UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

13. Ottimizzazioni sulla memoria

Compilatori – Middle end [I215-014]

Corso di Laurea in INFORMATICA
(D.M.270/04) [16-215]
Anno accademico 2024/2025

Prof. Andrea Marongiu
andrea.marongiu@unimore.it

Copyright note

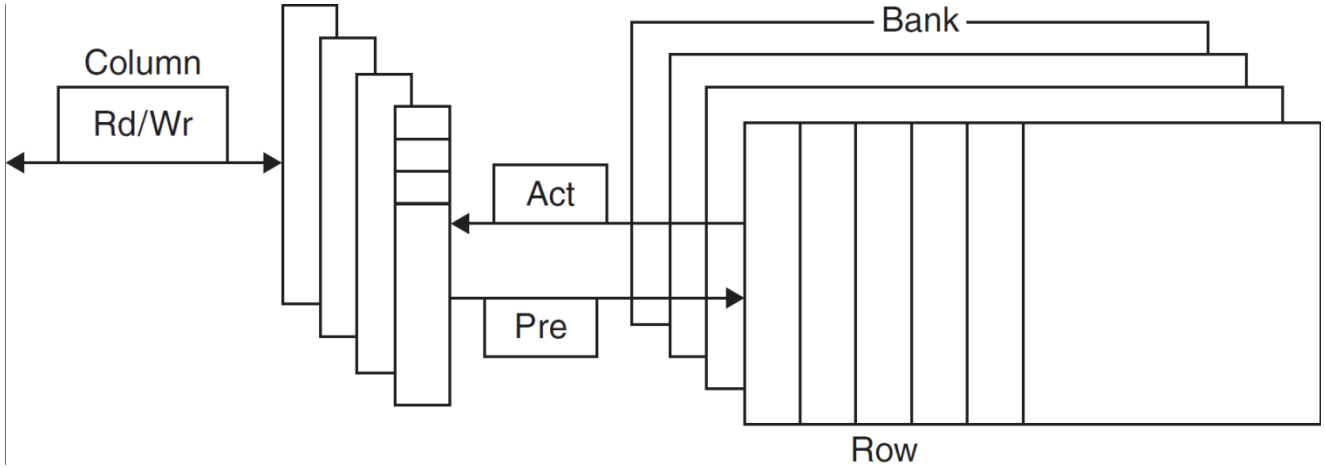
Credits

- Hennessy, Patterson, “Computer Organization and Design - RISC-V Edition : The Hardware Software Interface”, Morgan Kaufmann
- Marchetti-Spaccamela, La Sapienza Università di Roma, “Cache e gerarchia di memoria”
- Gibbons, Carnegie Mellon University, “Optimizing Compilers”
- Pekhimenko, University of Toronto, “Compiler Optimization”
- Juurlink, Technische Universitat Berlin, “Advanced Memory Hierarchy Design”

Outline

DRAM Technology

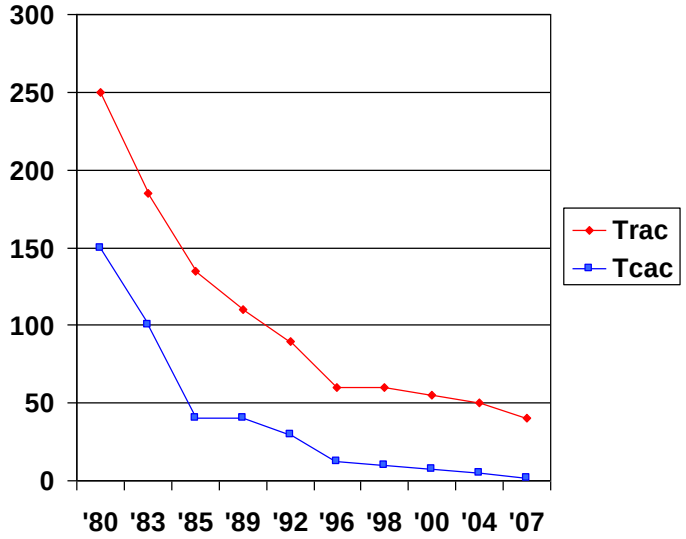
- Data stored as a charge in a capacitor
  - Single transistor used to access the charge
  - Must periodically be refreshed
    - Read contents and write back
    - Performed on a DRAM “row”



Advanced DRAM Organization

- Bits in a DRAM are organized as a rectangular array
  - DRAM accesses an entire row
  - Burst mode: supply successive words from a row with reduced latency
- Double data rate (DDR) DRAM
  - Transfer on rising and falling clock edges
- Quad data rate (QDR) DRAM
  - Separate DDR inputs and outputs

DRAM Generations

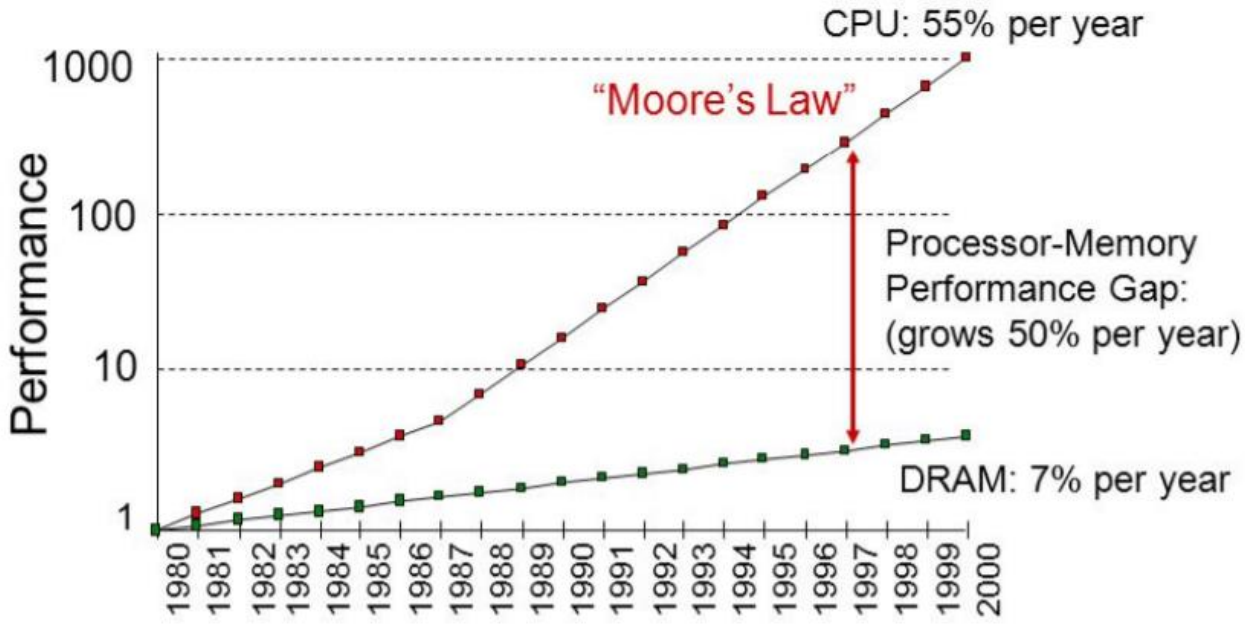


Year introduced	Chip size	$ per GiB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	$1,500,000	250 ns	150 ns
1983	256 Kibibit	$500,000	185 ns	100 ns
1985	1 Mebibit	$200,000	135 ns	40 ns
1989	4 Mebibit	$50,000	110 ns	40 ns
1992	16 Mebibit	$15,000	90 ns	30 ns
1996	64 Mebibit	$10,000	60 ns	12 ns
1998	128 Mebibit	$4,000	60 ns	10 ns
2000	256 Mebibit	$1,000	55 ns	7 ns
2004	512 Mebibit	$250	50 ns	5 ns
2007	1 Gibibit	$50	45 ns	1.25 ns
2010	2 Gibibit	$30	40 ns	1 ns
2012	4 Gibibit	$1	35 ns	0.8 ns

DRAM Performance Factors

- Row buffer
  - Allows several words to be read and refreshed in parallel
- Synchronous DRAM
  - Allows for consecutive accesses in bursts without needing to send each address
  - Improves bandwidth
- DRAM banking
  - Allows simultaneous access to multiple DRAMs
  - Improves bandwidth

The memory wall



- 1980 – No cache in microprocessor
- 1995 – Two-level cache on microprocessor

CPU
60% per yr
2X in 1.5 yrs

DRAM
9% per yr
2X in 10 yrs

- access to data is a limiting factor
  - ***we'd better spend transistors to improve memory behavior***

Are there other "types" of memory?

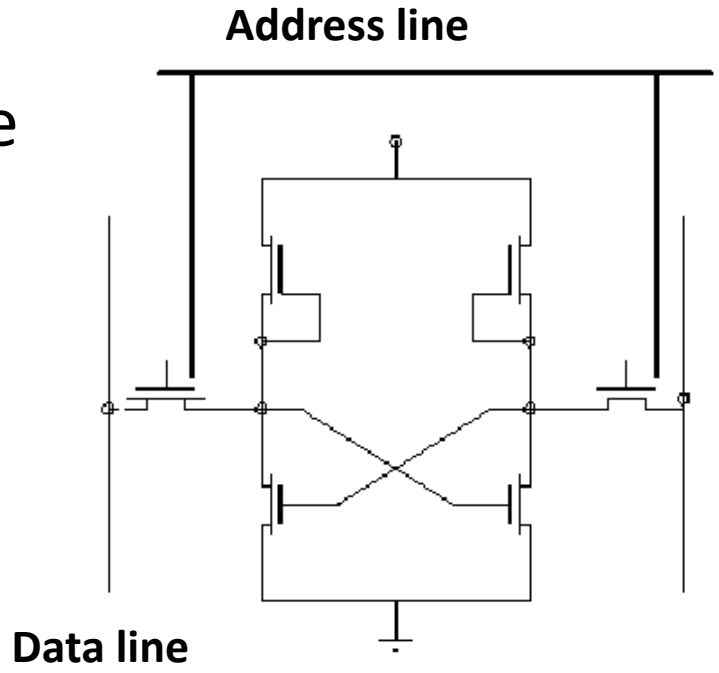
- Different technology than DRAM?
- With better performance than DRAM?

Static RAM (SRAM)

The elementary cell is composed of 6 MOC transistors MOS forming a FLIP-FLOP

The information remains stable as long as the circuit is powered

- Fast access time
- High density
- High cost



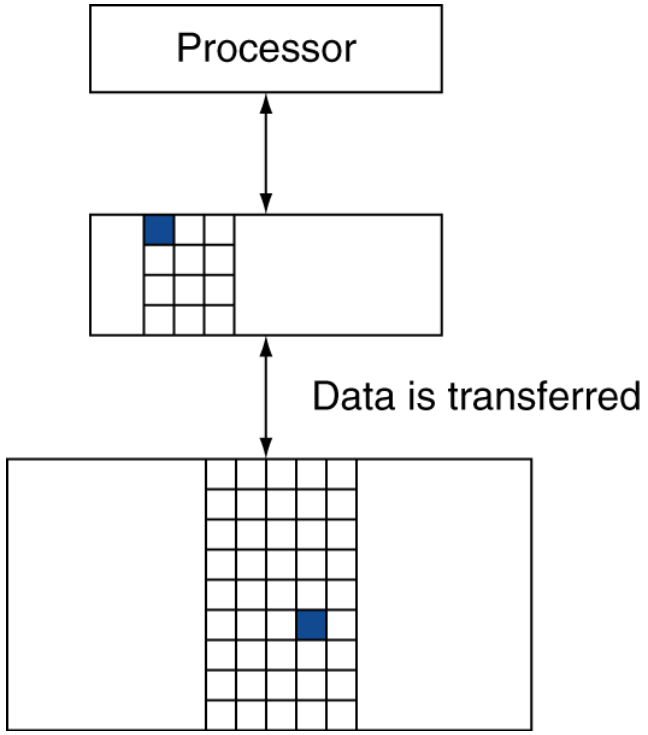
Memory Technology

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk

Memory Hierarchy Levels

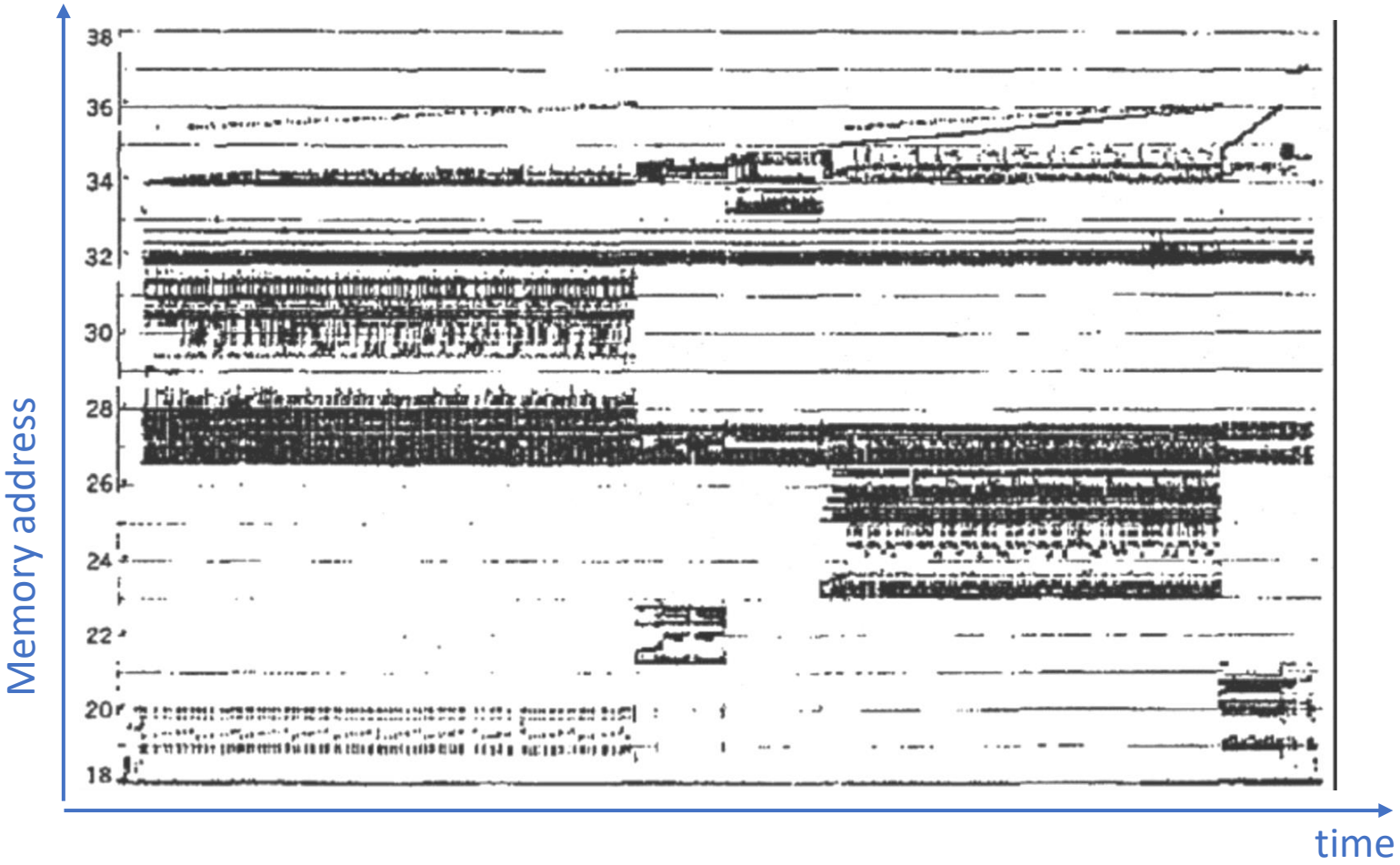
- Block (aka line): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses
      = 1 – hit ratio
  - Then accessed data supplied from upper level



Principle of Locality

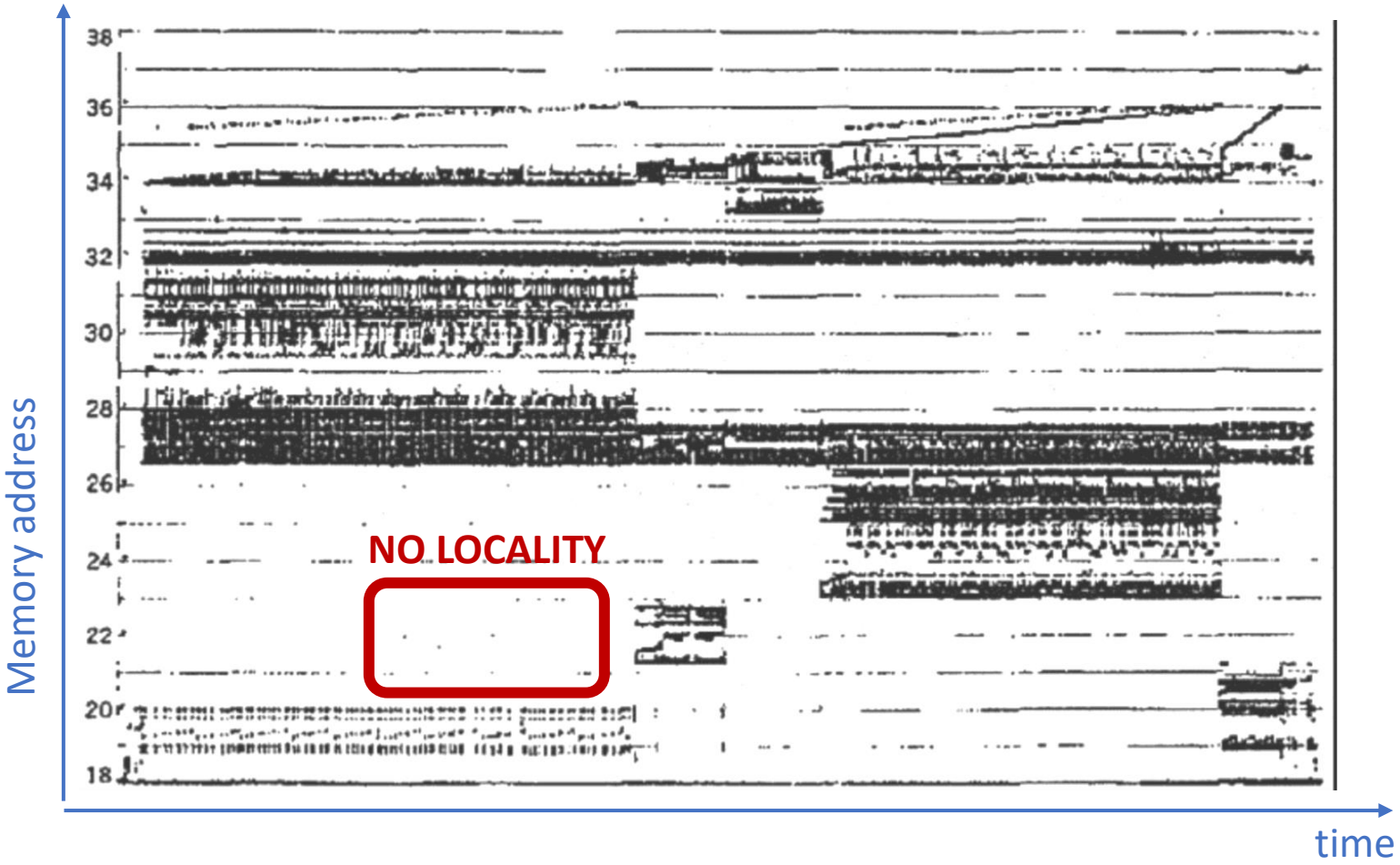
- Programs access a small proportion of their address space at any time
- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

Locality in programs



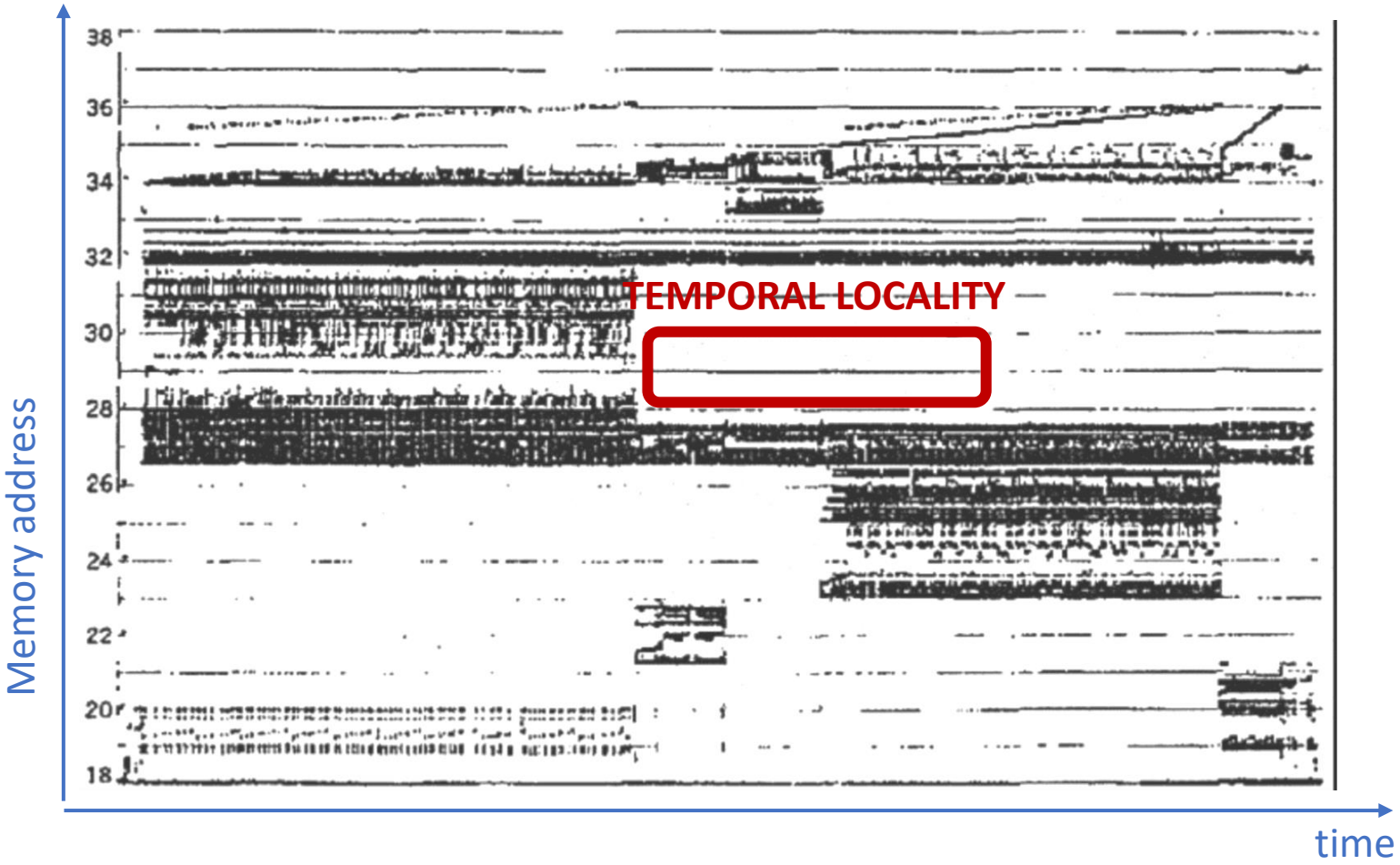
Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Locality in programs



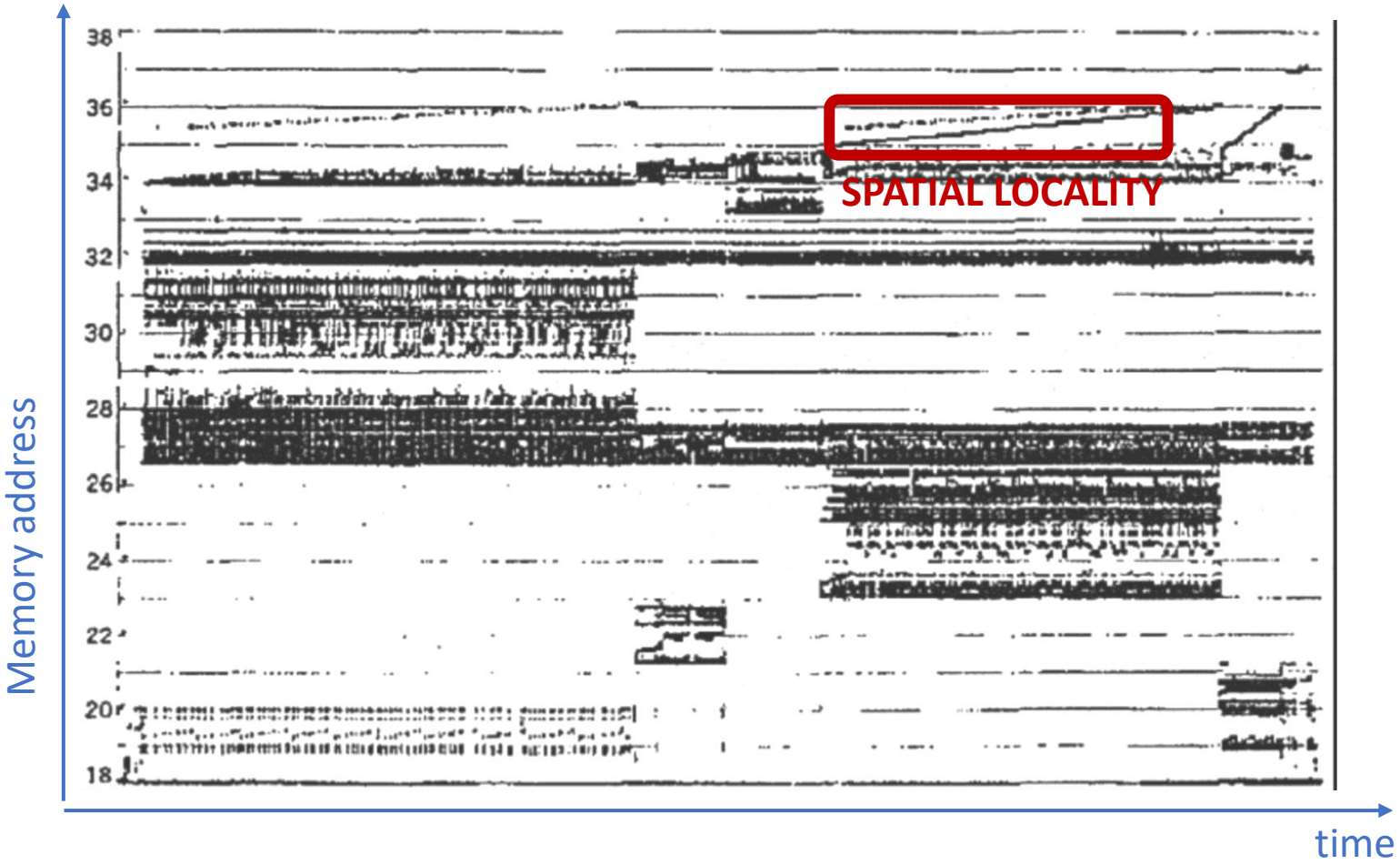
Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Locality in programs



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Locality in programs



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Taking Advantage of Locality

- Memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - Cache memory attached to CPU

Caches

- The term "cache" was used for the first time to indicate the level of the memoy hierarchy between the CPU and the main memory
  - but it is nowadays used to indicate any memory management scheme that exploits access locality
- Simple mechanism: The processor only interacts with the cache. Upon cache miss the requested word is copied from the main memory to the cache
- Need mechanisms for:
  - Knowing if data is present in the cache
  - If so, locate the data and access it
- These operations must execute as fast as possible, as this heavily impacts the performance of the memory system

Hit & Miss

- We have a **cache hit** when the address requested from the level above (e.g., the processor) are present in the cache.
- We have a **cache miss** when the address requested from the level above is not present in the cache and must be fetched from the level below.
- A key performance metric for the memory system is the **hit rate**, i.e., the percentage of memory accesses that hit in the cache.
- The **miss rate** is defined as *1-hit rate* and indicates the percentage of memory accesses that miss in the cache.
- **Hit time**: time required to fetch the data from the cache in case of a hit. Includes the time needed to determine if the access is a hit or a miss.
- **Miss time**: time required to replace data at the requested address from the lower level of the hierarchy.

Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Cache Misses

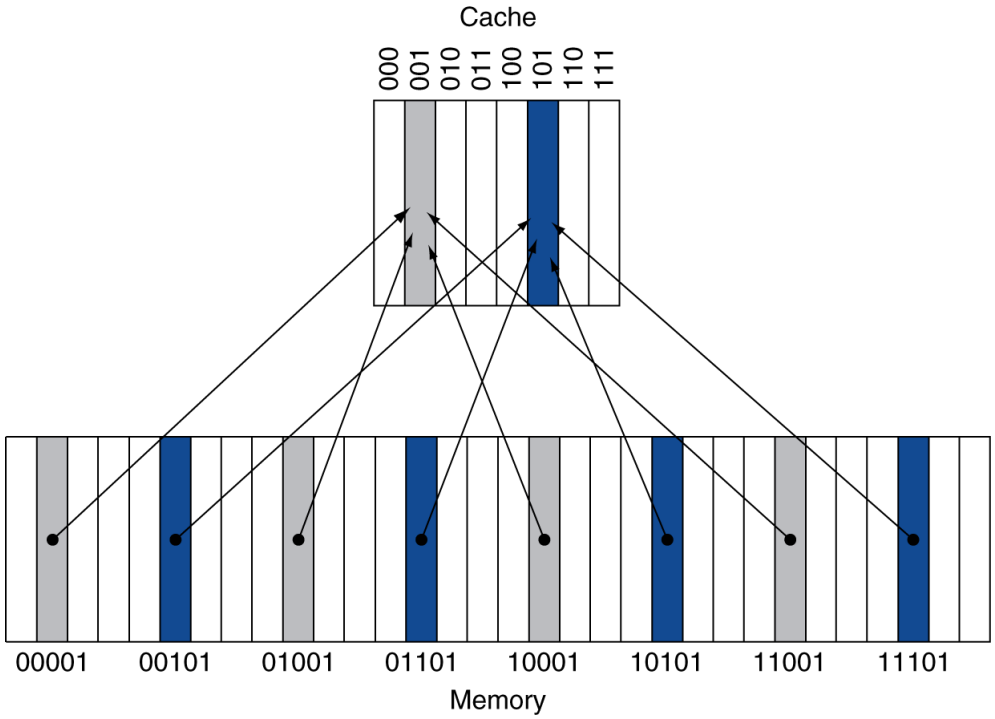
- Three types of cache miss in single-core processors:
- Compulsory (cold cache)
  - Happens when no data is present in the cache (typically at program startup)
- Conflict
  - Happens when two addresses in DRAM are mapped on the same cache line
- Capacity
  - Happens when there is no space left in the cache

Types of caches

- Direct-mapped caches
- Associative caches
- Which pros and cons?

Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)



- #Blocks is a power of 2
- Use low-order address bits

Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the tag
- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

Direct Mapped Cache

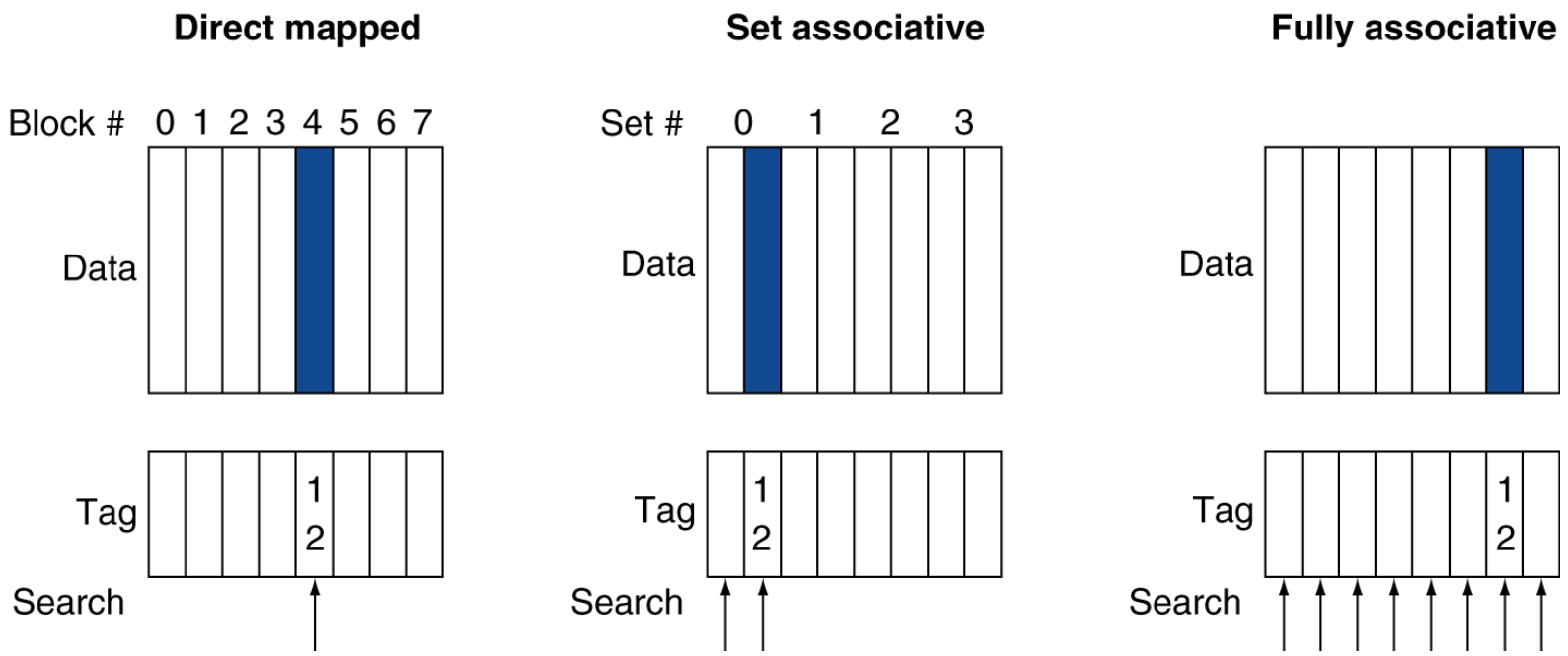
- PROs
- Very simple addressing scheme
- Very fast hit time

- CONs
- Subject to conflict misses
- Non-optimal use of cache space

Associative Caches

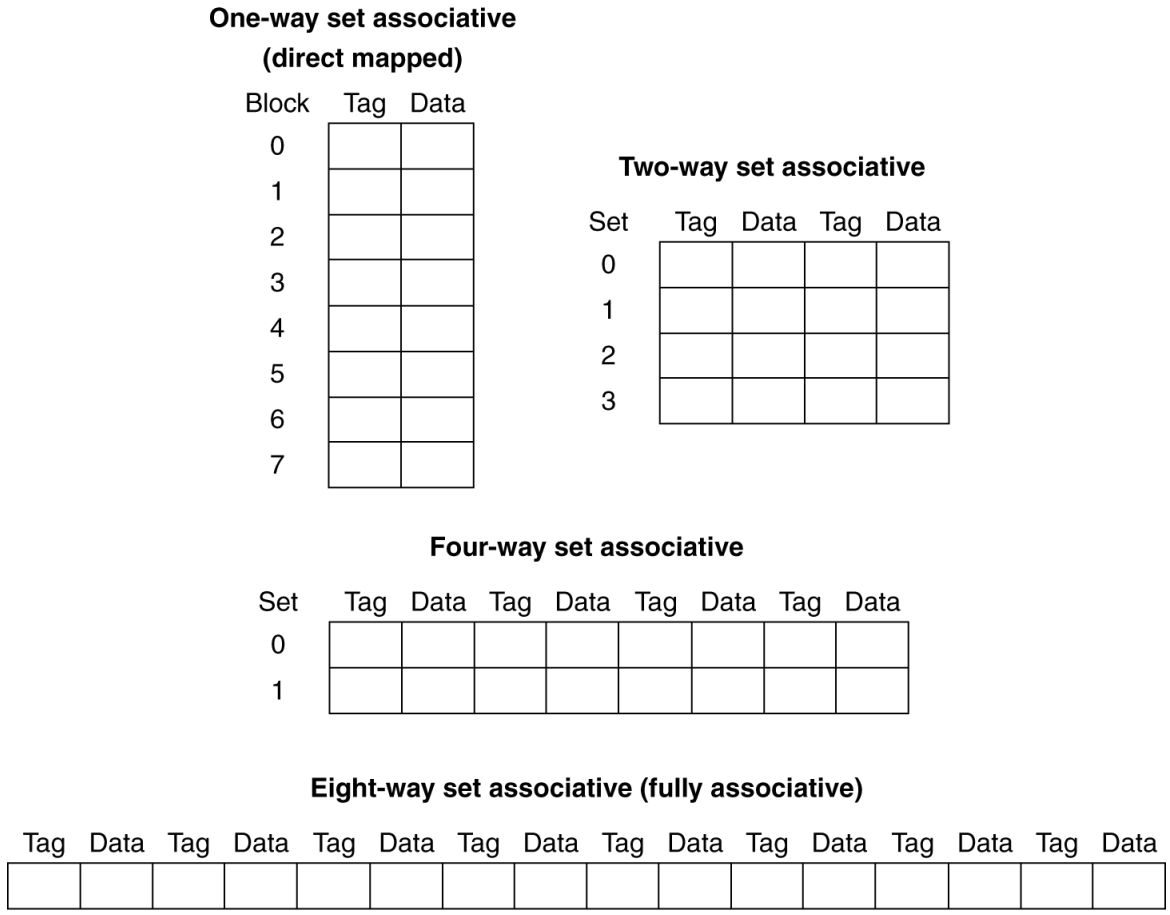
- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

Associative Cache Example



Spectrum of Associativity

- For a cache with 8 entries



Associative Caches

- PROs
- Better use of cache space
- Less prone to conflict miss

- CONs
- More expensive addressing scheme

Exploiting spatial locality

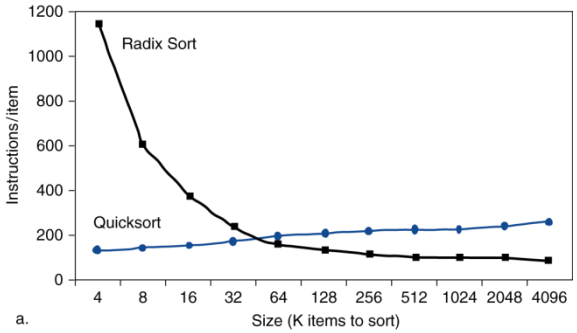
- To exploit spatial locality cache blocks (or lines) must be larger than a single word.
  - Upon a *miss* several adjacent words will be loaded in the cache line, which have a high probability of being requested in the future.

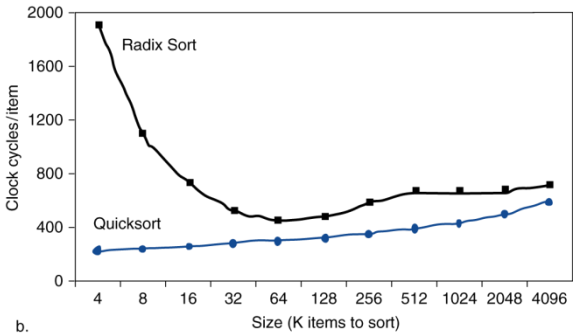
Block Size Considerations

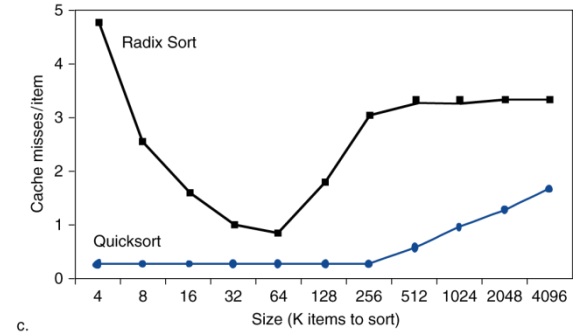
- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks $\Rightarrow$ fewer of them
    - More competition $\Rightarrow$ increased miss rate
  - Larger blocks $\Rightarrow$ pollution
- Larger miss penalty
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

Interactions with Software

- Misses depend on memory access patterns
  - Algorithm behavior
  - Compiler optimization for memory access







Cache summary

- **Cache hit:**
  - in-cache memory access—cheap
- **Cache miss:**
  - non-cached memory access—expensive
  - need to access next, slower level of hierarchy
- **Cache line size:**
  - # of bytes loaded together in one entry
  - typically a few machine words per entry
- **Capacity:**
  - amount of data that can be simultaneously in cache
- **Associativity:**
  - direct-mapped: only 1 address (line) in a given range in cache
  - n-way: n ≥ 2 lines w/ different addresses can be stored

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

Compiler optimizations for the Memory Hierarchy

Memory Hierarchy Optimizations

- Reduce ***memory latency***
  - The latency of a memory access is the time (usually in cycles) between a memory request and its completion
- Maximize ***memory bandwidth***
  - Bandwidth is the amount of useful data that can be retrieved over a time interval
- Manage overhead
  - Cost of performing optimization (e.g., copying) should be less than anticipated gain

Reuse and Locality

- Consider how data is accessed
  - ***Data reuse:***
    - Same or nearby data used multiple times
    - Intrinsic in computation
  - ***Data locality:***
    - Data is reused and is present in “fast memory”
    - Same data or same data transfer
- If a computation has reuse, what can we do to get locality?
    - Appropriate data placement and layout
    - Code reordering transformations

Optimizing Cache Performance

- Things to enhance:
  - temporal locality
  - spatial locality

- Things to minimize:
  - conflicts (i.e. bad replacement decisions)

What can the *compiler* do to help?

Two Things We Can Manipulate

- Time:
  - When is an object accessed?
- Space:
  - Where does an object exist in the address space?

How do we exploit these two levers?

Time: Reordering Computation

- What makes it difficult to know *when* an object is accessed?
- How can we predict a better time to access it?
  - What information is needed?
- How do we know that this would be safe?

Space: Changing Data Layout

- What do we know about an object's location?
  - scalars, structures, pointer-based data structures, arrays, code, etc.

- How can we tell what a better layout would be?
  - how many can we create?

- To what extent can we safely alter the layout?

Types of Objects to Consider

- Scalars
- Structures & Pointers
- Arrays

Scalars

- Locals
- Globals
- Procedure arguments

- Is cache performance a concern here?
- If so, what can be done?

```
int x;
double y;
foo(int a){
   int i;
   …
   x = a*i;
   …
}
```

Structures and Pointers

- What can we do here?
  - within a node
  - across nodes

```
struct {
    int count;
    double velocity;
    double inertia;
    struct node *neighbors[N];
} node;
```

- What limits the compiler's ability to optimize here?

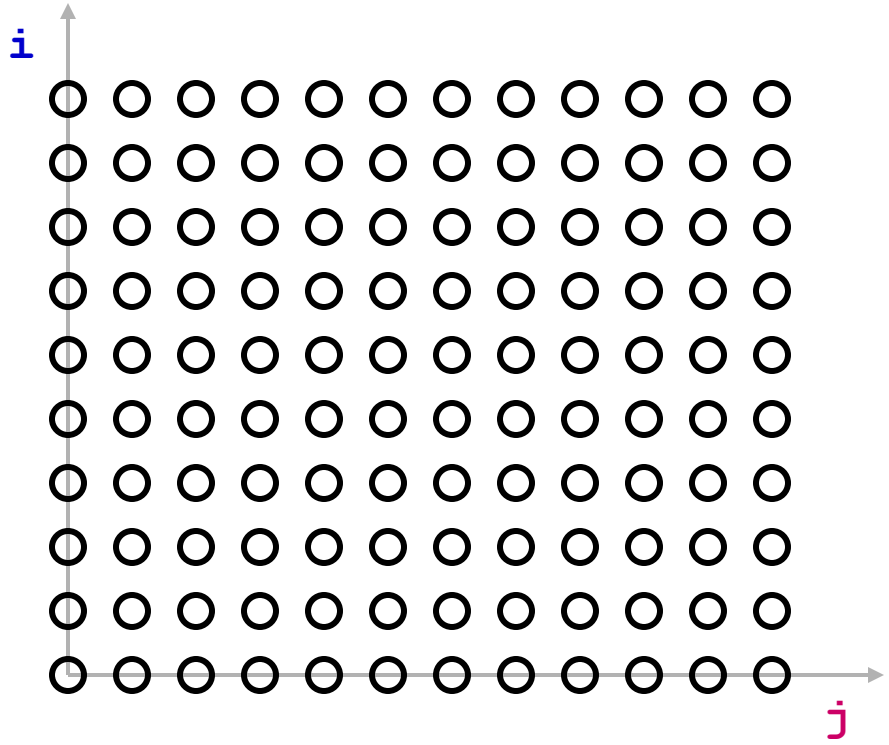
Arrays

```
double A[N][N], B[N][N];
…
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

- usually accessed within loops nests
  - makes it easy to understand “time”
- what we know about array element addresses:
  - start of array?
  - relative position within array

Iteration Space

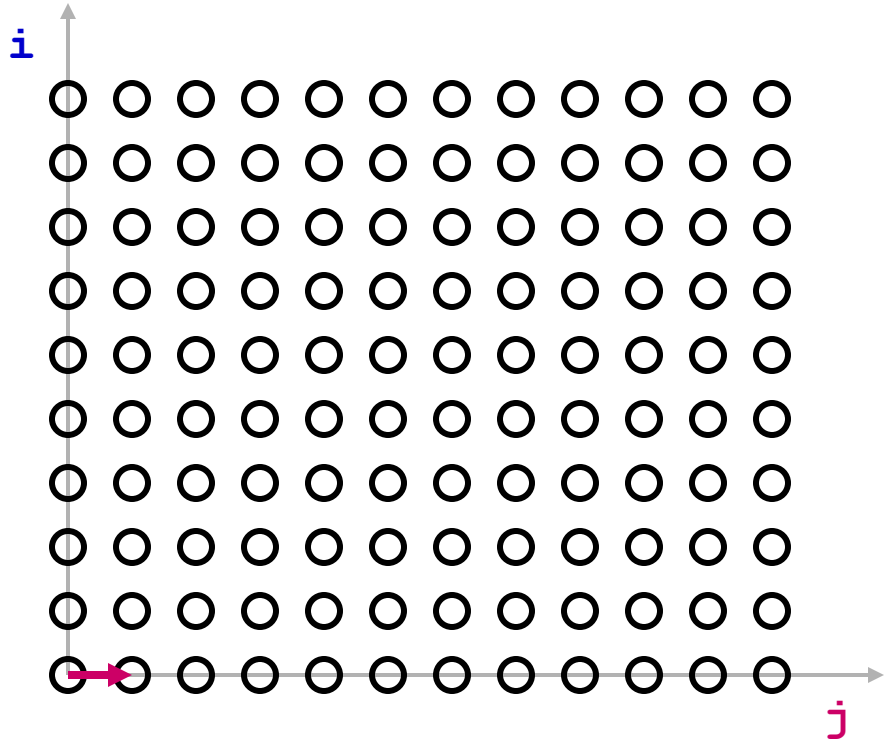
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



- each position represents an iteration

Visitation Order in Iteration Space

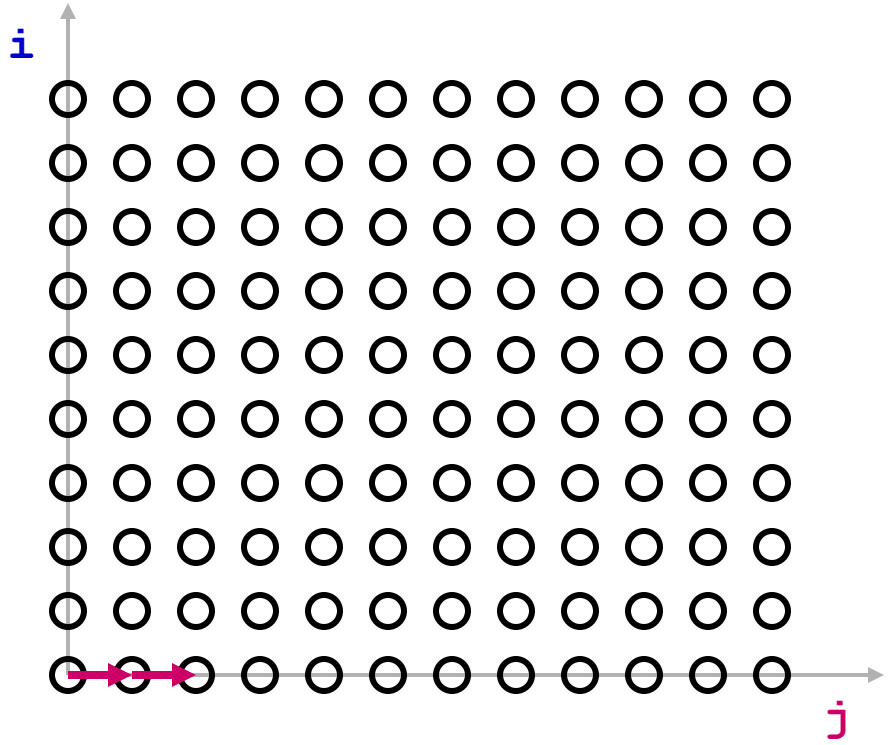
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



- each position represents an iteration

Visitation Order in Iteration Space

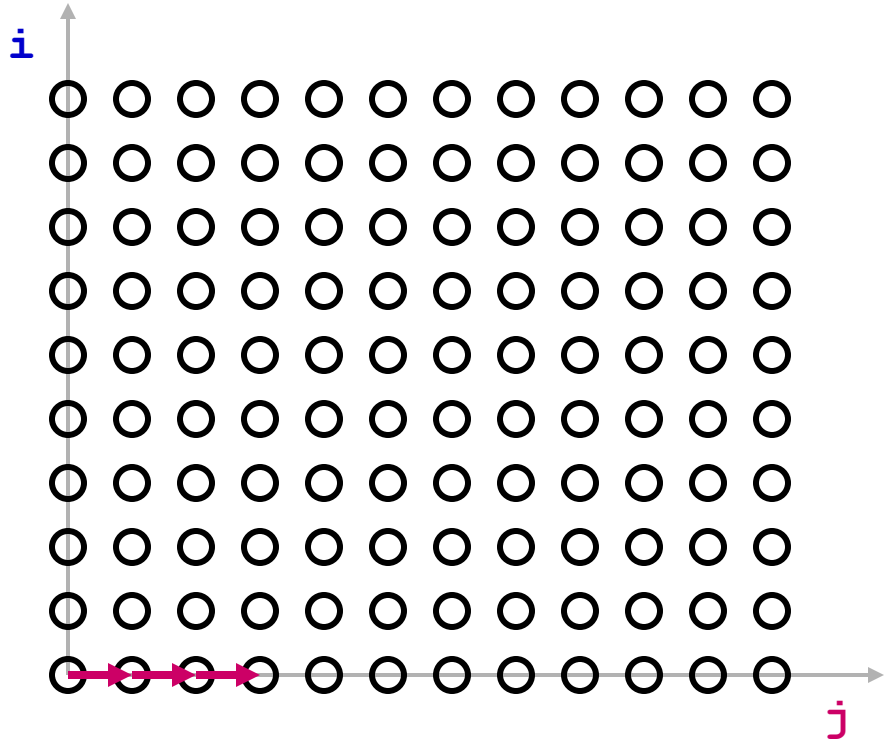
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



- each position represents an iteration

Visitation Order in Iteration Space

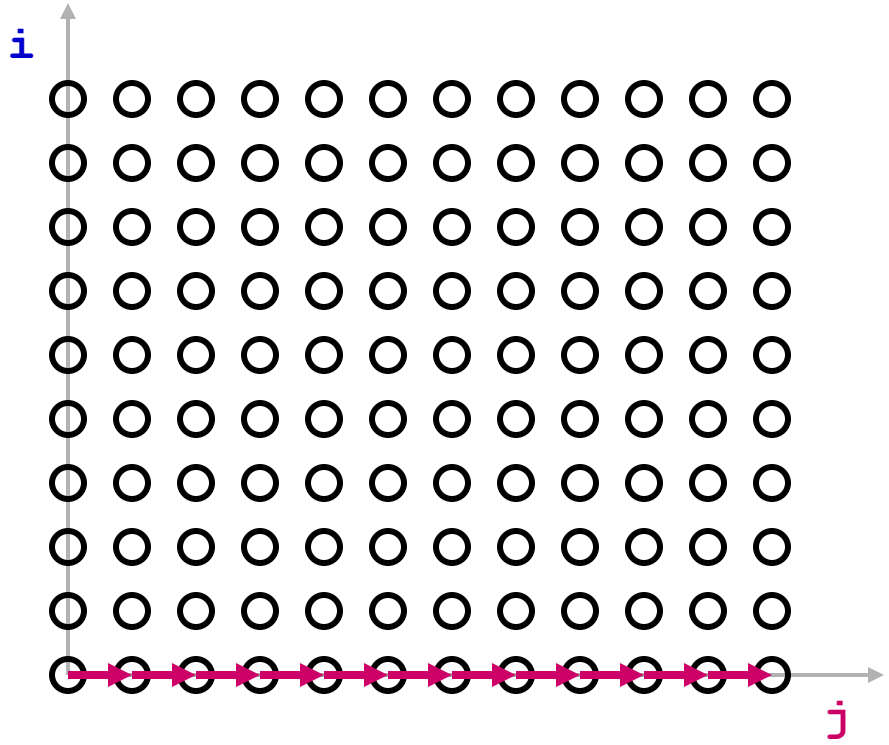
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



- each position represents an iteration

Visitation Order in Iteration Space

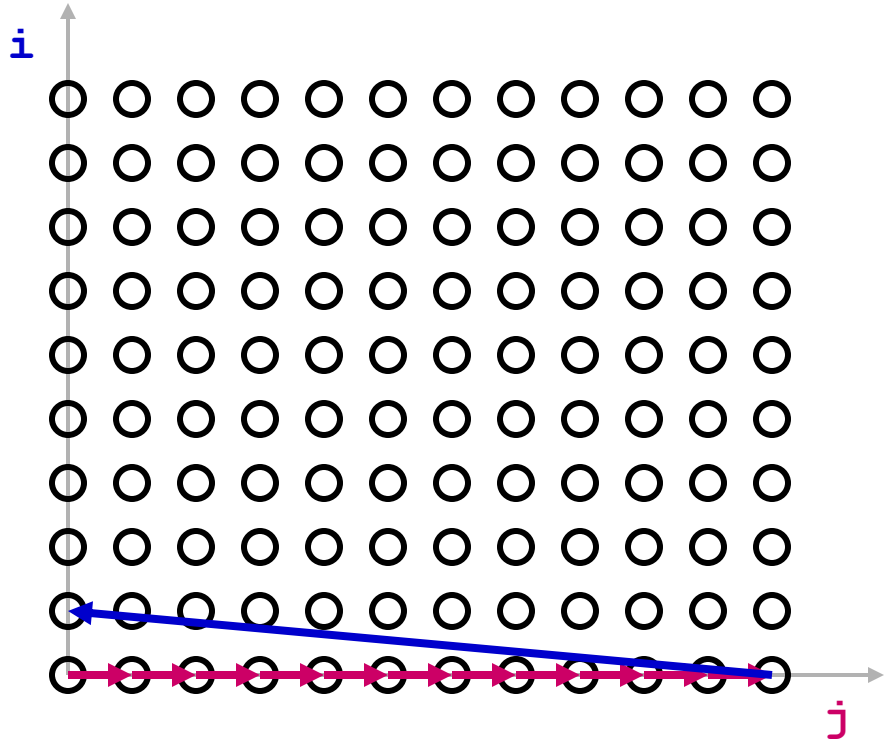
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



- each position represents an iteration

Visitation Order in Iteration Space

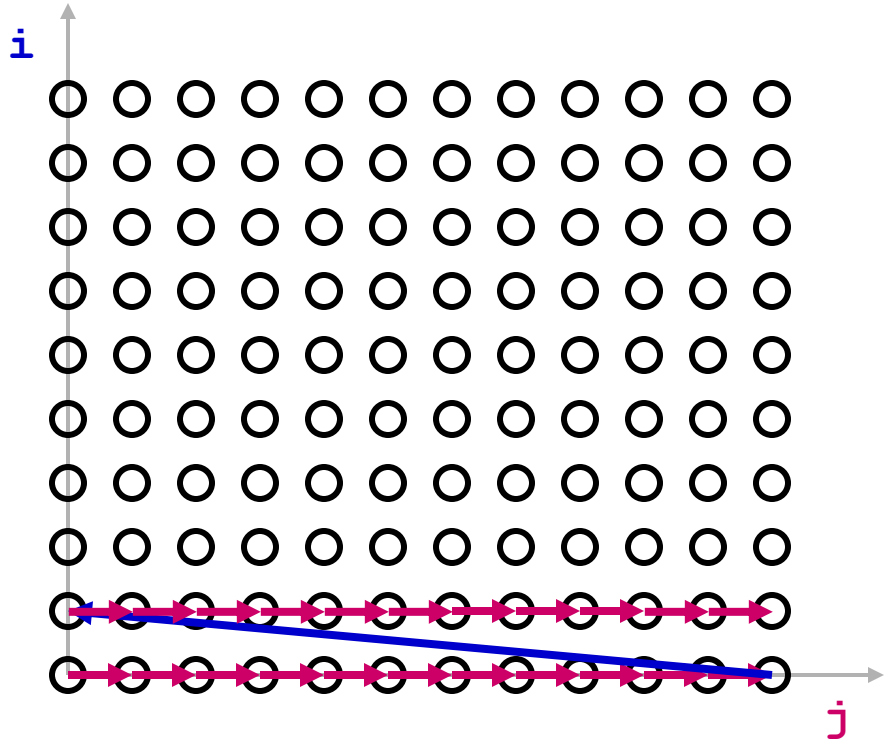
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



- each position represents an iteration

Visitation Order in Iteration Space

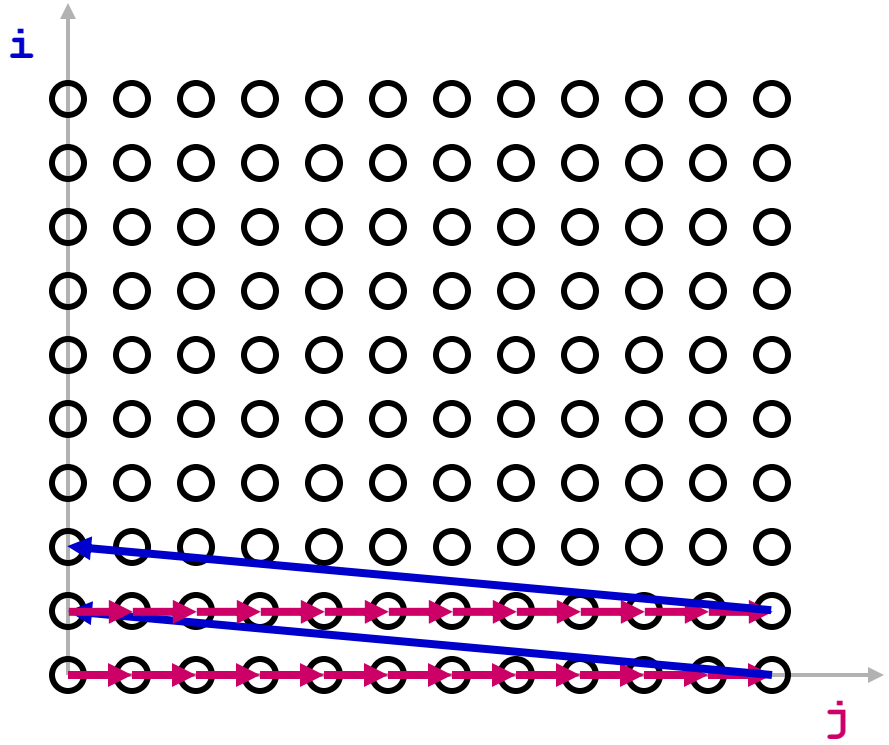
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



- each position represents an iteration

Visitation Order in Iteration Space

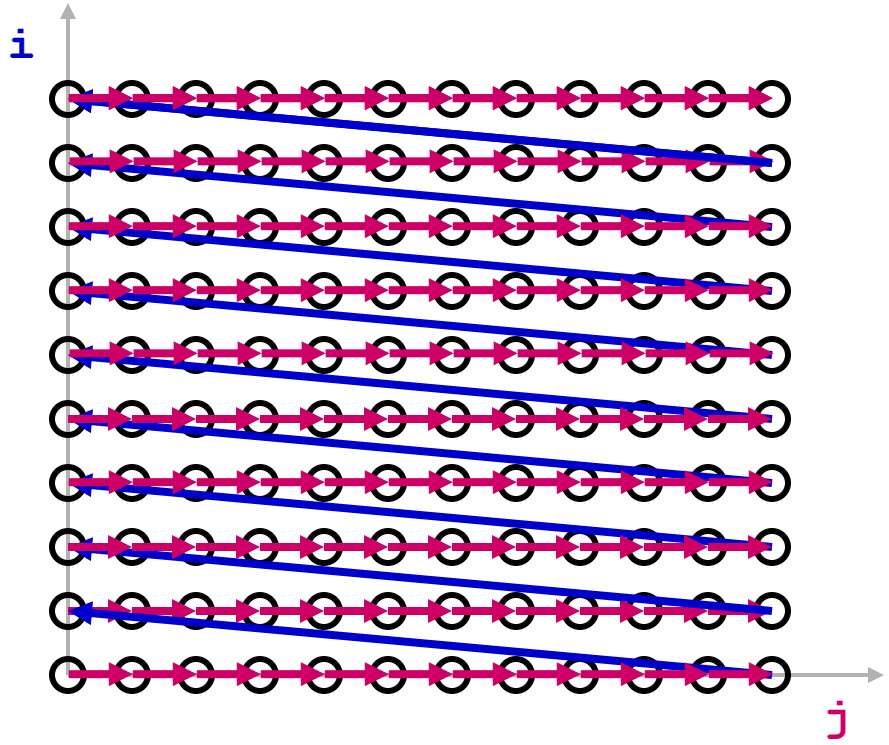
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



- each position represents an iteration

Visitation Order in Iteration Space

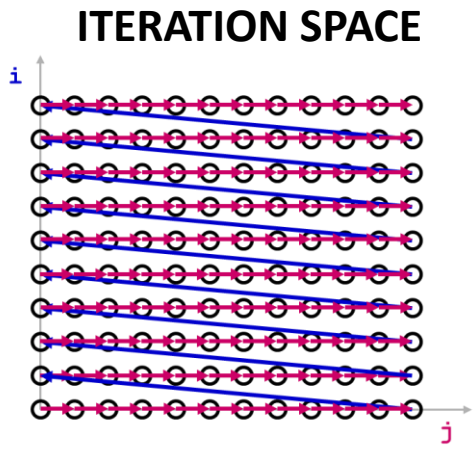
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



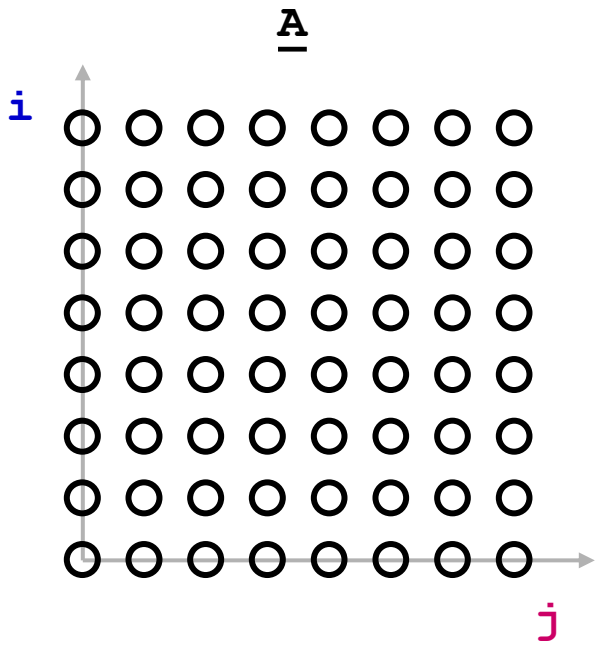
- Note: iteration space ≠ data space

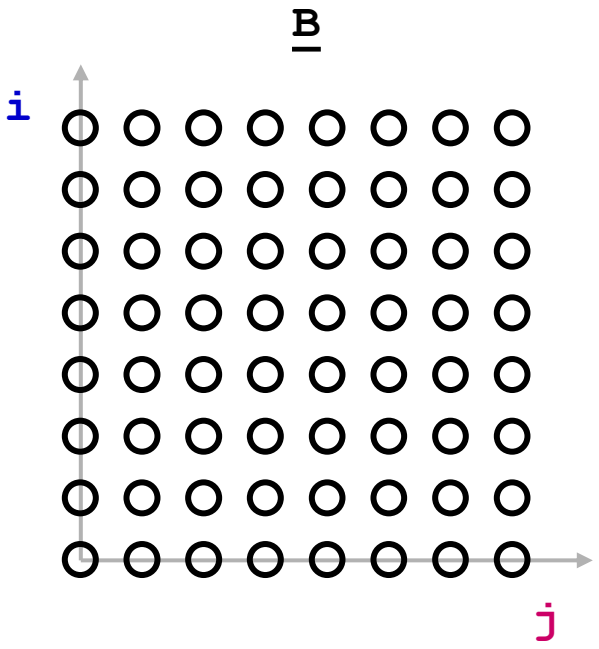
When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



DATA SPACE

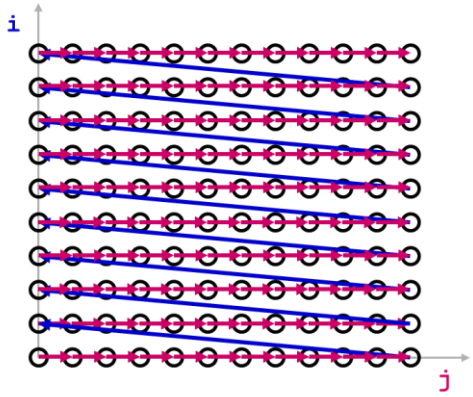




When Do Cache Misses Occur?

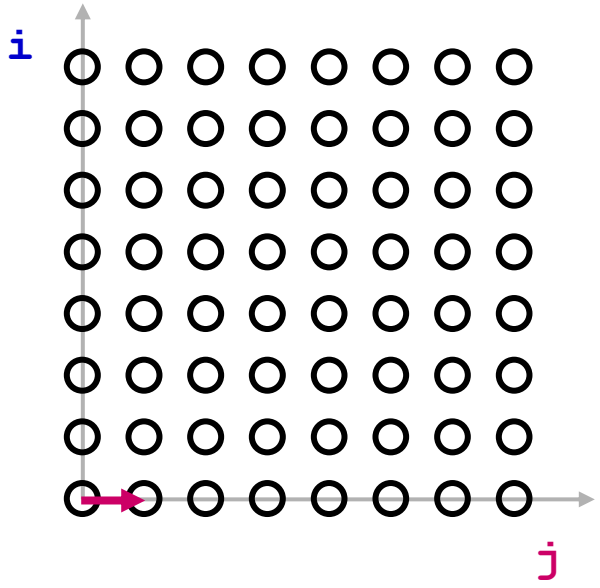
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

ITERATION SPACE

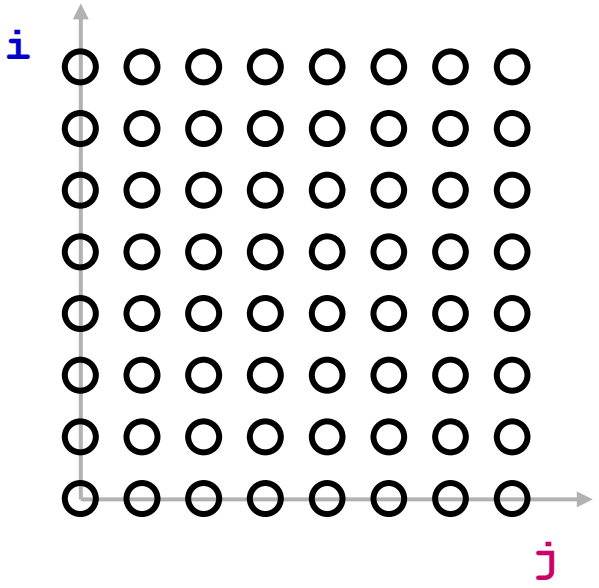


DATA SPACE

A

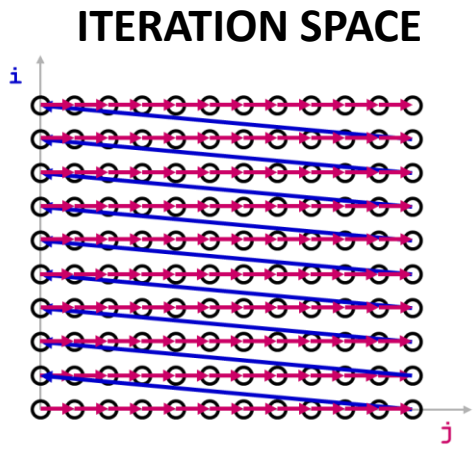


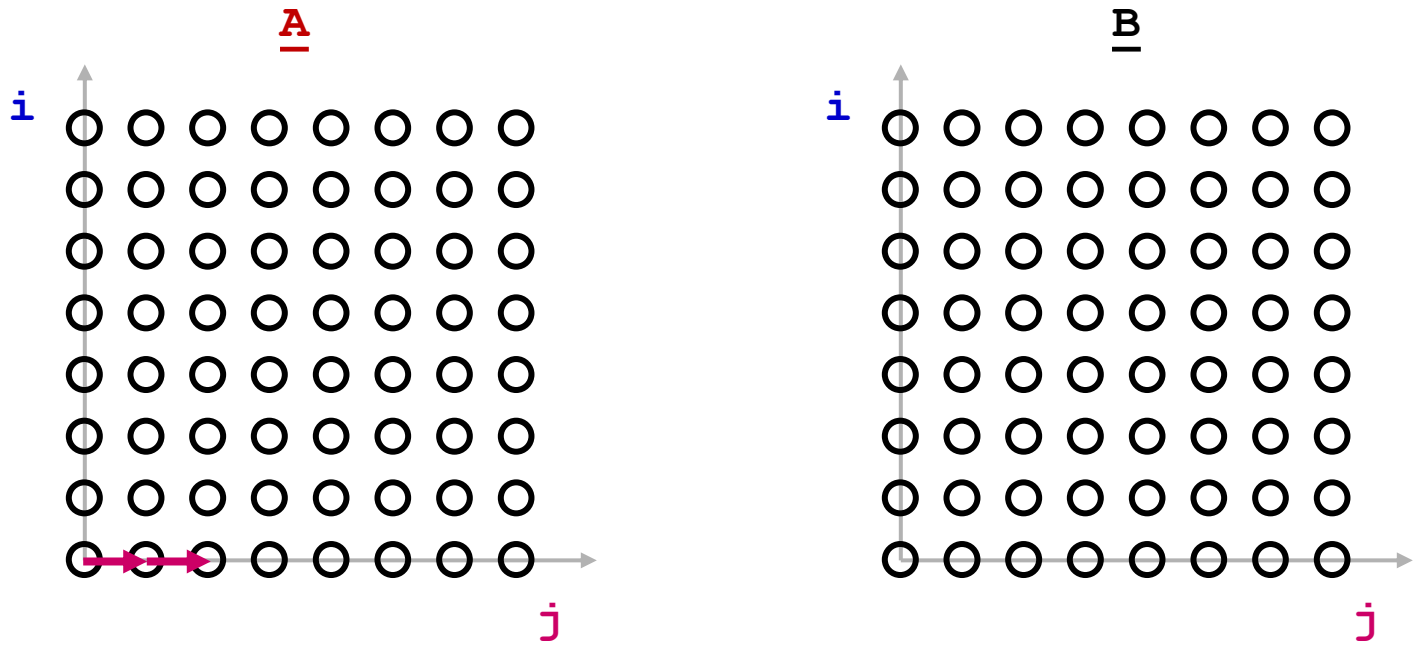
B



When Do Cache Misses Occur?

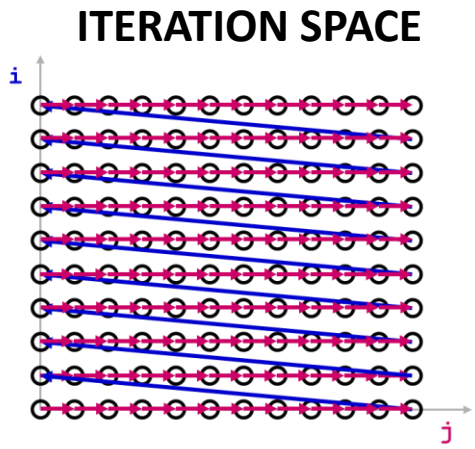
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

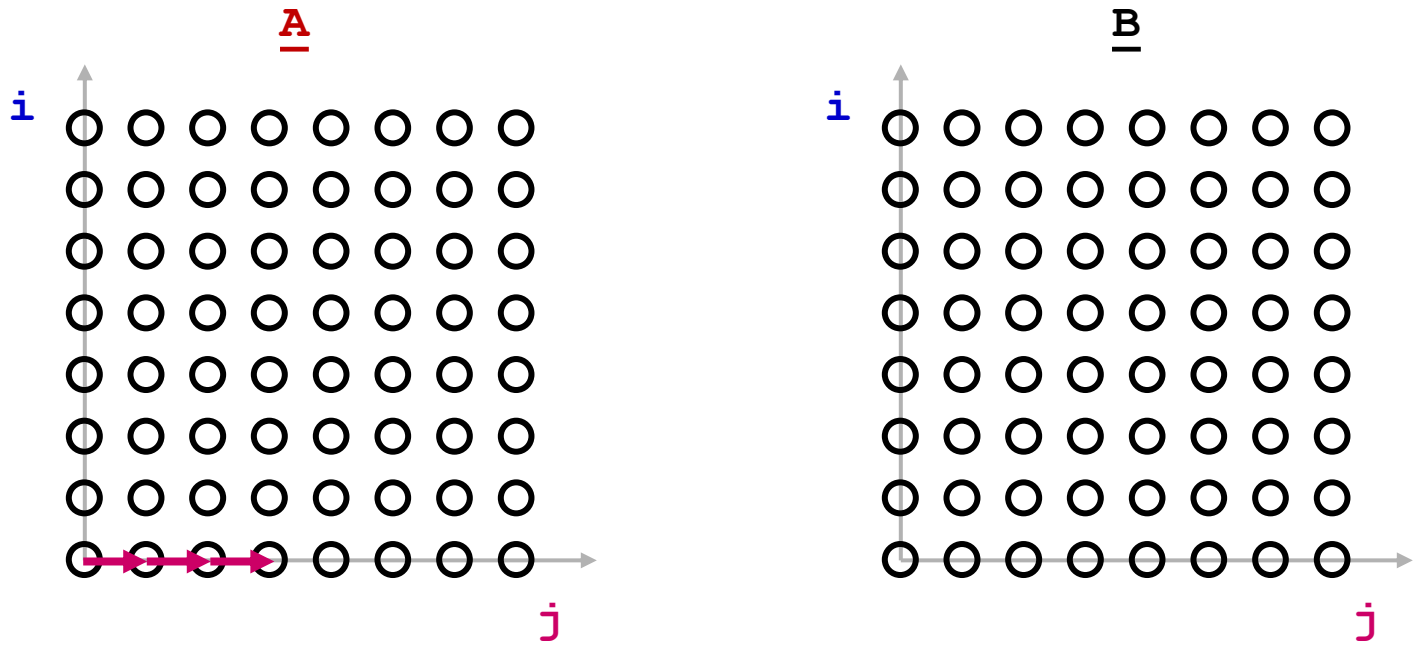




When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

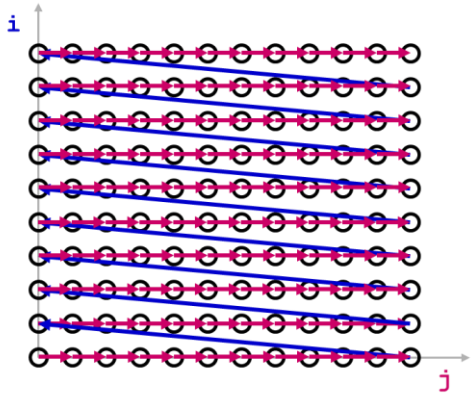




When Do Cache Misses Occur?

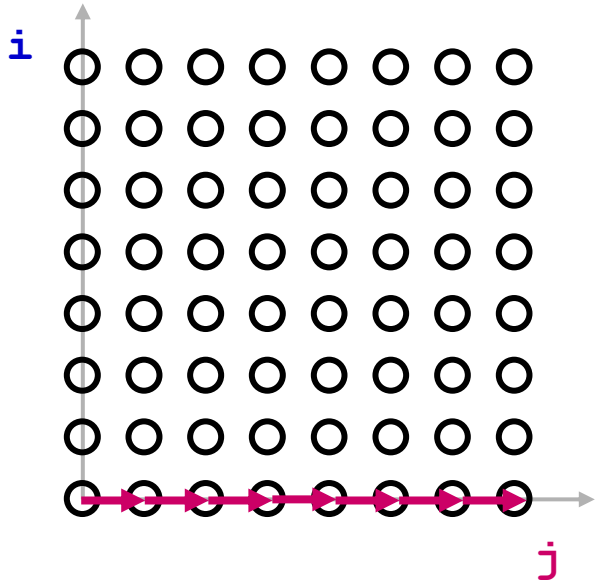
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

ITERATION SPACE

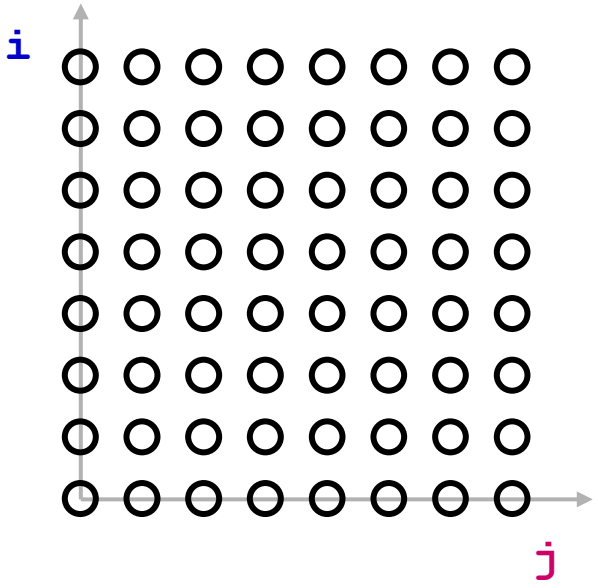


DATA SPACE

A



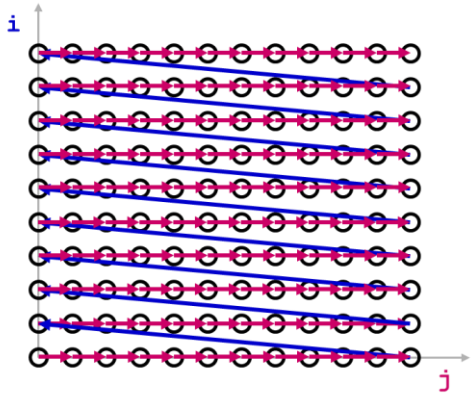
B



When Do Cache Misses Occur?

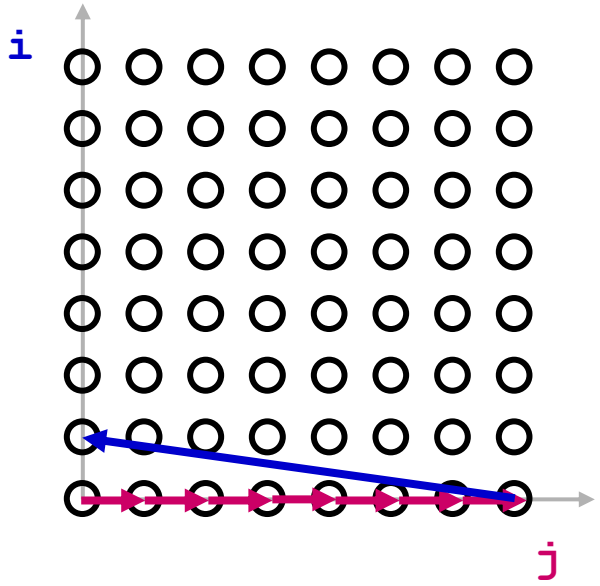
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

ITERATION SPACE

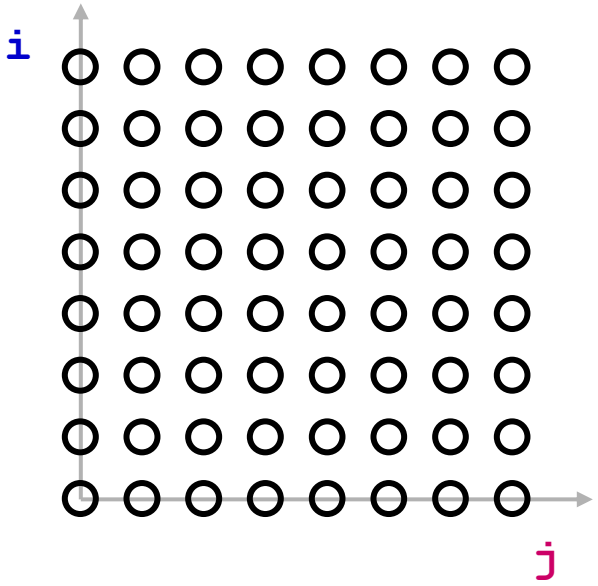


DATA SPACE

A



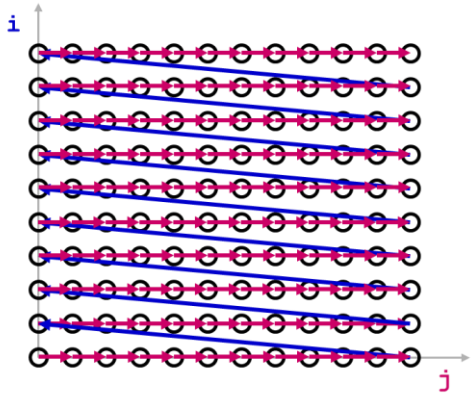
B



When Do Cache Misses Occur?

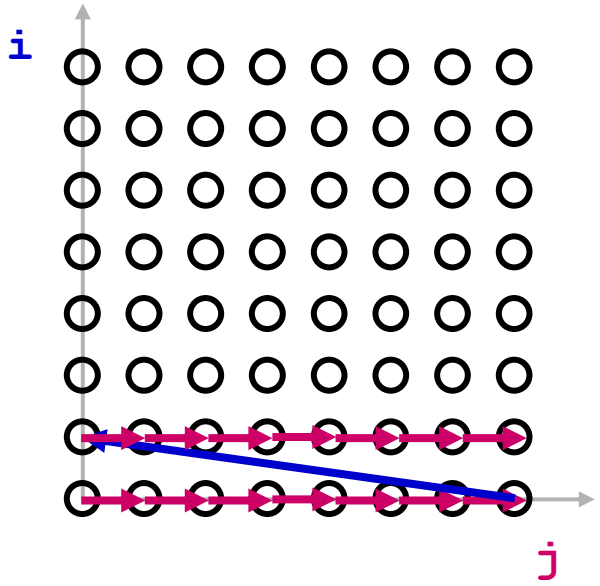
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

ITERATION SPACE

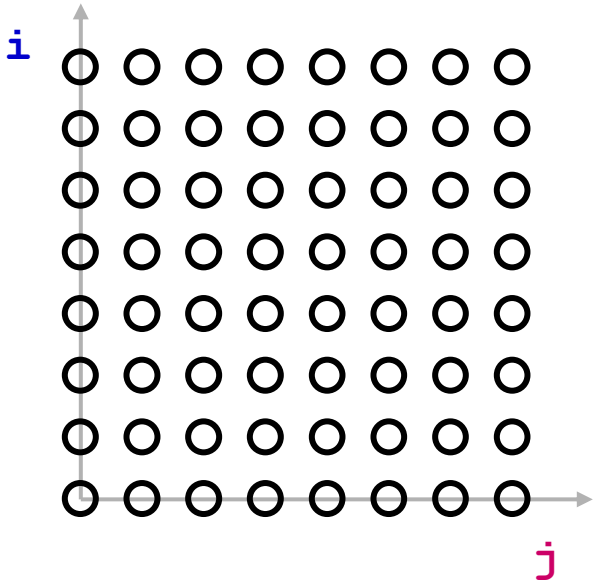


DATA SPACE

A



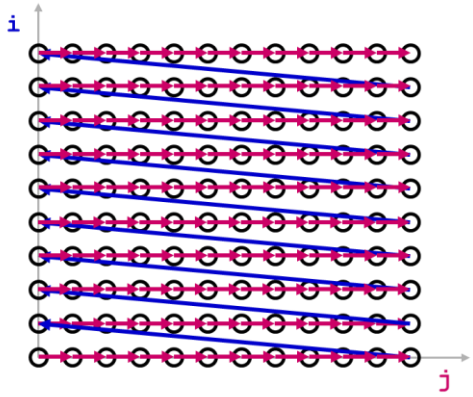
B



When Do Cache Misses Occur?

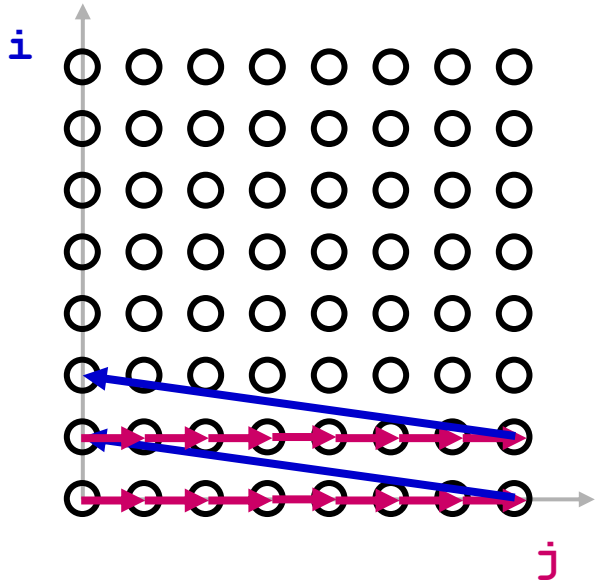
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

ITERATION SPACE

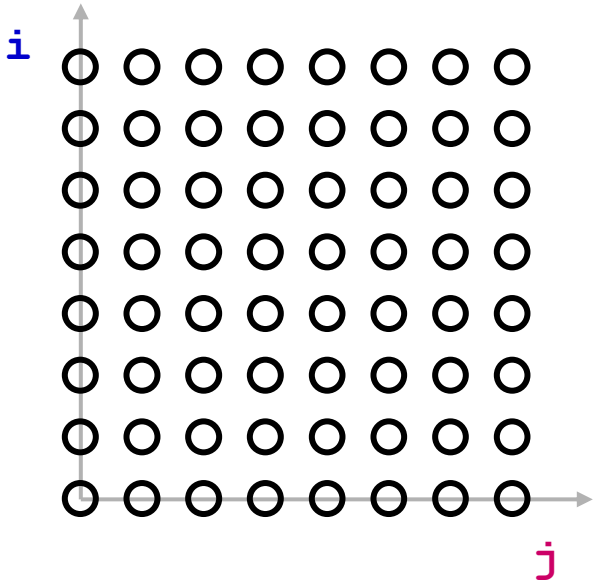


DATA SPACE

A



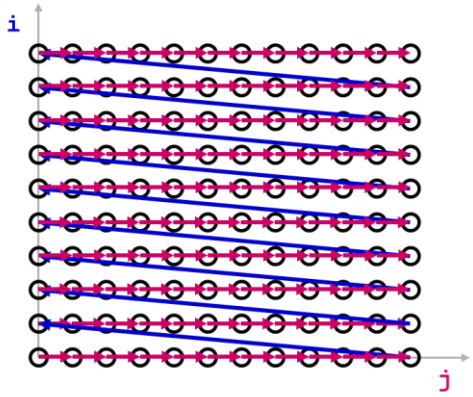
B



When Do Cache Misses Occur?

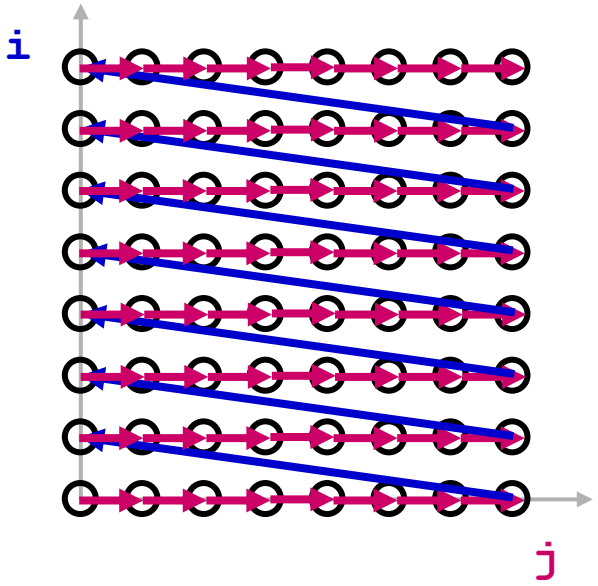
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

ITERATION SPACE

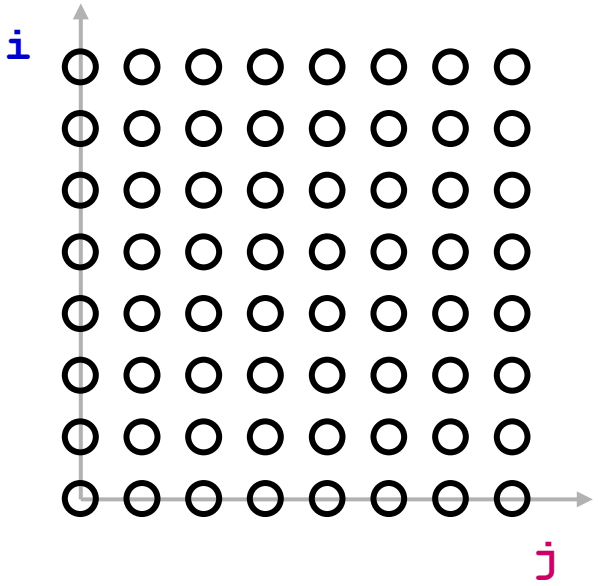


DATA SPACE

A



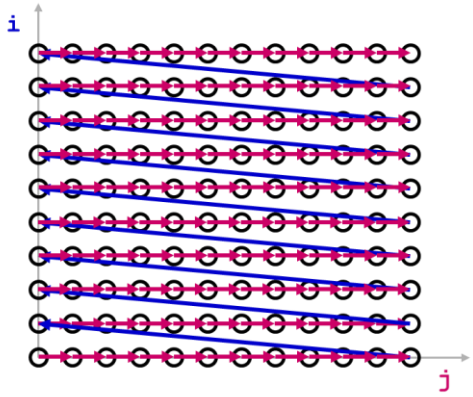
B



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

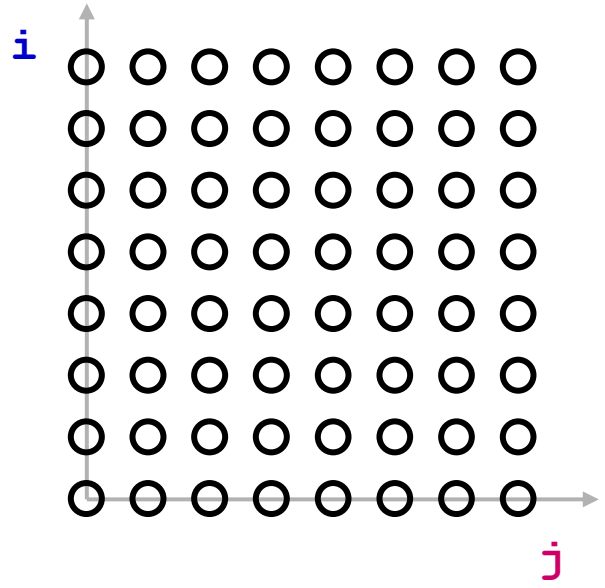
ITERATION SPACE



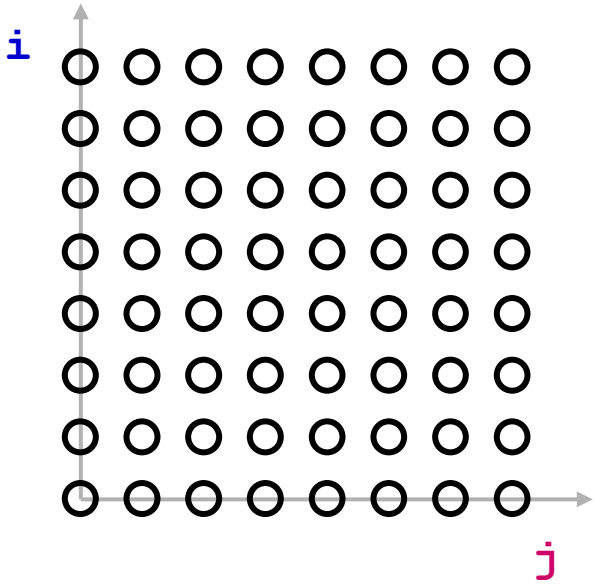
When do cache misses occur?

DATA SPACE

A



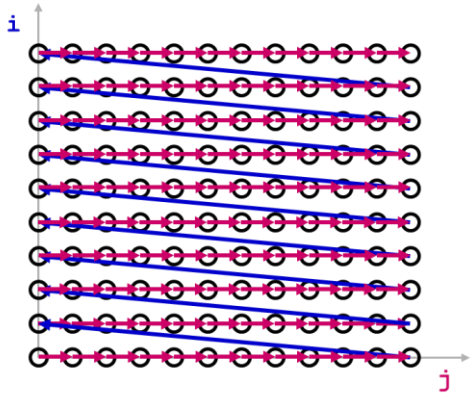
B



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

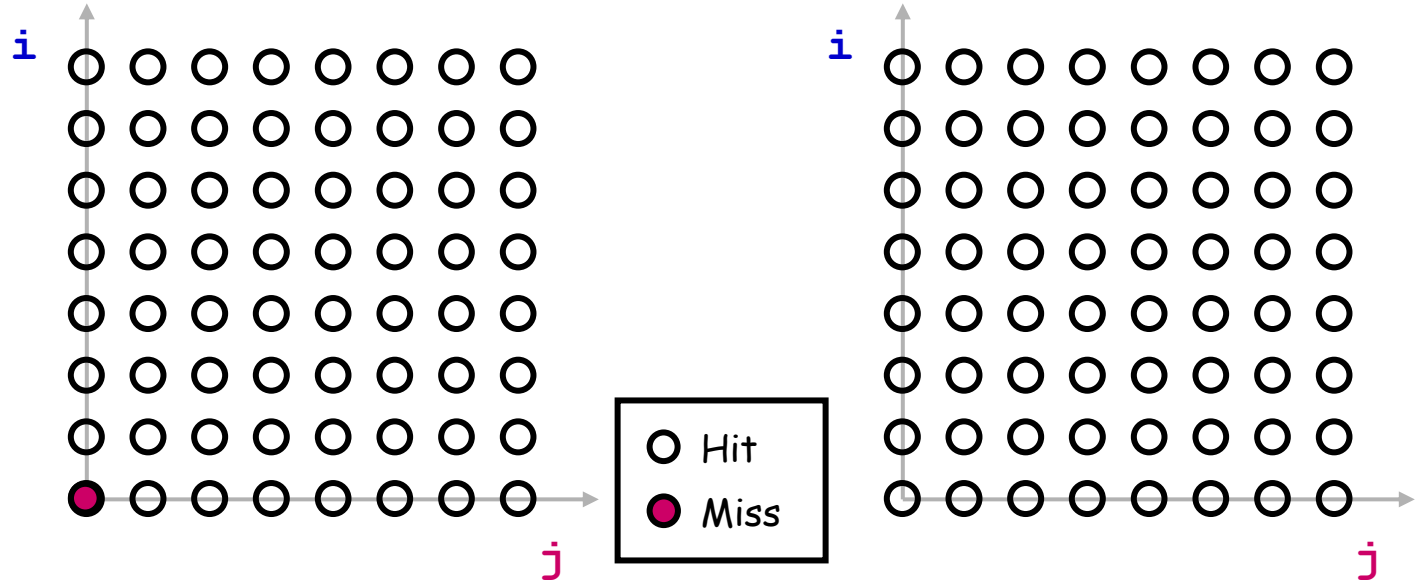
ITERATION SPACE



When do cache misses occur?

DATA SPACE

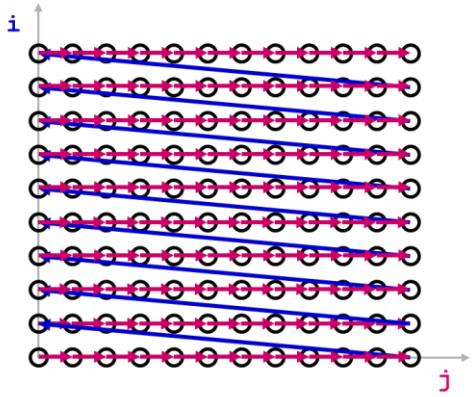
A Assuming 8 cache lines of 4 words B



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

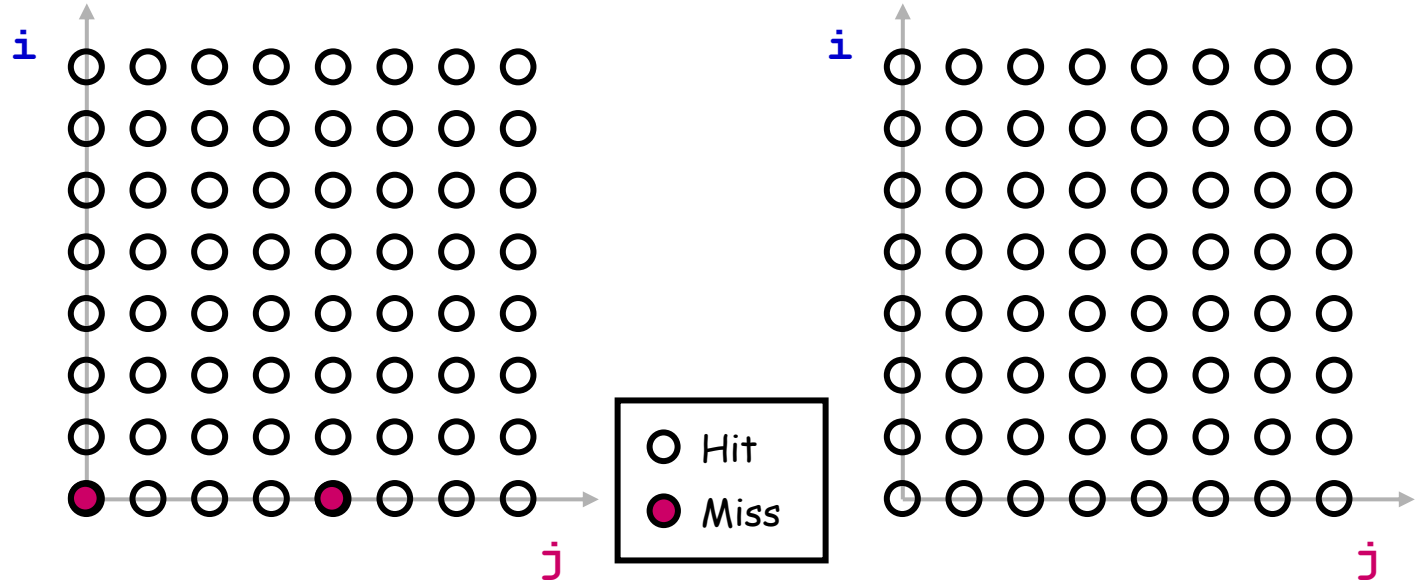
ITERATION SPACE



When do cache misses occur?

DATA SPACE

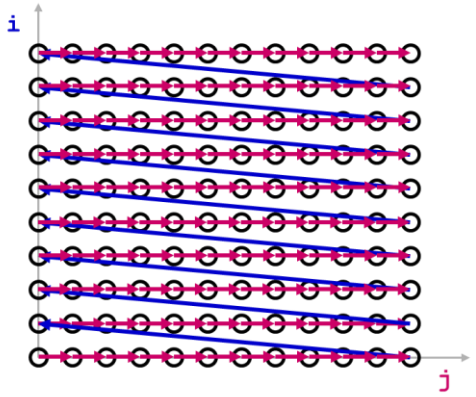
A Assuming 8 cache lines of 4 words B



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

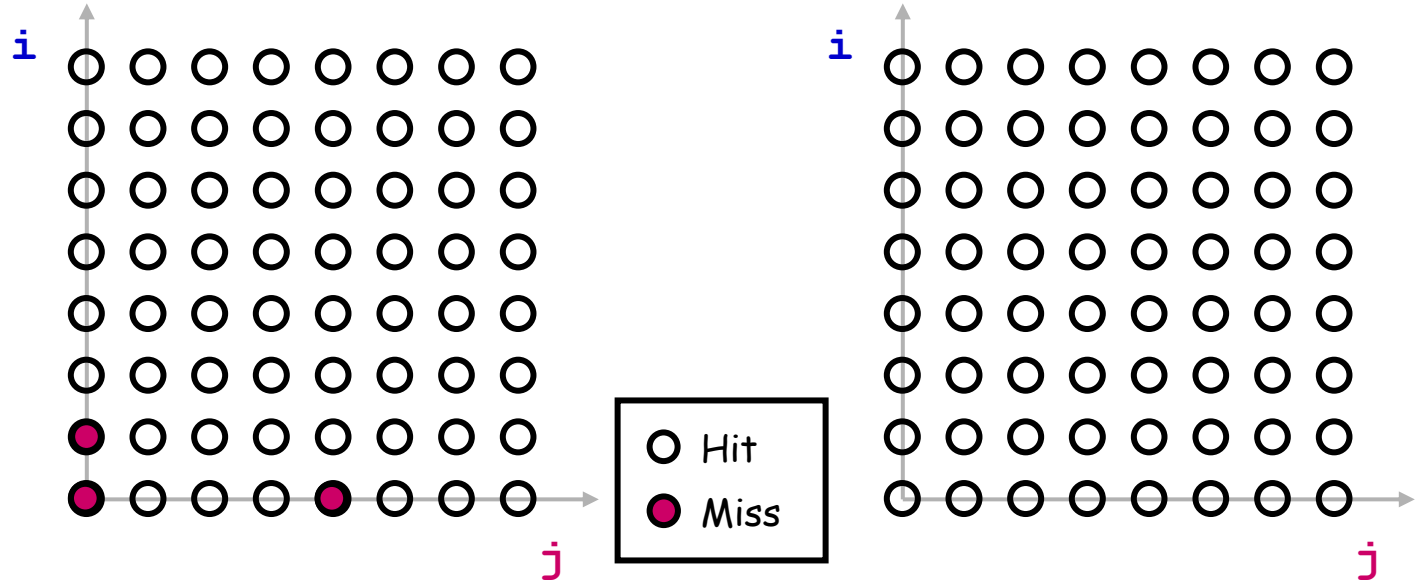
ITERATION SPACE



When do cache misses occur?

DATA SPACE

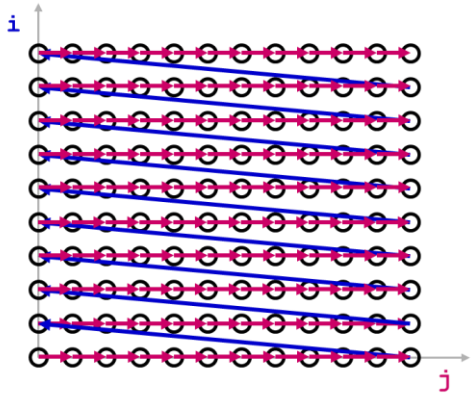
A Assuming 8 cache lines of 4 words B



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

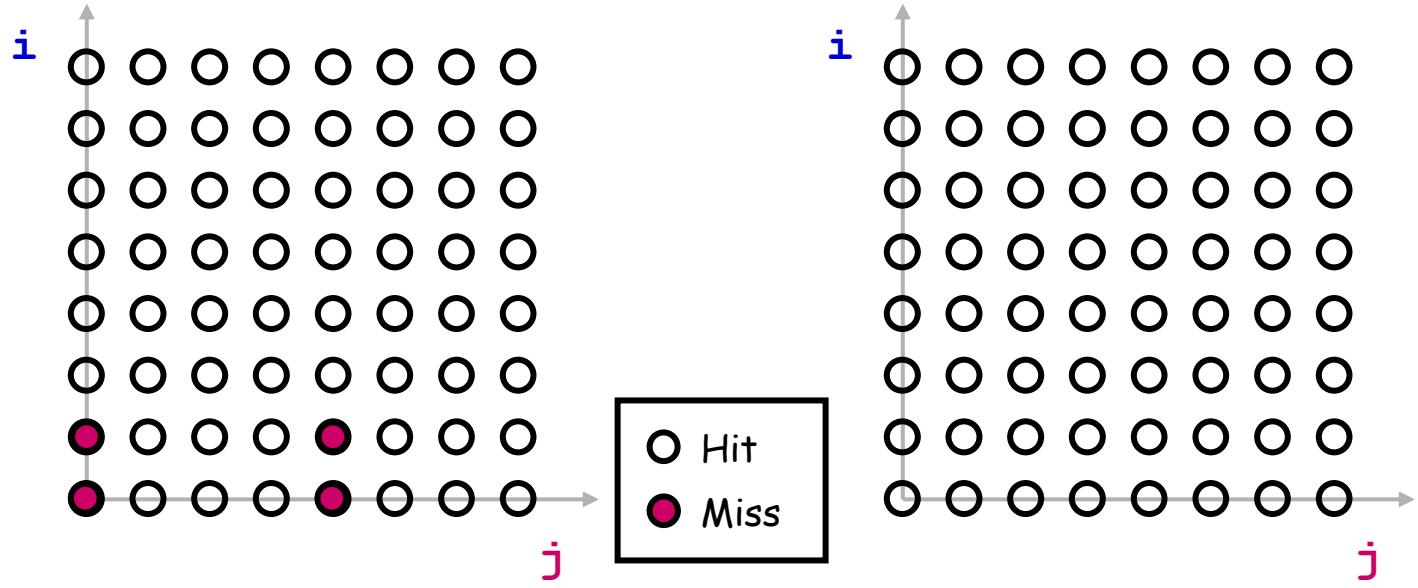
ITERATION SPACE



When do cache misses occur?

DATA SPACE

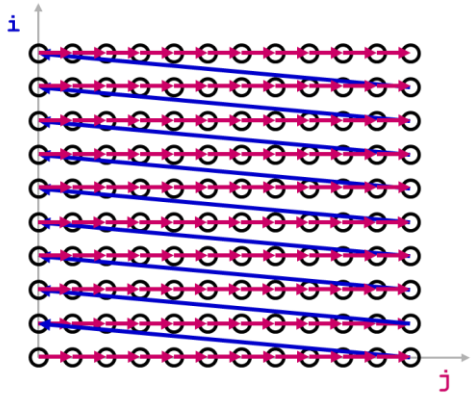
A Assuming 8 cache lines of 4 words B



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

ITERATION SPACE



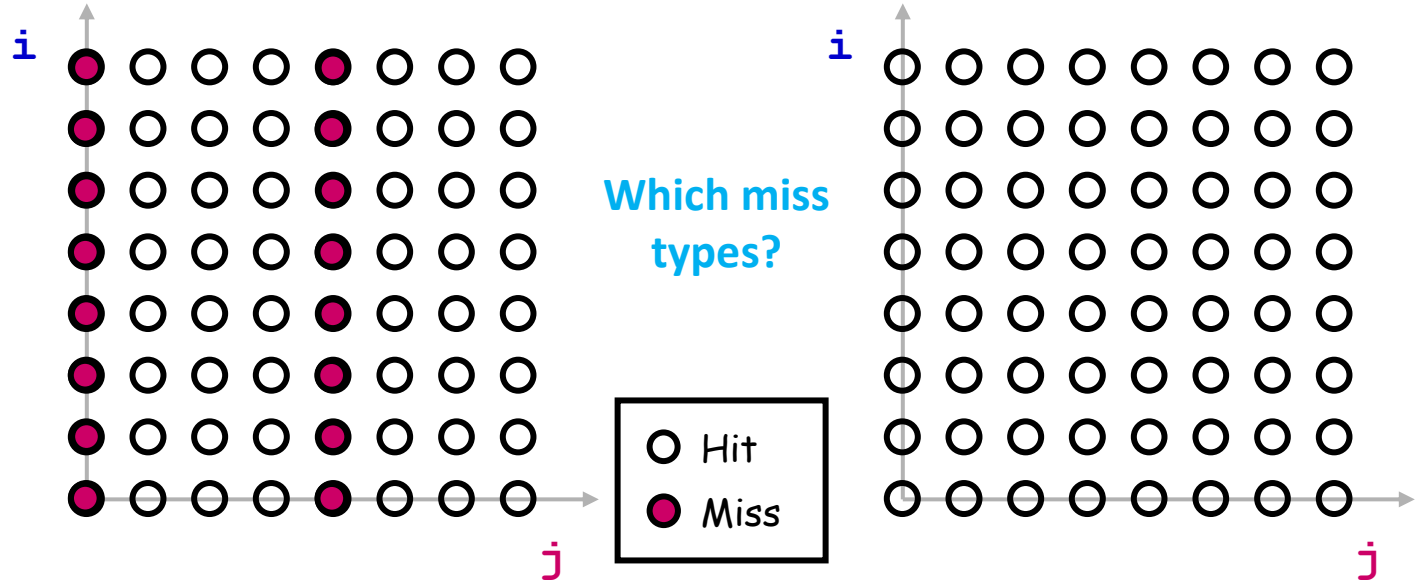
When do cache misses occur?

DATA SPACE

A

Assuming 8 cache lines of 4 words

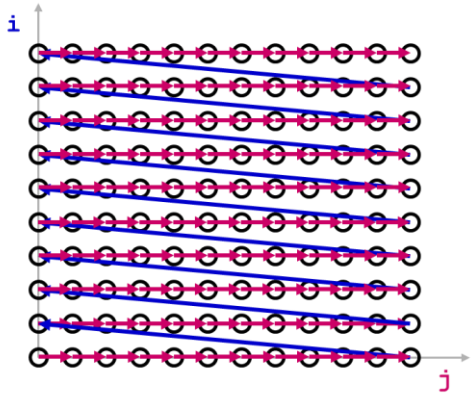
B



When Do Cache Misses Occur?

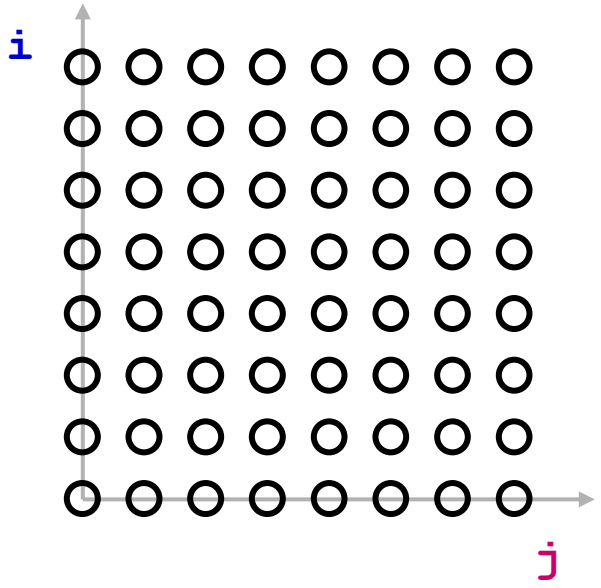
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

ITERATION SPACE

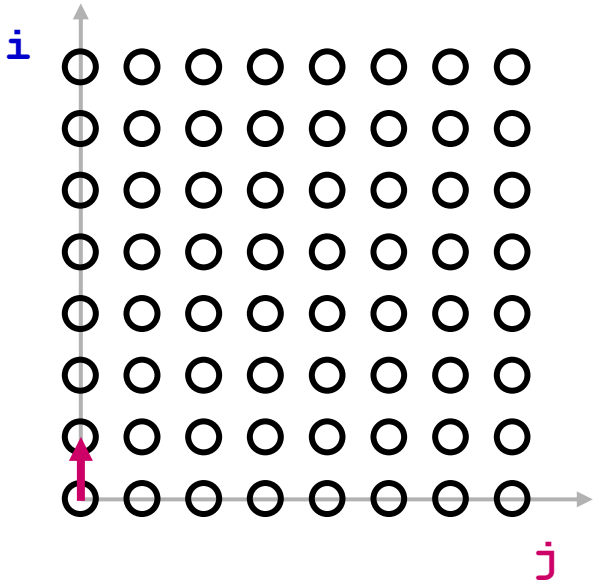


DATA SPACE

A

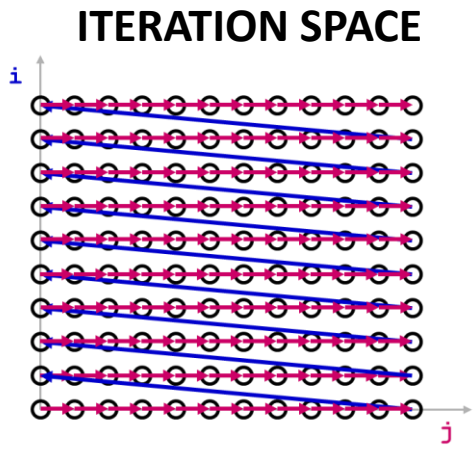


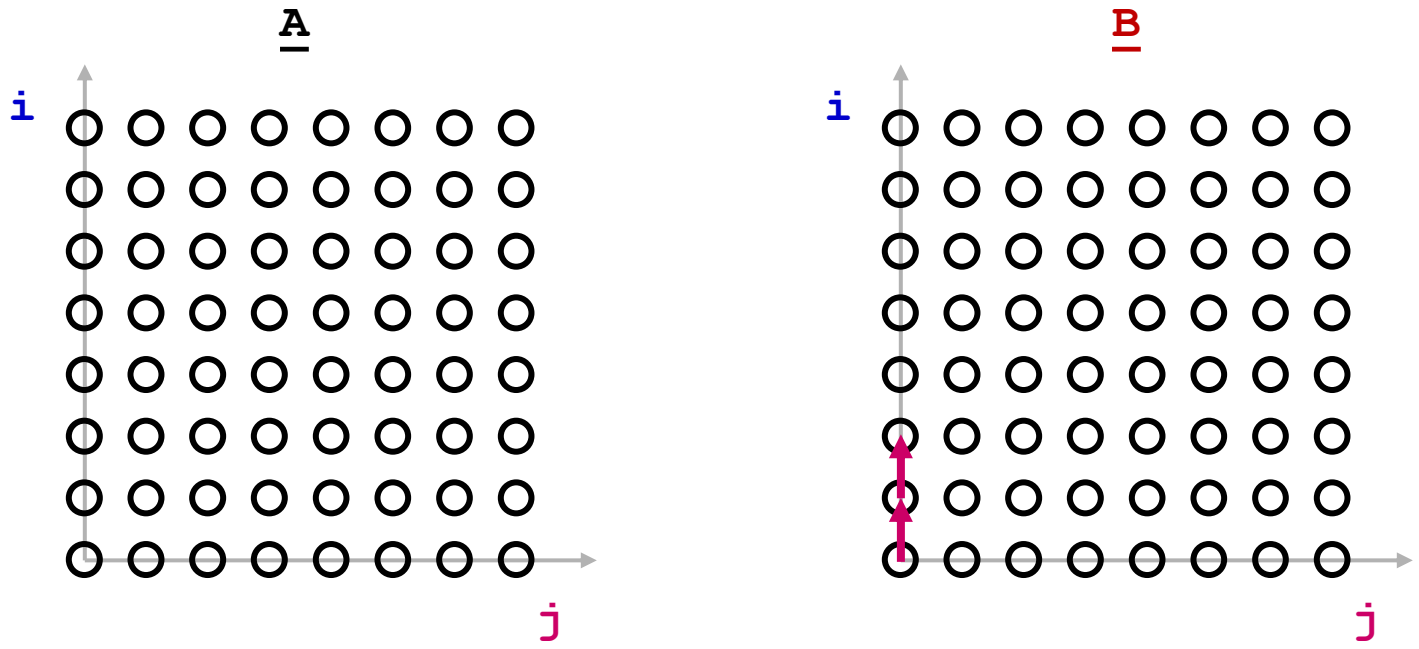
B



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

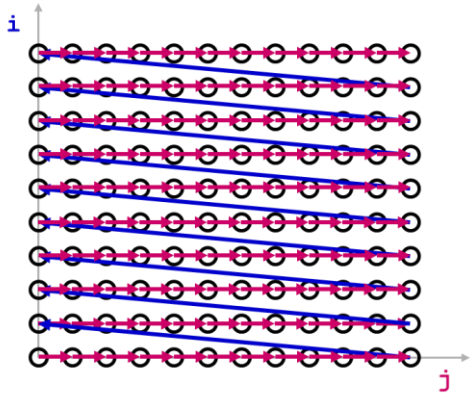




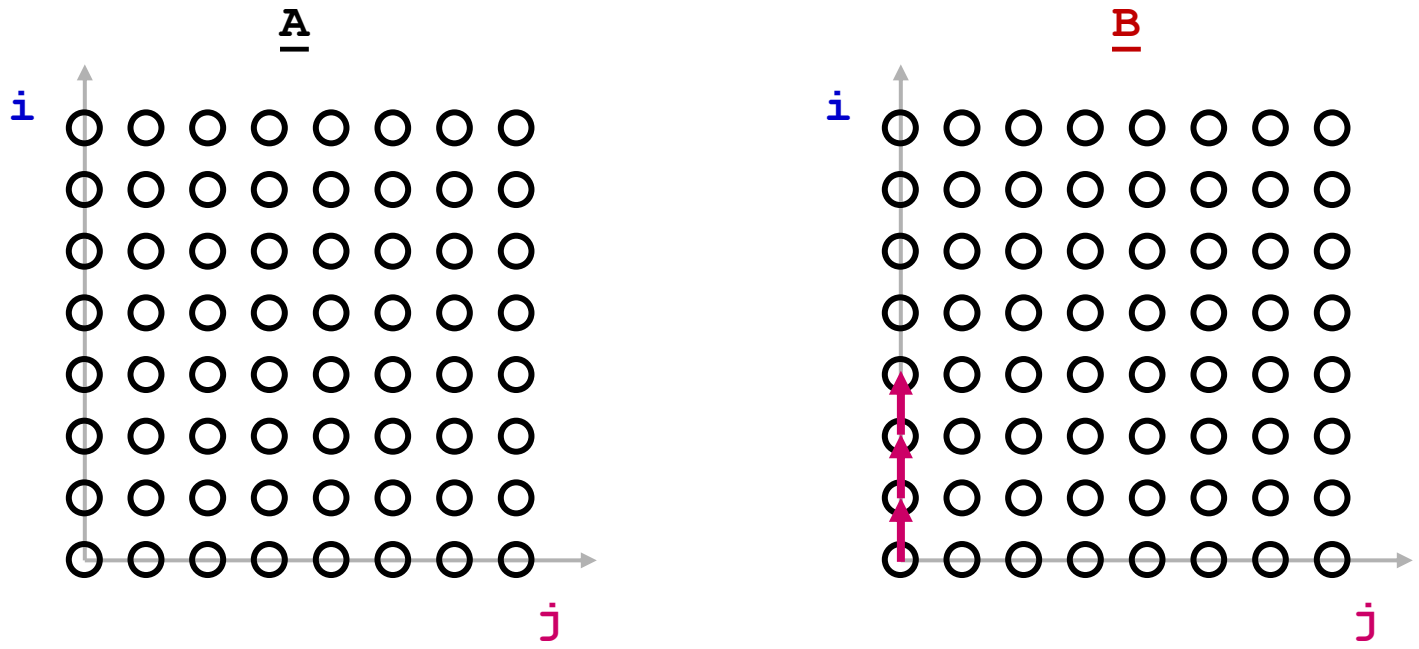
When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

ITERATION SPACE

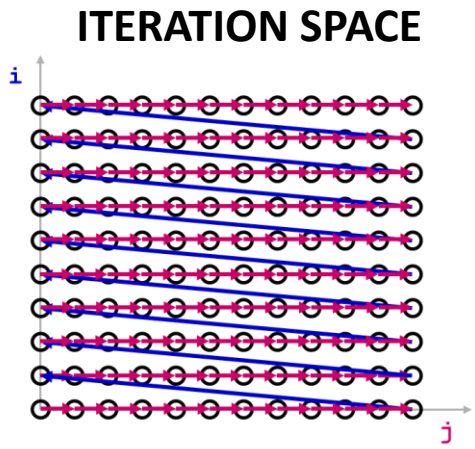


DATA SPACE

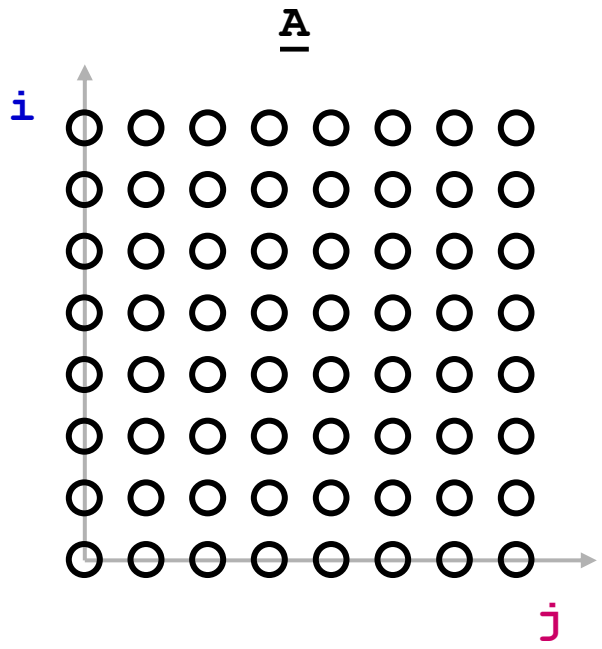


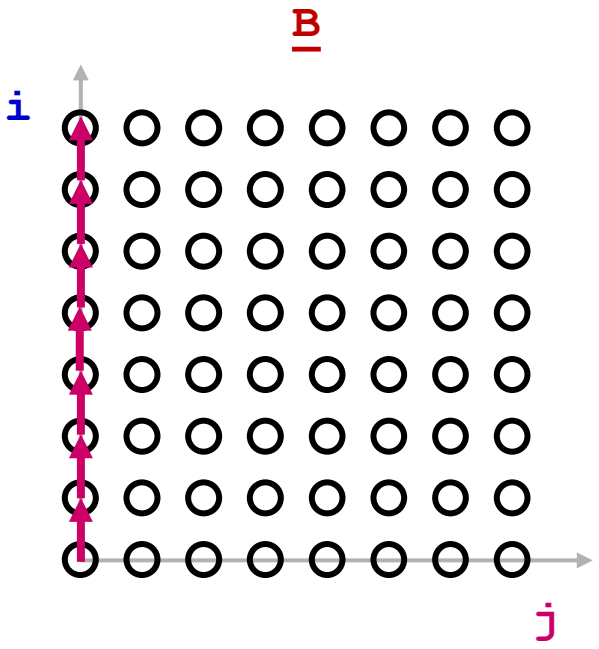
When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



DATA SPACE

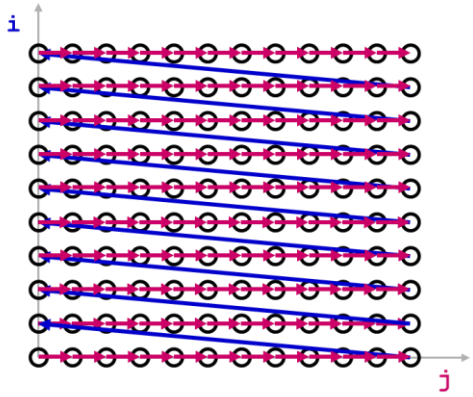




When Do Cache Misses Occur?

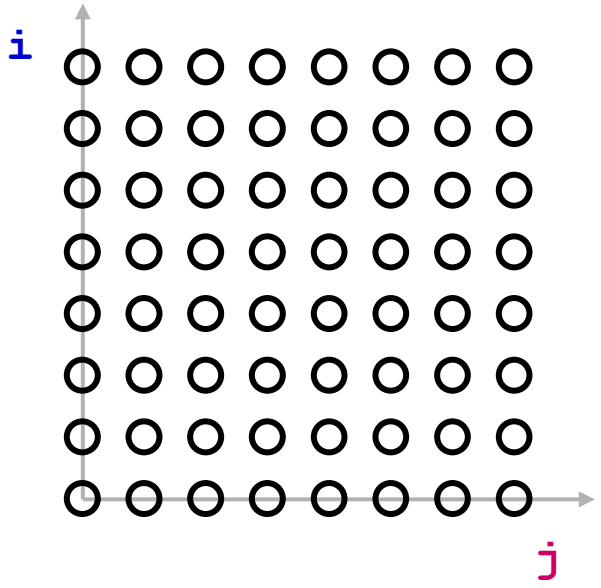
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

ITERATION SPACE

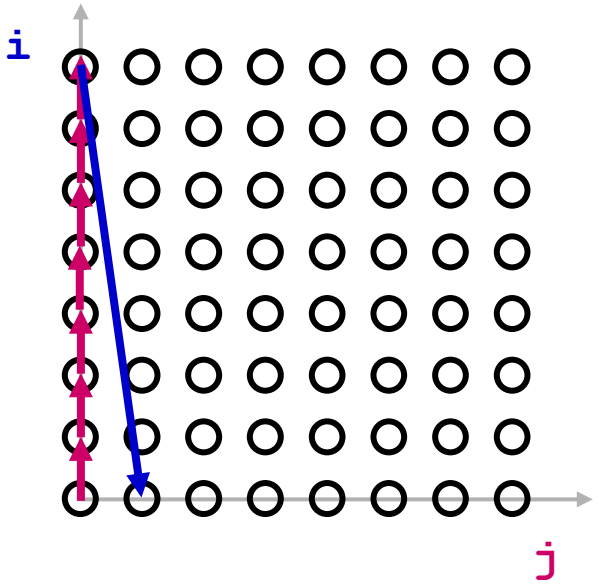


DATA SPACE

A

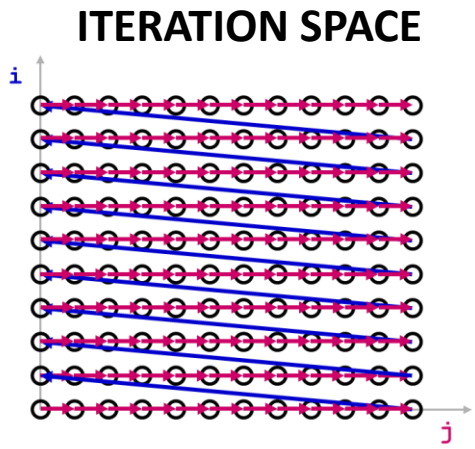


B

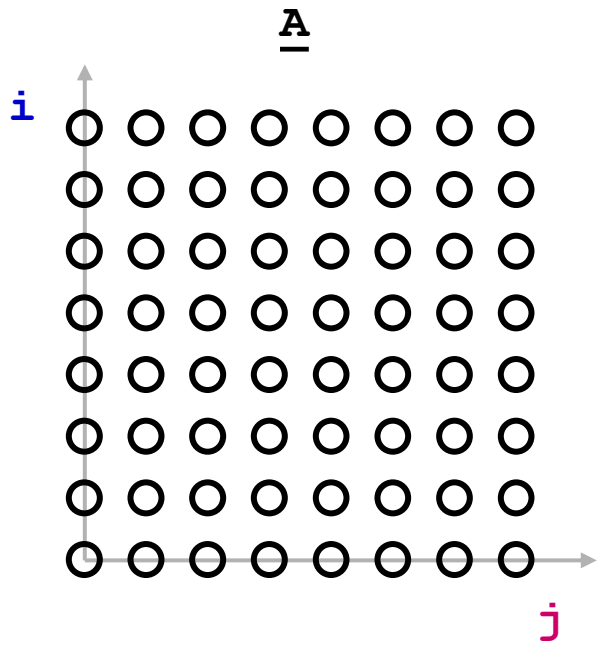


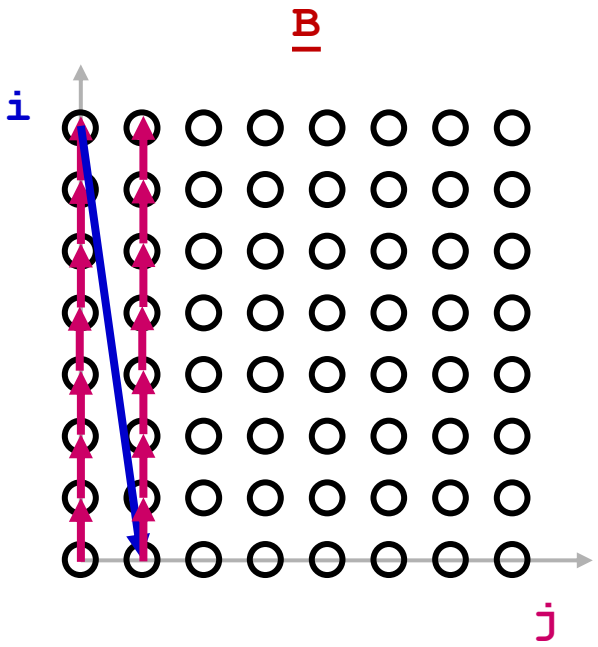
When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



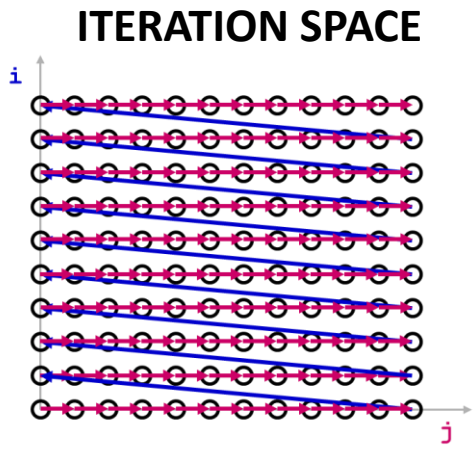
DATA SPACE

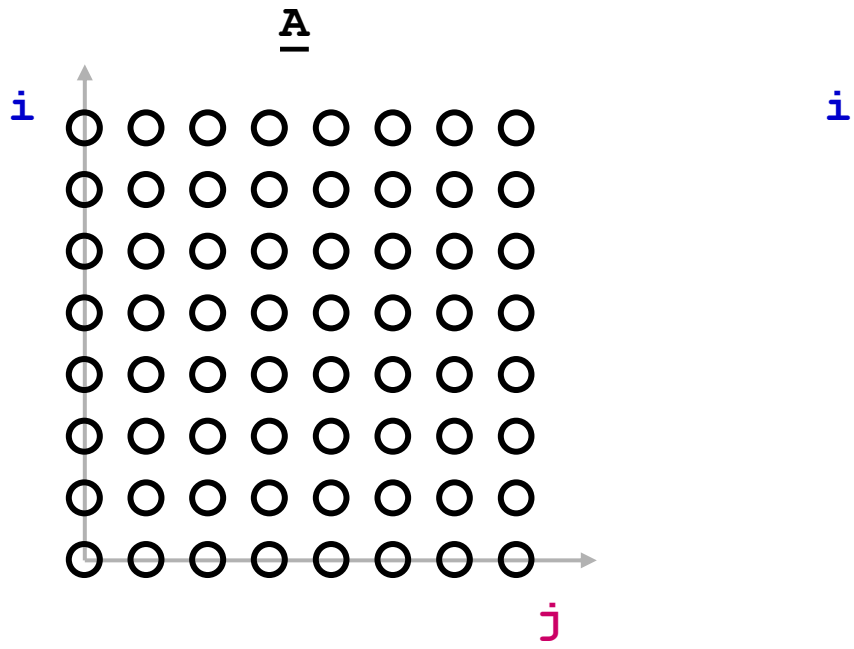




When Do Cache Misses Occur?

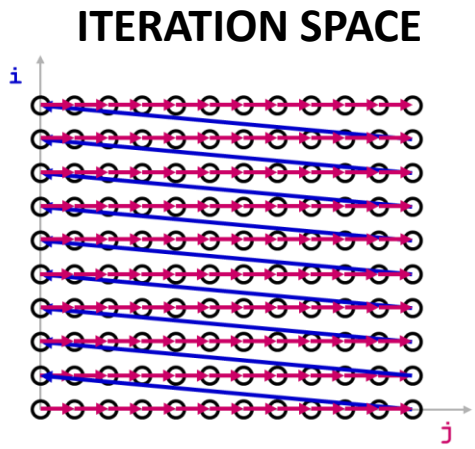
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



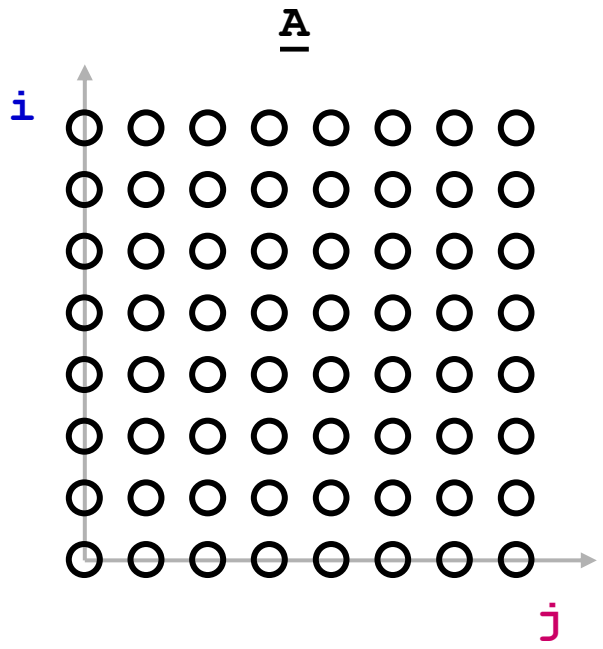


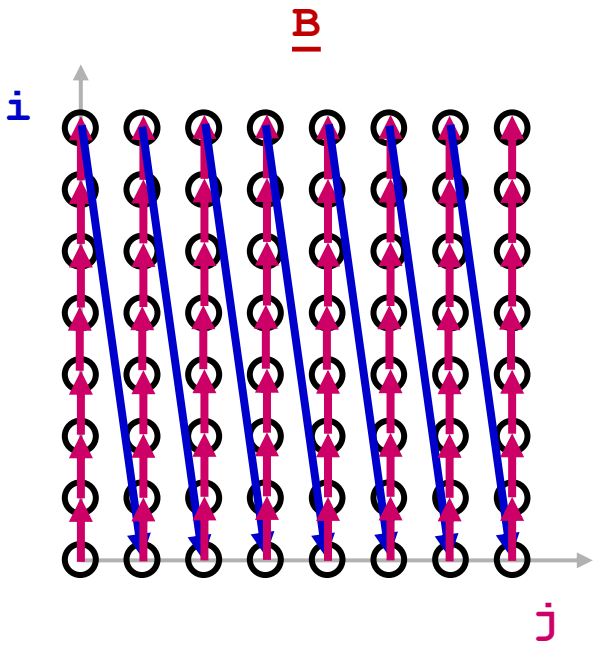
When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



DATA SPACE

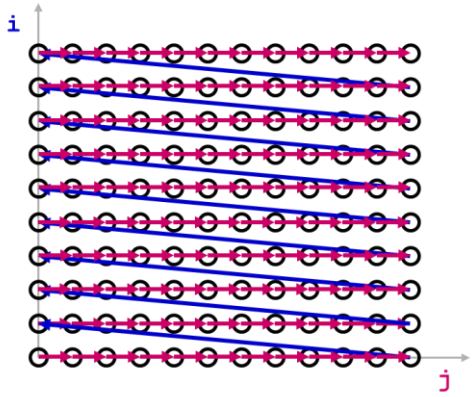




When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

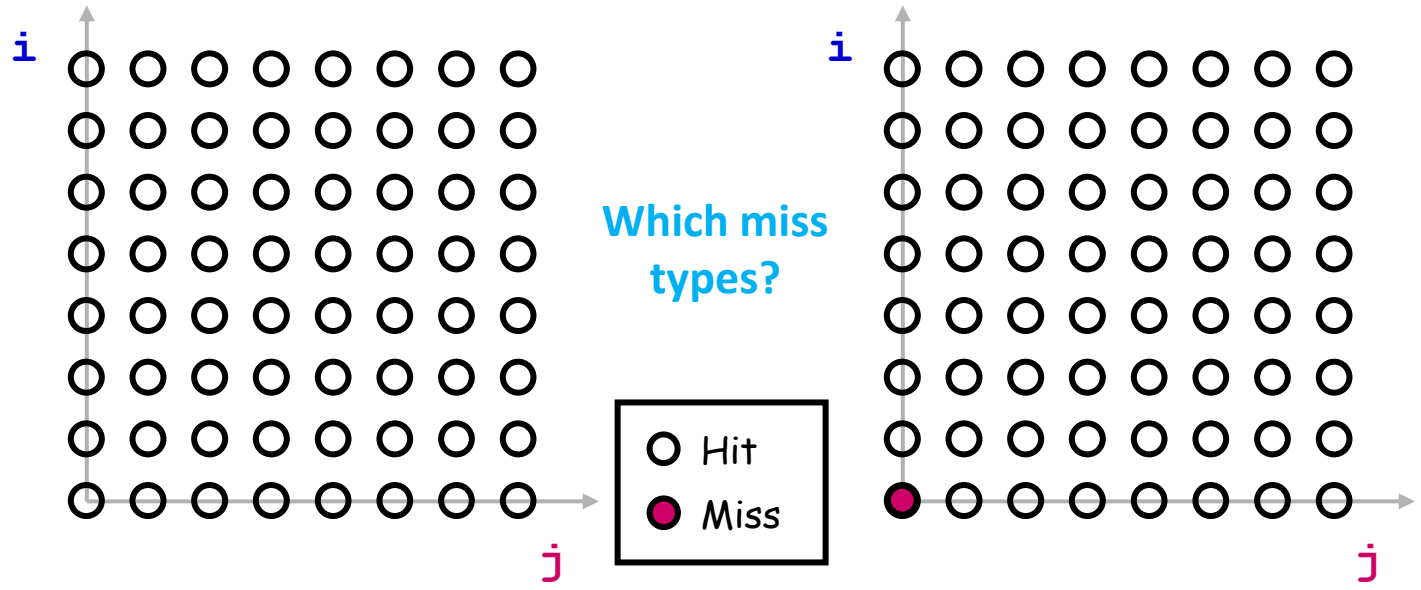
ITERATION SPACE



When do cache misses occur?

DATA SPACE

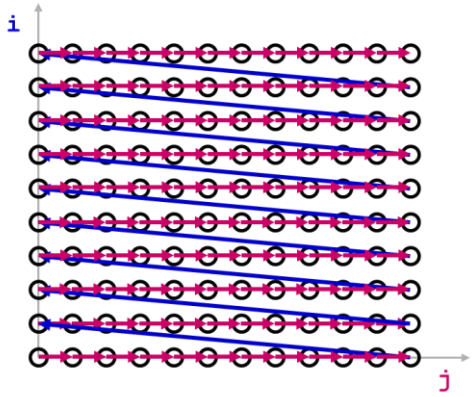
Assuming 8 cache lines of 4 words



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

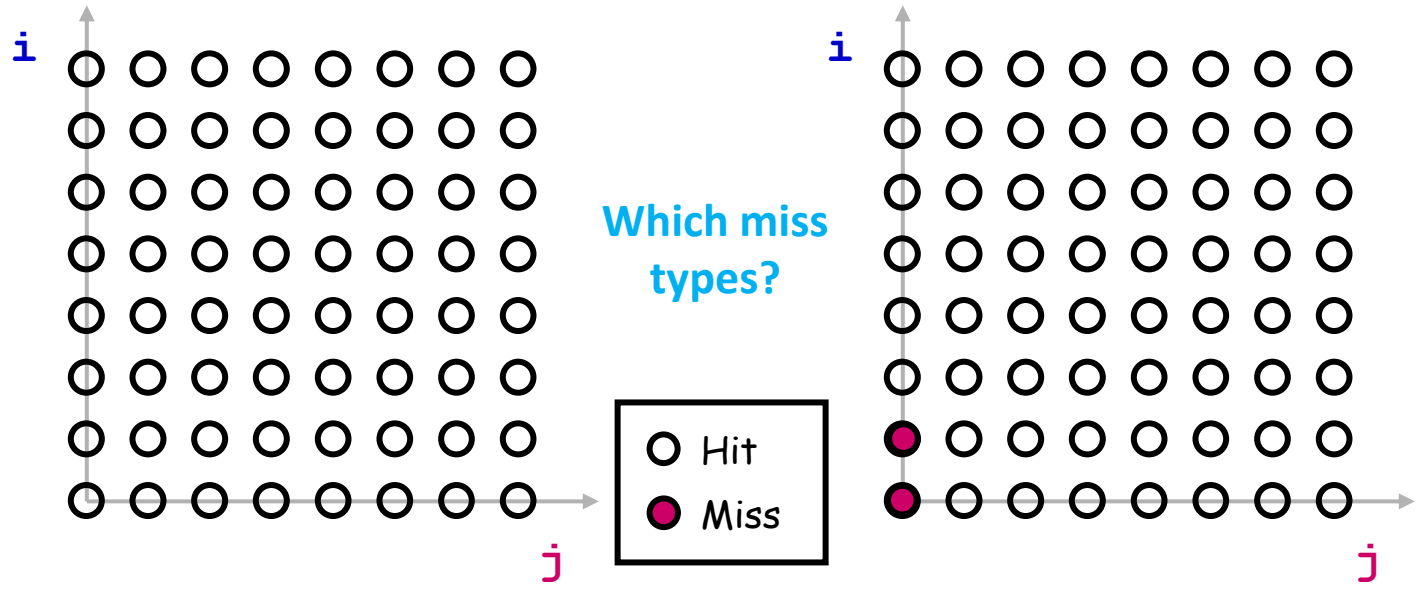
ITERATION SPACE



When do cache misses occur?

DATA SPACE

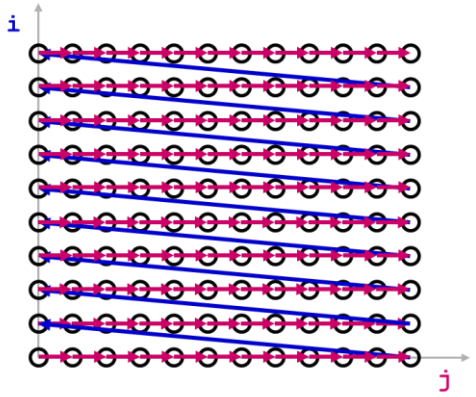
Assuming 8 cache lines of 4 words



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

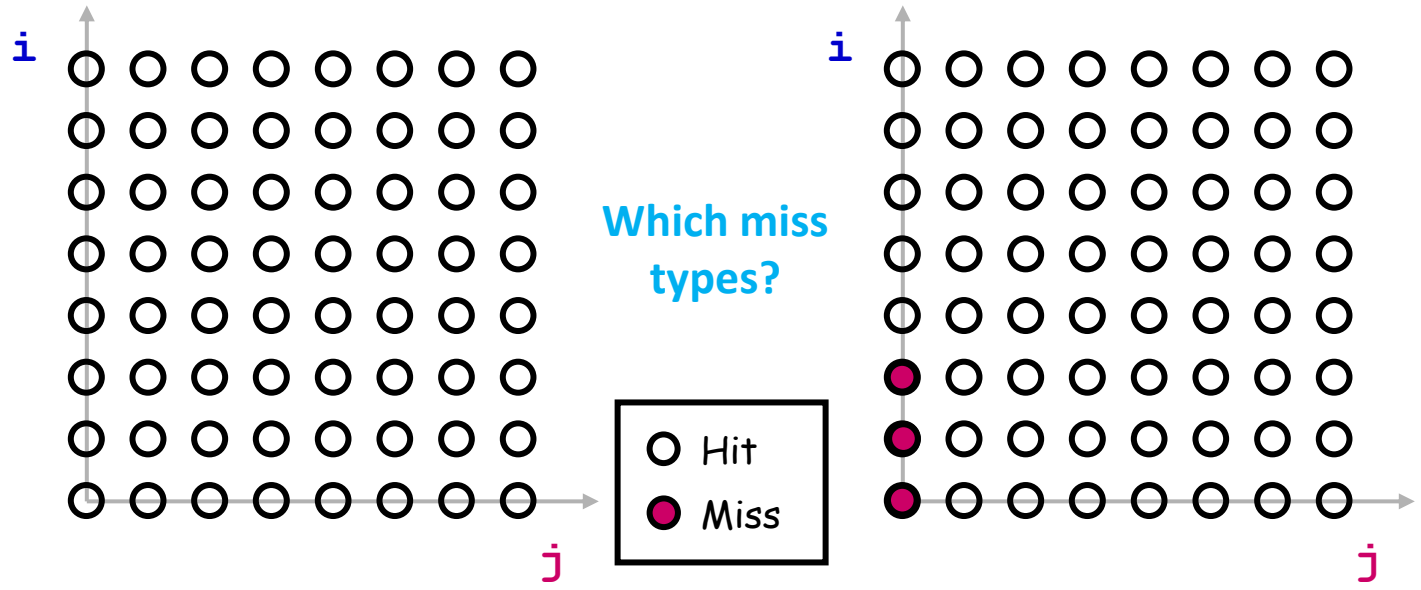
ITERATION SPACE



When do cache misses occur?

DATA SPACE

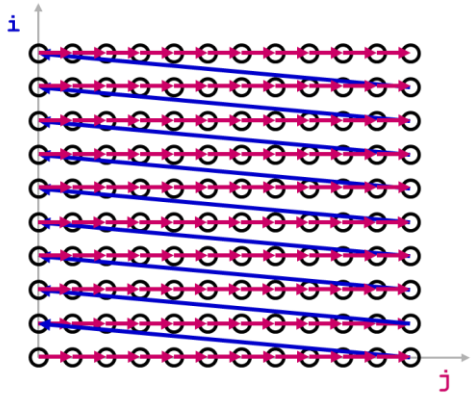
Assuming 8 cache lines of 4 words



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

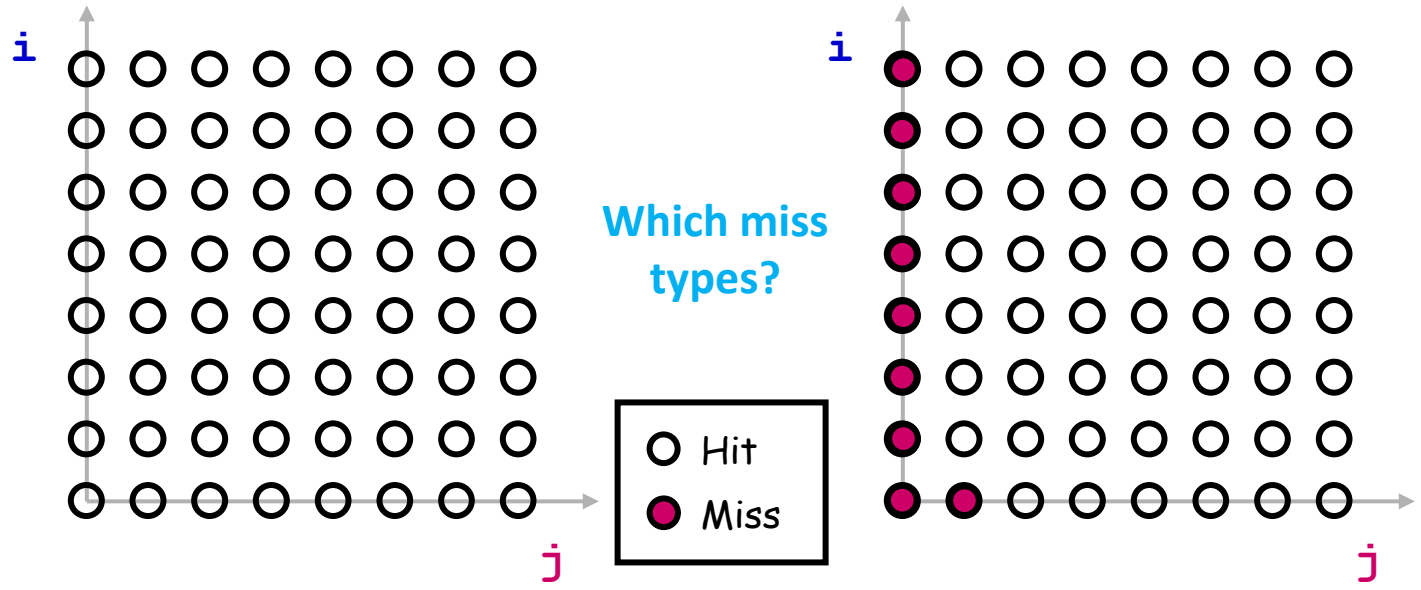
ITERATION SPACE



When do cache misses occur?

DATA SPACE

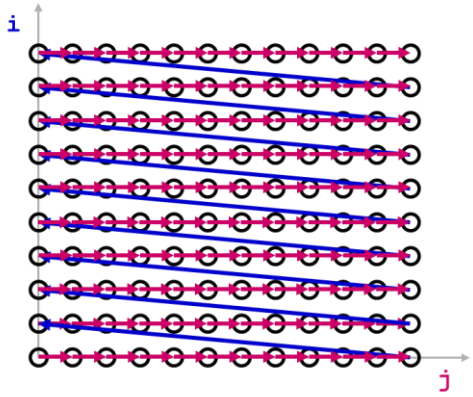
Assuming 8 cache lines of 4 words



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

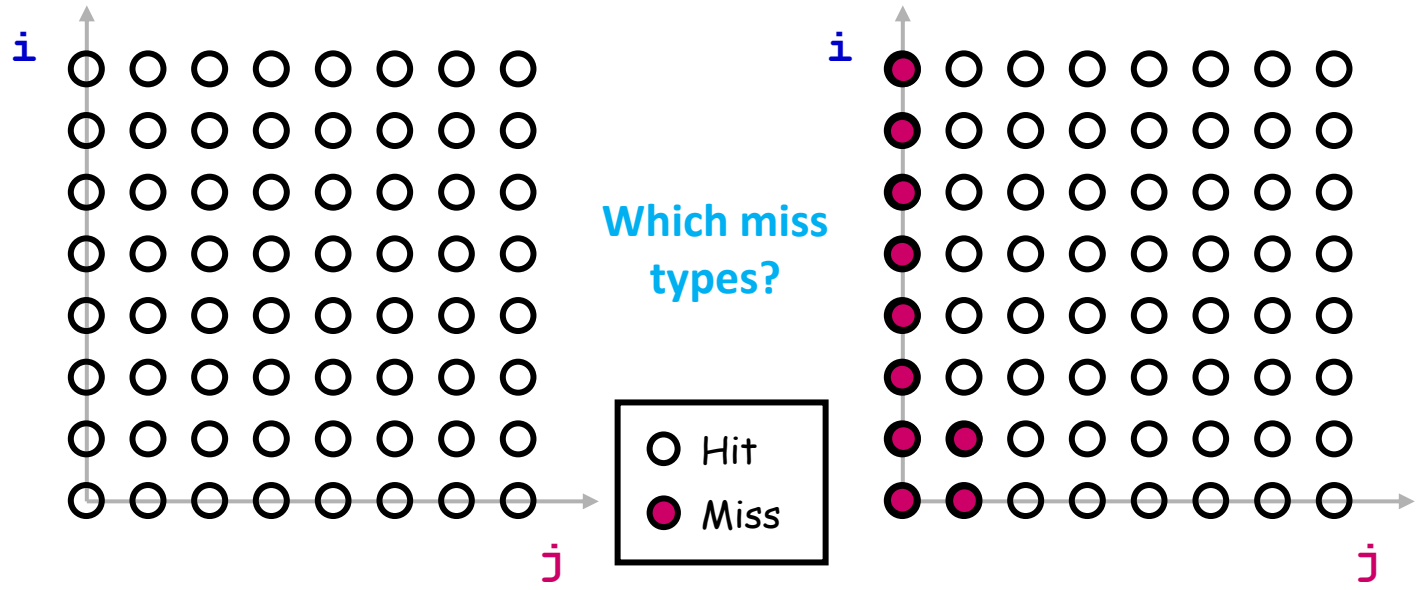
ITERATION SPACE



When do cache misses occur?

DATA SPACE

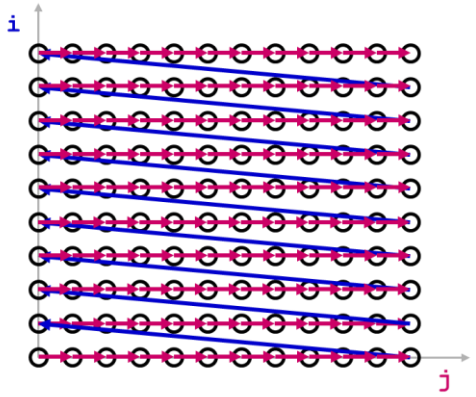
Assuming 8 cache lines of 4 words



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

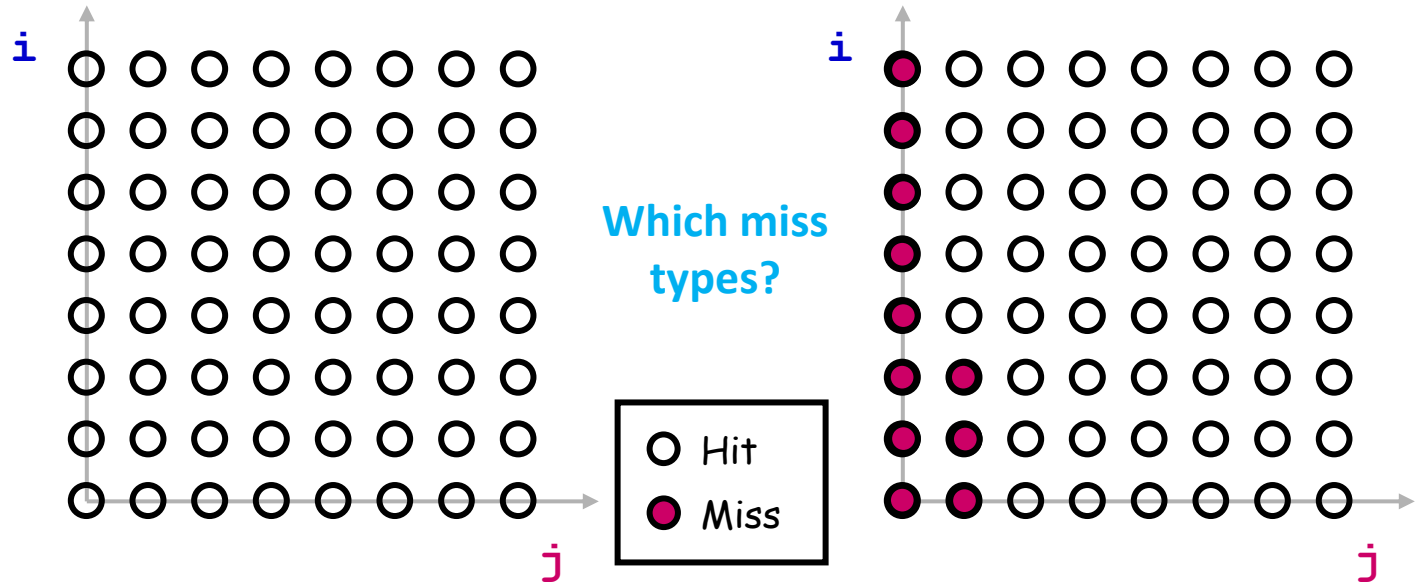
ITERATION SPACE



When do cache misses occur?

DATA SPACE

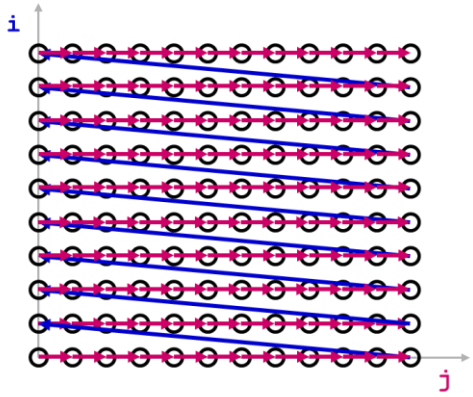
Assuming 8 cache lines of 4 words



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

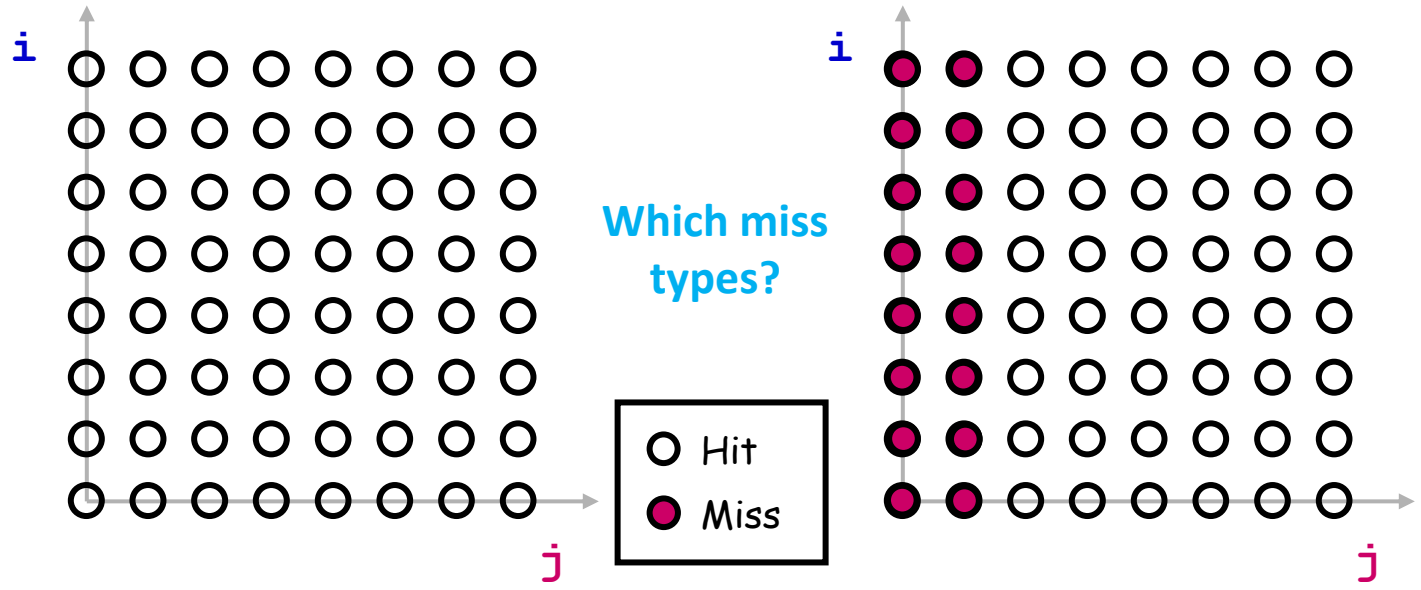
ITERATION SPACE



When do cache misses occur?

DATA SPACE

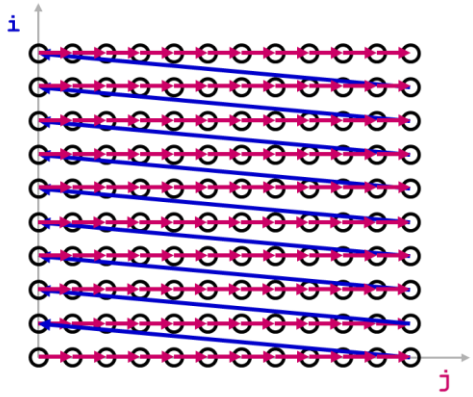
Assuming 8 cache lines of 4 words



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

ITERATION SPACE



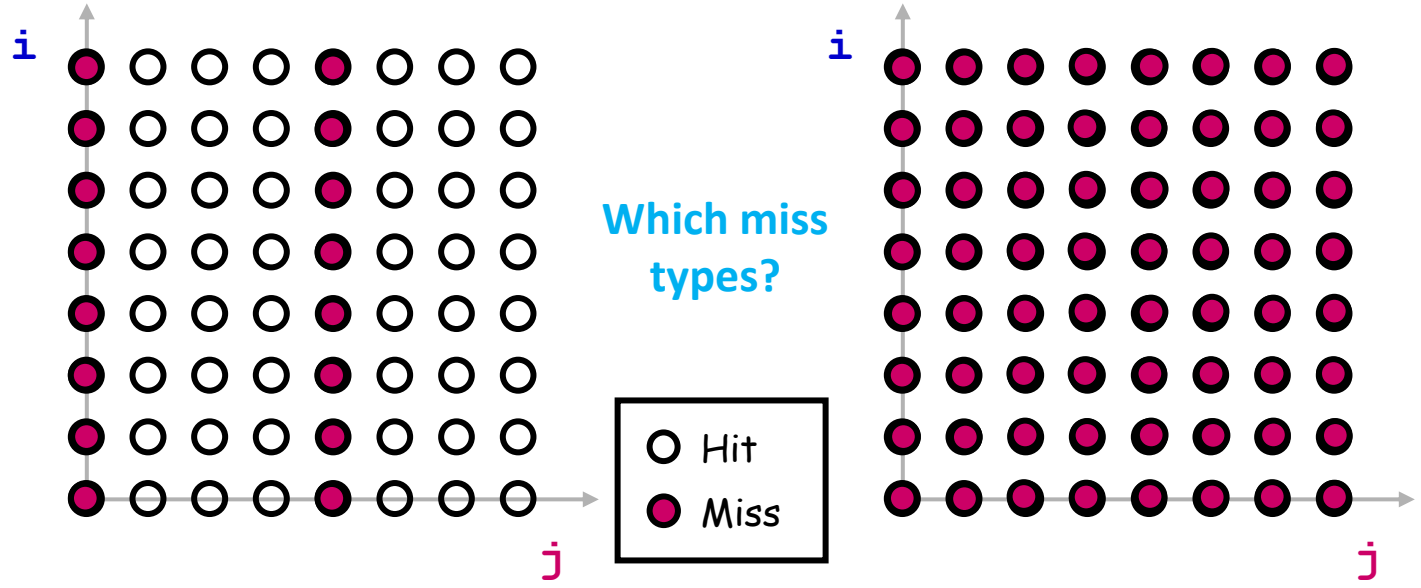
When do cache misses occur?

DATA SPACE

A

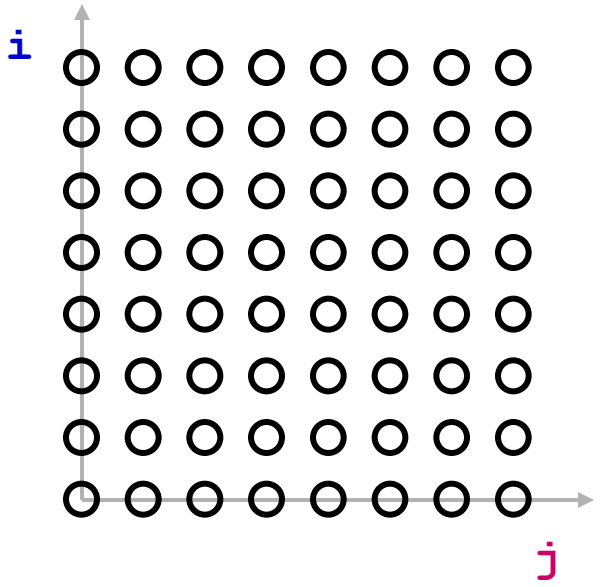
Assuming 8 cache lines of 4 words

B



When Do Cache Misses Occur?

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i+j][0] = i*j;
```



Cache Behavior of Array Accesses

- We need to answer the following questions:
  - when do cache misses occur?
    - use "locality analysis"
  - can we change the order of the iterations (or possibly data layout) to produce better behavior?
    - evaluate the cost of various alternatives
  - does the new ordering/layout still produce correct results?
    - use "dependence analysis"

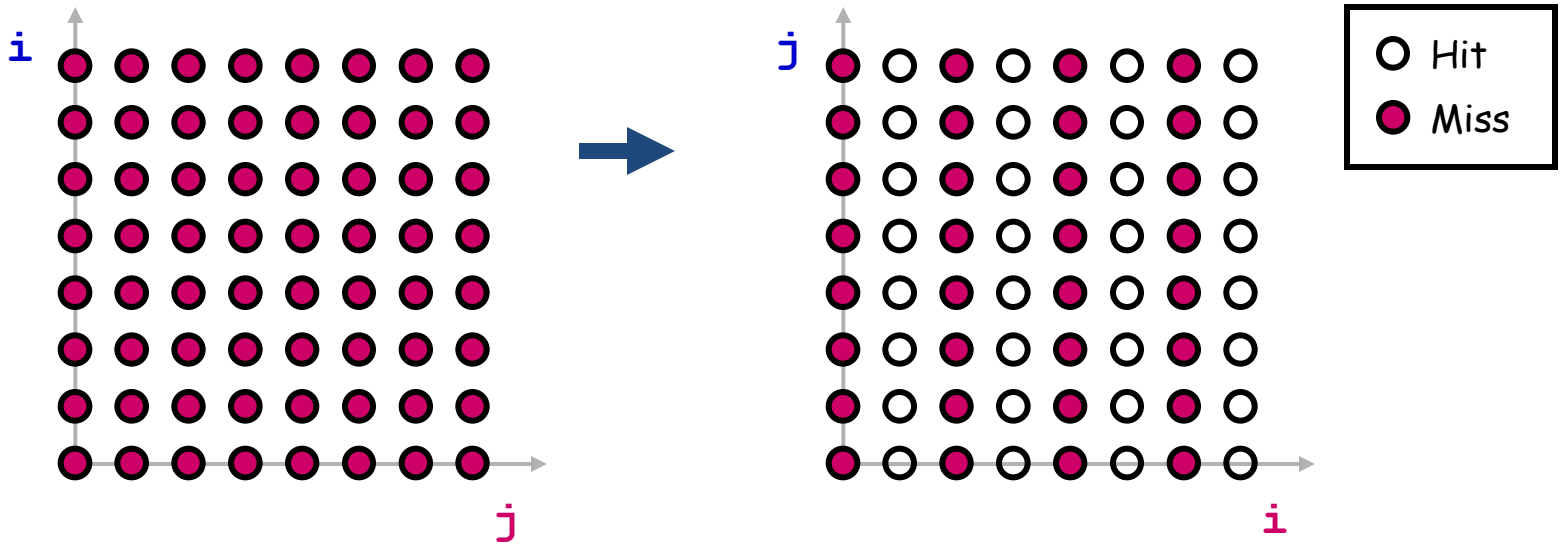
Examples of Loop Transformations

For improved cache behavior

- Loop Interchange
- Loop Tiling/Blocking
- …

Loop Interchange

```
for i = 0 to N-1                for j = 0 to N-1
   for j = 0 to N-1                for i = 0 to N-1
      A[j][i] = i*j;                  A[j][i] = i*j;
```



- *(assuming N is large relative to cache size)*

Permutation has many goals

- Locality optimization
  - Particularly, for spatial locality
- Rearrange loop nest to move parallelism to appropriate level of granularity
  - Inward to exploit fine-grain parallelism
  - Outward to exploit coarse-grain parallelism
- Also, to enable other optimizations

Cache Blocking in Two Dimensions

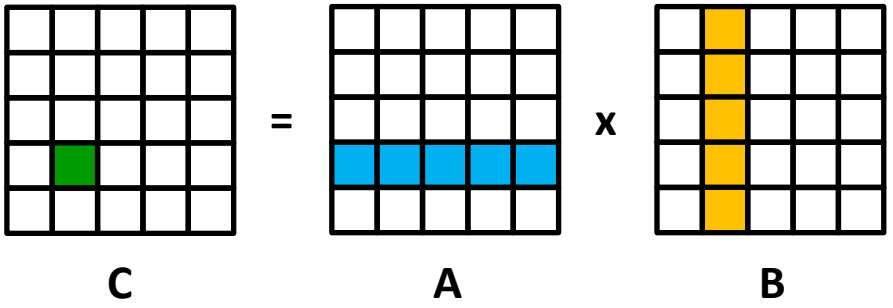
Matrix multiplication

```
for i = 0 to N-1
  for j = 0 to N-1
    for k = 0 to N-1
      c[i][j] += a[i][k]*b[k][j];
```

- Imagine a situation in which your data structure does not fit in cache, but has lots of reuse
- The traversal order is key to exploiting temporal (and spatial) locality

Cache Blocking in Two Dimensions

Matrix multiplication



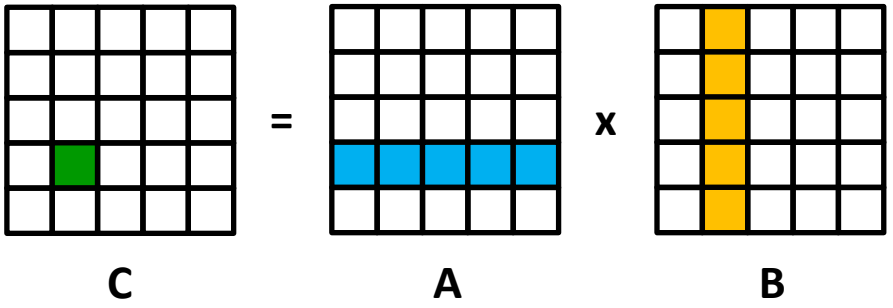

```
for i = 0 to N-1
  for j = 0 to N-1
    for k = 0 to N-1
      c[i][j] += a[i][k]*b[k][j];
```

1 element = cache line size = S
N elements = 1 matrix row does not fit in cache

- **In the innermost loop:**
- c[i][j] is accessed only once – no reuse
- a[i][k] entails accessing a whole A matrix row – doesn't fit in cache
- b[k][j] entails accessing a whole B matrix column – doesn't fit in cache and has poor spatial locality

Cache Blocking in Two Dimensions

Matrix multiplication

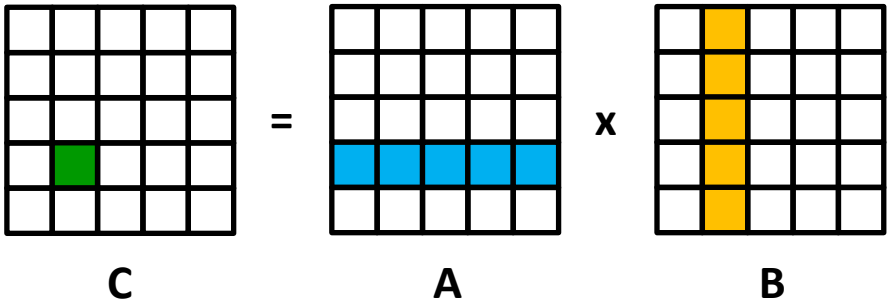



```
for i = 0 to N-1
  for j = 0 to N-1
    for k = 0 to N-1
      c[i][j] += a[i][k]*b[k][j];
```

- **How many cache misses?**
- C – NxN – Each row element is written N times in sequence (the element stays in cache)
- A – NxNxN – Each row is read N times in sequence (but a row doesn't fit in cache)
- B – NxNxN – Each matrix element is read once in column-major order

Cache Blocking in Two Dimensions

Matrix multiplication

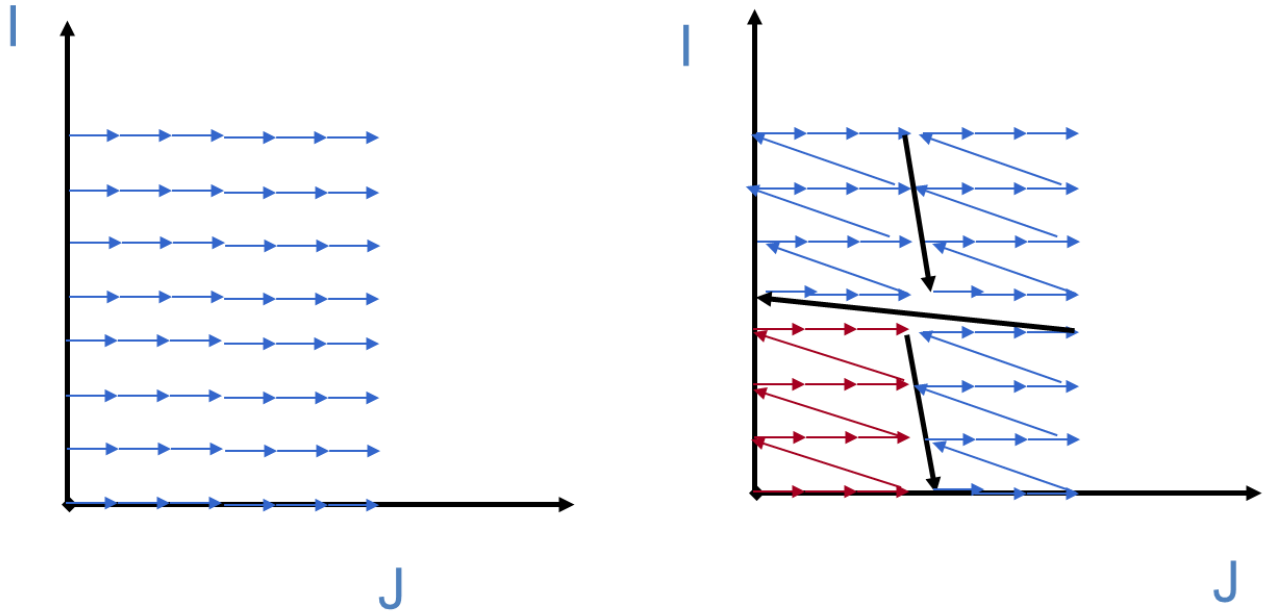



```
for i = 0 to N-1
  for j = 0 to N-1
    for k = 0 to N-1
      c[i][j] += a[i][k]*b[k][j];
```

- **How many cache misses? - $2N^3 + N^2$**
- C – NxN – Each row element is written N times in sequence (the element stays in cache)
- A – NxNxN – Each row is read N times in sequence (but a row doesn't fit in cache)
- B – NxNxN – Each matrix element is read once in column-major order

Cache Blocking in Two Dimensions

- Blocking reorders loop iterations to bring iterations that reuse data closer in time
- Goal is to retain in cache between reuse



Cache Blocking in Two Dimensions

Matrix multiplication

```
for i = 0 to N-1
  for j = 0 to N-1
    for k = 0 to N-1
      c[i][j] += a[i][k]*b[k][j];
```

```
for (jj=0; jj<N; jj+=B)
  for (kk=0; kk<N; kk+=B)
    for i = 0 to N-1
      for (j = jj; j < jj+B; j++)
        for (k = kk; k < kk+B; k++)
          c[i][j] += a[i][k]*b[k][j];
```

- **Cache blocking alters the traversal order of the matrices involved**

Cache Blocking in Two Dimensions

Matrix multiplication

```
for i = 0 to N-1
  for j = 0 to N-1
    for k = 0 to N-1
      c[i][j] += a[i][k]*b[k][j];
```

```
for (jj=0; jj<N; jj+=B)
  for (kk=0; kk<N; kk+=B)
    for i = 0 to N-1
      for (j = jj; j < jj+B; j++)
        for (k = kk; k < kk+B; k++)
          c[i][j] += a[i][k]*b[k][j];
```

- Cache blocking alters the traversal order of the matrices involved
- How many misses?

Tiling is Fundamental!

- Tiling is very commonly used to manage limited storage
  - Registers
  - Caches
  - Software-managed buffers
  - Small main memory
- Can be applied hierarchically

Determining Safety and Profitability

- Safety
  - Notion of reordering transformations
  - Based on data dependences
- Profitability
  - Reuse analysis (and other cost models)
  - Also based on data dependences, but simpler

Loop fusion

- Some programs have separate loops that access same array(s)
  - If the arrays are large the data is replaced before it is reused in the second loop
- **Loop fusion**: "fuse" loops into a single loop
  - Data fetched in the cache can be reused before being replaced
  - Reduces miss rate by improving temporal locality

Loop fusion

- Example

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    a[i][j] = 1/b[i][j]*c[i][j];

for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    d[i][j] = a[i][j]+c[i][j];
```

Loop fusion

- Example

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    a[i][j] = 1/b[i][j]*c[i][j];

for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    d[i][j] = a[i][j]+c[i][j];
```

Loop fusion

- Example

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    a[i][j] = 1/b[i][j]*c[i][j];

for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    d[i][j] = a[i][j]+c[i][j];
```

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    a[i][j] = 1/b[i][j]*c[i][j];
    d[i][j] = a[i][j]+c[i][j];
```

Loop fusion

- When is loop fusion allowed?
- Changing execution order should not violate dependencies

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

Software Prefetching

Coping with Memory Latency

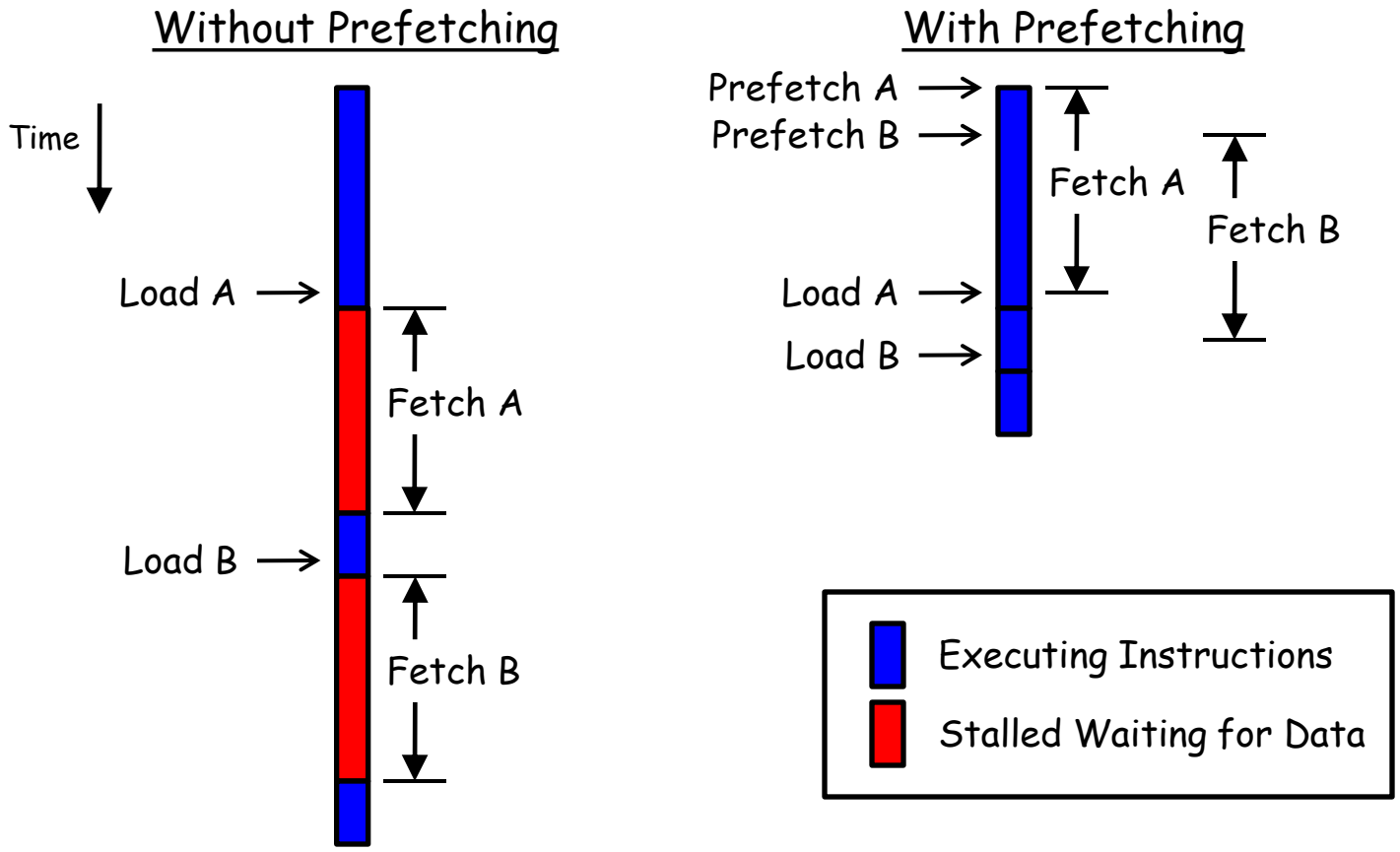
Reduce Latency:

- Locality Optimizations
  - reorder iterations to improve cache reuse

Tolerate Latency:

- Prefetching
  - move data close to the processor before it is needed

Tolerating Latency Through Prefetching



- overlap memory accesses with computation and other accesses

Compiler Algorithm

Analysis: what to prefetch

- Locality Analysis

Scheduling: when/how to issue prefetches

- Loop Splitting
- Software Pipelining

Loop Splitting

- Decompose loops to isolate cache miss instances
  - cheaper than inserting IF statements

Locality Type	Predicate	Loop Transformation
None	True	None
Temporal	`i` = 0	Peel loop `i`
Spatial	(`i` mod l) = 0	Unroll loop `i` by l

- Apply transformations recursively for nested loops
- Suppress transformations when loops become too large
  - avoid code explosion

Software Pipelining

$$\text{Iterations Ahead} = \left\lceil \frac{l}{s} \right\rceil$$

where l = memory latency, s = shortest path through loop body

Original Loop

```
for (i = 0; i<100; i++)
   a[i] = 0;
```

Software Pipelined Loop
(5 iterations ahead)

```
for (i = 0; i<5; i++)          /* Prolog */
   prefetch(&a[i]);

for (i = 0; i<95; i++) {       /* Steady State*/
   prefetch(&a[i+5]);
   a[i] = 0;
}

for (i = 95; i<100; i++)       /* Epilog */
   a[i] = 0;
```