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. Cache Efficient Data Structures (10 hrs)

3.1. Memory Hierarchy and Cache Efficiency

3.2. Principles of Cache-Efficient Data Structures

3.3. Cache-Conscious Algorithms and Cache-Oblivious Algorithms

3.4. Profiling Tools for Analyzing Memory and Cache Efficiency

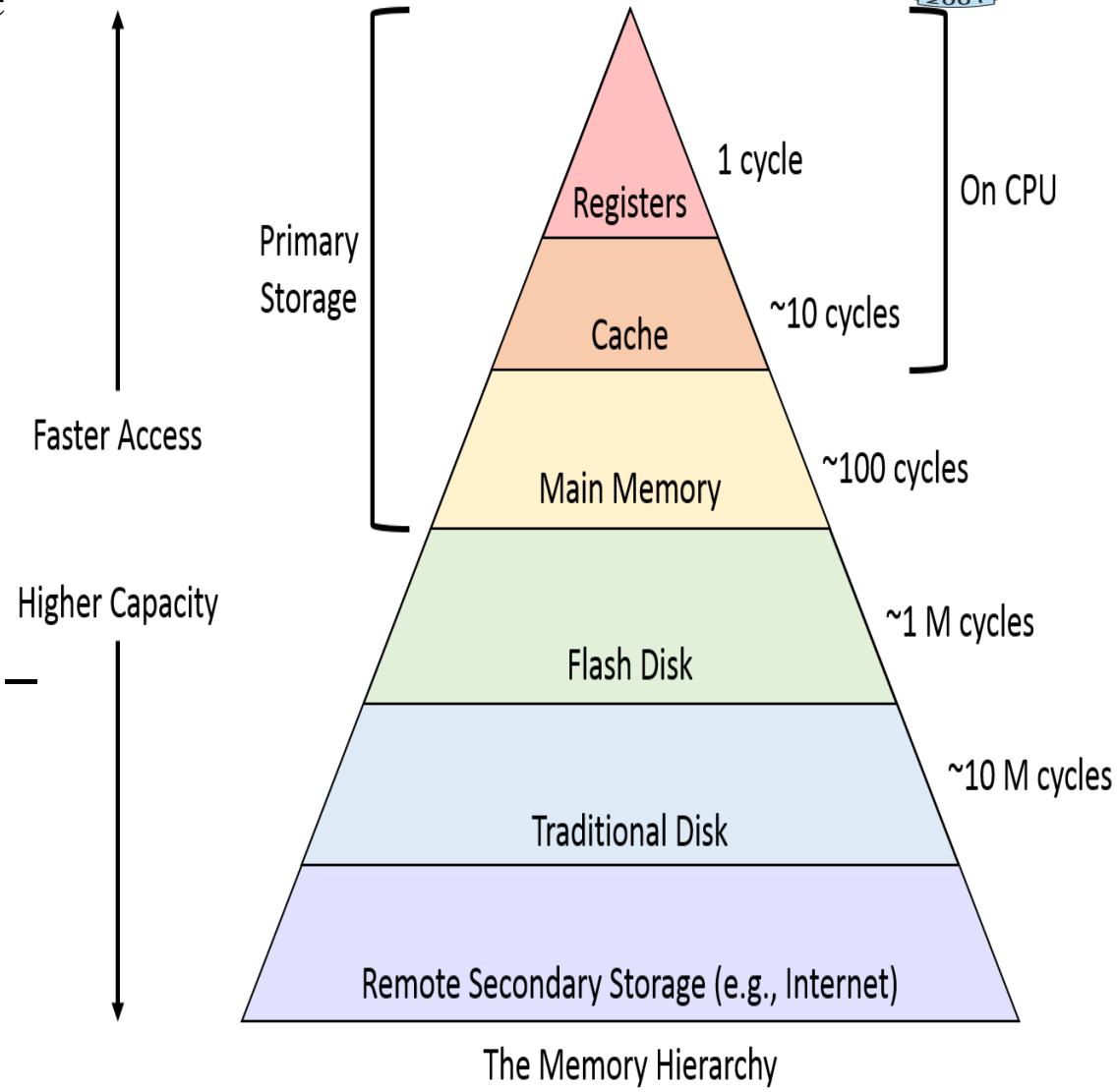
3.5. Case Studies in

- 3.5.1. Cache-Conscious B Tree
- 3.5.2. Cache-Conscious Merge Sort,
- 3.5.3. Cache-Oblivious merge sort
- 3.5.4. Profiling Tools: Valgrind and Cachegrind

Memory Hierarchy

- Memory Hierarchy is an enhancement to organize the memory such that it can minimize the access time.
- The Memory Hierarchy was developed based on a program behavior known as locality of references.
- **Two Main Types**

1. **External Memory or Secondary Memory:** Comprising of Magnetic Disk, Optical Disk, Magnetic Tape i.e. peripheral storage devices which are accessible by the processor via I/O Module.
2. **Internal Memory or Primary Memory** – Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by the processor.



The Memory Hierarchy

Cache Memory

- **Cache memory** is the high-speed memory. It is small but faster than RAM (the main memory).
- The CPU can access the cache more quickly than its primary memory.
- Cache memory can be used to synchronize with a high-speed CPU and to improve its overall performance.
- Only the CPU can access the cache memory. This memory can be the reserved part of a main memory/storage device outside the CPU.
- The cache holds the data and programs that the CPU uses frequently. So the information is instantaneously available for the CPU when it needs this information.
- If the CPU finds the required instructions or data in the system's cache memory, it does not need to access its primary memory or RAM. Thus, it speeds up the performance of the system by operating as a buffer between the RAM and the CPU.

Cache Memory

Types of Cache Memory

1. L1 Cache

- It is a first level of any cache memory, known as **L1 cache** or **Level 1** cache.
- In **L1 cache** memory, a very small memory exists inside the CPU itself. In case the CPU consists of four cores (A quad-core CPU), each core will then have its own L1 cache.
- This memory exists in the CPU, so that it can operate at the very same speed of CPU.
- Memory size of **L1 cache** ranges from 2KB to 64KB.
- The **L1 cache** has two further types of caches: The Instruction cache storing ($L1_i$) and the data cache storing ($L1_d$).

Cache Memory

Types of Cache Memory

2. L2 Cache

- It is refers to the **L2 cache** or Level 2 cache. The L2 cache may reside both inside or outside any CPU.
- All the cores of the CPU can consist of their separate (own) **L2 cache**, or they can even share a single **L2 cache** among themselves.
- In case, if it is outside the CPU, it connects with the CPU using a very high-speed bus.
- Memory size of **L2 cache** ranges from 256KB to 512KB.
- It is slower than **L1 cache** but faster than **L3 cache.**

Cache Memory

Types of Cache Memory

3. L3 Cache

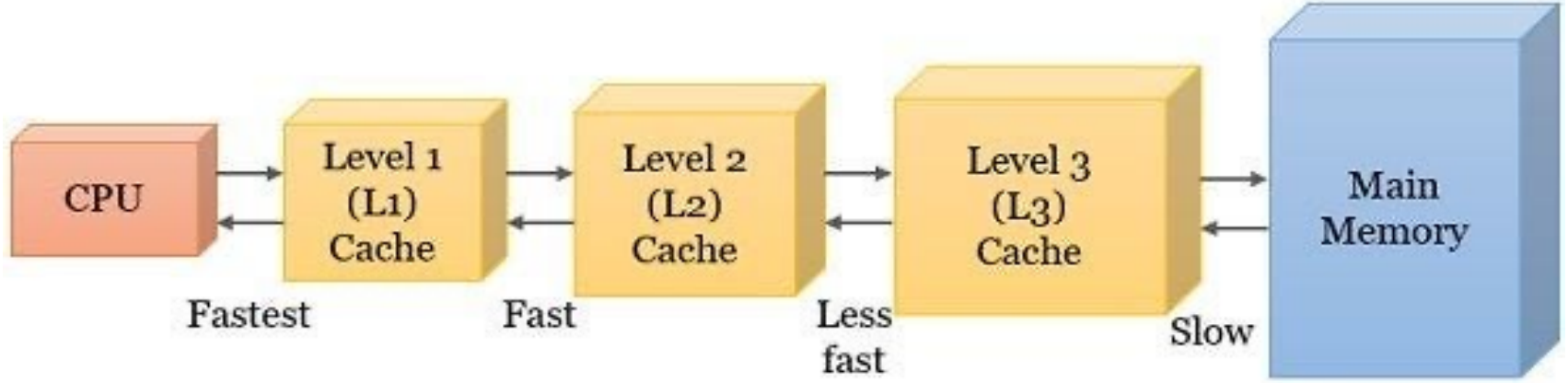
- This cache is known as the **L3 cache** or **Level 3** cache.
- This type of cache isn't present in all of the processors. Only a few of the high-end processors may contain this type of cache.
- **L3 cache** is used to enhance the overall performance of the **L1** and the **L2** cache.
- **L3 cache** is located outside a CPU, so it is shared by all CPU cores.
- Memory size of **L3 cache** ranges from 1MB to 8MB.
- **L3 cache** is slower than that of the **L1 cache** and **L2 cache** but much faster than the RAM.

Cache Memory

Types of Cache Memory

4. L4 Cache

- **L4 Cache** is a **level 4** cache and is an uncommon cache technology which is used in specific systems like Intel Broadwell i7-5775C. **L4 cache** comes after **L3 cache**.
- **L4 cache** is made up of DRAM or eDRAM. SRAM is the common choice of other caches.
- **L4 cache** is larger than L1, L2 and L3 cache.
- **L4 cache** is slower than L1, L2 and L3 cache.
- **L4 cache** is accessed by both CPU and GPU.
- **L4 cache** was first introduced by Intel through Haswell microarchitecture in 4th June, 2013.
- **L4 cache** had 128MB of L4 cache and was made up of embedded Dynamic Random Access Memory (eDRAM). It was linked to L3 cache and can be used by both CPU and on-die GPU.



Cache Organization in Computer

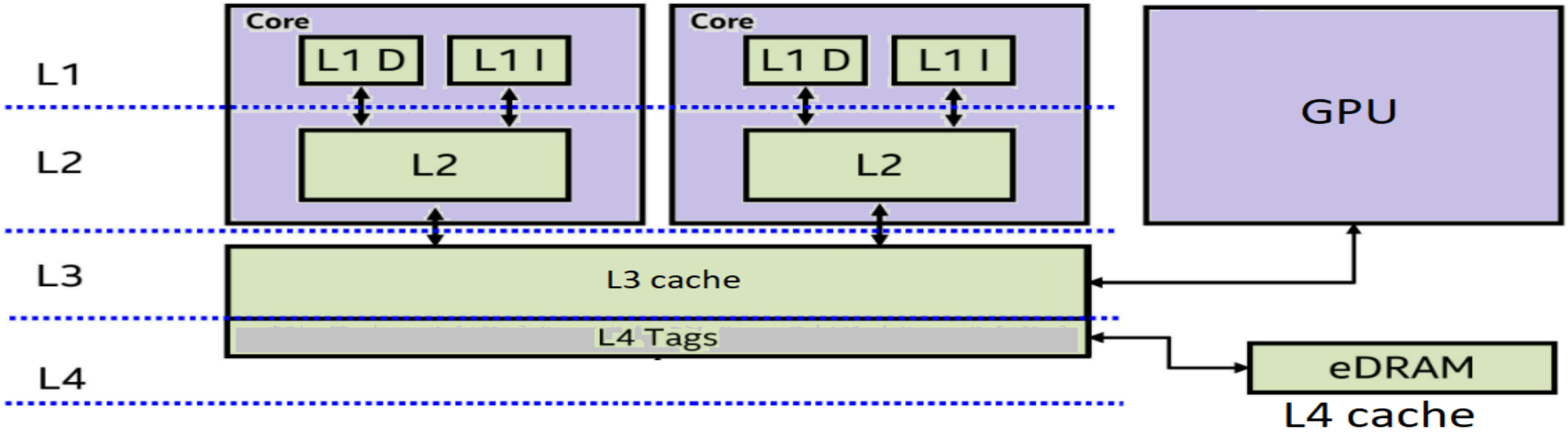


Figure: L4 cache along with placement of L1, L2 & L3 cache

Memory Hierarchy (revisited)

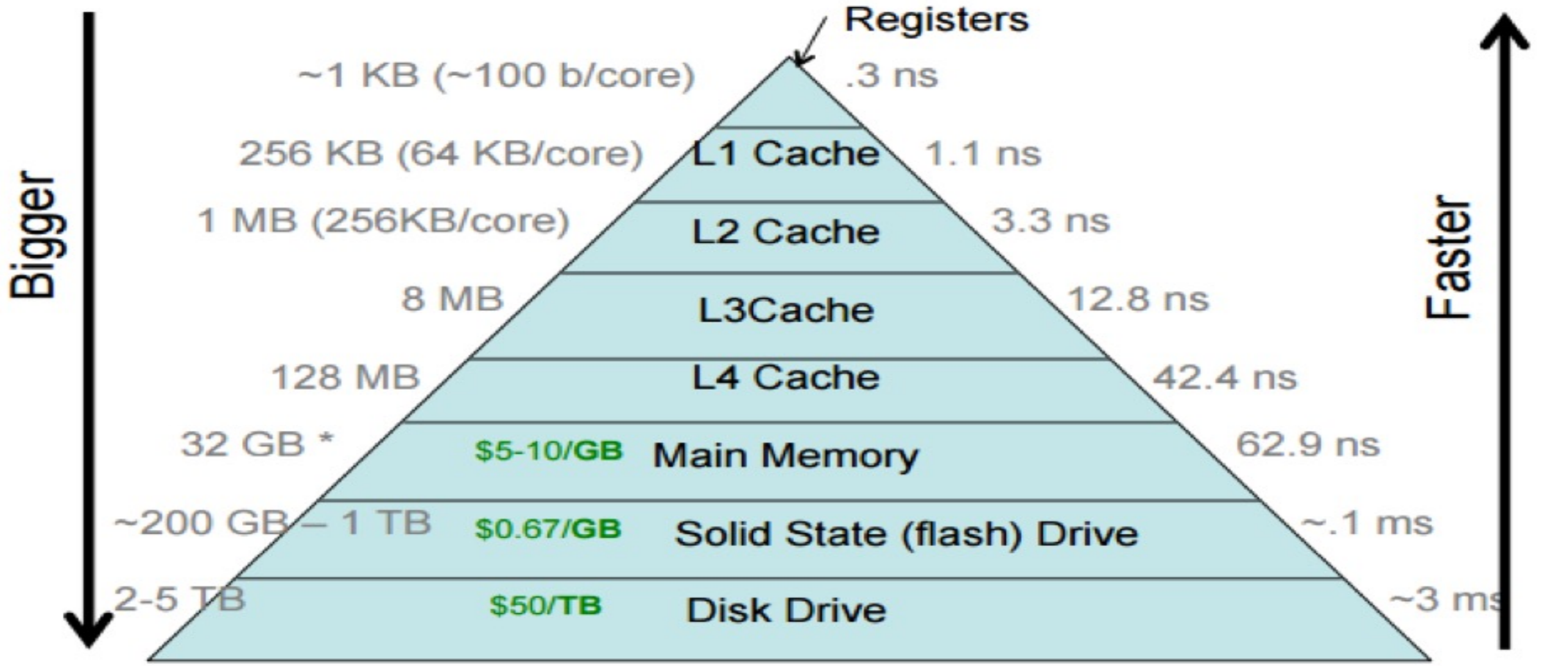


Fig. Memory hierarchy with access time and capacity (Ref:https://cs61.seas.harvard.edu/site/2022/Storage/#gsc.tab=0)

Caching Mechanism

- During each read query, it is first checked in CPU's **L1 cache**, if it is found then returned, else check **L2 cache** and so on till **L3 cache** or **L4 cache** (if present).
- If not found in any CPU caches, fetch from main memory and add to **L1, L2** and **L3 cache**.
- When a byte is read from main memory, not only the byte is returned but a block of bytes is returned known as the **cache line**.
- Generally the cache line is 64 bytes. Any read for a byte in the same cache line is read from either **L1, L2** or **L3 cache** because the cache line of 64 bytes is already cached in the CPU.
- *For e.g. if the cache lines are aligned such that 0–63 bytes is one cache line, 64-127 is next one and so on. Then if byte 10 is read, all bytes from 0 to 63 is transferred from RAM to CPU and cached. Next if request for byte 57 is made, it is already present in CPU cache since 0 <= 57 <= 63, hence read from cache.*
- A cache line read from main memory can occupy one of N cache slots i.e. if an existing cache line already occupies one of the N slots, then the incoming cache line can only occupy one of remaining N-1 slots.
- If all the N slots are occupied then we have to evict one of the N slots to make room for the new cache line. LRU policy is used to evict cache slots.

Cache Terminologies

- A **block** is a unit of data storage. Both the underlying storage and the cache are divided into blocks.
- In some caches, blocks have fixed size; in others, blocks have variable size.
- Each block of underlying storage has an **address**. Abstractly, address is represented as integers, though in a specific cache, an address might be a memory address, an offset into a file, a disk position, or even a web page's URL.
- The cache is divided into one or more **slots**. A cache slot can hold at most one block of underlying storage.
  - An **empty** slot is unused-it contains no data.
  - A **full** slot contains a cached version of a specified block of underlying storage.
  - Every full slot has an associated **tag**, which is the address of the corresponding block on underlying storage.

Cache Terminologies

- Some specialized terms are used for specific kinds of storage. For instance, a block in primary memory (or a slot in a processor cache) is called a **cache line**.
- Read caches must respond to user read requests for data at particular addresses.
- On each access, a cache typically checks whether the specified block is already loaded into a slot.
  - If it is not, the cache must first read the block from underlying storage into some slot.
  - This operation is called **filling** the cache slot.
  - Once the data is cached, the cache will return data from the slot.
- Write caches absorb user requests for writes at particular addresses.
  - A write cache will pass the written data to the underlying storage.
  - This operation is called **flushing** the corresponding slot.

Cache Terminologies

- A slot that has absorbed some writes, but not yet flushed, contains data that is "newer" than the underlying storage and is called **dirty**.
- A cache slot that is not dirty (that contains data not newer than the underlying storage) is called **clean**.
- A cache access is called a **hit** if the data is already loaded into a cache slot, and a **miss** otherwise.
- Cache hits are good because they are cheap.
- Cache misses are bad: they incur both the cost of accessing the cache and the cost of accessing the slower storage.
- High **cache hit rate** is desirable.
- The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the **hit ratio.**

Cache Efficiency and Cache Efficient Data Structure

- **Cache efficiency** refers to how effectively a program or system utilizes the cache memory to improve performance by minimizing costly accesses to slower levels of the memory hierarchy.

- A **cache-efficient data structure** is specifically designed to optimize the performance of a program by reducing the number of cache misses and improving cache locality when accessing memory.
- These data structures utilizes the characteristics of the memory hierarchy, particularly the cache, to minimize the costly delays of accessing data from slower levels of memory.

Cache Efficiency

Measuring Cache Efficiency

1. **Cache Hit Ratio**:
   - Ratio of cache hits to total memory accesses.
   - $Hit\ Ratio = Cache\ Hits / Total\ Accesses$
2. **Average Memory Access Time (AMAT)**:
   - $AMAT = Hit\ Time + Miss\ Rate \times Miss\ Penalty$
   - Lower AMAT indicates better cache efficiency.
3. **Cache Profiling Tools**:
   - Use tools like **valgrind** (e.g., cachegrind) or performance monitoring counters to analyze cache usage.

Applications of Cache Efficiency

- **High-performance computing (HPC)**: Optimizing simulations and scientific computations.
- **Databases**: Efficient query execution through cache-optimized indexing and storage structures.
- **Graphics and Gaming**: Real-time rendering and animation require efficient data access.
- **Operating Systems**: Efficient memory management and process scheduling.

Features of Cache Efficient Data Structure

- **Maximized Cache Locality**:
  - Data is arranged to take advantage of **spatial locality** (accessing memory locations close to each other) and **temporal locality** (reusing recently accessed data).
- **Optimized Memory Layout**:
  - Structures are laid out in memory to minimize **cache line conflicts** and ensure that frequently accessed elements are stored together.
- **Reduced Cache Misses**:
  - Access patterns are designed to reduce **compulsory**, **conflict**, and **capacity** cache misses.
- **Alignment with Cache Line Sizes**:
  - Data is grouped to fit within cache lines, making efficient use of the cache's bandwidth.
- **Predictable Access Patterns**:
  - Sequential or block-based access patterns are preferred to enable prefetching by modern CPUs.

Examples of Cache Efficient Data Structure

- **B-Trees and Cache-Optimized B-Trees**:
  - Nodes are grouped to fit within a cache line, reducing the number of cache misses during traversal.
- **Van Emde Boas (vEB) Tree**:
  - A recursive layout ensures that nodes in the same subtree are stored in contiguous memory locations, enhancing spatial locality.
- **Array-Based Structures**:
  - Arrays are inherently cache-efficient because elements are stored contiguously in memory.
- **Blocked Data Structures**:
  - Data is divided into blocks that fit within cache lines or cache sizes, such as blocked matrices used in numerical computations.
- **Cache-Optimized Hash Tables**:
  - Designed to reduce collisions and optimize probe sequences for better cache performance.

Principles of Cache Efficient Data Structure

1. **Utilize the Locality of Reference**:
   - **A. Spatial Locality**: Data near recently accessed memory locations are likely to be accessed soon.
     - When a memory location is accessed, nearby locations are also loaded into the cache.
     - Implementation:
       - Arrange data in contiguous memory locations
       - Access data sequentially whenever possible
       - Use array-based structures instead of linked structures when appropriate
     - **Example**: Storing binary tree nodes in a breadth-first layout in memory instead of using a pointer-based implementation.
   - **B. Temporal Locality**: Recently accessed data is likely to be accessed again.
     - Implementation:
       - Reuse data that's already in the cache
       - Design algorithms that access the same data repeatedly in short intervals.
       - Use buffering techniques to keep frequently used data in the cache.
     - **Example**: Matrix multiplication algorithms like **blocked matrix multiplication**, which break the matrix into smaller blocks that fit in cache and process them repeatedly.

Principles of Cache Efficient Data Structure

2. Optimize Cache Line Usage

- **Principle**: design the data structures in a way that minimizes cache misses and makes optimal use of the CPU cache.
- **Implementation**:
  - Minimize padding or gaps between data elements to prevent unused cache space.
  - Store related data contiguously in memory so that accessing one part of the data brings other useful data into the cache.
  - Ensure that different threads working on different data elements do not unintentional share a cache line (avoid false sharing)
  - Break loops into smaller chunks so that the working data set fits into the cache, reducing the need to reload data frequently.
  - Avoid accessing data structures with excessive indirection (e.g., pointer-chasing in linked lists).
- **Example**: Access arrays in a row-major order (in languages like C/C++) to utilize spatial locality.

Principles of Cache Efficient Data Structure

3. **Reducing Cache Misses**:
   - **Compulsory Misses**: Occur when data is accessed for the first time. Minimized by prefetching.
   - **Conflict Misses**: Occur when multiple data blocks map to the same cache line. Minimized by improving cache associativity.
   - **Capacity Misses**: Occur when the working set size exceeds the cache size. Minimized by optimizing the working set.
   - **Implementation**: Use smaller, cache-friendly working sets.
     - Design data structures to avoid frequent eviction of data from the cache.
   - **Example**: Use **cache-oblivious algorithms** that adapt to the cache size without explicit tuning.

4. **Batch and Sequential Access**
   - Access data in a predictable and sequential order to support prefetching mechanisms in modern CPUs.
   - **Implementation**:
     - Replace random-access patterns with linear scans.
     - Avoid data structures requiring non-sequential memory traversals, like traditional linked lists.
   - **Example**: Iterating over an array in order instead of using scattered pointers.

Principles of Cache Efficient Data Structure

5. Block Data Access

- Divide data into blocks that fit entirely in the cache to maximize locality and minimize cache eviction.
- **Implementation**:
  - Divide data into **chunks or tiles** for algorithms or traversal patterns.
  - Adjust block size based on the cache size of the target architecture.
- **Example**: Tiled processing for image processing or graph traversal.

6. Minimize Pointer Overhead

- Reduce the use of pointers that lead to non-contiguous memory access patterns.
- **Implementation**:
  - Use flat memory layouts or offsets instead of pointers.
  - When pointers are necessary, store them contiguously to improve prefetching.
- **Example**: Use an array of indices for adjacency lists in graph representation instead of separate linked lists.

Principles of Cache Efficient Data Structure

7. Adapt to Cache Hierarchy

- Design data structures that can work efficiently across different levels of the memory hierarchy (L1, L2, L3 caches).
- **Implementation**:
  - Use multi-level blocking for larger data sets to fit blocks into progressively larger caches.
  - Optimize for the smallest cache that can hold the working data set.
- **Example**: **Multi-level B-trees** used in databases to accommodate both CPU caches and disk storage.

8. Cache-Conscious Design

- Take cache properties, such as line size and associativity, into account when designing data structures.
- **Implementation**:
  - Align data structure size to cache line size.
  - Avoid designs that cause frequent cache line conflicts (e.g., multiple hot data items mapped to the same cache set).
- **Example**: **Cache-optimized hash tables** that reduce collisions and optimize probe sequences.

Principles of Cache Efficient Data Structure

9. Use of Cache-Aware and Cache-Oblivious Techniques

- Either design explicitly for a specific cache architecture (cache-aware) or use general techniques that work for all cache sizes (cache-oblivious).
- **Implementation**:
  - ***Cache-aware***: A data structure with knowledge of the cache parameters (details about the hardware e.g., cache size, block size, etc.).
  - The programmer adjusts the data structure or algorithm to minimize cache misses.
  - ***Cache-oblivious***: Use recursive divide-and-conquer techniques to access smaller subproblems that fit in the cache utilizing the cache hierarchy automatically.
  - It is designed without requiring any knowledge of the cache parameters.
  - Performs well across different architectures.
- **Example**: Cache-oblivious algorithms for sorting or matrix multiplication.

Cache Conscious Algorithms

Cache Conscious Algorithms

- **Cache-conscious algorithms** are specifically designed to improve performance by optimizing data access patterns to minimize cache misses and make better use of the CPU cache hierarchy.
- These algorithms aim to utilize spatial and temporal locality in data access, ensuring that data frequently accessed together is stored and processed together to reduce memory latency.
- Cache-conscious algorithms do not necessarily require **explicit knowledge** of cache parameters.
- Cache-conscious algorithms focus more on **data layout** and **access patterns** to ensure efficient memory usage.
- Cache-conscious algorithms may be **hardware-independent** or designed to work well across multiple architectures.

- **Analogy :** Think of cache-aware algorithms as a **tailored suit** (custom-fitted to a specific person, i.e., hardware) and cache-conscious algorithms as a **stretchable, universally fitting outfit** (designed to fit a wide range of people, i.e., hardware)

Designing Cache Conscious Algorithms

Following principles guideline for cache conscious algorithms effectively utilize cache memory, minimizing cache misses and improving performance.

1. Contiguous Allocation

- Memory allocated in a sequential block so that data elements are stored adjacently.
- Examples: Arrays, vectors, or custom array-like structures.
- Advantages: It Exploits spatial locality, as accessing one element likely brings nearby elements into cache.
- Drawback of Non-Contiguous Structures: Linked lists scatter nodes across memory, causing frequent cache misses.

- Implementation Example

Cache-Conscious Approach

```
int arr[100]; // Contiguous allocation
for (int i = 0; i < 100; i++) {
      sum += arr[i]; // Efficient memory
access due to sequential layout.
}
```

Cache Behavior: One cache line load may contain multiple elements, reducing cache misses.

Cache-Insensitive Approach (Linked List)

```
struct Node {
    int data;
    struct Node* next;
    };
    struct Node* head; // Memory scattered,
    higher cache misses.
```

Designing Cache Conscious Algorithms

2. Minimize Indirection

- **Indirection**: Accessing data via multiple references or pointers.
- **Problem with Indirection**: Each pointer dereference may fetch unrelated cache lines.
- **Impact**: Algorithms relying on linked structures experience frequent cache misses.
- **Solution**: Replace pointers with array indices or flat structures.
- **Example:Traditional Linked List**

```
struct Node {
    int data;
    struct Node* next;
};
```

A traditional linked list uses pointers to connect nodes. Since nodes are dynamically allocated, they are often scattered across non-contiguous memory locations. This leads to frequent cache misses as each pointer dereference accesses a potentially distant memory block.

- **Cache-Conscious Alternative (Index-Based)**

```
struct CompactNode {
        int data;
        int nextIndex; // Uses index instead of pointer
};
```

- **Benefit**: Index-based access keeps elements in contiguous memory for better spatial locality.

Designing Cache Conscious Algorithms

2. Minimize Indirection

Benefits of Contiguous Memory Access

1. **Array-Based Storage**:
   1. Nodes are stored in a single, contiguous array.
   2. Contiguous memory access improves **spatial locality**, as adjacent elements are likely to be loaded into the cache together.

```
CompactNode nodes[100]; // Contiguous array storage
```

2. **Reduced Cache Misses**:
   1. Accessing *nodes[current.nextIndex]* accesses the next element in a predictable memory location.
   2. This minimizes cache misses compared to following pointers that may lead to arbitrary memory locations.
3. **Preloading and Prefetching**:
   1. CPUs can efficiently prefetch contiguous memory locations into cache.
   2. With *nextIndex*, the CPU preloads future nodes, improving data availability.

Designing Cache Conscious Algorithms

2. Minimize Indirection

- **Cache Line Utilization**
  1. **Structure Packing**:
     - Using an int nextIndex instead of a pointer typically requires less memory.
     - For example, a 32-bit system pointer is 4 bytes, whereas int is often the same size but avoids the overhead of complex memory lookups.
  2. **Compact Representation**
     - **Memory Efficiency**: Reducing indirection and storing only minimal metadata (*nextIndex* instead of full pointer management) keeps data structures compact, fitting more nodes within a single cache line.

Designing Cache Conscious Algorithms

3. Align with Cache Lines

- **Structure Alignment:** Arrange data so that fields in a structure align with cache line boundaries.
- **Cache Line Size**: Often 64 bytes; misaligned structures span multiple cache lines, causing excessive cache loads.
- **Padding**: Add unused space to align fields properly.
- **Compiler Directives**: Use alignas or similar keywords for alignment.

Structure definition with misalignment

```
struct Unaligned {
        char c;  // 1 byte
        int x;  // 4 bytes, starts at offset 1 (misaligned)
        };
```

char c occupies **1 byte** at **offset 0**.int x is **4 bytes** but starts at **offset 1** (immediately after c), misaligning it. A typical cache line fetches memory in 64-byte blocks. Since x is misaligned, part of it may reside in one cache line and the rest in another.

Implications

•**Unaligned x leads to extra cache misses**. Accessing **x** may require fetching two cache lines:

- One from the cache block containing the first part of x.
- Another from the cache block containing the rest of x.

Designing Cache Conscious Algorithms

3. Align with Cache Lines

- **Structure definition with alignment**

```
struct Aligned {
    char c;          // 1 byte
    char padding[3]; // 3 bytes padding to align next field
    int x;           // Starts at a 4-byte boundary
};
```

- *char c* occupies **1 byte** at **offset 0**.*char padding[3]* occupies **3 bytes** at **offsets 1 to 3**, aligning the next field to a 4-byte boundary.
- *int x* starts at **offset 4**, ensuring it aligns with a **4-byte boundary** (a multiple of 4).

Designing Cache Conscious Algorithms

3. Align with Cache Lines

What Is Alignment?

- Alignment refers to the way data is positioned in memory so that its address is a multiple of a word size (often 4 or 8 bytes) or the size of a cache line (commonly 64 bytes). Proper alignment ensures data elements start and end within a single cache line, avoiding costly extra memory accesses.

Cache Line Basics

- **Cache Line Size**: Typically 64 bytes in modern processors.
- **Memory Blocks**: Memory is divided into blocks (or cache lines) that the CPU loads into cache.
- **Alignment of Data**: If a structure or variable is not aligned with cache line boundaries, data can straddle two cache lines, requiring additional memory fetches.

Designing Cache Conscious Algorithms

3. Align with Cache Lines

How Alignment Works in Cache

- When a processor fetches data:

1. **Aligned Data**:
    1. Suppose a cache line is 64 bytes, and a data element begins at address 0x100. The entire structure fits within one cache line.
    2. Example: Loading a structure with int key and float value that fits entirely in one cache line reduces latency.
2. **Unaligned Data**:
    1. If a structure spans address 0x104 to 0x108 but crosses to 0x110, two cache lines must be fetched.
    2. Example:
        1. Cache line 1: Data from 0x100 to 0x10F (partially filled).
        2. Cache line 2: Data from 0x110 to 0x11F.

Designing Cache Conscious Algorithms

3. Align with Cache Lines

- **Padding and Alignment Techniques**
  - **Manual Padding**: Adding unused space to align fields.

```
struct Padded {
    char c;              // 1 byte
    char padding[3];     // Padding for alignment
    int x;               // Aligned to 4-byte boundary
};
```

  - **Compiler Directives**:
  - alignas(N) in C++ or __attribute__((aligned(N))) in GCC can enforce specific alignment.

- **Why Alignment Matters**

1. Fitting Within Cache Lines

1. Cache lines (usually 64 bytes) are the smallest chunks of memory transferred between main memory and cache.
2. Aligned fields avoid spanning multiple cache lines, reducing the number of memory accesses.

2. CPU Efficiency

1. Most modern CPUs are optimized to fetch and operate on aligned data. Misalignment causes performance penalties as the CPU must handle split reads or writes.

3. Align with Cache Lines

Example: Structure of Arrays (SOA)

- In SOA, data for each field is stored in separate, contiguous arrays.

```
struct SOA {
    int data[100];      // Array for integers
    float weight[100];  // Array for floats
};
```

- **Explanation**:
  - Each array stores a specific field for all objects.
  - Accessing data sequentially for one field benefits from spatial locality, as elements are stored contiguously in memory.
  - Suitable for operations that process individual fields (e.g., all weights).
- **Access Pattern Example**

```
for (int i = 0; i < 100; i++) {
    process(SOA.weight[i]);  // Efficient cache access due to spatial locality
}
```

3. Align with Cache Lines

Example: Array of Structure (AOS)

- In AOS, each element is a structure, and these structures are stored in an array.

```
struct Element {
    int data;
    float weight;
};
Element array[100];  // Array of structures
```

- **Explanation**:
  - Each object (element) is stored contiguously, but different fields of the object may be scattered within each structure.
  - Suitable for operations where you process complete objects (data and weight together).
- **Access Pattern Example**

```
for (int i = 0; i < 100; i++) {
    process(array[i].data, array[i].weight); // Accessing both fields per iteration
}
```

- **Choosing SOA or AOS**
  - **SOA**: Prefer for **cache-conscious** algorithms that operate on individual fields, such as SIMD or parallel processing.
  - **AOS**: Use for general-purpose, object-oriented designs where all fields are accessed together.

Designing Cache Conscious Algorithms

4. Compact Representations

- **Compact Data Structures**: Use minimal memory to store only necessary fields.
- **Reduces Memory Footprint**: More data fits in cache, lowering cache misses.
- **Avoid Wasted Space**: Combine fields or use smaller data types when possible.
- **Example: Inefficient**

```
struct Color {
    int red;    // 4 bytes
    int green;  // 4 bytes
    int blue;   // 4 bytes
};
```

- **Compact Version**

```
struct CompactColor {
    unsigned char red;   // 1 byte
    unsigned char green; // 1 byte
    unsigned char blue;  // 1 byte
};
```

- **Impact**: More *CompactColor* elements fit into the same cache line, reducing misses.
- The compact version uses **3 bytes** instead of **12 bytes**, fitting more color entries into a single cache line.
- Analyze structures to eliminate fields that are not needed or can be derived from others.
- Use memory alignment and structure reordering to minimize wasted space due to padding.

Designing Cache Conscious Algorithms

5. Partitioning

- Partitioning refers to dividing large data structures into smaller, manageable chunks or blocks that fit efficiently within cache lines.
- It improves memory locality, reduces cache misses, and enhances performance, particularly for large-scale data processing.
- Partitioning ensures that each accessed portion of the data resides in cache for faster access.

Reasons for Partitioning

1. **Cache Line Size Limitation**
   - Cache lines in modern CPUs are typically **64 bytes**. Data exceeding this size may require multiple memory accesses.
2. **Cache Capacity**
   - L1, L2, and L3 caches have limited sizes. Partitioning helps by working on smaller subsets of data that fit into the cache hierarchy, reducing **capacity misses**.

Partitioning Techniques

1. **Blocking (Tiling)**
   - Blocking divides large arrays or matrices into smaller blocks that can fit into cache.
   - It is common in **matrix multiplication** and similar computations.

5. Partitioning (Partitioning Techniques)

Example: Matrix Multiplication without Blocking

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Issues with Cache Usage

Rows and columns are repeatedly accessed, causing frequent cache misses when matrices are too large.

Improved Code Using Blocking

```
#define BLOCK_SIZE 64
for (int i = 0; i < n; i += BLOCK_SIZE){
    for (int j = 0; j < n; j += BLOCK_SIZE){
        for (int k = 0; k < n; k += BLOCK_SIZE){
            for (int ii = i; ii < i + BLOCK_SIZE; ii++){
                for (int jj = j; jj < j + BLOCK_SIZE; jj++){
                    for (int kk = k; kk < k + BLOCK_SIZE; kk++){
                        C[ii][jj] += A[ii][kk] * B[kk][jj];
                    }
                }
            }
        }
    }
}
```

Explanation of Blocking Benefits

- Each **BLOCK_SIZE × BLOCK_SIZE** sub-matrix is loaded into cache, avoiding frequent memory accesses for the entire array.
- **Spatial locality** is improved as elements within each block are accessed together.

Designing Cache Conscious Algorithms

5. Partitioning (Partitioning Techniques)

- **Partitioning for Divide and Conquer**
  - Partitioning is also integral to algorithms like **Quicksort** and **Merge Sort**.
  - **Quicksort Example**
    - **Partition Step**: Divides an array into smaller sections around a pivot, ensuring recursive steps work on smaller chunks.
    - Partitioning in Quicksort helps because:
      - Small partitions fit into cache better than processing the entire array at once.
      - Recursive partitioning narrows the active working set of data, keeping cache usage efficient.
- **Data Partitioning for Hash Tables**
  - When designing **cache-aware hash tables**, partitioning can be applied:
  - **Bucket Partitioning**: Divide the key space into buckets that fit within cache.
  - Each partition represents a smaller, cache-friendly subset of the table.

Designing Cache Conscious Algorithms

5. Partitioning

Benefits of Partitioning

1. **Improves Cache Line Efficiency**
   - By keeping partitions within a single cache line, memory accesses are localized, reducing the number of cache misses.
2. **Reduces Cache Pollution**
   - Operating on smaller partitions limits the need to evict other cache-resident data unnecessarily.
3. **Better Prefetching**
   - Smaller, sequential partitions are easier for the CPU to prefetch compared to arbitrary access patterns.

Cache-Oblivious Algorithms

Cache-Oblivious Algorithms

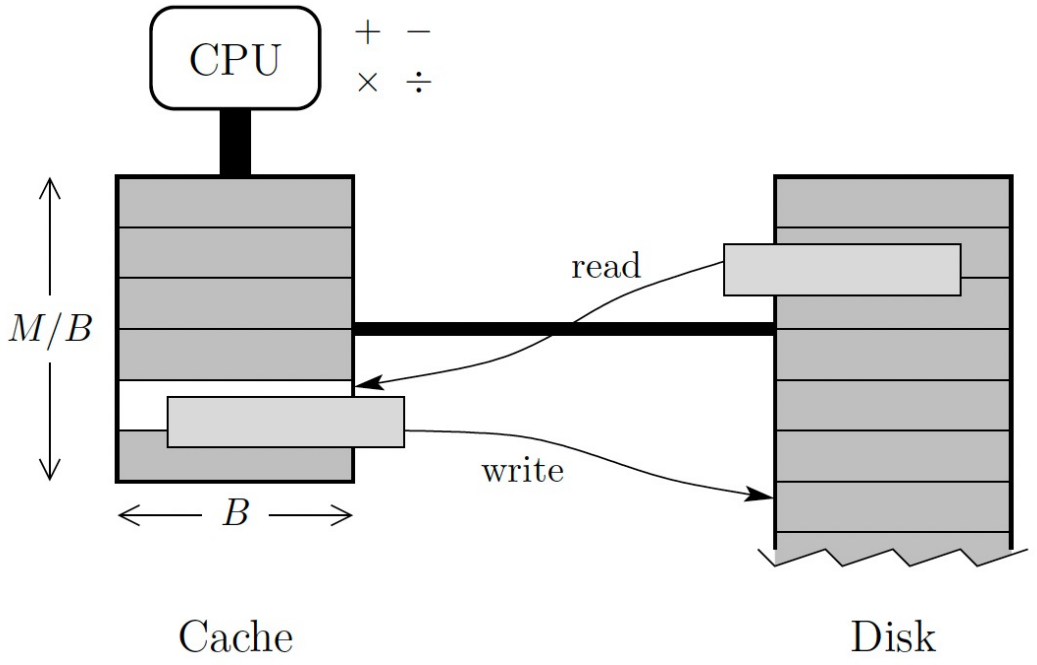
- **Cache oblivious** is a way of achieving the effectiveness of algorithms in arbitrary memory hierarchies without the use of multi-level memory models, since cost, running time are crucial things in multilevel memory hierarchies.
- Cache Oblivious Models are built in a way so they can be independent of constant factors, like the size of the cache memory.
- It is designed for memory hierarchy which separates computer storage into a hierarchy based on response time. Cache oblivious algorithms are asymptotic only when they ignore the constant factors.
- Cache oblivious notion is to design **cache-efficient** and **disk-efficient** algorithms and data structures.
- Cache-oblivious algorithms perform well on a **multilevel memory hierarchy** without knowing any parameters of the hierarchy, but to have the consciousness of their existence. This means that they just need to ignore the constant factors to work efficiently.

Cache-Oblivious Algorithms

External Memory Model

- To contrast with the cache-oblivious model, let us considered the standard model of a two-level memory hierarchy with block transfers which is refers as the external-memory model, the I/O model, the disk access model, or the cache-aware model.
- The model defines a computer as having two levels (as in fig below: A two-level memory hierarchy shown with one block replacing another.)

1. the cache which is near the CPU, cheap to access, but limited in space; and
2. the disk which is distant from the CPU, expensive to access, but nearly limitless in space.



Cache-Oblivious Algorithms

External Memory Model

- It has a **2-level memory hierarchy** which consists of cache (M where M ≥B) and blocks of data (B) which transfer between the cache and the disk.
- The disk is split up into blocks of B elements, and accessing one of them on disk, copies its entire block to the cache.
- The access time makes a whole lot of difference between the cache and disk. The general idea behind the read operation is to read the formerly stored data and the write operation stores a new value in memory.
- When the CPU or processor accesses a memory location, if that block of data is already in the cache, then it is known as **cache-hit,** cache-cost to access it is 0.
- And if it is not in cache already, then it is a **cache-miss.**
- This memory is then accessed from the disk and transferred to the cache (M). The cache cost would be 1 for this case.
- Before fetching a block from disk when the cache is already full, the algorithm must decide which block to evict from cache. (Cache replacement algorithms like LRU, Optimum, FIFO etc)

Cache-Oblivious Algorithms

Cache-Oblivious Model

- The major principle idea of cache-oblivious model is to design external-memory algorithms without knowing B and M.
- Consequences of Cache-oblivious model
  - If a cache-oblivious algorithm performs well between two levels of the memory hierarchy (nominally called cache and disk), then it must automatically work well between any two adjacent levels of the memory hierarchy.
  - If the number of memory transfers is optimal up to a constant factor between any two adjacent memory levels, then any weighted combination of these counts (with weights corresponding to the relative speeds of the memory levels) is also within a constant factor of optimal.
  - Another, more practical consequence is self-tuning. Typical cache-efficient algorithms require tuning to several cache parameters which are not always available from the manufacturer and often difficult to extract automatically. Parameter tuning makes code portability difficult.
- Unlike external-memory model, algorithms in the cache-oblivious model cannot explicitly manage the cache (issue block-read and block-write requests), since the block and cache sizes are unknown.
- This automatic-management model more closely matches physical caches other than the level between main memory and disk: which block to replace is normally decided by the cache hardware according to a fixed page replacement strategy, not by a general program.

Cache-Oblivious Algorithms

Associative Cache:

- The cache is generally distinguished by three factors-
  - **Associativity(A)-** The associativity **A** specifies the no. of different frames or lines(B) resided to the main memory. If the block from the main memory(Disk) can reside in any frame or line then the associativity is fully satisfied.
  - **Block(B)-** Block is the part of the minimum memory access size.
  - **Capacity(C)-** Capacity is part of the minimum memory access size.
- A cache is equal to **C** bytes. But due to physical constraint, the cache is divided into cache frames of size **B**. These factors can have an effect on a particular cache.
- When a memory address is not in the cache (**cache-miss**), we must bring a block or line to the cache and it should decide where it should be mapped to in the cache. The Cache model presumes that any cache line or block can be mapped to any location in the cache memory. Most caches are 2-way, 4-way, 8-way, 16-way associative.

Cache-Oblivious Algorithms

What is set-associative mapping?

- It is the mapping of a particular block of main memory(Disk, in our case) to a cache set. This occurs when a cache-miss happens.
- Cache lines are assembled into sets where each set contains **k** number of lines.
- Then, a block of main memory is mapped to a set of cache. We can map to any freely available cache-line from the main memory.
- The set of the cache to which a unique main-memory can be mapped is given by-

Cache set number = [Main Memory block address] modulo [number of sets in the cache]

Cache-Oblivious Algorithms

Optimal Cache Replacement Policy:

- When a cache miss occurs, a new cache line is mapped from the main memory to the cache.
- But before fetching a block from the main memory if the cache is already full, there must be a way to evict the current existing line from the cache.
- There is no optimal replacement in reality because it requires us to know the future cache-miss which is unpredictable.
- In real-world caches do not know the future, and employ more realistic page replacement strategies such as evicting the least-recently-used block (LRU) or evicting the oldest block (FIFO).

Cache-Oblivious Algorithms

Why use Cache-Oblivious Algorithm?

- The cache-oblivious model is invented so that a data structure can reflect some properties of cache consciousness.
- Cache oblivious algorithm inherits some properties of register machines which usually consists of a small amount of fast storage and random access machine model.
- But they have their own discrepancies. The register is a constantly accessible location available to the processor of a computer system. They consist of small memories of fast storage. These small memories are used when a computer loads data from a sizeable memory into registers to do arithmetic operations.
- There is a reason why the cache-oblivious are cache conscious; i.e. because of a hierarchy of cache memory which is inspected through the cache-oblivious model.
- The principle which holds all of them together is to design external memory algorithms without knowing the size of the cache and the blocks in the cache.

Cache-Oblivious Algorithms

Tall Cache Assumption

- It is common to assume that a cache is taller than it is wide, that is, the number of blocks, M/B, is larger than the size of each block, B. Often this constraint is written $M = \Omega(B^2)$, **where**,
  M is the size of the cache.
  B is the size of the cache line.
  Ω symbol is used to represent the lower bound of the algorithm or data. And that is the top speed any algorithm can get to.

Cache-Oblivious Algorithms

Example of Cache-Oblivious Algorithm

Scanning

Traversal and Aggregates

- Suppose we need to traverse all of the elements in a set, e.g., to compute an aggregate (sum, maximum, etc.). On a flat memory hierarchy (uniform-cost RAM), such a procedure requires $\Theta(N)$ time for N elements. In the external-memory model, if we store the elements in $\lceil N/B \rceil$ blocks of size B, then the number of blocks transfers is $\lceil N/B \rceil$.
- To achieve a similar bound in the cache-oblivious model, we can lay out the elements of the set in a contiguous segment of memory, in any order, and implement the N-element traversal by scanning the elements one-by-one in the order they are stored.
- This layout and traversal algorithm do not require knowledge of B (or M).

- Scanning N elements stored in a contiguous segment of memory costs at most $\lceil N/B \rceil + 1$ memory transfers.

Cache-Oblivious Algorithms

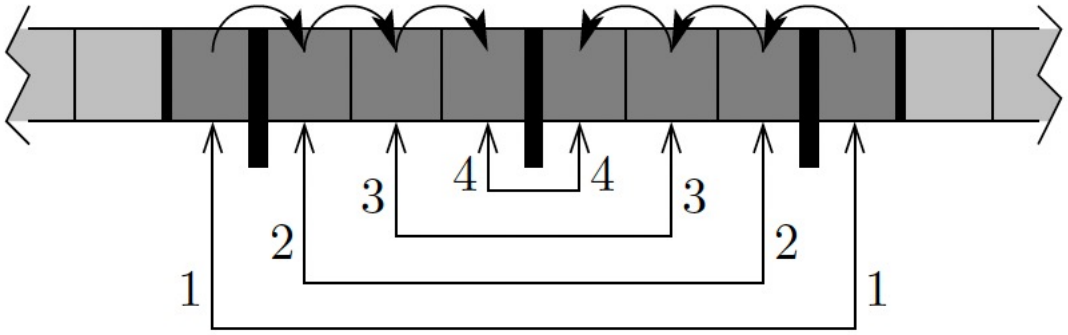
Array Reversal

- Array reversal is reversing the elements in an array without any extra storage. Bentley's array-reversal algorithm constructs two parallel scans from both sides, each from the opposite ends of the array. At each scan or step, the two elements trade their position to each other.
- **Advantages:**
  - The cache-oblivious algorithm uses the same number of memory reads as a single scan, provided that $M \geq 2B$,
  - The algorithm helps to implement the N-element traversal by scanning the elements one by one in the order they are stored.
- **Limitation:**
  - The algorithms perform worse than the RAM-based and cache-aware algorithms when data are stocked into the main memory.

Cache-Oblivious Algorithms

Bentley's array-reversal algorithm

```
void reverse_array(int arr[], int n) {
    int left = 0, right = n - 1;
    while (left < right) {
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
        left++;
         right--;
    }
}
```



Cache-Oblivious Algorithms

Matrix Multiplication

- The goal is to compute the product of two $n \times n$ matrices A and B to produce a matrix C, where:

  $C[i][j] = \sum_{k=0}^{n-1} A[i][k] \times B[k][j]$

- A naive matrix multiplication algorithm would access elements of A row by row(*stored in row-major order*) and B column by column(stored in column-major order), leading to cache inefficiencies because data from B is accessed non-contiguously. This can result in frequent cache misses.
- A cache-oblivious approach recursively divides the matrices into smaller submatrices (blocks), similar to how cache-conscious methods work, but without tuning for the block size. Dividing the problem down into smaller subproblems ensures that submatrices fit into cache, minimizing cache misses.

$$\begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix}$$

$$= \begin{pmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{pmatrix}$$

- **Approach**:
  1. **Recursive Decomposition:**
     - Reduce an N×N multiplication problem down to eight (N/2) × (N/2) multiplication subproblems.

Cache-Oblivious Algorithms

Matrix Multiplication

2. **Matrix Multiplication on Submatrices**:
   - Perform matrix multiplication on these smaller submatrices recursively.
3. **Combine the Results**:
   - Use the results of the submatrix multiplications to compute the final product.

- To make small matrix blocks fit into blocks or main memory, the matrix is not stored in row-major or column-major order, but rather in a recursive layout.
- Each matrix **A** is laid out so that each block $\mathbf{A_{11}}$,$\mathbf{A_{12}}$,$\mathbf{A_{21}}$,$\mathbf{A_{22}}$ occupies a consecutive segment of memory, and these four segments are stored together in an arbitrary order.

Matrix Multiplication (Pseudocode)

```
function multiply(A, B, C, n):
    if n == 1:
        C[0][0] += A[0][0] * B[0][0]
    else:
// Split matrices A, B, and C into 4 smaller submatrices each
        A11, A12, A21, A22 = split(A, n)
        B11, B12, B21, B22 = split(B, n)
        C11, C12, C21, C22 = split(C, n)

        // Recursively multiply submatrices
        multiply(A11, B11, C11, n/2)
        multiply(A12, B21, C11, n/2)
        multiply(A11, B12, C12, n/2)
        multiply(A12, B22, C12, n/2)
        multiply(A21, B11, C21, n/2)
        multiply(A22, B21, C21, n/2)
        multiply(A21, B12, C22, n/2)
        multiply(A22, B22, C22, n/2)
```

```
function split(M, n):
//Split matrix M of size n x n into 4 submatrices
    mid = n / 2
    A11 = M[0..mid-1, 0..mid-1]
    A12 = M[0..mid-1, mid..n-1]
    A21 = M[mid..n-1, 0..mid-1]
    A22 = M[mid..n-1, mid..n-1]
    return A11, A12, A21, A22
```

Base case: If the matrix size is 1×1, multiply the two scalar values.

Recursive step: Split the matrices A, B, and C into 4 submatrices of half the size, and recursively perform matrix multiplication on these smaller submatrices.

Split function: Divides a matrix of size n×n into 4 smaller matrices. Each submatrix will be of size n/2×n/2.

Cache-Oblivious Algorithms

Matrix Multiplication

How Cache-Obliviousness Works

- The algorithm ensures that submatrices are smaller and fit into the cache at different levels of the memory hierarchy.
- **Spatial locality**: When accessing a submatrix, the data from that submatrix is likely to remain in the cache for further access.
- **Temporal locality**: As recursion unfolds, the algorithm exploits the cache for recently accessed data, leading to fewer cache misses.

Performance Considerations

- The algorithm has the same asymptotic complexity as the standard matrix multiplication algorithm, $O(n^3)$.
- The primary improvement is in reducing cache misses, which speeds up the computation in practice by minimizing memory access time.

Complexity Calculation

Cache-Oblivious Matrix Multiplication

The cache-oblivious approach uses a **divide-and-conquer strategy** to recursively divide the matrices into smaller submatrices. The key idea is to exploit spatial and temporal locality by operating on submatrices that fit into the cache.

Recursive Division

1. Divide each matrix into four quadrants:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

2. Recursively compute the products of the submatrices:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Recurrence Relation

Let $T(n)$ be the time complexity of multiplying two $n \times n$ matrices using the cache-oblivious approach. The recurrence relation is derived as follows:

1. The problem is divided into 8 subproblems (matrix multiplications) of size $n/2 \times n/2$.
2. The additions of submatrices take $O(n^2)$ time.

Thus, the recurrence relation is:

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

Solving the Recurrence Relation

We solve the recurrence using the **Master Theorem** or by **expansion**. Let's use the expansion method.

Step 1: Expand the Recurrence

Start with the recurrence:

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

Assume $O(n^2) = cn^2$, where c is a constant. Then:

$$T(n) = 8T\left(\frac{n}{2}\right) + cn^2$$

Now, expand $T(n/2)$:

$$T\left(\frac{n}{2}\right) = 8T\left(\frac{n}{4}\right) + c\left(\frac{n}{2}\right)^2$$

$$T\left(\frac{n}{2}\right) = 8T\left(\frac{n}{4}\right) + c\frac{n^2}{4}$$

Complexity Calculation

Substitute this back into the original equation:

$$T(n) = 8\left[8T\left(\frac{n}{4}\right) + c\frac{n^2}{4}\right] + cn^2$$

$$T(n) = 64T\left(\frac{n}{4}\right) + 8 \cdot c\frac{n^2}{4} + cn^2$$

$$T(n) = 64T\left(\frac{n}{4}\right) + 2cn^2 + cn^2$$

$$T(n) = 64T\left(\frac{n}{4}\right) + 3cn^2$$

Step 2: Continue Expanding

Expand $T(n/4)$:

$$T\left(\frac{n}{4}\right) = 8T\left(\frac{n}{8}\right) + c\left(\frac{n}{4}\right)^2$$

$$T\left(\frac{n}{4}\right) = 8T\left(\frac{n}{8}\right) + c\frac{n^2}{16}$$

Substitute this back into the equation:

$$T(n) = 64\left[8T\left(\frac{n}{8}\right) + c\frac{n^2}{16}\right] + 3cn^2$$

$$T(n) = 512T\left(\frac{n}{8}\right) + 64 \cdot c\frac{n^2}{16} + 3cn^2$$

$$T(n) = 512T\left(\frac{n}{8}\right) + 4cn^2 + 3cn^2$$

$$T(n) = 512T\left(\frac{n}{8}\right) + 7cn^2$$

Step 3: Generalize the Pattern

After k expansions, the equation becomes:

$$T(n) = 8^kT\left(\frac{n}{2^k}\right) + cn^2\left(1 + 2 + 3 + \cdots + k\right)$$

The sum $1 + 2 + 3 + \cdots + k$ is $\frac{k(k+1)}{2}$, so:

$$T(n) = 8^kT\left(\frac{n}{2^k}\right) + cn^2 \cdot \frac{k(k+1)}{2}$$

Step 4: Determine the Base Case

The recursion stops when the matrix size is 1×1, i.e., $\frac{n}{2^k} = 1$. Solving for k:

$$\frac{n}{2^k} = 1$$

$$2^k = n$$

$$k = \log_2 n$$

Step 5: Substitute k Back into the Equation

Substitute $k = \log_2 n$ into the generalized equation:

$$T(n) = 8^{\log_2 n} T(1) + cn^2 \cdot \frac{(\log_2 n)(\log_2 n + 1)}{2}$$

Simplify $8^{\log_2 n}$:

$$8^{\log_2 n} = (2^3)^{\log_2 n} = 2^{3 \log_2 n} = n^3$$

Thus:

$$T(n) = n^3 T(1) + cn^2 \cdot \frac{(\log_2 n)(\log_2 n + 1)}{2}$$

Since $T(1)$ is a constant (time to multiply two 1×1 matrices), we can write:

$$T(n) = O(n^3) + O(n^2 \log^2 n)$$

The dominant term is $O(n^3)$, so:

$$T(n) = O(n^3)$$

Basic Matrix Multiplication Complexity

- **Naive matrix multiplication** has time complexity:

  $$O(N^3)$$

  but causes $O(N^3)$ **cache misses** because it accesses elements row-wise and column-wise inefficiently.

- **Blocked matrix multiplication** reduces cache misses using **explicit tiling**, but requires knowledge of B and M.

- **Cache-oblivious matrix multiplication** achieves cache efficiency **without knowing cache parameters** by using recursive partitioning.

Complexity Calculation

Cache Complexity

Let *Q(N)* represent the **number of cache misses** for multiplying two N×N matrices.

Recursive Matrix Multiplication Steps

1.Divide:

1. The N×N matrices A, B, and C are split into **four** (N/2)×(N/2) submatrices.
2. **8 recursive calls** are made

2.Conquer:

1. Each recursive call multiplies smaller submatrices.

3.Combine:

1. The resulting submatrices are summed element-wis

This leads to the recurrence:

$$Q(N) = 8Q(N/2) + O(N^2/B)$$

where:

- $8Q(N/2)$: Recursively multiplying 8 smaller matrices.
- $O(N^2/B)$: Cache misses from scanning matrix elements.

Why is Cache Miss Complexity $O(N^2/B)$ for Scanning Matrix Elements?

Case 1: Row-wise Traversal

- When scanning an $N \times N$ **matrix row-wise**, elements are stored **contiguously in memory.**
- Each cache block holds B **elements.**
- **Number of cache misses per row:**

$$N/B$$

(since each cache miss brings in a whole block of B elements).

- **Total misses for all N rows:**

$$N \times (N/B) = N^2/B$$

3. Solving the Recurrence Using Recursion Tree

We expand the recurrence step by step.

Step 1: Expanding the Recurrence

Expanding for a few levels:

First Expansion

$$Q(N) = 8Q(N/2) + O(N^2/B)$$

Second Expansion

$$Q(N/2) = 8Q(N/4) + O((N/2)^2/B)$$

Substituting:

$$Q(N) = 8[8Q(N/4) + O(N^2/4B)] + O(N^2/B)$$

$$= 64Q(N/4) + 8O(N^2/4B) + O(N^2/B)$$

Third Expansion

$$Q(N/4) = 8Q(N/8) + O(N^2/16B)$$

Substituting:

$$Q(N) = 512Q(N/8) + 64O(N^2/16B) + 8O(N^2/4B) + O(N^2/B)$$

Step 2: Identifying the Pattern

At level k, we have:

$$Q(N) = 8^k Q(N/2^k) + \sum_{i=0}^{k-1} 8^i O\left(\frac{N^2}{2^{2i}B}\right)$$

Since the recursion stops when $N/2^k = 1$, solving for k:

$$N = 2^k$$

$$k = \log_2 N$$

Step 3: Summing Up the Cache Misses

From the recurrence tree:

$$Q(N) = \sum_{i=0}^{\log_2 N} 8^i O\left(\frac{N^2}{2^{2i}B}\right)$$

$$= O\left(\frac{N^2}{B} \sum_{i=0}^{\log_2 N} \frac{8^i}{4^i}\right)$$

Since $8^i/4^i = 2^i$, the summation gives:

$$Q(N) = O\left(\frac{N^2}{B}\sum_{i=0}^{\log_2 N} 2^i\right)$$

Using the geometric sum formula:

$$\sum_{i=0}^{\log_2 N} 2^i = 2^{\log_2 N+1} - 1 = 2N - 1 = O(N)$$

Thus:

$$Q(N) = O\left(\frac{N^3}{B}\right)$$

4. Refining the Cache Complexity

To improve accuracy, we consider the effect of **cache size** M.

Since recursive calls **operate on submatrices**, an optimal recursion depth is reached when the submatrices fit into cache.

This occurs when:

$$(N/\sqrt{M}) \times (N/\sqrt{M}) \text{ fits in cache}$$

which introduces an extra factor $\sqrt{M}$ in the denominator.

Thus, the final cache complexity is:

$$Q(N) = O\left(\frac{N^3}{B\sqrt{M}} + \frac{N^2}{B}\right)$$

where:

- $O(N^2/B)$: Cache misses from scanning.
- $O(N^3/B\sqrt{M})$: Cache misses from multiplication.

Profiling Tools for Analyzing Memory and Cache Efficiency

Profiling Tools

- Profiling tools are essential for analyzing memory and cache efficiency in software applications.
- They provide insights into memory usage patterns, allocation and deallocation behavior, and cache hit/miss rates, enabling developers to identify and optimize performance bottlenecks

Key profiling tools

- **Valgrind:** A powerful open-source suite for debugging and profiling C and C++ applications. It includes tools like Memcheck for detecting memory errors and Callgrind for call graph and cache profiling
- **Intel VTune Profiler:** A comprehensive performance analysis tool from Intel that provides deep insights into hardware-level performance, including memory and cache usage.
- **CLR Profiler:** A tool specifically designed for profiling .NET applications, focusing on memory allocation and garbage collection.
- It helps identify areas where memory usage can be reduced and garbage collection can be optimized.

Profiling Tools

Key profiling tools

- **JProfiler:** A commercial Java profiler that offers a wide range of features, including memory profiling, thread analysis, and performance monitoring.
  - It provides detailed information about memory usage and allocation patterns.

- **dotTrace:** A .NET performance profiler from JetBrains that provides comprehensive analysis of memory and performance bottlenecks in .NET applications.
  - It offers features like memory allocation tracking, garbage collection analysis, and performance timeline visualization.

- **Python cProfile:** A built-in profiling module for Python that provides insights into function call counts and execution times.
  - It helps identify performance-critical functions and areas for optimization

Profiling Tools

These tools offer various features such as:

- **Memory usage tracking:** Monitoring memory allocation and deallocation, identifying memory leaks, and analyzing memory usage patterns.
- **Cache hit/miss analysis:** Analyzing cache usage patterns, identifying cache misses, and optimizing data access patterns to improve cache efficiency.
- **Performance profiling:** Analyzing function call counts, execution times, and other performance metrics to identify performance bottlenecks.
- **Thread and concurrency analysis:** Analyzing thread usage and synchronization patterns to identify and resolve concurrency issues.
- **Visualization tools:** Providing graphical representations of memory usage, performance timelines, and other data to facilitate analysis and understanding.

Students Presentations

3.5. Case Studies in

- 3.5.1. Cache-Conscious B Tree
- 3.5.2. Cache-Conscious Merge Sort,
- 3.5.3. Cache-Oblivious merge sort
- 3.5.4. Profiling Tools: Valgrind and Cachegrind

References

1. Erik D. Demaine ;*Cache-Oblivious Algorithms and Data Structures* MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139, USA, edemaine@mit.edu
2. Memory Hierarchy: https://cs61.seas.harvard.edu/site/2022/Storage/#gsc.tab=0
3. Various resources Like books, Lecture slides from different universities, Web Links, AI tools etc.