Family Name: SOLUTIONS First Name: Student ID:

Final Exam

Computer Architecture (263-2210-00L)

ETH Zürich, Fall 2019

Prof. Onur Mutlu

Problem 1 (40 Points):	GPUs and SIMD	
Problem 2 (35 Points):	Vector Processing	
Problem 3 (25 Points):	Emerging Memory Technologies	
Problem 4 (40 Points):	Data Prefetching	
Problem 5 (30 Points):	Asymmetric Multicore	
Problem 6 (25 Points):	Bottleneck Acceleration	
Problem 7 (35 Points):	Cache Coherence	
Problem 8 (35 Points):	Memory Consistency	
Total (265 Points):		

Examination Rules:

1. Written exam, 180 minutes in total.
2. No books, no calculators, no computers or communication devices. 10 single-sided A4 pages of handwritten notes are allowed.
3. Write all your answers on this document, space is reserved for your answers after each question. Blank pages are available at the end of the exam.
4. Clearly indicate your final answer for each problem. Answers will only be evaluated if they are readable.
5. Put your Student ID card visible on the desk during the exam.
6. If you feel disturbed, immediately call an assistant.
7. Write with a black or blue pen (no pencil, no green or red color).
8. Show all your work. For some questions, you may get partial credit even if the end result is wrong due to a calculation mistake. If you make assumptions, state your assumptions clearly and precisely.
9. Please write your initials at the top of every page.

Tips:

- **Be cognizant of time.** Do not spend too much time on one question.
- **Be concise.** You may be penalized for verbosity.
- **Show work when needed.** You will receive partial credit at the instructors' discretion.
- **Write legibly.** Show your final answer.

This page intentionally left blank

1 GPUs and SIMD [40 points]

We define the *SIMD utilization* of a program that runs on a GPU as the fraction of SIMD lanes that are kept busy with *active threads* during the run of a program. As we saw in lecture and practice exercises, the SIMD utilization of a program is computed across the *complete run* of the program.

The following code segment is run on a GPU. A warp in the GPU consists of 32 threads, and there are 32 SIMD lanes in the GPU. Each thread executes **a single iteration** of the shown loop. Assume that the data values of the arrays `A` and `B` are already in vector registers so there are no loads and stores in this program. The value of `k` is constant across all iterations and $0 < k \leq 32$. (Hint: Notice that there are 2 instructions in each iteration. The two comparisons in the `if` statement are executed as a single instruction.)

```
for (i = 0; i < 3072; i++) {
    if (i % k == 0 || A[i % k] > 0) { // Instruction 1
        B[i] = A[i] + 1;              // Instruction 2
    }
}
```

Please answer the following four questions.

(a) [5 points] How many warps does it take to execute this program?

96 warps.

Explanation:
The number of warps is calculated as:
$\#Warps = \lceil \frac{\#Total\_threads}{\#Warp\_size} \rceil$,

where
$\#Total\_threads = 3072 = 3 \times 2^{10}$ (i.e., one thread per loop iteration),

and
$\#Warp\_size = 32 = 2^5$ (given).

Thus, the number of warps needed to run this program is:
$\#Warps = \lceil \frac{3 \times 2^{10}}{2^5} \rceil = 3 \times 2^5 = 96$.

(b) [10 points] What needs to be true about array `A` to achieve 100% utilization? Show your work. (Hint: The warp scheduler does not issue instructions where no threads are active).

For each `k`, array elements `A[1]` to `A[k - 1]` should have positive values.

Explanation:
Threads with an ID `i` that is a multiple of `k` (or 0) execute `Instruction 2` for any value of `A[i]`. The rest of threads enter the `if` statement if `A[1]` to `A[k - 1]` have positive values.

(c) [15 points] Provide an analytical expression that determines the SIMD utilization as a function of `k`. Show your work.

$SIMD\_utilization = \frac{3072+\frac{3072}{k}+\frac{\alpha}{k-1}\times(3072-\frac{3072}{k})}{3072+3072} = \frac{k+1+\alpha}{2\times k}$, with $0 \leq \alpha \leq$ k - 1.

Explanation:
An analytical expression for the $SIMD\_utilization$ should be a fraction with the denominator equal to 3072 + 3072, because at least one thread of each warp executes `Instruction 2` (notice $k \leq \#Warp\_size$).

The numerator sums the number of threads that execute `Instruction 1`, which is 3072, and the number of threads that execute `Instruction 2`.

One every `k` threads executes `Instruction 2` because its `i` is a multiple of `k`, i.e., $\frac{3072}{k}$ threads.

The rest of threads execute `Instruction 2` if the corresponding element of `A` (i.e., `A[1]` to `A[k - 1]`) has a positive value. We refer to the number of these elements that are positive as α, with $0 \leq \alpha \leq$ k - 1.

(d) [10 points] What needs to be true about array `A` and `k` to achieve SIMD utilization of $\frac{2}{3}$? Show your work. (Please cover all cases in your answer.)

With the analytical expression from part c, we obtain that `k` should be a multiple of 3 and $\alpha = \frac{k}{3} - 1$, i.e., $\frac{k}{3} - 1$ elements among `A[1]` to `A[k - 1]` should have positive values.

2 Vector Processing [35 points]

A vector processor implements the following ISA:

Opcode	Operands	Latency (cycles)	Description
LD	V_{STR}, #n	1	$V_{STR} \leftarrow$ n (V_{STR} = Vector Stride Register)
LD	V_{LEN}, #n	1	$V_{LEN} \leftarrow$ n (V_{LEN} = Vector Length Register)
LDM	V_i	1	$V_{MSK} \leftarrow LSB(V_i)$ (V_{MSK} = Vector Mask Register)
CM		1	$V_{MSK} \leftarrow 1$ (clears V_{MSK} to enable writeback for all vector elements)
VLD	V_i, #Address	X, pipelined	$V_i \leftarrow Mem[Address]$
VST	V_i, #Address	X, pipelined	$Mem[Address] \leftarrow V_i$
VADD	V_i, V_j, V_k	4, pipelined	$V_i \leftarrow V_j + V_k$
VNOT	V_i	4, pipelined	$V_i \leftarrow BitwiseNOT(V_i)$
VCMPZ	V_i, V_j, V_k	4, pipelined	if($V_j == V_k$) $V_i \leftarrow$ 0xFFFF; else $V_i \leftarrow$ 0x0000

Assume the following:

- For the vector instructions (i.e., VLD, VST, VADD), the above table denotes the latency of processing a single vector element.
- All vector units are fully pipelined, and thus a vector unit outputs a vector element each cycle after the first element.
- The latency of VLD and VST is unknown to you and is denoted as **X**.
- The processor dispatches instructions to execution units in the program order.
- An execution unit cannot start executing a new vector instruction until the execution unit completes the execution of the already dispatched instruction, i.e., until all pipeline stages are cleared.
- The VLD and VST instructions share the same load/store execution unit. Similarly, VADD, VNOT, and VCMPZ share the same arithmetic execution unit. LD, LDM, and CM instructions have their own execution units.
- The size of a vector data element is 4 bytes.
- Each vector register V_i contains V_{LEN} vector elements. The total number of vector registers is 8.
- The LD and LDM instructions execute in one single cycle.
- LDM moves the least-significant bit (LSB) of each vector element in a vector register V_i into the corresponding position in V_{MSK}.
- V_{STR} and V_{LEN} are 16-bit registers. V_{MSK} has V_{LEN} bits.
- V_{MSK} enables predicated execution. Assume a simple implementation in which all V_{LEN} operations are executed, but the result writeback is turned off according to V_{MSK} (0 means writeback is turned off). Assume the LDM instruction is not subject to the mask (i.e., the result writeback is turned on for every vector element). Except LDM and CM, all other vector instructions are subject to the mask register.
- V_{MSK} is initialized with all 1's in the beginning of a program so that writeback is on for all vector elements until the V_{MSK} is modified.
- The main memory is byte addressable.
- The main memory has 64 banks. Vector elements stored in consecutive memory addresses are interleaved between the memory banks. For instance, if a vector element at address A maps to bank B, a vector element at address $A + 4$ maps to bank $(B + 1)$ % 64, where % is the modulo operator.
- There is one single memory port, which is used for reads and writes.
- The processor *does not* support chaining between vector functional units.

(a) [5 points] Assuming a vector stride of 1, what could be the **maximum** value of VLD/VST latency X if the processor does not stall when executing a single VLD or VST instruction? Explain why.

$X = 64$ cycles.

Explanation:
Since the memory has 64 banks, a VLD or VST instruction can take at most 64 cycles to complete execution for a single vector element. This is because consecutive elements map to consecutive banks, and a bank must be freed before another access to the same bank arrives. With 64-cycle VLD/VST latency, a bank completes serving a vector element and in the next cycle the bank starts serving the next element. If the VLD/VST latency is more than 64 cycles, there will be stalls due to bank contention.

Consider the following piece of code:

```
for (i = 0; i < 32; i++){
    if (A[i] != B[i])
        C[i] = B[i];
    else
        C[i] = A[i] + B[i];
}
```

(b) [15 points] Translate the code into assembly language with the **minimum number of instructions** by using the provided ISA. Note that you may need to make use of the mask register V_{MSK} to write a program with a minimum number of instructions. Also, note that you need to set the Vector Length (V_{LEN}) and the Vector Stride (V_{STR}) registers appropriately.

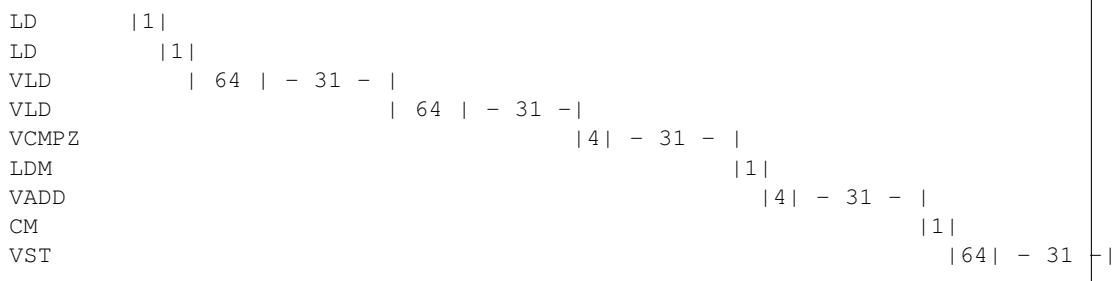
We need minimum 9 instructions.

```
LD VLEN, 32        # Load Vector Length Register
LD VSTR, 4         # Load Vector Stride Register
VLD V1, A          # Read from array A
VLD V2, B          # Read from array B
VCMPZ V3, V1, V2   # Compare V1 to V2
LDM V3             # set the mask register
VADD V2, V1, V2    # Conditionally add A and B based on the mask's status
CM                 # enable writeback to all vector elements
VST C, V2          # Write to array C
```

(c) [15 points] What is the total number of cycles needed to execute the program in part (b)? As a VLD/VST latency, use the maximum latency value you found in part (a). Show your work.

359 cycles.

Explanation:



3 Emerging Memory Technologies [25 points]

Computer scientists at ETH developed a new non-volatile memory technology, ETH-RAM. The ETH-RAM's access latency is close to that of DRAM while providing a higher density compared to the latest DRAM technologies. However, ETH-RAM has one shortcoming: it has limited endurance, i.e., a memory cell fails after 10^7 writes are performed to the cell (known as cell wear-out).

A bright ETH student has built a computer system using ETH-RAM as main memory. ETH-RAM exploits a perfect wear-leveling mechanism, i.e., a mechanism that equally distributes the writes over all of the cells of the main memory.

(a) [15 points] This student is worried about the lifetime of the computer system she has built. She executes a test program to wear out the entire ETH-RAM *as quickly as possible*. The test program runs special instructions to bypass the cache hierarchy and repeatedly writes data into different pages until all the ETH-RAM cells are worn-out. The student's measurements show that ETH-RAM stops functioning (i.e., all its cells are worn-out) in 2.5 years. Assume the following:

- The processor is in-order, and there is no memory-level parallelism.
- It takes 16 ns to send a memory request from the processor to the memory controller, and it takes 26 ns to send the request from the memory controller to ETH-RAM. The write latency of ETH-RAM is 86 ns. The total latency of a write request is 128 ns (16 ns + 26 ns + 86 ns), which cannot be overlapped by the latency of another write request (i.e., write requests are fully serialized).
- ETH-RAM requests are issued at page-level granularity. Thus, each write request writes 4096 bytes to memory.
- ETH-RAM works in the multi-level cell (MLC) mode in which each memory cell stores 4 bits of a page.

What is the capacity of ETH-RAM? Show your work. *Hint:* 2.5 years $\approx 8 \times 10^{16}$ ns.

$2^3 \times 10^{16} = \frac{size}{2^{12}} \times 10^7 \times (16 + 26 + 86)$
$size = \frac{2^3 \times 2^1 2 \times 10^{16}}{2^7 \times 10^7}$
$size = 256\ GBytes$

Explanation:

- Each memory cell should receive 10^7 writes.
- Since ETH-RAM is word addressable, the required amount of writes is equal to $\frac{size}{2^{12}} \times 10^7$.
- The processor is in-order and there is no memory-level parallelism, so the total latency of each memory access is equal to 16 + 26 + 86 = 128 ns.

(b) [10 points] The student decides to improve the lifetime of ETH-RAM cells by using the single-level cell (SLC) mode. When ETH-RAM is used in SLC mode, the lifetime of each cell improves by a factor of 10, and the write latency decreases by $n\%$. By how much must ETH-RAM's write latency decrease, assuming the lifetime of the system using SLC mode increases by 2x? Assume we repeat the experiment in part (a), with everything else remaining the same in the system, and the capacity of ETH-RAM while working in the MLC mode to be **S**. Show your work.

$2^4 \times 10^{16} = \frac{S}{2^{14}} \times 10^8 \times (16 + 26 + 86 \times p)$
$p = \frac{2^{18} \times 10^8 - 42 \times S}{86 \times S}$

Explanation:

- Each memory cell should receive $10 \times 10^7 = 10^8$ writes.
- The memory capacity is reduced by 4x since we are using SLC: Capacity $= \frac{S}{2^2}$.
- The required amount of writes is equal to $\frac{S}{S^2} \times \frac{1}{2^{12}} \times 10^8$.
- The system's lifetime improves two times (from 8 $\times 10^{16}$ ns to 16 $\times 10^{16}$ ns).

4 Data Prefetching [40 points]

Qualtel is designing a next-gen low-power mobile processor codenamed Nemo. You and your colleagues are tasked with designing the prefetcher for Nemo. Nemo has a single core, one level of cache, and a DRAM-based main memory system.

You need to examine different prefetcher designs and analyze the trade-offs involved.

- For all parts of this question, you need to compute the *coverage*, *accuracy* and *bandwidth* over of the prefetcher in its **steady state**.
- If there is a request to a cache block that has gone to main memory, a new request for the same cache block will not go to main memory as the outstanding request has not yet completed. Instead the new request will be merged with the already outstanding request in the MSHR.

You run an application `libclassical` that has the following memory access pattern (note that these are cache block addresses):

$$A,\ A+1,\ A+2,\ A+7,\ A+8,\ A+9,\ A+14,\ A+15,\ A+16,\ A+21,\ A+22,\ A+23,\ ...$$

Assume this pattern continues for a long time.

(a) [5 points] You first design a stride prefetcher that observes the last three cache block requests. If there is a constant stride S between the last three requests, the prefetcher issues a prefetch to the next cache block using the stride S. In absence of a constant stride, the prefetcher refrains from prefetching. What is the coverage of your stride prefetcher for `libclassical`? Show your work. Prefetcher coverage is defined as

$$\frac{Total\ number\ of\ correctly\ predicted\ prefetch\ requests}{Total\ number\ of\ unique\ cache\ block\ requests\ without\ the\ prefetcher}$$

0%

Explanation: The prefetcher will learn a constant stride of +1 by seeing, say, cache blocks 14, 15, and 16 and trigger a prefetch to cache block 17. But the prefetched blocks will always be useless; none of the demand requests will be covered by the prefetcher.

(b) [5 points] What is the the accuracy of your stride prefetcher for `libclassical`? Show your work. Prefetcher accuracy is defined as

$$\frac{Total\ number\ of\ correctly\ predicted\ prefetch\ requests}{Total\ number\ of\ prefetched\ requests}$$

0%

Explanation: None of the prefetched cache blocks will be accurate.

(c) [10 points] Your colleague designs a new prefetcher that, on a cache block access, prefetches the next N cache blocks. The coverage and accuracy of this prefetcher are 66.67% and 50% respectively for `libclassical`. What is the value of N? Show your work.

$N = 2$

Explanation: For an example, the prefetcher will issue prefetch requests to cache blocks 22 and 23 after seeing the request to 21. Similarly, it will issue prefetches to cache blocks 23 (which will be merged in MSHR) and 24 after seeing request to 22. Hence, every two out of three demand accesses will be covered. And every three out of six prefetch requests will be correct.

(d) [5 points] The bandwidth overhead of the prefetcher can be defined as

$$\frac{Total\ number\ of\ unique\ cache\ block\ requests\ with\ the\ prefetcher}{Total\ number\ of\ unique\ cache\ block\ requests\ without\ the\ prefetcher}$$

What is the bandwidth overhead of this next-N-block prefetcher for `libclassical`? Show your work.

5/3

Explanation: For every group of three demanded cache blocks, the prefetcher will fetch two additional cache blocks.

(e) [5 points] What is the minimum value of N required to achieve a 100% prefetch coverage for `libclassical`? Show your work. Remember that you should consider the prefetcher's coverage in its steady state.

$N = 5$

Explanation: In this case, the prefetcher can issue prefetch for, say, cache block 28 by seeing a request to cache block 23

(f) [5 points] What is the bandwidth overhead at this value of N? Show your work.

7/3

(g) [5 points] However, you are not happy with the bandwidth overhead required to achieve a prefetch coverage of 100% with a next-N-block prefetcher. You aim to design a prefetcher that achieves a coverage of 100% with a 1× bandwidth overhead. Propose a prefetcher design that accomplishes this goal. Be concrete and clear.

This is an open-ended question.
One solution can be to design a multi-stride prefetcher that can learn a chain of frequently-occurring strides, like $(+1, +1, +5)$. This learning mechanism will cover all accesses without adding any bandwidth overhead.

5 Asymmetric Multicore [30 points]

A microprocessor manufacturer asks you to design an asymmetric multicore processor for modern workloads. Your design contains one large core and several small cores, which share the same die. Assume the total die area is A units. The table below describes the area, performance, and power specifications for each core type.

Type of Core	Area (mm^2)	Performance	Dynamic Power (W)	Static Power (W)
Large	S	$\sqrt{S}$	S	$\frac{1}{4} \times S$
Small	1	1	1	0.5

The serial portion of a workload executes only on the large core, while the parallel portion executes on both large and small cores. On this multiprocessor, we will execute a workload where a fraction P of its work is parallel, and $1-P$ of its work is serial. You will fit as many small cores as possible, after placing the large core. Consider the following two configurations:

- Configuration X: A = 32, S = 4.
- Configuration Y: A = 32, S = 16.

Please answer the following questions. Show your work. Express your equations and solve them.

(a) [6 points] For what values of P does the workload run faster on Y than on X? Show your work.

$P < \frac{60}{64} \Rightarrow P < 0.94$

Explanation:
Given that the large core occupies 4 units, the number of small core for Configuration X is $32 - 4 = 28$. Thus, we calculate the speedup for this configuration following Amdahl's law, as:
$Speedup_x = \frac{1}{\frac{1-P}{\sqrt{4}} + \frac{P}{\sqrt{4}+(32-4)\times 1}}$
$Speedup_x = \frac{1}{\frac{1-P}{2} + \frac{P}{2+28}}$
$Speedup_x = \frac{1}{\frac{1-P}{2} + \frac{P}{30}}$

Given that the large core occupies 16 units, the number of small core for Configuration Y is $32 - 16 = 16$. Thus, we calculate the speedup for this configuration following Amdahl's law, as:
$Speedup_y = \frac{1}{\frac{1-P}{\sqrt{16}} + \frac{P}{\sqrt{16}+(32-16)\times 1}}$
$Speedup_y = \frac{1}{\frac{1-P}{4} + \frac{P}{4+16}}$
$Speedup_y = \frac{1}{\frac{1-P}{4} + \frac{P}{20}}$

$Speedup_y > Speedup_x$
$\Rightarrow {Speedup_y}^{-1} < {Speedup_x}^{-1}$
$\Rightarrow \frac{1}{\frac{1-P}{4} + \frac{P}{20}} < \frac{1}{\frac{1-P}{2} + \frac{P}{30}}$
$\Rightarrow 24 \times P - \frac{8 \times P}{3} < 20$
$\Rightarrow \frac{64 \times P}{3} < 20$
$\Rightarrow P < \frac{60}{64}$

(b) [6 points] For what values of P does the workload consumes less energy when running on Y than on X? Show your work.

$P < \frac{75}{94} \Rightarrow P < 0.80$

Explanation:
We calculate the energy consumption as:
$E_{total} = E_{large} + E_{small}$
$E_{large} = (P_{large\_dynamic} + P_{large\_static}) \times t_{serial} + (P_{large\_dynamic} + P_{large\_static}) \times t_{parallel}$
$E_{small} = (P_{small\_dynamic} + P_{small\_static}) \times t_{parallel} + (P_{small\_static}) \times t_{serial}$

Thus:
$Energy_y = 20 \times \frac{1-P}{4} + 20 \times \frac{P}{20} + 16 \times [1.5 \times \frac{P}{20} + 0.5 \times \frac{1-P}{4}]$
$Energy_x = 5 \times \frac{1-P}{2} + 5 \times \frac{P}{30} + 28 \times [1.5 \times \frac{P}{30} + 0.5 \times \frac{1-P}{2}]$
$Energy_y < Energy_x$
$P < \frac{75}{94}$

5.1 Accelerating Single-Thread Execution

Assume that two large cores can operate in a collaborative manner to achieve the single-thread performance of an even "larger" core that is **N**× faster than the largest core on the chip. When executing the serial portion of the workload, the functional units of both large cores are merged into the same pipeline to have a faster core. The collaborative execution mode is only enabled during the serial portion of the workload. During the parallel portion, the two large cores separate from each other (on-the-fly) and operate as two independent cores. The serial portion executes only on the "dual-core", and the parallel portion executes on all the cores. The table below describes the area and performance specifications for each core for this design.

Type of Core	**Area (mm^2)**	**Performance**
$Large_1$	S_1	$\sqrt{S_1}$
$Large_2$	S_2	$\sqrt{S_2}$
$Large_1 + Large_2$	$S_1 + S_2$	$\mathbf{N} \times \sqrt{Max(S_1, S_2)}$
Small	1	1

Consider the following configuration:

- Configuration Z: A = 32, $S_1 = 9$, $S_2 = 4$.

(c) [6 points] For what values of N does the workload run faster on Z than on X? Assume $P = 0.8$. Show your work.

$N > 0.71$

Explanation:
$Speedup_z = \frac{1}{\frac{1-P}{N\times(\sqrt{9})} + \frac{P}{3+2+19}}$
$Speedup_x = \frac{1}{\frac{1-P}{2} + \frac{P}{2+28}}$
$\Rightarrow \frac{1-P}{3\times N} + \frac{P}{24} < \frac{1-P}{2} + \frac{P}{30}$
$\Rightarrow \frac{3\times N}{1-P} > \frac{2\times 30\times 24}{30\times 24\times(1-P)+2\times 24\times P-2\times 30\times P}$
$\Rightarrow N > \frac{480-480\times P}{720-732\times P}$
$\Rightarrow N > \frac{480-480\times 0.8}{720-732\times 0.8}$
$\Rightarrow N > 0.71$

(d) [6 points] For what values of P does the workload run faster on Z than on X? Assume $N = 1.5$. Show your work.

$P < \frac{100}{103} \Rightarrow P < 0.97$

Explanation:
$\frac{1-P}{0.5\times(\sqrt{9})} + \frac{P}{3+2+19} < \frac{1-P}{2} + \frac{P}{2+28}$
$\Rightarrow P < \frac{100}{103}$

(e) [6 points] Suppose you are executing workloads where a fraction P of its work is *infinitely* parallelizable. Which configuration would you choose? Z or Y? Why?

Z, since it provides speedup for larger values of P (i.e., $P_Y < P_Z \Rightarrow 0.94 < 0.97$)

6 Bottleneck Acceleration [25 points]

In this question, you are asked to analyze the performance and scalability benefits of accelerating the critical section in the following piece of code.

```
while (!problem_solved){
    Lock(X) //start of the critical section
        SubProblem = PQ.Dequeue();
    Unlock(X) //end of the critical section

    problem_solved = Solve(SubProblem)
    if(problem_solved)
      break;
}
```

Assume that

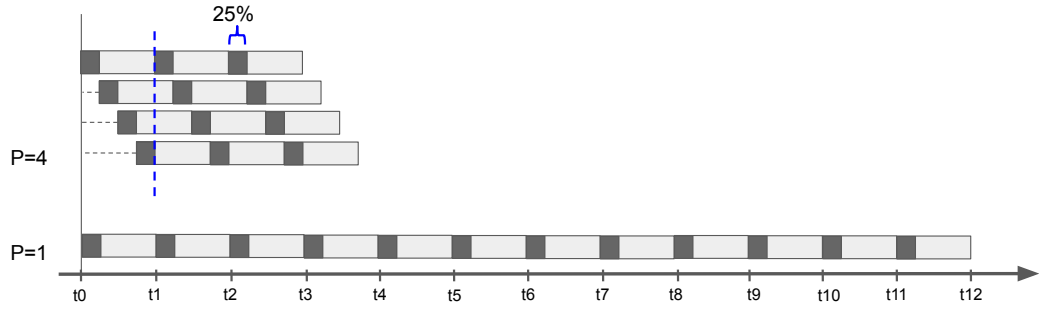
- the while loop iterates 12 times.
- there is no data dependency across iterations.

(a) [15 points] We observe that the application's performance saturates when the iterations are distributed across four threads (i.e., running more than four threads does not improve the performance). We are interested in the ratio of time spent executing critical sections to the total execution time for each iteration. Calculate the range of all possible values for this ratio. (Hint: The performance saturates with P threads if it can achieve speedup from increasing the number of threads from P-1 to P, and not from P to P+1.)

(1/3,1/4]

Explanation:

The application will be scalable to four threads if the ratio of critical section execution time to the total execution time is within this range : (1/3,1/4]



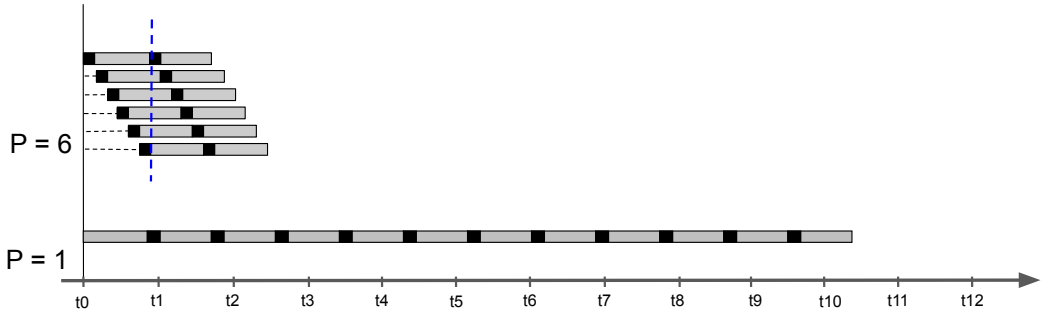
(b) [10 points] In order to improve the performance and scalability of the program, we decide to accelerate the critical section by migrating its execution to a more powerful core. The execution of the critical section on the powerful core completes faster, including the cost of migration between cores. We use the powerful core only for executing the critical section, while the rest of the iteration executes on smaller less powerful cores. Then, we observe that the application's scalability increases up to six cores.

Assume that the critical section initially takes the minimum ratio of execution time that you found in the previous section. How much does the powerful core increase the performance of the critical section in this case? Show the range for possible performance improvements.

[6/4,5/4)

Explanation:

In this case, the ratio of the critical section execution time to the total execution time is within this range: (1/5,1/6]
Compared to the initial case, in which the critical section takes 1/4 of the execution time, the speedup is within this range: [6/4,5/4)



7 Cache Coherence [35 points]

We have a system with 4 processors {P0, P1, P2, P3} that can access memory at byte granularity. Each processor has a private data cache with the following characteristics:

- Capacity of 256 bytes
- Direct-mapped
- Write-back
- Block size of 64 bytes

Each processor has also a dedicated private cache for instructions. The characteristics of the instruction caches are not necessary to solve this question.

All data caches are connected to and actively snoop a global bus, and cache coherence is maintained using the MESI protocol, as we discussed in class. Note that on a write to a cache block in the S state, the block will transition directly to the M state. The range of accessible memory addresses is from `0x00000` *to* `0xfffff`.

The semantics of the instructions used in this question is the following:

Opcode	Operands	Description
ld	rx,[ry]	rx ← Mem[ry]
st	rx,[ry]	rx → Mem[ry]
addi	rx,#VAL	rx ← rx + VAL
j	TARGET	jump to TARGET
beq	rx,ry,TARGET	if([rx]==[ry]) jump to TARGET

Each processor executes the following instructions in a *sequentially consistent* manner:

	P0
0	`ld r1,[r2]`
1	`addi r1,#1`
2	`st r1,[r2]`
3	`LP : ld r1,[r2]`
4	`beq r1,r4,END`
5	`j LP`
6	`END: st r4, [r2]`

	P1
0	`LP : ld r1,[r4]`
1	`beq r1,r3,END`
2	`j LP`
3	`END: addi r1,#1`
4	`st r1,[r4]`
-	
-	

	P2
0	`LP : ld r1,[r5]`
1	`beq r1,r3,END`
2	`j LP`
3	`END: addi r1,#1`
4	`st r1,[r5]`
-	
-	

	P3
0	`LP : ld r1,[r2]`
1	`beq r1,r3,END`
2	`j LP`
3	`END: addi r1,#1`
4	`st r1,[r2]`
-	
-	

The initial state of the caches is unknown. After an arbitrarily large amount of time, all cores finish executing their code. The *final* tag store state of each *data* cache is as follows:

Final Tag Store States

Cache for P0		
Set	*Tag*	*MESI state*
0	0x100	M
1	0xfff	M
2	0x010	S
3	0x110	I

Cache for P1		
Set	*Tag*	*MESI state*
0	0x100	I
1	0xfff	I
2	0x010	S
3	0x110	I

Cache for P2		
Set	*Tag*	*MESI state*
0	0x100	I
1	0xfff	I
2	0x011	E
3	0x110	S

Cache for P3		
Set	*Tag*	*MESI state*
0	0x100	I
1	0xff1	S
2	0x010	I
3	0x10f	S

(a) [25 points] What are the initial values of the registers in each of the 4 processors that ensure that the above final tag store states are deterministic (i.e., the final states are independent of the order in which the memory requests are issued to memory)? Explain your answer.

Solution:
P0: r1: X1, r2: 0x100YZ, r3: C , r4: C+4
P1: r1: X2, r4: 0x100YZ, r3: C+1
P2: r1: X3, r5: 0x100YZ, r3: C+2
P3: r1: X4, r2: 0x100YZ, r3: C+3

Where X1, X2, X3, and X4 can be any value (these values are overwriten), Y is an integer number between 0x0 and 0x3, Z is an integer number between 0x0 and 0xF, and C is any number. The values of r3 in P1, P2, and P3 are interchangeable (e.g., the nex r3 values are also valid: P1: r3 = C+2, P3: r3 = C+1, P2: r3 = C+3).

Explanation:
We need to find a solution that 1) does not have infinite loops (i.e., the program finish execution), 2) the final state of the tag store is deterministic, and 3) the final states are the ones provided in figure.We observe that 1) each individual thread reads and writes a unique memory position, and 2) all threads write to memory at the end of the execution. Because only P0 has cache blocks in Modified estate, we can conclude that all the other threads have to be accessed to the same cache block. Otherwise, there would be some modified block in P1, P2, or P3.
Because the only cache block that is present in all processors is the block in set 0, we conclude that the content of the register r2 in P0, r4 in P1, r5 in P2 and r2 in P3 are accessing an address in the same block, which is in Set 0, with tag 0x100. Because Set 0, Tag 0x100 is only valid and with M state in P0, we conclude that P0 should execute its last store instruction always the last. We can ensure this by trigering the END of all processors in sequence. For doing that, all processors to access exactly the same byte in the cache block. Therefore, all processors should access the same address 0x100YZ, where Y is an integer number between 0x0 and 0x3 (the 2 most significant bits of Y represent the set 0), and Z is an integer number between 0x0 and 0xF.
The content of r3 in P0 can be any value C, and the content of r3 in P1, P2 and P3 should contain consecutive values. For example, this are some valid values: P1-r3 = C+1, P2-r3 = C+2, P3-r3 = C+3, and this is other combination of valid values: P3-r3 = C+1, P1-r3 = C+2, P2-r3 = C+3. This are the only values that ensure that all threads finish in the same deterministic order. Finally, r4 in P0 should have the value C+4 to ensure that is the last one into writing to memory.

(b) [10 points] Fill in the following tables with the *initial* tag store states (i.e., *Tag* and *MESI* state) before having executed the instructions shown above. Answer X if a tag value is unknown, and for the *MESI* states, write in *all possible values* (i.e., `M`, `E`, `S`, and/or `I`).

Initial Tag Store States

Cache for P0		
Set	*Tag*	*MESI state*
0	`X`	`M, E, S, I`
1	`0xfff`	`M`
2	`0x010`	`S`
3	`0x110`	`I`

Cache for P1		
Set	*Tag*	*MESI state*
0	`X`	`M, E, S, I`
1	`0xfff`	`I`
2	`0x010`	`S`
3	`0x110`	`I`

Cache for P2		
Set	*Tag*	*MESI state*
0	`X`	`M, E, S, I`
1	`0xfff`	`I`
2	`0x011`	`E`
3	`0x110`	`S`

Cache for P3		
Set	*Tag*	*MESI state*
0	`X`	`M, E, S, I`
1	`0xff1`	`S`
2	`0x010`	`I`
3	`0x10f`	`S`

8 Memory Consistency [35 points]

A programmer writes the following two C code segments. She wants to run them concurrently on a multicore processor, called SC, using two different threads, each of which will run on a different core. The processor implements *sequential consistency*, as we discussed in the lecture.

	Thread T0
Instr. T0.0	`X[0] = 2;`
Instr. T0.1	`flag[0] = 1;`
Instr. T0.2	`a = X[0]*2;`
Instr. T0.3	`b = Y[0]-1;`
Instr. T0.4	`c = X[0];`

	Thread T1
Instr. T1.0	`X[0] = 1;`
Instr. T1.1	`X[0] += 2;`
Instr. T1.2	`while(flag[0] == 1);`
Instr. T1.3	`a = flag[0];`
Instr. T1.4	`X[0] = 2;`
Instr. T1.5	`Y[0] = 10;`

`X` and `flag` have been allocated in main memory. Thread 0 and Thread 1 have their private processor registers to store the values of `a` , `b`, and `c`. A read or write to any of these variables generates a single memory request. The initial values of all memory locations and variables are 1. Assume each line of the C code segment of a thread is a *single* instruction.

(a) [5 points] Do you find something that could be wrong in the C code segments? Explain your answer.

Thread 1 will never finish.

Explanation:
The while loop in instruction T1.2 is an infinite loop, because the value of `flag[0]` is 1 since the beginning of the program.

(b) [10 points] What could be possible final values of `X[0]` in the SC processor, after executing both C code segments? Explain your answer. Provide all possible values.

2, 3, or 4.

Explanation:
The sequential consistency model ensures that the operations of each individual thread are executed in the order specified by its program. Across threads, the ordering is enforced by the use of `flag[0]`. Thread 1 will remain in instruction T1.2 until `flag[0]` has a value that is not 1. However, thread 1 will never finish execution. There are *at least* three possible sequentially-consistent orderings that lead to *at most* three different values of `X` at the end:
Ordering 1: T1.0 → T1.1 → T0.0, Final value: `X[0]` = 2.
Ordering 2: T0.0 → T1.0 → T1.1, Final value: `X[0]` = 3.
Ordering 3: T1.0 → T0.0 → T1.1, Final value: `X[0]` = 4.

(c) [5 points] What could be possible final values of `a` in the SC processor, after executing both C code segments? Explain your answer. Provide all possible values.

2, 4, 6, or 8.

Explanation:
The value of `a` is twice the value of `X[0]`:
Ordering 1: T1.0 → T1.1 → T0.0 → T0.2, Final value: `X[0]` = 4.
Ordering 2: T0.0 → T1.0 → T1.1 → T0.2, Final value: `X[0]` = 6.
Ordering 3: T1.0 → T0.0 → T1.1 → T0.2, Final value: `X[0]` = 8.

Ordering 4: T0.0 → T0.1 → T1.0 → T0.2, Final value: `X[0]` = 2.

(d) [5 points] What could be possible final values of `b` in the SC processor, after both threads finish execution? Explain your answer. Provide all possible values.

0.

Explanation:
Because the value of `b` depends only on the value of `Y[0]` (instruction T0.3). The initial value of `Y[0]` is 1. Instruction T1.4 will not be executed as T1 enters an infinite loop after executing instruction T1.2.

(e) [10 points] With the aim of achieving higher performance, the programmer tests her code on a new multicore processor, called NC, that does not implement memory consistency. Thus, there is *no* guarantee on the ordering of instructions as seen by different cores.

What is the final value of `X[0]` in the NC processor, after executing both threads? Explain your answer.

1, 2, 3, or 4.

Explanation:
Since there is no guarantee of a strict order of memory operations, as seen by different cores, instruction T1.1 could complete before or after instruction T1.0, from the perspective of the core that executes thread 0. If instruction T1.1 completes before instruction T1.0, from the perspective of the core that executes T0, instruction T0.0 could complete before or after instruction T1.0. Thus, there are at least five possible weakly-consistent orderings that lead to different values of `X[0]` at the end:
Ordering 1: T0.0 → T1.1 → T1.0, Final value: `X[0]` = 1.
Ordering 2: T1.4 → T0.0 → T1.1 → T1.0, Final value: `X[0]` = 1.
Ordering 3: T1.0 → T1.1 → T0.1, Final value: `X[0]` = 2.
Ordering 4: T0.0 → T1.0 → T1.1, Final value: `X[0]` = 3.
Ordering 5: T1.0 → T0.0 → T1.1, Final value: `X[0]` = 4.