Data Structure and Algorithm Design

ME/MSc (Computer) – Pokhara University

Prepared by:
Assoc. Prof. Madan Kadariya (NCIT)
Contact: madan.kadariya@ncit.edu.np

Chapter 3:
Cache Efficient Data Structure (10 hrs)

Outline

3.5.1. Cache-Conscious B Tree

3.5.2. Cache-Conscious Merge Sort,

Cache-Conscious B-Tree

Cache-Conscious B-Tree

- A **cache-conscious B-tree** is specifically designed to optimize memory access patterns based on CPU cache architectures.
- The goal is to minimize cache misses by aligning the B-tree nodes with cache line sizes ensuring when a no

Key Aspects:

1. **Node Structure and Memory Layout**:
   - In a traditional B-tree, nodes optimize tree balancing and search efficiency, while in a cache-conscious B-tree, nodes are reorganized to fit into CPU cache lines(typically 64 bytes in modern CPUs), improving memory access efficiency.
2. **Alignment and Packing**:
   - Keys are packed into nodes, ensuring that once a node is loaded into the cache, the keys that are likely to be accessed together are also brought into cache simultaneously, reducing cache misses.
3. **Data Locality**:
   - The child pointers are also placed in the node layout to ensure that when a key is accessed, its corresponding child pointer (if necessary) is likely to be in the same cache line, improving locality.

Cache-Conscious B-Tree

- **Example**: In a traditional B-tree with a fanout of 4, each internal node holds 3 keys and 4 pointers, but nodes may not align with cache lines, causing inefficient cache usage. In a cache-conscious B-tree, nodes are packed and aligned to fit into the CPU's cache line, allowing all keys and pointers to be fetched in a single cache load, reducing cache misses and memory accesses.
- Complexity: Both B-trees and cache-conscious B-trees have the same **asymptotic time complexity** of *O(log_b N)* for search, insert, and delete operations. However, cache-conscious B-trees improve **real-world performance** by reducing **cache misses**and optimizing memory access, resulting in better **amortized performance** compared to traditional B-trees.
- A **Cache-Sensitive B+Tree (CSB+Tree)** optimizes memory access by aligning nodes to cache lines, reducing cache misses and improving performance for large datasets. It preserves the B+ tree structure, supporting efficient search, insert, and delete operations with improved cache locality.

Cache-Oblivious merge sort

Cache-Oblivious merge sort

Sequential Merge Sort: In each step, it sorts a subarray , starting with the entire array and recursing down to smaller and smaller subarrays.

- Merge sort is a classical sorting algorithm using a divide-and-conquer approach. The initial unsorted list is first divided in half, each half sublist is then applied the same division method until individual elements are obtained. Pairs of adjacent elements/sublists are then merged into sorted sublists until the one fully merged and sorted list is obtained.
- It is a stable sorting algorithm.
- The best, average and waste case time complexity of merge sort is ***O(nlogn)***

```
MERGE-SORT(A, p, r)
if p ≥ r                              // zero or one element?
    return
q = ⌊(p + r)/2⌋                       // midpoint of A[p : r]
MERGE-SORT(A, p, q)                   // recursively sort A[p : q]
MERGE-SORT(A, q + 1, r)               // recursively sort A[q + 1 : r]
// Merge A[p : q] and A[q + 1 : r] into A[p : r].
MERGE(A, p, q, r)
```

Cache-Oblivious merge sort

- Cache Oblivious Models are built in a way so they can be independent of constant factors, like the size of the cache memory.
- Cache-oblivious merge sort is a variant of merge sort that is designed to work efficiently across all levels of a multi-level memory hierarchy (such as CPU caches and RAM) without requiring explicit knowledge of cache sizes or block sizes. It is a **cache-oblivious algorithm**, meaning it optimizes cache performance without tuning for specific hardware parameters.

Key Concepts

- **Divide and Conquer**: Like traditional merge sort, cache-oblivious merge sort recursively divides the array into smaller subarrays until they become trivially small.
- **Recursive Merging**: Instead of a naive merge that accesses data in a non-optimal pattern, cache-oblivious merge sort ensures that subarrays fit in cache as much as possible.
- **Implicit Cache Optimization**: It leverages the structure of recursive calls to maximize **cache efficiency**, even without explicitly knowing the cache size.

Cache-Oblivious merge sort

Cache Efficiency

- The traditional merge sort incurs many **cache misses** because merging involves non-sequential memory access.
- Cache-oblivious merge sort **minimizes cache misses** by ensuring that most memory accesses are within **a single cache line** at each level of recursion.
- The algorithm works well on hierarchical memory systems (L1, L2, RAM, disk, etc.) without requiring tuning.

Time Complexity

- **Worst-case time complexity**: $O(n \log n)$ (same as classic merge sort)
- **Expected cache complexity**: $O((n/B)\log_{M/B}(n/B))$ where:

  M = size of cache, B = block size of cache, n = size of input

Cache-Oblivious merge sort

Given k sorted sequences, each containing n/k elements, the merge process follows a **divide-and-conquer** strategy:

1. Split the ***k*** sequences into two equal halves:
   1. **Left group**: ***k/2*** sequences
   2. **Right group**: ***k/2*** sequences
2. Recursively merge the two groups.
3. Finally, merge the two resulting sequences.
   - This recursive splitting forms a **binary merge tree** with depth ***O(logk)***

Cache-Oblivious merge sort

Time Complexity Analysis

The total number of elements is n, and at each level of recursion, every element is merged exactly once. The merge process consists of $O(logk)$ levels.

1. **Number of Merge Operations per Level**:
   At each level, we perform $O(k)$ merging steps, and each merge operation involves **linear** time complexity.
2. **Total Work Done**:
   - Each level processes $O(n)$ elements.
   - Since we have $O(logk)$ levels, the total time complexity is:

     $T(n,k)=O(nlogk)$

- Thus, the **final time complexity** for merging ***k*** sorted sequences is: $O(nlogk)$

Cache-Oblivious merge sort

Cache Complexity Analysis

To analyze the **cache complexity**, we use the Ideal Cache Model where:

- **M** is the cache size.
- **B** is the cache block size.
- **Q(n,k)** is the cache complexity (number of cache misses).

1. **Cache Misses in a Single Merge Operation**:
   - A two-way merge scans two sequences sequentially, leading to $O(\frac{n}{B})$ cache misses.
   - Since we use a recursive binary merge tree, the recursion depth is $O(\log k)$
2. **Total Cache Misses Over All Levels**:
   - At each level, merging is performed sequentially, leading to: $O((\frac{n}{B})\log k)$
   - However, considering **multi-level memory hierarchy**, the cache-oblivious model refines this to: $Q(n,k)=O((\frac{n}{B})\log_{M/B}(n/B))$
   - This represents the **optimal** number of cache misses for hierarchical memory.

Cache-Oblivious merge sort

Conclusion

- The recursive merging strategy ensures that elements fit within cache as early as possible.
- **Time Complexity**: $O(nlogk)$, which is optimal.
- **Cache Complexity**: $Q(n,k)=O((\frac{n}{B})log_{M/B}(n/B))$, significantly better than a naive merge.