Exam 3 Notes

Amdahl's Law

Amdahl's Law is a principle that helps in understanding the potential speedup in the overall performance of a system when only part of the system is improved. It's particularly relevant in the context of parallel computing and optimizing performance through hardware upgrades or software optimization.

The formula for Amdahl's law is

$$S = \frac{1}{(1-\alpha) + \alpha/k}$$

where in the aforementioned formula the variables are

- S: The maximum possible speedup of the entire process.
- α: The proportion of the process that benefits from the improved system performance.
- k: The ratio of how much a process can be sped up.

Sector Access

Accessing a sector in a disk involves reading from or writing to a specific, small, fixed-size portion of the disk. Disks, whether they are hard disk drives (HDDs) or solid-state drives (SSDs), are organized into platters (for HDDs) or blocks (for SSDs), which are further divided into tracks and sectors. Let's focus primarily on HDDs for the classic understanding of disk sectors, though the general concept applies to SSDs with some differences due to their lack of moving parts.

Sector Definition

A sector is the smallest storage unit on a disk that can be read from or written to. Historically, sectors on hard drives have been 512 bytes in size, but newer hard drives use a larger sector size of 4096 bytes (or 4K), known as Advanced Format (AF). Each sector has its own unique address, which the disk controller uses to read or write data.

Accessing A Sector

To access a sector, the disk's read/write head must be positioned over the correct track (for HDDs, this is a circular path on the surface of a platter) and then wait for the disk to rotate until the desired sector is under the head. This process involves two main components:

1. **Seek Time**: The time it takes for the read/write head to move to the correct track. Seek time can vary significantly, depending on how far the head needs to move.
2. **Rotational Latency**: Once the head is over the correct track, it must wait for the disk to rotate the correct sector under the head. The average rotational latency depends on the rotation speed of the disk, measured in revolutions per minute (RPM).

The formula for calculating the time it takes to access a sector is made up of three parts: average seek time, average rotation, and average transfer time. Formally as an expression this is

$$T_\alpha = T_\sigma + T_\rho + T_\tau = T_\sigma + \frac{1}{2} \cdot \left(\frac{60}{\text{Rot. Rate}}\right) \cdot 1000 + \frac{60}{\text{Rot. Rate}} \cdot \left(\frac{1}{\text{Sectors / Track}}\right) \cdot 1000$$

where σ is the average seek time, ρ is the average rotation, and τ is the average sectors per track.

Pipeline Speed Up

Pipeline speed up is a concept in computer architecture that refers to the increase in processing speed that can be achieved by using an instruction pipeline. The basics of pipelining are

- **Instruction Pipeline**: It's like an assembly line in a factory. Each stage of the pipeline completes part of the instruction. While one stage is processing one part of an instruction, another stage can process a different part of another instruction.
- **Stages**: Common stages in a simple instruction pipeline include instruction fetch, instruction decode, execute, memory access, and write-back.

Combinatorial Logic

Combinatorial logic functions are a foundational concept in digital circuit design. They are logic circuits whose outputs depend only on the current state of their inputs and not on any prior history (in contrast to sequential logic circuits, which have outputs that depend on a combination of current input states and historical input states).

Solving Combinatorial Logic Problems

When solving a combinatorial logic problem (in the context of what was seen in the quizzes) the goal is to maximize the clock speed. When a register is added between stages, it allows the stages to operate independently in a pipelined fashion. To maximize the clock speed, we want to minimize the delay of the longest pipeline in the process. The steps for determining the highest possible clock speed is:

1. Add a register between the first two stages, calculate the delay for the first delay plus the delay of the newly added register, calculate the delay for the rest of the process, and keep track of the largest delay in this scenario.
2. Add a register between the last two stages, calculate the delay for first two delays plus the delay of the newly added register, calculate the delay for the rest of the process, and keep track of the largest delay in this scenario.
3. Determine the smallest delay between the two largest delays found in step 1 and 2, and calculate the clock speed with the smallest delay with the following formula

$$\text{C.S} = \frac{1}{\alpha \cdot 10^{-12}}$$

where α is the smallest of the two largest delays in ps (pico seconds $10^{-12}(s)$). The resulting calculation is in Hz (hertz $(1/s)$).

Memory Aliasing

Memory aliasing refers to a situation in computer systems where two or more different memory addresses refer to the same physical memory location. This can occur in various contexts and can have both intentional and unintentional consequences. The common causes for memory aliasing are

- **Pointers In Programming**: In languages like C or C++, if two or more pointers point to the same memory address, changing the memory value through one pointer affects the value seen by all pointers aliasing that address.
- **CPU Caches**: Multiple cache lines may map to the same memory location, especially in systems with virtually indexed, physically tagged caches.
- **Memory-Mapped I/O**: Devices mapped to the same address space can cause aliasing, where different device registers are accessed using the same memory addresses.

- **Virtual Memory Systems**: Different virtual addresses may map to the same physical address through the page table mechanism, either within the same process or across different processes.
- **Compiler Optimizations**: When the compiler tries to optimize code, it assumes that different variables occupy different memory locations. Aliasing can break these assumptions and lead to incorrect optimizations.

Memory aliasing has several implications when it happens, here are some examples of these implications

- **Correctness**: It can lead to bugs that are difficult to track down because the same memory is being manipulated from multiple reference points.
- **Performance**: Aliasing can hinder certain optimizations because the compiler must assume that operations affecting one alias could affect all aliases.
- **Consistency**: In multi-threaded environments, memory aliasing complicates the coherence protocols that ensure memory consistency across different CPU cores and caches.

Program Optimizations

We can optimize code in numerous ways, cutting down on executions, function calls, etc.

Machine Independent Optimization

Machine independent optimizations are code transformations that improve performance and are not specific to any particular machine architecture. These optimizations generally improve the efficiency of the code by reducing the number of instructions, improving algorithmic complexity, or enhancing data access patterns. They are typically performed by the compiler at a high level, without considering the specifics of the underlying hardware.

```
// Without
int compute_area(int width, int height) {
    int area1 = width * height;
    int area2 = width * height; // Redundant computation
    return area1 + area2;
}
// With
int compute_area(int width, int height) {
    int area = width * height; // Compute once
    return area + area; // Reuse the result
}

```

Loop Unrolling

Loop unrolling is an optimization technique that aims to increase a program's execution speed by reducing or eliminating the overhead of loop control. By executing more than one iteration of the loop per cycle through the loop control code, it can also improve the opportunities for other optimizations, such as instruction pipelining in the processor.

```
// Without
for (int i = 0; i < N; ++i) {
    dest[i] = src[i] + 1;
}
// With
// Handle the main part of the loop in steps of 4 to reduce loop overhead.
for (int i = 0; i < N; i += 4) {
    dest[i] = src[i] + 1;
    dest[i+1] = src[i+1] + 1;
    dest[i+2] = src[i+2] + 1;
    dest[i+3] = src[i+3] + 1;
}

// Handle the remainder of the elements that didn't fit into groups of 4
for (int i = N - N % 4; i < N; ++i) {
    dest[i] = src[i] + 1;
```

```
}

```

Unrolling And Multiple Accumulators

Loop unrolling with multiple accumulators is a further enhancement of the loop unrolling optimization. In this technique, not only is the loop unrolled to reduce the loop overhead, but multiple accumulator variables are also used to hold intermediate results. This can reduce dependencies between loop iterations and allow for more parallelism, especially on hardware that can perform multiple operations simultaneously.

```
// Without
int sum = 0;
for (int i = 0; i < N; ++i) {
    sum += array[i];
}
// With
int sum0 = 0, sum1 = 0, sum2 = 0, sum3 = 0;
for (int i = 0; i < N; i += 4) {
    sum0 += array[i];
    sum1 += array[i + 1];
    sum2 += array[i + 2];
    sum3 += array[i + 3];
}
int totalSum = sum0 + sum1 + sum2 + sum3;

// Handle the remainder of the elements that didn't fit into groups of 4
for (int i = N - N % 4; i < N; ++i) {
    totalSum += array[i];
}

```

Strength Reduction

Strength reduction is an optimization technique where more expensive operations are replaced with equivalent but less costly operations. It's particularly effective in loops where an expensive operation, like multiplication or division, can be replaced with addition or subtraction.

```
// Without
for (int i = 0; i < N; ++i) {
    array[i * 4] = i * 4; // Multiplication inside loop
}
// With
for (int i = 0, j = 0; i < N; ++i, j += 4) {
    array[j] = j; // Replace multiplication with addition
}

```

Parallel Accumulators

Parallel accumulators are an optimization technique used to minimize dependencies between successive iterations of a loop that would otherwise limit the degree of parallelism achievable. By using separate accumulator variables in a loop that performs a reduction (like summing values), a program can take advantage of parallel execution capabilities of modern processors.

```
// Without
int sum = 0;
for (int i = 0; i < N; ++i) {
    sum += array[i];
}
// With
int sum1 = 0, sum2 = 0;
for (int i = 0; i < N / 2; ++i) {
    sum1 += array[2 * i];
    sum2 += array[2 * i + 1];
}
```

```
    int totalSum = sum1 + sum2;

    // Handle the case where N is odd
    if (N % 2 != 0) {
        totalSum += array[N - 1];
    }

```

Common Subexpression Elimination

Common subexpression elimination (CSE) is an optimization technique that identifies instances of identical expressions being evaluated multiple times, and eliminates the redundancy by computing the expression once and reusing the result. This reduces the computation time and can also decrease code size, improving cache performance.

```
// Without
int width = 5;
int height = 10;
int area = width * height; // Computed here
int perimeter = 2 * (width + height);
int doubleArea = 2 * (width * height); // Computed again - a common subexpression
// With
int width = 5;
int height = 10;
int area = width * height; // Compute once
int perimeter = 2 * (width + height);
int doubleArea = 2 * area; // Reuse the computed area

```

Inlining

Inlining is an optimization where the compiler replaces a function call with the actual code of the function. It eliminates the overhead associated with function calls such as parameter passing, return value computation, and stack frame management. This is especially beneficial for small, frequently called functions.

```
// Without
int square(int num) {
    return num * num;
}

int main() {
    int total = 0;
    for (int i = 0; i < N; ++i) {
        total += square(i);
    }
    return total;
}
// With
int main() {
    int total = 0;
    for (int i = 0; i < N; ++i) {
        int temp = i * i; // Inlining 'square(i)'
        total += temp;
    }
    return total;
}

```

Spatial Locality

Spatial locality is a principle that helps to optimize how computer systems access and store data, and it's particularly relevant in the context of memory hierarchies, including caches. The concept of spatial locality refers to the tendency of a processor to access data locations that are physically close to recently accessed locations. The formal definition of spatial locality is

- **Spatial Locality**: The principle that if a particular storage location is accessed, the locations with nearby addresses are likely to be accessed soon. This is due to the structure of most programs, where data is often organized sequentially in memory (like arrays or adjacent fields in a structure).

Cache

A cache in computing is a smaller, faster storage layer that stores copies of data from a more substantial, slower storage layer. The primary goal of a cache is to increase data retrieval performance by reducing the time it takes to access data. Caches are ubiquitous in computer systems, found in web browsers, operating systems, and most importantly, within the CPU to speed up access to memory.

Caches consist of fundamental parameters that are used to determine quantities like the Cache Offset, Cache Index, and Cache Tag.

The following are fundamental parameters regarding caches:

$S = 2^s$	(Number of Sets)
$s = \log_2(S)$	(Number of *Set Index* Bits)
E	(Number of Lines Per Set)
$B = 2^b$	(Block Size (Bytes))
$b = \log_2(b)$	(Number of *Block Offset* Bits)
$M = 2^m$	(Maximum Number of Unique Memory Addresses)
$m = \log_2(m)$	(Number of Physical (Main Memory) Address Bits)
$t = m - (s + b)$	(Number of *Tag* Bits)
$C = B \cdot E \cdot S$	(Cache Size (Bytes))

Cache Offset

The cache offset, also known as the block offset, refers to the position of a byte or word within a cache block or line. When data is loaded into the cache, it's not loaded in single bytes but rather in blocks of contiguous bytes. The offset specifies the exact byte within this block where the desired data is located. It's used to pinpoint the data within the cache line once the correct cache line is identified.

Cache Index

The cache index helps in determining which cache line (or slot) within the cache is to be used for storing and retrieving a specific block of data. The cache is divided into several lines, and the index specifies which line data is stored in or should be looked for. The index is derived from the memory address being accessed, usually by taking certain bits from the middle of the address.

Cache Tag

The cache tag stores information about the data's identity in the cache line. It's used to verify that the data in the cache line is the same as the data being requested by the CPU. When a memory address is accessed, a part of it is used to form the cache tag. The cache controller compares this tag with the tags in the cache to determine if the requested data is present (a hit) or not (a miss).

Cache Hit Or Miss

- **Cache Hit**: A cache hit occurs when the data requested by the CPU is found in the cache. This means the CPU can directly read from the cache without having to access the slower main memory, leading to faster data retrieval.

- **Cache Miss**: A cache miss occurs when the requested data is not found in the cache. This forces the CPU to fetch the data from the main memory, which is a slower process. A cache miss also triggers the process of loading the requested data into the cache for future access, possibly replacing existing data based on the cache's replacement policy.