Problem 2 General Questions (20 credits)

a) In class, we discussed different *database processing models*. Please **fill** in the gaps below with the processing model that best fits the sentence. Each model should fit in **only one** sentence.

0 1 2 3

I. The _____ model can lead to a significant function call overhead due to its pipelining approach.

II. The _____ model has the potential for excellent performance, provided that a specific parameter is chosen correctly.

III. The _____ model involves tight loops, and modern compilers can effectively optimize it through techniques like loop unrolling.

b) **Name** one disadvantage each for **Fully Associative** and **Direct-Mapped** caches. **Explain** why **Set-Associative** caches are a compromise.

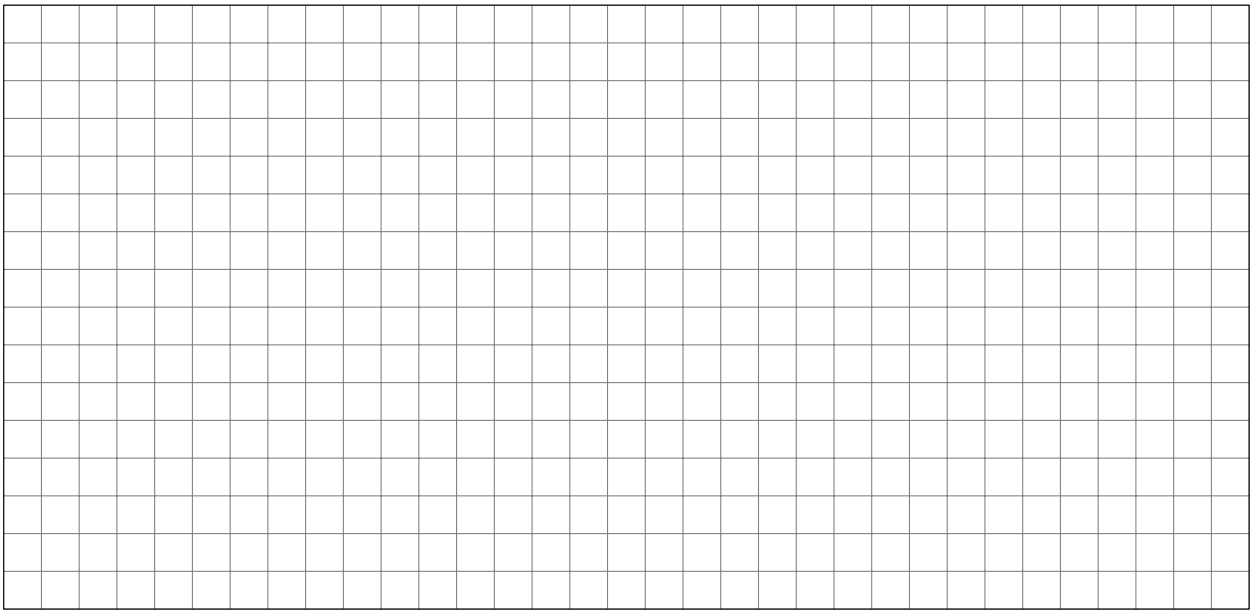
0 1 2 3 4

c) Your program just accessed a **virtual address**, but the data is not in memory. **Briefly** explain how the **page fault** is handled by the system.

0 1 2 3 4

0 1 2 3 4

d) **Briefly** explain how **multi-socket NUMA systems** affect the design of **data structures and algorithms** and the **concurrency** for data processing.



0 1 2 3 4 5

e) **Name** one advantage and one disadvantage each for *column stores* and *row stores*. Which one is better for **OLAP** workloads?

Problem 3 Data Hazards and Instruction Execution (15 credits)

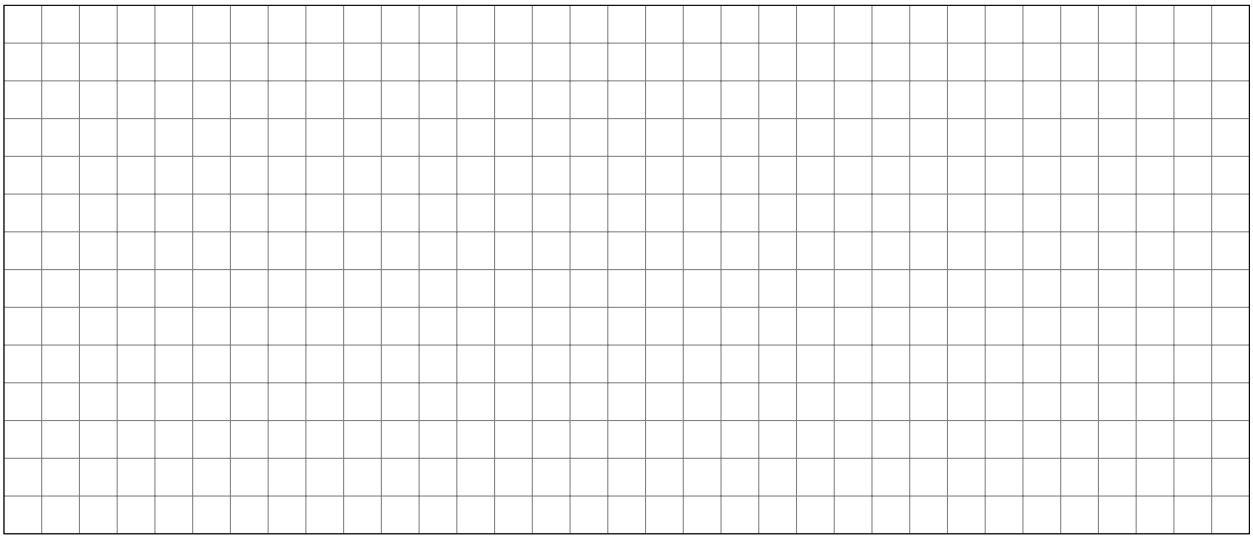
You are an employee of the big software company *HolyC International* and encounter the function `foo` in a critical code path. You know that `array` can contain up to **millions** of entries and the value distribution is **unpredictable**. You work on a common x86-64 desktop machine and can only use a C compiler that does not apply optimizations.

```
uint32_t foo(uint8_t* array, size_t array_size, uint8_t selector) {
    uint32_t sum = 0;
    for(size_t i = 0; i < array_size; i++) {
        if(array[i] > selector ) {
            sum = sum + ((uint32_t) array[i]);
        }
    }
    return sum;
}
```

Hint: The data types `uint8_t` and `uint32_t` are 8-bit and 32-bit large and stores unsigned integers. You can assume to work on a common x86-64 desktop machine.

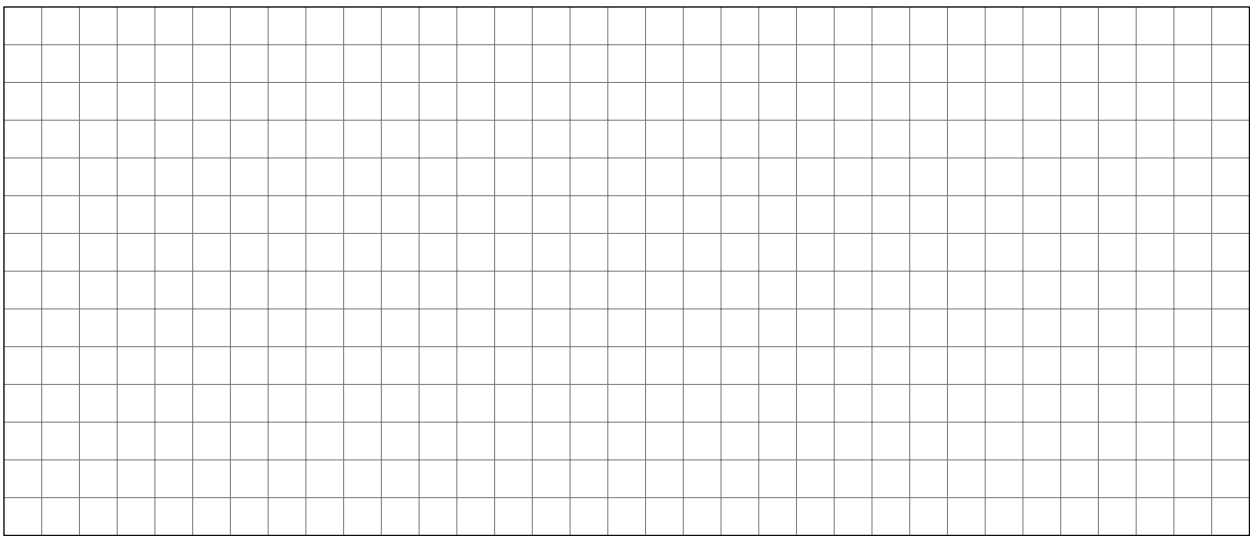
a) **Sketch** the expected execution time of the function `foo` in proportion to the selectivity of the predicate `selector` using a graph. **Briefly** explain your sketch.

0 1 2 3 4



b) Use **software predication** to rewrite function `foo`, so run time is independent of predicate selectivity.

0 1 2 3



0 1 2 3

c) Is your rewrite of the function `foo` **always** faster than the original version? **Briefly** justify your answer.

After your co-workers heard about your amazing work on analyzing function `foo` they ask you to have a look at the function `bar`. One of your co-workers already implemented a compiler pass that applies simple **loop unrolling**, but the performance of `bar` still is not as good as expected. The function is executed on an **in-order** x86-64 CPU.

```
void bar(uint8_t* array, size_t array_size, uint8_t constant) {
        for(size_t i = 0; i < array_size; i++) {
                array[i] = array[i] + constant;
        }
}
```

Hint: The data type `uint8_t` is 8-bit large and stores unsigned integers.

0 1 2

d) **Name** one problem simple loop unrolling has with instruction ordering and **briefly** describe it.

0 1 2 3

e) **Name** another compiler pass that can be applied **after** loop unrolling to improve its performance. **Briefly** describe the idea behind this compiler pass.