# Memory Performance of Algorithms

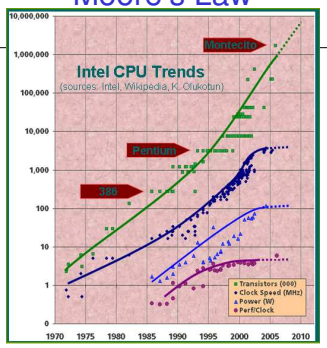
CSE 326
Data Structures
Lecture 4

## Algorithm Performance Factors

- Algorithm choices (asymptotic running time)
  - › $O(n^2)$ or $O(n \log n)$ …
- Data structure choices
  - › List or Arrays
- Language and Compiler
  - › C, C++, Java, Fortran
- Memory performance
  - › How near is the data to the processor

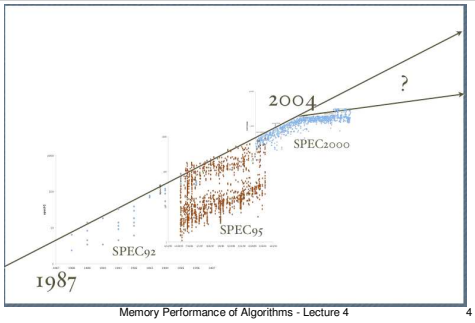


## Moore's Law





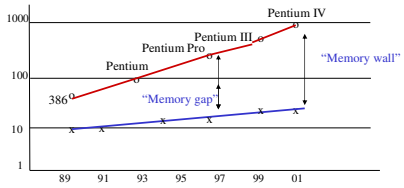

## Performance on Benchmarks





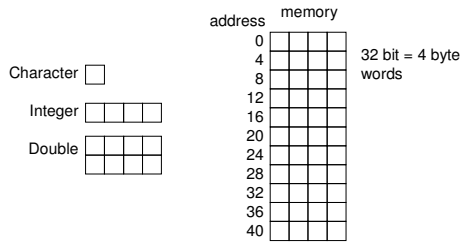

## Processor-Memory Performance Gap

- x86 CPU speed (100x over 10 years)



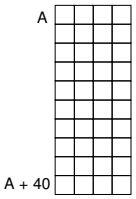



## Program Model of Memory I

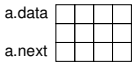


## Program Model of Memory II

Array A[0,9] of integers




Record = struct = data object
a.data : double
a.next : pointer or reference




A pointer or reference is simply an integer that represents a memory address

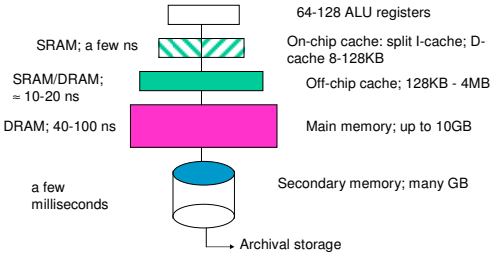


## Memory Model vs. Reality

- The program memory model is very simple and elegant
- The reality is not.
- Physical memory is organized in a hierarchy.
  - Accessing memory close to the processor is fast
  - Accessing memory far from the processor is slower
- Caching allows for accessed data to be moved closer to the processor.
  - There is a win if that data is accessed again

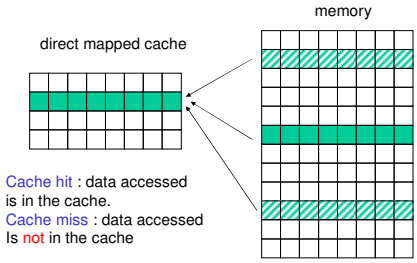


## Levels in the Memory Hierarchy





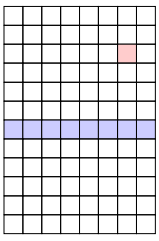

## The Cache



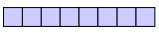

Cache hit : data accessed is in the cache.
Cache miss : data accessed Is not in the cache



## Memory Blocks



Addressable unit, usually a byte



Memory block – unit of memory transferred as a whole from memory to cache. Sometimes called "cache line". Usually, 32 64 bytes (but growing in size). Memory block size usually greater than word size



## Why Memory Blocks

- Time to transfer x bytes is given by
  - $T(x) = a + bx$. (a is latency, b ~ 1/bandwidth)
  - Average time per byte is $a/x + b$
- Because a is large relative to b, it pays to transfer more than one byte at a time.
  - The hope is that bytes near the accessed byte will be accessed soon – good spatial locality.

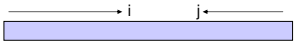
## Locality

- Spatial locality : addresses near a recently accessed byte are accessed also.
- Temporal locality : the same address that was accessed recently is accessed again.



## Examples of Locality

- Good spatial locality
  - Quicksort – the array is scanned

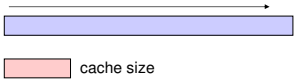



- Poor spatial locality
  - Binary search – jump around the array



## Examples of locality

- Good temporal locality
  - For loop index i in a tight loop.
    for i = 1 to n do { …}
- Poor temporal locality
  - Repeated long scans that exceed the cache size, like in iterative merge sort.



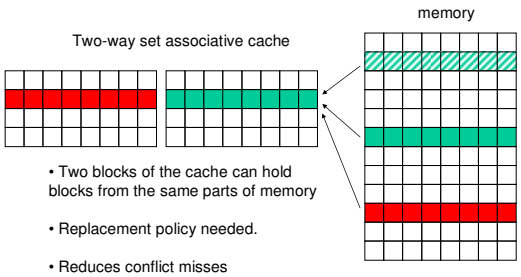



## Classifying Cache Misses

- Compulsory misses – first time a block is accessed
  - Can never be avoided
- Capacity misses – data structure does not fit in cache
  - Can be avoided by algorithmic design.
- Conflict misses – several accessed blocks map to the same location in cache
  - Conflict misses are not much of a problem because modern caches are set associative.



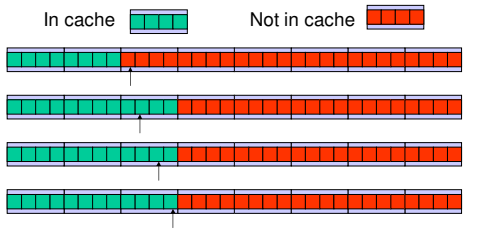
## Set Associative Cache




- Two blocks of the cache can hold blocks from the same parts of memory
- Replacement policy needed.
- Reduces conflict misses

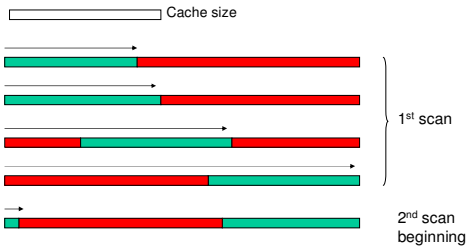


## Cache Misses for Scans




1/B misses per access where B is number of access per line

## Repeated Long Scans






## Repeated Long Scans

- Have good spatial locality
- Poor temporal locality
- If there are B accesses per memory block then 1/B of the accesses are cache misses.

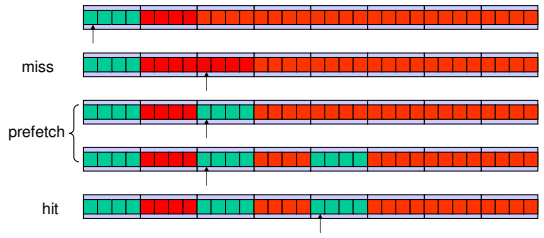


## Hardware Prefetching to the Rescue

- The hardware keeps track of integer variables to see if they are regularly incremented (or decremented).
- If so, the anticipated blocks are loaded into the cache in parallel with computation.



## Prefetching



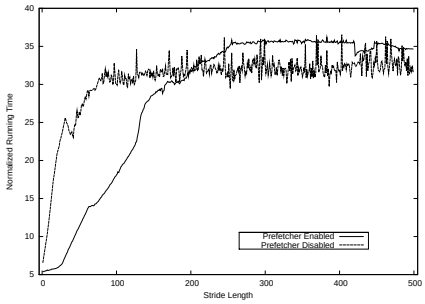



## Hardware Prefetcher

- Simple test to observe the prefetcher's effectiveness
- Access every $n^{th}$ byte in main memory
- Array of 40 million bytes, traversed 10 times
- Expect prefetcher speedup for $n \leq 256$



## Prefetching Performance Study

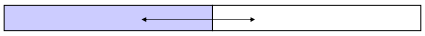


## Cache Friendly Algorithms

- Algorithms with good spatial and temporal locality.
  - Divide and Conquer is generally good.
- Algorithms with regular scans with short stride.
  - There is a limit (maybe 8) on the number of simultaneous scans that are supported by modern hardware.



## Insertion Sort is Prefetch Friendly



- The main loop scans to the right.
- The inner loop scans to the left.

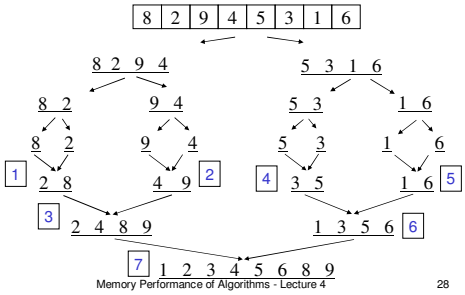


## Mergesort

- Recursive Mergesort has good spatial and temporal locality.
- When doing long merges, Mergesort uses three scans with short strides.

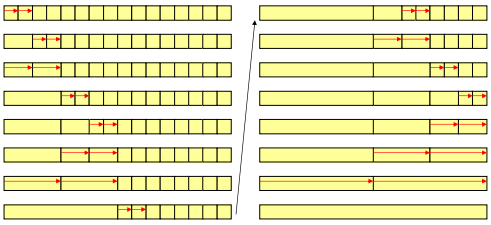


## Recursive Mergesort



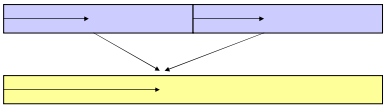



## Recursive Mergesort





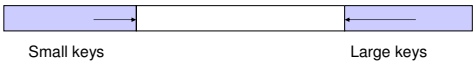

## Merging



Merging is prefetch friendly
As a result iterative mergesort is cache friendly

## Quicksort is Cache Friendly

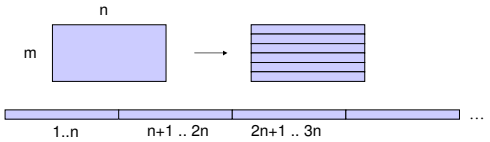
- It is divide and conquer with good temporal and spatial locality.
- It is prefetch friendly with two scans per partition.





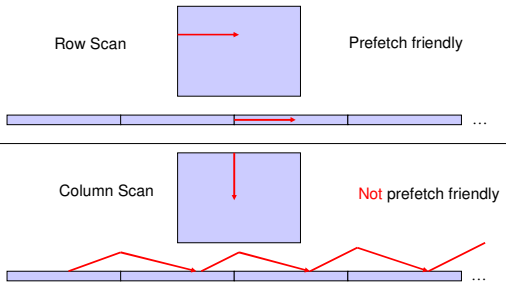

## Matrices

- A matrix $A_{ij}$ for $1 \le i \le m$, $1 \le j \le n$ with m rows and n columns is represented by a two dimensional array. A[1..m][1..n] = A[1..m,1..n].
- Generally 2-d arrays are stored by row.






## Row and Column Scans






## Example: All Pairs Shortest Path

- Input
  - N cities
  - For every pair of cities i, j there may be a non-stop flight of cost c(i,j).
  - If there is no flight then the cost c(i,j)=∞
  - c(i,i) = 0
- Output c*(i,j) = shortest path from i to j

$$= \min\{c(i_0,i_1) + c(i_1,i_2) + \cdots + c(i_{m-1},i_m)\}$$

where $i = i_0$ , $j = i_m$



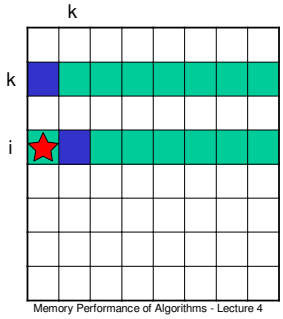
## Floyd-Warshall Algorithm

```
C* := C
for k = 1 to n
  for i = 1 to n
    for j = 1 to n
      C*[i,j] = min(C*[i,j], C*[i,k] + C*[k,j])
```

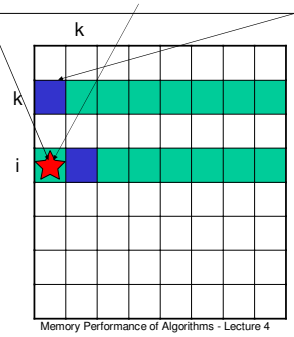
j scans in the inner loop.



## C*[i,j] = min(C*[i,j], C*[i,k] + C*[k,j])

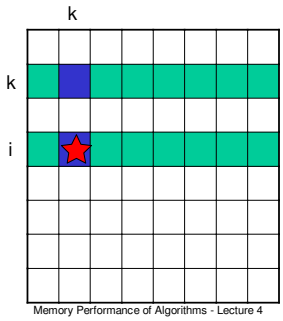


C*[i,j] = min(C*[i,j], C*[i,k] + C*[k,j])



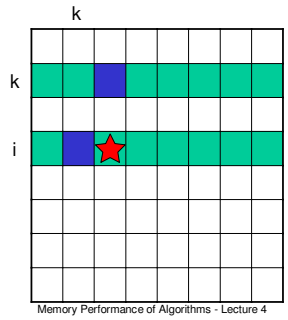



C*[i,j] = min(C*[i,j], C*[i,k] + C*[k,j])



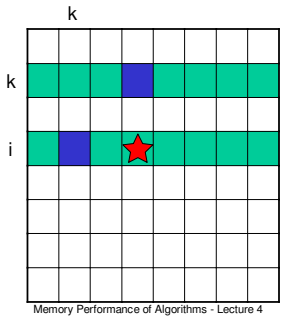



C*[i,j] = min(C*[i,j], C*[i,k] + C*[k,j])



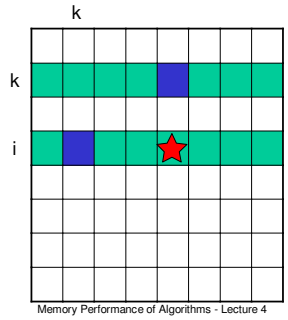



C*[i,j] = min(C*[i,j], C*[i,k] + C*[k,j])



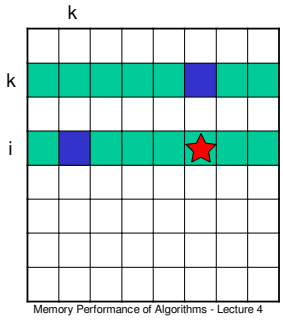



C*[i,j] = min(C*[i,j], C*[i,k] + C*[k,j])

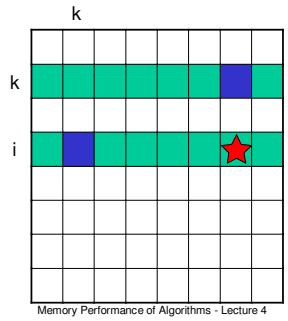





C*[i,j] = min(C*[i,j], C*[i,k] + C*[k,j])

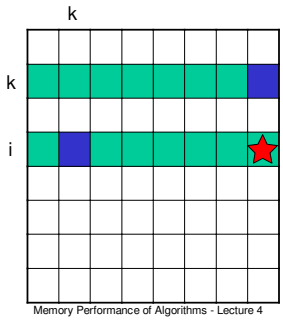


## C*[i,j] = min(C*[i,j], C*[i,k] + C*[k,j])






## C*[i,j] = min(C*[i,j], C*[i,k] + C*[k,j])





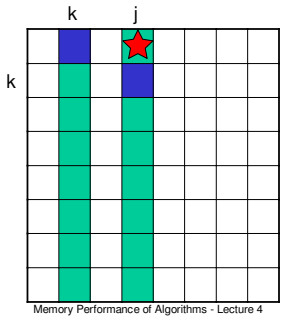

## Floyd-Warshall Algorithm By Column

```
C* := C
for k = 1 to n
  for j = 1 to n
    for i = 1 to n
      C*[i,j] = min(C*[i,j], C*[i,k] + C*[k,j])
```

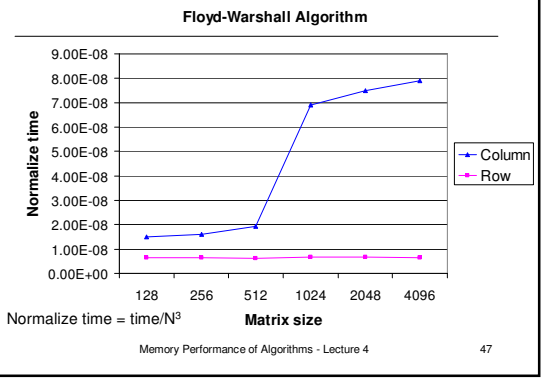
Switch the order in two inner loops.



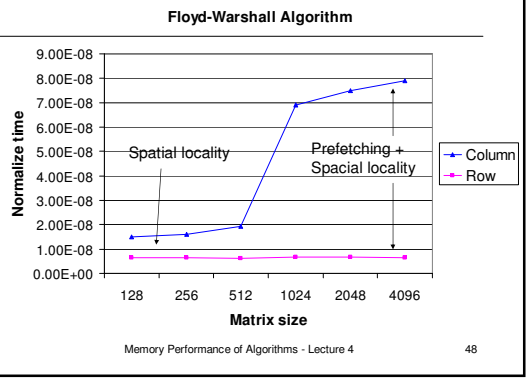
## C*[i,j] = min(C*[i,j], C*[i,k] + C*[k,j])







## Summary

- Good spatial and temporal locality and scans with short strides reduce cache misses and improve performance in modern computers.
- Warning: These are only constant time speed-ups. Asymptotic speeds-up by better algorithms are usually a bigger win.