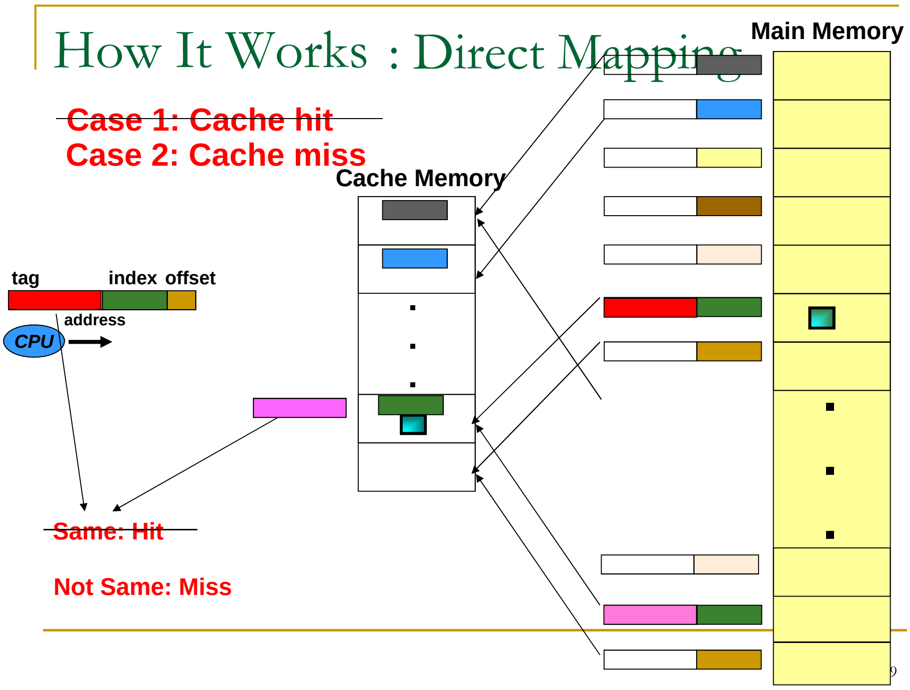
Summing up

- Cache organized in terms of **blocks**, memory locations that share the same address bits other than lsbs. Main memory too.
- Address used as

tag	**Index into directory**	**block offset**

8

How It Works : Direct Mapping



Cache Terminology

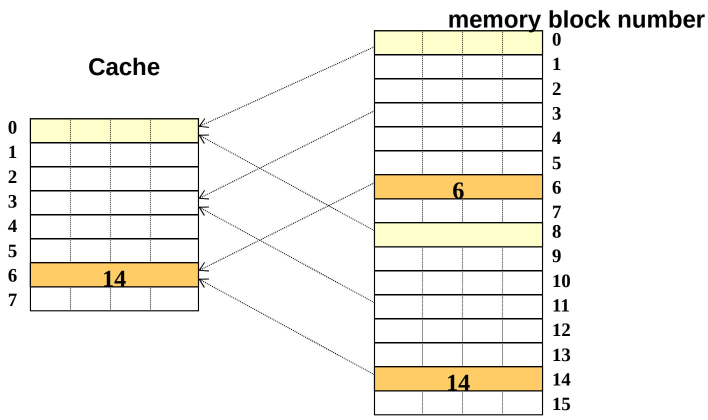
- Cache hit: A memory reference where the required data is found in the cache
- Cache Miss: A memory reference where the required data is not found in the cache
- Hit Ratio: # of hits / # of memory references
- Miss Ratio = (1 - Hit Ratio)
- Hit Time: Time to access data in cache
- Miss Penalty: Time to bring a block to cache

Cache Organizations

1. Where can a block be placed in the cache?
   - Direct mapped, Set Associative
2. How to identify a block in cache?
   - Tag, valid bit, tag checking hardware
3. Replacement policy?
   - LRU, FIFO, Random …
4. What happens on writes?
   - Hit: When is main memory updated?
     - Write-back, write-through
   - Miss: What happens on a write miss?
     - Write-allocate, write-no-allocate

Block Placement: Direct Mapping

- A memory block goes to the unique cache block (memory block no.) mod (# cache blocks)



Identifying Memory Block (DM Cache)

Assume 32-bit address space, 16 KB cache, 32byte cache block size.

Offset field -- to identify bytes in a cache line

Offset Bits = log (32) = 5 bits

No. of Cache blocks = 16KB/ 32 = 512

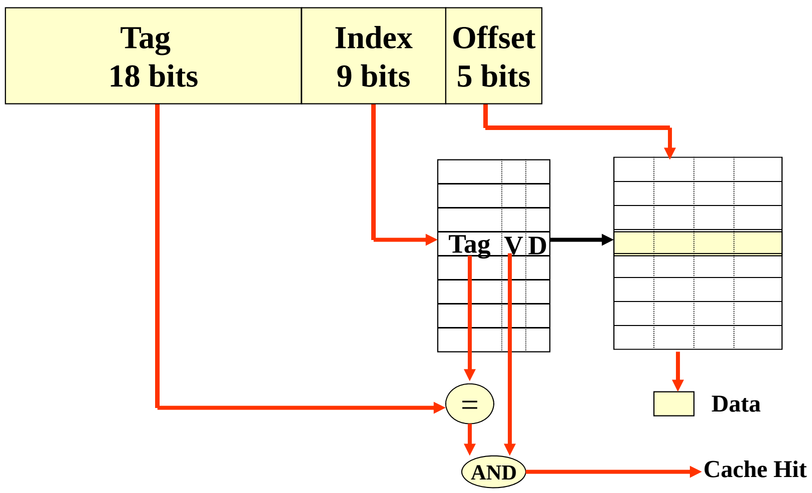
Index Bits = log (522) = 5 bits

Tag -- identify which memory block is in this cache block -- remaining bits (= 18bits)

Tag	Index	Offset
18 bits	9 bits	5 bits

13

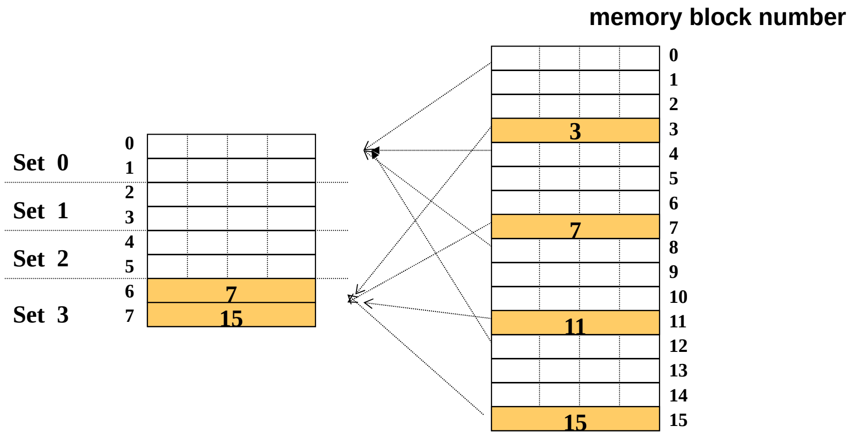
Accessing Block (DM Cache)



14

Block Placement: Set Associative

- A memory block goes to unique set, and within the set to any cache block



Identifying Memory Block (Set Associative Cache)

Assume 32-bit address space, 16 KB cache, 32byte cache block size, 4-way set-associative.

Offset field -- to identify bytes in a cache line

Offset Bits = log (32) = 5 bits

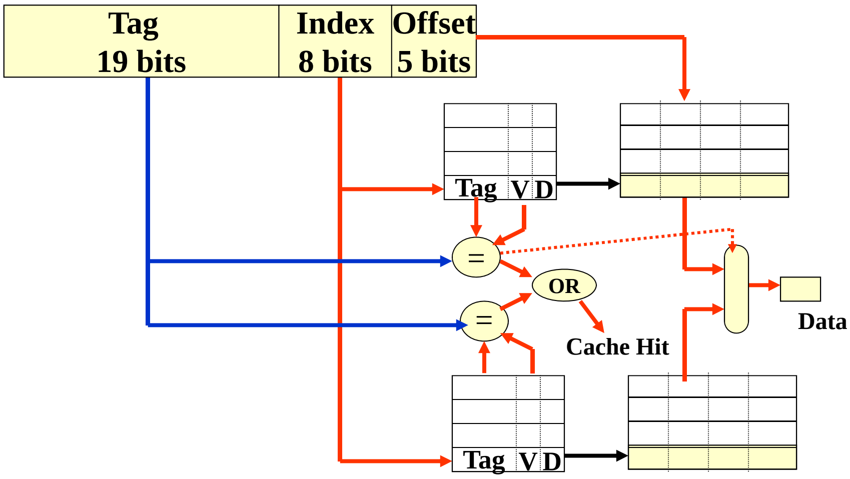
No. of Sets = Cache blocks / 4 = 512/4 = 128

Index Bits = log (128) = 7 bits

Tag -- identify which memory block is in this cache block -- remaining bits (= 20 bits)

Tag	Index	Offset
20 bits	7 bits	5 bits

Accessing Block (2-w Set-Associative)



17

Block Replacement

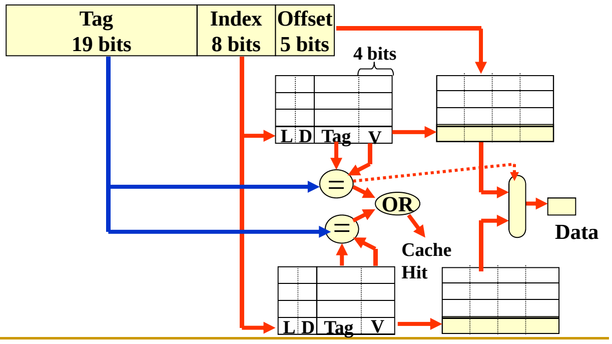
- Direct Mapped: No choice is required
- Set-Associative: Replacement strategies
  - First-In-First-Out (FIFO)
    - simple to implement
  - Least Recently Used (LRU)
    - complex, but based on (temporal) locality, hence higher hits
  - Random
    - simple to implement

18

Block Replacement…

•Hardware must keep track of LRU information

•Separate valid bits for each word (or sub-block) of cache can speedup access to the required word on a cache miss



Write Policies

When is Main Memory Updated on Write Hit?

- Write through: Writes are performed both in Cache and in Main Memory
  - \+ Cache and memory copies are kept consistent
  - -- Multiple writes to the same location/block cause higher memory traffic
  - -- Writes must wait for longer time (memory write)

  Solution: Use a Write Buffer to hold these write requests and allow processor to proceed immediately

Write Policies…

- Write back: writes performed only on cache. Modified blocks are written back in memory on replacement
  - o Need for dirty bit with each cache block
  - + Writes are faster than with write through
  - + Reduced traffic to memory
  - -- Cache & main memory copies are not always the same
  - -- Higher miss penalty due to write-back time

21

Write Policies…

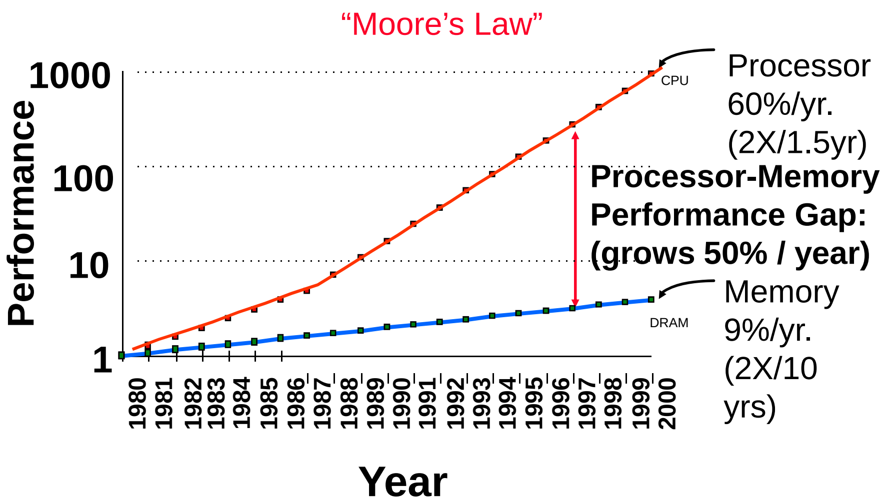
What happens on a Write Miss?

- Write-Allocate: allocate a block in the cache and load the block from memory to cache.
- Write-No-Allocate: write directly to main memory.

- Write allocate/no-allocate is orthogonal to write-through/write-back policy.
  - Write-allocate with write-back
  - Write-no-allocate with write-through: ideal if mostly-reads-few-writes on data

22

What Drives Computer Architecture?



23

Cache and Programming

- Objective: Learn how to assess cache related performance issues for important parts of our programs
- Will look at several examples of programs
- Will consider only data cache, assuming separate instruction and data caches
- Data cache configuration:
  - Direct mapped 16 KB write back cache with 32B block size

Tag : 18b	Index: 9b	Offset: 5b

24

Example 1: Vector Sum Reduction

```
double A[2048]; sum=0.0;
for (i=0; i<2048, i++)
    sum = sum +A[i];
```

- To do analysis, must view program close to machine code form (to see loads/stores)

```
Loop:   FLOAD   F0,   0(R1)
        FADD    F2,   F0,   F2
        ADDI    R1,   R1,   8
        BLE     R1,   R3,   Loop
```

25

Example 1: Vector Sum Reduction

- To do analysis:
  - Observe loop index i, sum and &A[i] are implemented in registers and not load/stored inside loop
  - Only A[i] is loaded from memory
  - Hence, we will consider only accesses to array elements

26

Example 1: Reference Sequence

- load A[0] load A[1] load A[2] … load A[2047]
- Assume base address of A (i.e., address of A[0]) is 0xA000, 10|10 0000 000|0 0000
  - Cache index bits: 100000000 (value = 256)
- Size of an array element (double) = 8B
- So, 4 consecutive array elements fit into each cache block (block size is 32B)
  - A[0] – A[3] have index of 256
    100000001 00000
    100000001 01000
    100000001 10000
    100000001 11000
  - A[4] – A[7] have index of 257 and so on

Example 1: Cache Misses and Hits

A[0]	0xA000	256	Miss	Cold start
A[1]	0xA008	256	Hit	
A[2]	0xA010	256	Hit	
A[3]	0xA018	256	Hit	
A[4]	0xA020	257	Miss	Cold start
A[5]	0xA028	257	Hit	
A[6]	0xA030	257	Hit	
A[7]	0xA038	257	Hit	
..	..	..	..	
..	..	..	..	
A[2044]	0xDFE0	255	Miss	Cold start
A[2045]	0xDFE8	255	Hit	
A[2046]	0xDFF0	255	Hit	
A[2047]	0xDFF8	255	Hit	

Hit ratio of our loop is 75% -- there are 1536 hits out of 2048 memory accesses

This is entirely due to spatial locality of reference.

Cold start miss: we assume that the cache is initially empty. Also called a Compulsory Miss

If the loop was preceded by a loop that accessed all array elements, the hit ratio of our loop would be 100%, 25% due to temporal locality and 75% due to spatial locality

Example 1 with double A[4096]

Why should it make a difference?

- Consider the case where the loop is preceded by another loop that accesses all array elements in order
- The entire array no longer fits into the cache – cache size: 16KB, array size: 32KB
- After execution of the previous loop, the second half of the array will be in cache
- Analysis: our loop sees misses as we just saw
- Called Capacity Misses as they would not be misses if the cache had been big enough

29

Example 2: Vector Dot Product

double A[2048], B[2048], sum=0.0;

for (i=0; i<2048, i++) sum = sum +A[i] * B[i];

- Reference sequence:
  - load A[0] load B[0] load A[1] load B[1] …
- Again, size of array elements is 8B so that 4 consecutive array elements fit into each cache block
- Assume base addresses of A and B are 0xA000 and 0xE000

0000 10 | 10 0000 000 | 0 0000

0000 11 | 10 0000 000 | 0 0000

30

Example 2: Cache Hits and Misses

A[0]	0xA000	256	Miss	Cold start
B[0]	0xE000	256	Miss	Cold start
A[1]	0xA008	256	Miss	Conflict
B[1]	0xE008	256	Miss	Conflict
A[2]	0xA010	256	Miss	Conflict
B[2]	0xE010	256	Miss	Conflict
A[3]	0xA018	256	Miss	Conflict
B[3]	0xE018	256	Miss	Conflict
..	..	..	..	
..	..	..	..	
B[1023]	0xFFF8	511	Miss	Conflict

Conflict miss: a miss due to conflicts in cache block requirements from memory accesses of the same program

Hit ratio for our program: 0%

Source of the problem: the elements of arrays A and B accessed in order have the same cache index

Hit ratio would be better if the base address of B is such that these cache indices differ

Example 2 with Padding

- Assume that compiler assigns addresses as variables are encountered in declarations
- To shift base address of B enough to make cache index of B[0] different from that of A[0]

  double A[2052], B[2048];
- Base address of B is now 0xE020
  - 0xE020 is 1110 0000 0010 0000
  - Cache index of B[0] is 257; B[0] and A[0] do not conflict for the same cache block
- Whereas Base address of A is 0xA000 which is 1010 0000 0000 0000 – cache index is 256
- Hit ratio of our loop would then be 75%

Example 2 with Array Merging

What if we re-declare the arrays as

```
struct {double A, B;} array[2048];
for (i=0; i<2048, i++)
    sum += array[i].A*array[i].B;
```

Hit ratio: 75%

33

Example 3: DAXPY

- Double precision **Y** = **aX** + **Y**, where **X** and **Y** are vectors and **a** is a scalar

```
double  X[2048],  Y[2048],  a;
 for (i=0; i<2048;i++)
     Y[I] = a*X[I]+Y[I];
```

- Reference sequence
  - load X[0] load Y[0] store Y[0] load X[1] load Y[1] store Y[1] …
- Hits and misses: Assuming that base addresses of X and Y don't conflict in cache, hit ratio of 83.3%

34

Example 4: 2-d Matrix Sum

```
double A[1024][1024], B[1024][1024];
for (j=0;j<1024;j++)
    for (i=0;i<1024;i++)
        B[i][j] = A[i][j] + B[i][j];
```

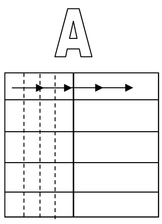
- Reference Sequence:

  load A[0,0] load B[0,0] store B[0,0]

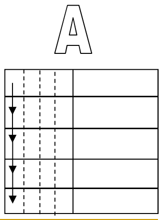
  load A[1,0] load B[1,0] store B[1,0] **…**
- Question: In what order are the elements of a multidimensional array stored in memory?

Storage of Multi-dimensional Arrays

- Row major order
  - Example: for a 2-dimensional array, the elements of the first row of the array are followed by those of the 2nd row of the array, the 3rd row, and so on
  - This is what is used in C



- Column major order
  - A 2-dimensional array is stored column by column in memory
  - Used in FORTRAN



Example 4: 2-d Matrix Sum

```
double A[1024][1024], B[1024][1024];
for (j=0;j<1024;j++)
    for (i=0;i<1024;i++)
        B[i][j] = A[i][j] + B[i][j];
```

- Reference Sequence:

  load A[0,0] load B[0,0] store B[0,0]

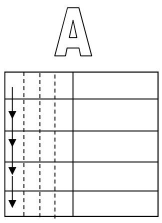
  load A[1,0] load B[1,0] store B[1,0] …

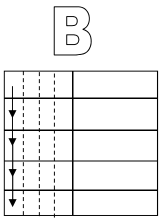
Example 4: Hits and Misses

- Reference order and storage order for an array are not the same
- Our loop will show no spatial locality
  - Assume that packing has been to eliminate conflict misses due to base addresses
  - Reference Sequence:

    load A[0,0] load B[0,0] store B[0,0]

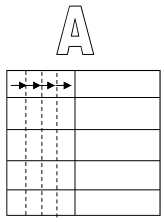
    load A[1,0] load B[1,0] store B[1,0] …
  - Miss(cold), Miss(cold), Hit for each array element
  - Hit ratio: 33.3%
  - Question: Will A[0,1] be in the cache when required?

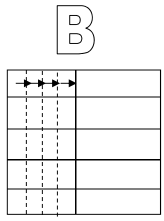




Example 4 with Loop Interchange

```
double A[1024][1024], B[1024][1024];
 for (i=0;i<1024;i++)
     for (j=0;j<1024;j++)
            B[i][j] = A[i][j] + B[i][j];
```





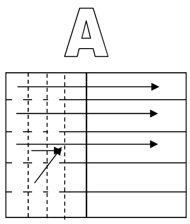
- Reference Sequence:

  load A[0,0] load B[0,0] store B[0,0]

  load A[0,1] load B[0,1] store B[0,1] …
- Hit ratio: 83.3%

Is Loop Interchange Always Safe?

```
for (i=2047; i>1; i--)
~~for (i=1; i<2048; i++)~~
    for (j=1; j<2048; j++)
        A[i][j] = A[i+1][j-1] + A[i][j-1];
```



A[1,1] = A[2,0]+A[1,0]

A[2,1] = A[3,0]+A[2,0]

…

A[1,2] = A[2,1]+A[1,1]

A[1,1] = A[2,0]+A[1,0]

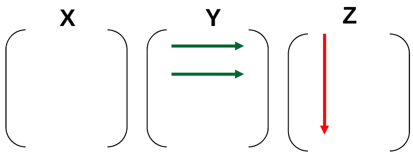
A[1,2] = A[2,1]+A[1,1]

…

A[2,1] = A[3,0]+A[2,0]

Example 5: Matrix Multiplication

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      for (k=0; k<N; k++)
         X[i][j] += Y[i][k] * Z[k][j];
```



/ Dot product inner loop

Reference Sequence:

Y[0,0], Z[0,0], Y[0,1], Z[1,0], Y[0,2], Z[2,0] … X[0,0],
Y[0,0], Z[0,1], Y[0,1], Z[1,1], Y[0,2], Z[2,1] … X[0,1],
…
Y[1,0], Z[0,0], Y[1,1], Z[1,0], Y[1,2], Z[2,0] … X[1,0],
…

With Loop Interchanging

- Can interchange the 3 loops in any way
- Example: Interchange i and k loops

```
double X[N][N], Y[N][N], Z[N][N];
for (k=0; k<N; k++)
    for (j=0; j<N; j++)
        for (i=0; i<N; i++)
            X[i][j] += Y[i][k] * Z[k][j];
```

- For inner loop: Z[k][j] can be loaded into register once for each (k,j), reducing the number of memory references

Let's try some Loop Unrolling Instead

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      for (k=0; k<N; k+=2)          Unroll k loop
         X[i][j] += Y[i][k]*Z[k][j] + Y[i][k+1]*Z[k+1][j];
```

Exploits spatial locality for array Z?

43

Let's try some Loop Unrolling Instead

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0; i<N; i++)
  for (j=0; j<N; j+=2)          Unroll j loop
    for (k=0; k<N; k++) {
        X[i][j] += Y[i][k]*Z[k][j];
        X[I][j+1] += Y[I][k]*Z[k][j+1];
    }
```

Exploits spatial locality for array Z

44

Let’s try some Loop Unrolling Instead

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0; i<N; i++)
    for (j=0; j<N; j+=2)                                    Unroll j loop
      for (k=0; k<N; k+=2){                                 Unroll k loop
         X[i][j]      += Y[i][k]*Z[k][j]      +Y[i][k+1]*Z[k+1][j];
         X[i][[j+1]   += Y[i][k]*Z[k][j+1]    +Y[i][k+1]*Z[k+1][j+1];
      }
```

Exploits spatial locality for array Z

Exploits temporal locality for array Y

Blocking or Tiling

45

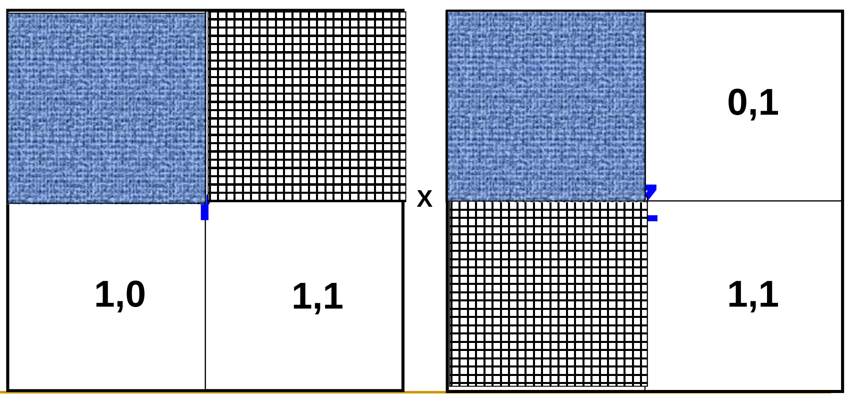
Blocking/Tiling

X   Y x Z

0,0   0,0x0,0 + 0,1x1,0

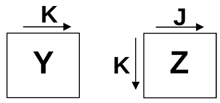
1,0   1,0x0,0 + 1,1x1,0

Idea: Since problem is with accesses to array Z, make full use of elements of Z when they are brought into the cache



46

Blocked Matrix Multiplication




```
for (J=0; J<N; J+=B)
for (K=0; K<N; K+=B)
for (i=0; i<N; i++)
for (j=J; j<min(J+B,N); j++){
   for (k=K, r=0; k<min(K+B,N); k++)
      r += Y[i][k] * Z[k][j];
    X[i][j] += r;
}
```

47

Revisit Example 1 : with double A[4096]

```
double A[4096];
sum=0.0;
for (i=0; i<4096, i++)
    sum = 0.0;
for (i=0; i<4096, i++)
    sum = sum +A[i];
```

- The entire array no longer fits into the cache – cache size: 16KB, array size: 32KB
- After execution of the previous loop, the second half of the array will be in cache
- Analysis: our loop sees misses as we just saw
- Called Capacity Misses as they would not be misses if the cache had been big enough

48

Some Homework

1. Implementing Matrix Multiplication

   Objective: Best programs for multiplying 1024x1024 double matrices on any 2 different machines that you normally use.

   Techniques: Loop interchange, blocking, etc

   Criterion: Execution time

   Report: Program and execution times