HW4 - Arman Behnam - A20522906

My Email: abehnam@hawk.iit.edu

5.1

5.1.1

In a 16-byte cache block, 44 32-bit integers can be stored, since each 32-bit integer occupies 4 bytes.

# Cache block size and size of each integer in bytes
cache_block_size_bytes = 16
integer_size_bytes = 4 # 32-bit integer
# Number of integers that can be stored in a 16-byte cache block
num_integers_in_cache_block = cache_block_size_bytes / integer_size_bytes
print(num_integers_in_cache_block)

5.1.2

In the given C code snippet, temporal locality refers to the reuse of specific data and/or resources within relatively small-time durations:

- A[I][J]: exhibits temporal locality due to its use within the inner loop. Since A[I][J] is accessed multiple times in quick succession when I is constant and J is varying, there is a high likelihood that A[I][J] will be loaded into cache and reused before it gets evicted.
- B[I][0]: shows strong temporal locality. For each iteration of the outer loop (where I is the loop variable), B[I][0] is accessed 8000 times (once for each iteration of the inner loop). This repeated access over a short duration increases the chances of B[I][0] being held in the cache and reused.
- A[J][I]: Its temporal locality is not as strong as A[I][J] or B[I][0]. This is because the values of J and I both change with each iteration of the inner loop, leading to more scattered access patterns. However, for a fixed I in the outer loop, A[J][I] is accessed 8000 times in the inner loop, giving it some degree of temporal locality.

5.1.3

A[I][J] strongly exhibits spatial locality due to contiguous row-wise access, A[J][I] shows lesser spatial locality due to its column-wise access in a row-major stored array, and B[I][0] has spatial locality when considering the change in I in the outer loop, although it's more dominated by temporal locality.

5.1.4

To determine how many 16-byte cache blocks are needed to store all the 32-bit matrix elements being referenced in the provided MATLAB code, we need to understand the data layout and access pattern. MATLAB uses column-major order for storing matrices, which means elements within the same column are stored contiguously in memory.

The matrices A and B have sizes 8x8000 and 8x1, respectively. Since each element is a 32-bit (4-byte) integer, we can calculate the total number of cache blocks required as follows:

- Determine the number of elements in each matrix.
- Calculate the total memory required to store these elements.
- Divide this total memory by the size of a cache block (16 bytes) to find the number of cache blocks required

```
# Matrix dimensions and sizes
rows_A = 8
cols_A = 8000
size_B = 8  # Assuming B is a column vector with 8 elements

# Total number of elements in each matrix
total_elements_A = rows_A * cols_A
total_elements_B = size_B

# Total memory required for all elements in bytes
total_memory_bytes = (total_elements_A + total_elements_B) * integer_size_bytes

# Number of cache blocks required
cache_blocks_needed = total_memory_bytes / cache_block_size_bytes
cache_blocks_needed
```

To store all the 32-bit matrix elements referenced in the MATLAB code, approximately 16, 002 16-byte cache blocks are needed

5.1.5

In the Matlab code, where matrix elements within the same column are stored contiguously, temporal locality is exhibited in the following ways:

- A(I, J): shows strong temporal locality. In the inner loop, A(I, J) is accessed repeatedly with the same I value but different J values. Although Matlab stores data in column-major order, the temporal locality is more about the reuse of the same memory location within a short period, which is evident here as the same element A(I, J) is written to in each iteration of the inner loop.
- B(I, 1): B(I, 0) in code is a typo and should be B(I, 1). This variable exhibits temporal locality because it is accessed multiple times in quick succession for each iteration of the outer loop. For each value of I, B(I, 1) is used in every iteration of the inner loop, making it a candidate for caching and reuse within a short time frame.
- A(J, I): shows some degree of temporal locality. For a fixed value of I in the outer loop, A(J, I) is accessed 8000 times in the inner loop. Although the access pattern is column-wise (which is optimal for Matlab's storage scheme), the repeated access of the same column elements in a short duration contributes to temporal locality.

5.1.6

In the Matlab code, where elements within the same column are stored contiguously due to its column-major order, spatial locality is exhibited in the following variables:

- A(J, I): This variable shows strong spatial locality. Since Matlab stores arrays in column-major order, elements in the same column (I fixed, J varying) are contiguous in memory. As J increments in the inner loop, subsequent elements in the column are accessed, benefiting from spatial locality as they are likely to be in the same or adjacent cache blocks.
- A(I, J): The spatial locality for A(I, J) is less pronounced compared to A(J, I). In column-major order, elements of different columns are not contiguous. However, as I changes in the outer loop and J in the inner loop, there might be some spatial locality when adjacent elements in a row are in the same or adjacent cache blocks, although this is not optimal for MATLAB's storage scheme.
- B(I, 1): The spatial locality for B(I, 1) is minimal in this context. B is a column vector, and elements are accessed sequentially as I increment in the outer loop. This sequential access pattern does exhibit spatial locality. However, since B(I, 1) is primarily accessed once per iteration of the outer loop, its spatial locality is not as strong as that of A(J, I).

5.2

5.2.1

To analyze the cache behavior for the given list of 32-bit memory addresses in a direct-mapped cache with 16 one-word blocks, we need to break down the steps:

- Convert each memory address to its binary representation.
- Determine the number of bits for the index and the tag in the cache. Since the cache has 16 one-word blocks, the index is used to identify which of these 16 blocks the memory address maps to. With 16 blocks, we need 4 bits for the index (since 24=1624=16). The remaining bits in the 32-bit address will be used for the tag.
- Split each binary address into the tag and the index.
- Determine if each reference is a cache hit or miss, assuming the cache starts empty.

Here's the analysis for each of the given memory address references in terms of binary address, tag, index, and whether each reference is a hit or miss in a direct-mapped cache with 16 one-word blocks:

Address 3:

- Binary: 00000000000000000000000000000011
- Tag: 0000000000000000000000000000
- Index: 0011
- Miss

Address 180:

- Binary: 00000000000000000000000010110100
- Tag: 0000000000000000000000001011
- Index: 0100
- Miss

Address 43:

- Binary: 00000000000000000000000000101011
- Tag: 0000000000000000000000000010
- Index: 1011
- Miss

Address 2:

- Binary: 00000000000000000000000000000010
- Tag: 0000000000000000000000000000
- Index: 0010
- Miss

Address 191:

- Binary: 00000000000000000000000010111111
- Tag: 0000000000000000000000001011
- Index: 1111
- Miss

Address 88:

- Binary: 00000000000000000000000001011000
- Tag: 0000000000000000000000000101
- Index: 1000
- Miss

Address 190:

- Binary: 00000000000000000000000010111110
- Tag: 0000000000000000000000001011
- Index: 1110
- Miss

Address 14:

- Binary: 00000000000000000000000000001110
- Tag: 0000000000000000000000000000
- Index: 1110
- Miss

Address 181:

- Binary: 00000000000000000000000010110101
- Tag: 0000000000000000000000001011
- Index: 0101
- Miss

Address 44:

- Binary: 00000000000000000000000000101100
- Tag: 0000000000000000000000000010
- Index: 1100
- Miss

Address 186:

- Binary: 00000000000000000000000010111010
- Tag: 0000000000000000000000001011
- Index: 1010
- Miss

Address 253:

- Binary: 00000000000000000000000011111101
- Tag: 0000000000000000000000001111
- Index: 1101
- Miss

```
def address_to_binary(address):
    """Converts a decimal address to a 32-bit binary string."""
    return format(address, '032b')

def get_tag_index(binary_address):
    """Extracts the tag and index from the binary address."""
    # In a 32-bit address with 16 blocks (4 bits for index), the last 4 bits are for index and the rest
are for tag
    tag = binary_address[:-4]
    index = binary_address[-4:]
    return tag, index

# List of memory addresses
addresses = [3, 180, 43, 2, 191, 88, 190, 14, 181, 44, 186, 253]

# Process each address
cache_data = []
cache_content = [None] * 16  # Represents the 16 blocks of the cache, initially empty
for address in addresses:
    binary_address = address_to_binary(address)
    tag, index = get_tag_index(binary_address)
    index_decimal = int(index, 2)  # Convert index to decimal to access the cache array

    # Check for hit or miss
    if cache_content[index_decimal] == tag:
        hit_miss = "Hit"
    else:
        hit_miss = "Miss"
        cache_content[index_decimal] = tag  # Update the cache with the new tag

    cache_data.append((address, binary_address, tag, index, hit_miss))

cache_data
```

5.2.2

Here's the analysis for each of the given memory address references in terms of binary address, tag, index, and whether each reference is a hit or miss in a direct-mapped cache with two-word blocks and a total of 8 blocks:

Address 3:

- Binary: 00000000000000000000000000000011
- Tag: 00000000000000000000000000000
- Index: 011
- Miss

Address 180:

- Binary: 00000000000000000000000010110100
- Tag: 00000000000000000000000010110
- Index: 100
- Miss

Address 43:

- Binary: 00000000000000000000000000101011
- Tag: 00000000000000000000000000101
- Index: 011
- Miss

Address 2:

- Binary: 00000000000000000000000000000010
- Tag: 00000000000000000000000000000
- Index: 010
- Miss

Address 191:

- Binary: 00000000000000000000000010111111
- Tag: 00000000000000000000000010111
- Index: 111
- Miss

Address 88:

- Binary: 00000000000000000000000001011000
- Tag: 00000000000000000000000001011
- Index: 000
- Miss

Address 190:

- Binary: 00000000000000000000000010111110
- Tag: 00000000000000000000000010111
- Index: 110
- Miss

Address 14:

- Binary: 00000000000000000000000000001110
- Tag: 00000000000000000000000000001
- Index: 110
- Miss

Address 181:

- Binary: 00000000000000000000000010110101
- Tag: 00000000000000000000000010110
- Index: 101
- Miss

Address 44:

- Binary: 00000000000000000000000000101100
- Tag: 00000000000000000000000000101
- Index: 100
- Miss

Address 186:

- Binary: 00000000000000000000000010111010
- Tag: 00000000000000000000000010111
- Index: 010

- Miss

Address 253:

- Binary: 00000000000000000000000011111101
- Tag: 000000000000000000000000011111
- Index: 101
- Miss

5.3

5.3.1

In the described direct-mapped cache design, the bits of the 32-bit address are divided as follows for tag, index, and offset:

- Tag: Bits 31–10
- Index: Bits 9–5
- Offset: Bits 4–0

The offset in the cache address is used to identify the specific word within a cache block. Since the offset in this case uses bits 4–0, it consists of 5 bits. The number of bits in the offset determines the block size in the following way:

- The size of the block in bytes is 2number of offset bits2number of offset bits.
- Since each word is 32 bits (or 4 bytes) in this context, the number of words in each block is 2number of offset bits/42number of offset bits/4.

The cache block size in this direct-mapped cache design is 88 words.

5.3.2

In the provided cache design, the index is determined by bits 9–5 of the 32-bit address. This range indicates that the index consists of 5 bits. The number of cache entries can be calculated as 2number of index bits2number of index bits. Since there are 5 index bits, the cache has 2525 entries. The cache has 3232 entries.

5.3.3

To determine the ratio between the total bits required for the cache implementation and the data storage bits, we need to consider the following components:

1. Data Storage Bits: the product of the number of cache entries and the size of each cache block in bits.
2. Tag Bits per Entry: the tag size for each entry. The tag size can be determined from the address bits.
3. Total Bits for Tags: This is the product of the number of cache entries and the number of tag bits per entry.

4. Total Cache Size: This is the sum of the total data storage bits and the total tag bits.

Given:

- 32-bit address.
- Tag: Bits 31–10 (22 bits).
- Index: Bits 9–5 (5 bits) → 32 entries.
- Offset: Bits 4–0 (5 bits) → Cache block size is 8 words (32 bytes).

The ratio between the total bits required for the cache implementation and the data storage bits is approximately 1.0861.086. This means that for every bit of actual data stored, an additional 0.0860.086 bits are used for the cache's structural overhead, primarily for storing the tag information.

5.3.4

To determine the number of blocks replaced in the cache, we need to analyze the cache references and see how many times a new block is brought into the cache, replacing an existing one. The cache is initially empty, and as new addresses are referenced; blocks will be loaded into the cache. A block is replaced when a new address maps to a cache entry that is already occupied by a different block.

Given the cache design:

- Cache block size: 32 bytes (from previous calculation).
- Number of cache entries: 32 (from the index bits).

Since the addresses are byte-addressed, we first convert them to block addresses by dividing each by the block size. If a block maps to an already occupied entry with a different tag, it indicates a replacement: 0, 4, 16, 132, 232, 160, 1024, 30, 140, 3100, 180, 2180.

In the cache with the given references, a total of 44 blocks are replaced

5.3.5

To calculate the hit ratio, we need to know the total number of cache accesses and the number of these accesses that resulted in cache hits. The hit ratio is defined as the number of hits divided by the total number of accesses.

From the provided addresses: 0, 4, 16, 132, 232, 160, 1024, 30, 140, 3100, 180, 2180.

We can count each access and determine whether it's a hit or a miss. A hit occurs when the accessed address maps to a cache block that's already loaded in the cache with the correct tag. Otherwise, it's a miss. The hit ratio for the cache with the given references is approximately 0.3330.333 or 33.333.3. This means that about one-third of the cache accesses resulted in hits.

5.3.6

To list the final state of the cache, we need to represent each valid entry as a record of the form <index, tag, data>. For this, we'll use the final state of the cache from our previous calculations, determine the index and tag for each cache entry, and list the data (which is the block address).

Given:

- Cache design with 22 tag bits, 5 index bits, and 5 offset bits.
- Addresses: 0, 4, 16, 132, 232, 160, 1024, 30, 140, 3100, 180, 2180.
- We've already converted these addresses to block addresses and determined their corresponding cache entries.

The final state of the cache, with each valid entry represented as a record of <index, tag, data>, is as follows:

1. <0, 0, 96>: Index 0, Tag 0, Data (Block Address) 96.
2. <4, 0, 68>: Index 4, Tag 0, Data (Block Address) 68.
3. <5, 0, 5>: Index 5, Tag 0, Data (Block Address) 5.
4. <7, 0, 7>: Index 7, Tag 0, Data (Block Address) 7.

These entries represent the cache's content at the end of the sequence of references. The 'data' in each record is the block address corresponding to that cache entry.

5.5

5.5.1

To calculate the miss rate for the address stream of a video streaming workload, and to understand its sensitivity to cache and working set size, we need to consider the nature of the accesses and the cache configuration:

1. **Cache Configuration**:
   - Cache size: 64 KiB.
   - Block size: 32 bytes.
   - Direct-mapped cache implies one block per cache line.
2. **Working Set:**
   - Working set size: 512 KiB.
   - Address stream: 0, 2, 4, 6, 8, 10, 12, 14, 16, ..., accessing data sequentially.
3. **Miss Rate Calculation:**
   - Given the sequential nature of the address stream, each new address after the first set of blocks will result in a cache miss, as the block corresponding to that address won't be in the cache.
4. **Sensitivity to Cache or Working Set Size:**
   - If the cache size increases, more blocks can be stored, reducing the miss rate up to a point. However, for large enough working set (like 512 KiB), the miss rate will still be significant.

- If the working set size decreases while keeping the cache size constant, the miss rate will reduce as more data can be retained in the cache.

5. **Categorization of Misses (3C Model):**
   - **Compulsory Misses**: These are the initial misses experienced when a block is accessed for the first time. This streaming workload will predominantly experience compulsory misses due to the sequential nature of data access.
   - **Capacity Misses**: These occur when the cache cannot hold all the blocks needed for the workload. Given the working set size is much larger than the cache, capacity misses will also be significant.
   - **Conflict Misses**: These are less relevant in a streaming workload with sequential access, especially in a large direct-mapped cache.

The miss rate is the ratio of the number of unique blocks accessed to the total number of accesses. The miss rate for the given address stream in a video streaming workload is approximately 0.031250.03125 or 3.1253.125.

Sensitivity to Cache or Working Set Size

- **Cache Size**: If the cache size is increased, more blocks can be stored, potentially reducing the miss rate. However, given the sequential nature of the address stream and the large working set size, the miss rate will still be significant as each new block accessed is likely to be a miss.
- **Working Set Size**: If the working set size is reduced (smaller than the cache size), the miss rate would decrease significantly, as more of the working set could be retained in the cache.

Categorization of Misses

- The misses experienced in this workload are predominantly **compulsory misses** since each block is accessed sequentially for the first time.
- There are also **capacity misses** because the working set size (512 KiB) is significantly larger than the cache size (64 KiB), implying not all data can be retained in the cache simultaneously.
- **Conflict misses** are not a primary concern in this scenario due to the direct-mapped cache and the sequential nature of the access pattern.

5.5.2

The recalculated miss rates for different cache block sizes are as follows:

- **16 bytes block size**: Miss rate is approximately 0.06250.0625 or 6.256.25.
- **64 bytes block size**: Miss rate is approximately 0.0156250.015625 or 1.56251.5625.
- **128 bytes block size**: Miss rate is approximately 0.00781250.0078125 or 0.781250.78125.

Analysis of Locality: This workload is exploiting **spatial locality**. Spatial locality refers to the use of data elements within relatively close storage locations. In this streaming workload, where data is accessed sequentially (0, 2, 4, 6, 8, 10, ...), consecutive data elements are likely to be located close to each other in memory. Larger block sizes capture more of this sequential data, which is why the miss rate decreases as the block size increases – more adjacent data is included in each cache block, making it more likely that subsequent accesses hit in the cache.

5.6

5.6.1

To determine the clock rates for processors P1 and P2, we can use their L1 cache hit times, as it's assumed that the L1 hit time determines the cycle time for each processor. The clock rate of a processor is inversely proportional to its cycle time. The cycle time, in this context, is equal to the L1 cache hit time.

Given the cycle time (L1 hit time) for each processor:

- P1: L1 Hit Time = 0.66 ns
- P2: L1 Hit Time = 0.90 ns

The clock rate will be given in Hertz (Hz), and since the times are in nanoseconds, the resulting clock rates will be in GHz (1 GHz = 1×1091×109 Hz). The respective clock rates for processors P1 and P2, based on their L1 cache hit times, are as follows:

- Processor P1: The clock rate is approximately 1.5151.515 GHz.
- Processor P2: The clock rate is approximately 1.1111.111 GHz.

5.6.2

The Average Memory Access Time (AMAT) for a processor can be calculated using the formula:
AMAT=Hit Time + (Miss Rate × Miss Penalty)

Where:

- Hit Time is the time it takes to access the L1 cache (L1 Hit Time).
- Miss Rate is the percentage of accesses that result in a cache miss.
- Miss Penalty is the time it takes to handle a cache miss, typically the time to access the next level of memory, in this case, the main memory.

Given data:

- Main memory access time: 70 ns.
- P1: L1 Size 2 KiB, Miss Rate 8.0%, L1 Hit Time 0.66 ns.
- P2: L1 Size 4 KiB, Miss Rate 6.0%, L1 Hit Time 0.90 ns.

The miss penalty is the same for both processors as they access the same main memory. Average Memory Access Time (AMAT) for processors P1 and P2 is as follows:

- Processor P1: AMAT is approximately 6.266.26 ns.
- Processor P2: AMAT is approximately 5.105.10 ns.

These values indicate the average time it takes for each processor to access memory, taking into account both the time for cache hits and the additional time incurred on cache misses.

5.6.3

To calculate the total Cycles Per Instruction (CPI) for processors P1 and P2, considering memory stalls, we use the formula: *Total CPI=Base CPI + Memory Stall CPI*

Where:

- Base CPI is given as 1.0 (without any memory stalls).
- Memory Stall CPI can be calculated as the product of the memory access portion of all instructions and the Average Memory Stall Cycles per instruction.

The Average Memory Stall Cycles per instruction can be calculated using: *Average Memory Stall Cycles = Miss Rate × Miss Penalty in Cycles*

Given that memory accesses are 36% of all instructions and the miss penalty is the Average Memory Access Time minus the L1 Hit Time, let's calculate the Total CPI for P1 and P2. The processor with the lower Total CPI is faster.

First, we need to convert the Average Memory Access Time and L1 Hit Time from nanoseconds to cycles. This is done by dividing the times in nanoseconds by the cycle time (also in nanoseconds) for each processor. The cycle time for each processor is equal to its L1 Hit Time. The total Cycles Per Instruction (CPI) for processors P1 and P2 are:

- Processor P1: Total CPI is approximately 1.2441.244.
- Processor P2: Total CPI is approximately 1.1011.101.

Based on these calculations, Processor P2 is faster. It has a lower total CPI, indicating that it can execute instructions with fewer additional cycles due to memory stalls, compared to Processor P1.

```
# Base CPI without memory stalls
base_cpi = 1.0
# Memory accesses are 36% of all instructions
memory_access_portion = 36 / 100  # converting percentage to decimal
# Convert AMAT and L1 hit times from ns to cycles for each processor
amat_P1_cycles = amat_P1_ns / hit_time_P1_ns
amat_P2_cycles = amat_P2_ns / hit_time_P2_ns
hit_time_P1_cycles = hit_time_P1_ns / hit_time_P1_ns
hit_time_P2_cycles = hit_time_P2_ns / hit_time_P2_ns
# Calculate Average Memory Stall Cycles per instruction
memory_stall_P1_cycles = miss_rate_P1 * (amat_P1_cycles - hit_time_P1_cycles)
memory_stall_P2_cycles = miss_rate_P2 * (amat_P2_cycles - hit_time_P2_cycles)
# Calculate Total CPI for P1 and P2
total_cpi_P1 = base_cpi + (memory_access_portion * memory_stall_P1_cycles)
total_cpi_P2 = base_cpi + (memory_access_portion * memory_stall_P2_cycles)
print(total_cpi_P1, total_cpi_P2)
```