CSE 326
Lecture 21

Radix Sort

Winter Quarter 1999

Radix Sort

- Perhaps the first mechanical sorting algorithm ever used. Used in the 1890 U.S. census by Hollerith.
- Does not use comparisons like quicksort, mergesort and relatives
- Very fast O(n) time for fixed length objects.
- Radix parameter should be chosen well to optimize performance

Example (Pass 1)

- 6 bit numbers with a radix of 2 (number of bits retired per pass)

101101
001100
111001
111000
000101
010010
101111
110100
100111
001101
101011
100001

traverse up

bucket 00

bucket 01

bucket 10

bucket 11

imagine a
stack of cards
going into a machine

Example (Pass 2)

001100
111000
110100
101101
111001
000101
001101
100001
010010
101111
100111
101011

bucket 00

bucket 01

bucket 10

bucket 11

Example (Pass 3)

100001
010010
110100
000101
100111
111000
111001
101011
001100
101101
001101
101111

bucket 00

bucket 01

bucket 10

bucket 11

Radix Sort Parameters

- n keys
- b bit integers - how to choose r

  - r divides b
  - $n/2^r$ is not too small - this is the average number of keys per bucket
  - 2^r not excessively large

  Example: 1 million 64 bit integers, r = 16 is an excellent choice
  1 million / 2^{16} is approximately 15
- w length alphabetic strings - here we are in base 128 (instead of base 2)

  number of buckets is 128 (1 character at a time) or 128^2=16,384 (2 characters at a time)
- variable length alphabetic strings can be done

  1. Sort all strings by length (this can be done using buckets)
  2. Radix sort longest strings first, include shorter strings in the pass where the characters in the shorter strings become significant.

Correctness and Analysis of Radix Sort

- Invariant of radix sort:

  After i iterations of radix sort the last i*r bits of all the keys are sorted.
- Integers case only: (Assume that r divides b)

  b/r passes

  in each pass n keys are moved and 2^r buckets are traversed

  Total time = $O(b/r\ (n + 2^r))$

  Since b and r are constants then total time = O(n). Linear time!

Alternative Implementations of Radix Sort

- Linked list implementation

  Array of size n plus array of pointers of size 2^r (buckets)

  For b/r passes scan the array right to left inserting the keys into the appropriate bucket. Scan the buckets left to right inserting the keys back into the main array.
- Array implementation

  Two arrays of size n plus a count array of size 2^r

  b/r passes:

  1. Scan the source array counting the number of keys destined for each bucket.
  2. Compute the offsets for the buckets from the counts.
  3. Scan the source array inserting the keys in the destination array. With each key inserted decrement the appropriate offset. (reverse role of source and destination for next pass)

Example of Array Algorithm

	Source	Destination
1	101101	
2	001100	
3	111001	
4	111000	
5	000101	
6	010010	
7	101111	
8	110100	
9	100111	
10	001101	
11	101011	
12	100001	

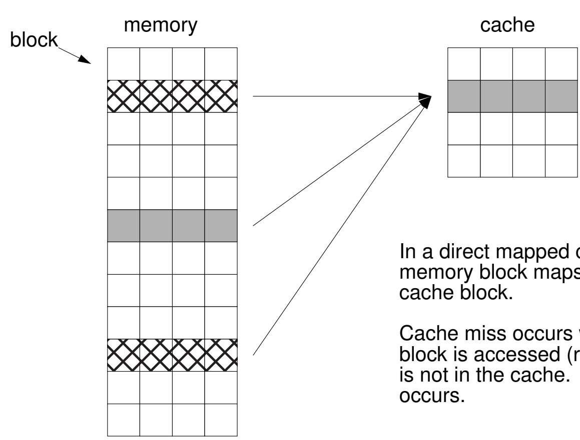
Counts	
00	3
01	5
10	1
11	3

Offsets	
00	3
01	8
10	9
11	12

Performance of Sorting

- Affected by the number of instructions executed
  - Theoretically closely related to number of comparisons for comparison sorts like insertion sort, quicksort, mergesort, and heapsort
  - Practically many other factors including: list vs. array, use of recursion
- Affected by the number of cache misses incurred.
  - compulsary misses - first time a cache block is read
  - capacity misses - data structure does not fit in cache
  - conflict misses - several blocks in memory map to the same location in the cache
- Good cache locality lowers the number of cache misses
  - temporal locality - a block that was visited recently is visited again
  - spatial locality - data near each other are visited together

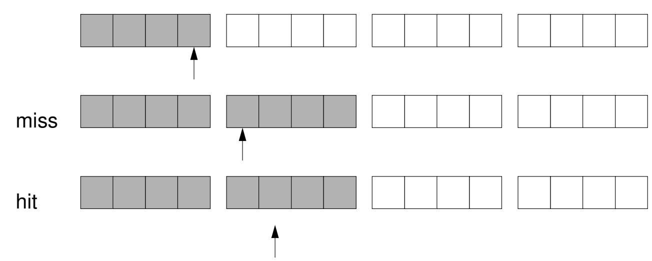
The Cache



In a direct mapped cache each memory block maps to a single cache block.

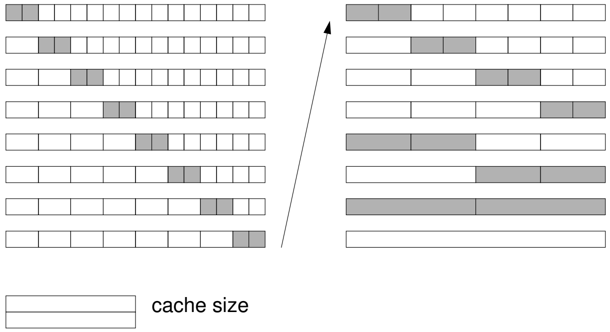
Cache miss occurs when a memory block is accessed (read or write) that is not in the cache. Otherwise, a hit occurs.

Traversing an Array



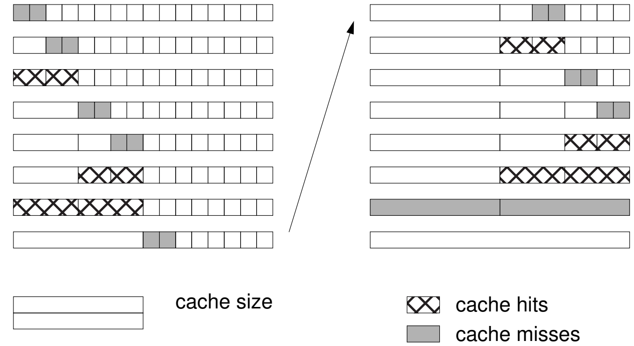
One miss for every 4 accesses in a traversal.

Cache Performance of Iterative Mergesort



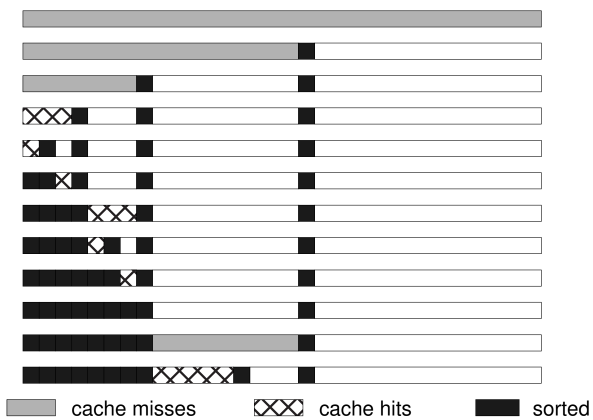
No temporal locality!

Cache Performance of Recursive Mergesort



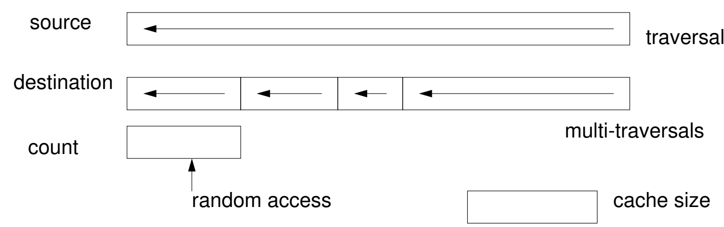
Good cache performance!

Cache Performance of Quicksort



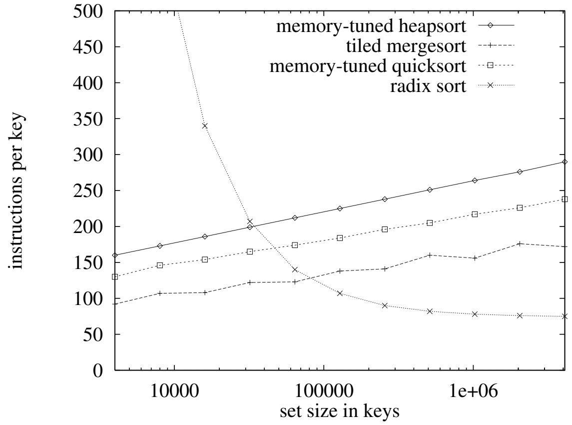
Good cache performance!

Cache Performance of Radix Sort

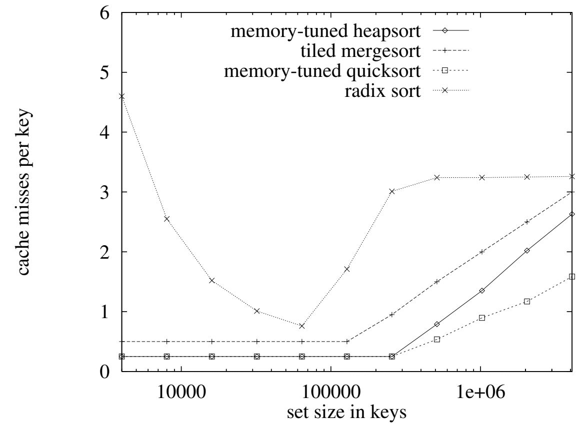


Very little temporal locality!

Sorting Instruction Count



Sorting Cache Misses



Sorting Execution Time