CMU 18-447 Introduction to Computer Architecture, Spring 2013

Final Exam

Date: Mon., 5/6

Instructor: Onur Mutlu
TAs: Justin Meza, Yoongu Kim, Jason Lin

Name:

Legibility & Name (6 Points):	
Problem 1 (35 Points):	
Problem 2 (35 Points):	
Problem 3 (30 Points):	
Problem 4 (30 Points):	
Problem 5 (30 Points):	
Problem 6 (35 Points):	
Problem 7 (55 Points):	
Problem 8 (55 Points):	
Bonus (44 Points):	
Total (355 Points):	

Instructions:

1. This is a closed book exam. You are allowed to have three letter-sized cheat sheets.
2. No electronic devices may be used.
3. This exam lasts 3 hours.
4. Clearly indicate your final answer for each problem.
5. Please show your work when needed.
6. Please write your initials at the top of every page.
7. Please make sure that your answers to all questions (and all supporting work that is required) are contained in the space required.

Tips:

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

1. The GPU Strikes Back! [35 points]

We define the *SIMD utilization* of a program run on a GPU as the fraction of SIMD lanes that are kept busy with *active threads* during the run of a program.

The following code segment is run on a GPU. Each thread executes **a single iteration** of the shown loop. Assume that the data values of the arrays A, B, and C are already in vector registers so there are no loads and stores in this program. (Hint: Notice that there are 5 instructions in each thread.) A warp in the GPU consists of 32 threads, and there are 32 SIMD lanes in the GPU.

```
for (i = 0; i < (512*1024); i++) {
    if (B[i] < 0) {
       A[i] = A[i] * C[i];
       B[i] = A[i] + B[i];
       C[i] = C[i] + 1;
       B[i] = B[i] + 1;
    }
}
```

(a) How many warps does it take to execute this program?

Warps = (Number of threads) / (Number of threads per warp)
Number of threads = 2^{19} (i.e., one thread per loop iteration).
Number of threads per warp = $32 = 2^5$ (given).
Warps = $2^{19}/2^5 = 2^{14}$

(b) When we measure the SIMD utilization for this program with one input set, we find that it is 9/40. What can you say about arrays A, B, and C? Be precise.

A: Nothing.

B: 1 in every 32 of B's elements are negative.

C: Nothing.

(c) Is it possible for this program to yield a SIMD utilization of 20% (circle one)?

YES **NO**

If YES, what should be true about arrays A, B, and C for the SIMD utilization to be 20%? Be precise.

A:

B:

C:

If NO, explain why not.

The smallest SIMD utilization possible is the same as part (b), 36/160, but this is greater than 20%.

(d) Is it possible for this program to yield a SIMD utilization of 100% (circle one)?

YES NO

If YES, what should be true about arrays A, B, C for the SIMD utilization to be 100%? Be precise.

A: Nothing.

B: Either:
(1) All of B's elements are less than 0, or
(2) All of B's elements are greater than or equal to 0.

C: Nothing.

If NO, explain why not.

2. Potpourri Express [35 points]

(a) **Prefetcher Timeliness**

Suppose we are characterizing the performance of a processor that has a single-level, direct-mapped cache. We consider the addition of a newly-designed fancy prefetcher to this processor. We take an important application, and measure its performance with and without prefetching enabled. We measure the *timeliness* of the prefetcher as the fraction of prefetched blocks that were needed by the processor after being inserted into the L1 cache divided by all prefetched blocks. We find that, when prefetching is enabled, performance improves by 50% and the timeliness is 0. What is going on? (In other words, why is there performance improvement even though the prefetcher is *not timely*?)

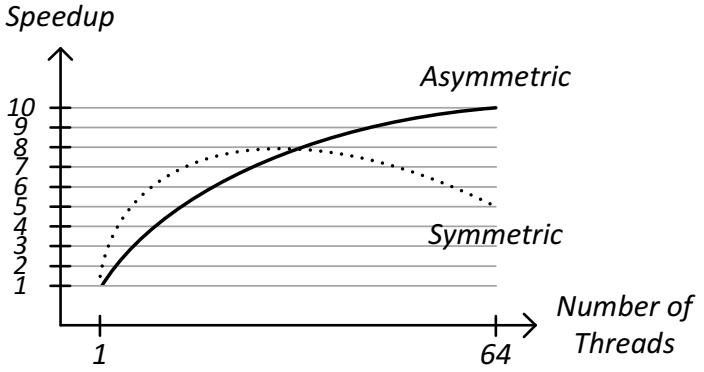
There are prefetches that are used before being inserted into the cache, i.e., the prefetcher covers some of the latency of cache misses but not all (leading to a timeliness of 0 with the described metric).

Then, we take another application, and measure its performance on the same processor, with and without prefetching enabled. When prefetching is enabled, performance degrades by 50% and the timeliness is 1 (i.e., the prefetcher is *perfectly timely*). What is going on?

The prefetcher is inaccurate, leading to cache pollution and bandwidth contention (even though accurate prefetches are perfectly timely).

(b) **Asymmetry**

The figure below shows the speedup curve for a workload on two systems: symmetric multicore and asymmetric multicore. Assume an area budget of 64 small cores.



What is the performance improvement of the asymmetric multicore over the symmetric one?

10/8 = 1.25 (you need to compare the two systems at their best number of threads, to make a fair comparison)

(c) **Routing Algorithms**

Routing decisions in a deterministic or oblivious routing algorithm are made without taking into account [Network state], whereas those in an adaptive routing algorithm are made by taking into account [Network state].

(d) **Path Diversity**

Remember we discussed the notion of "path diversity" in class. This notion refers to the phenomenon that many different paths can exist from a source node to a destination node.

(i) Rank the following topologies in terms of their path diversity (the one with higher diversity should appear on the left):

- (T) Torus
- (R) Ring
- (M) Mesh

[T] > [M] > [R]

(ii) Rank the following routing algorithms in terms of their ability to exploit path diversity (the one with higher ability should appear on the left):

- (D) Dimension-order routing
- (B) Bufferless deflection routing
- (V) Valiant's algorithm

[B] > [V] > [D]

(e) **Friendly Advice**

Your friend says: "Let's use a bus to connect these two caches because the bus ensures memory operations to two different cache blocks to be serialized. This would ease our implementation of the MESI protocol."

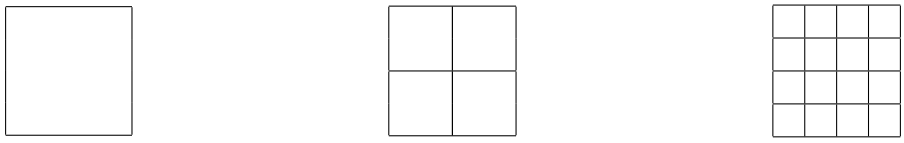
What would you say to your friend? Why? (Hint: Is there anything wrong with your friend's statement?)

Your friend is confusing the two distinct concepts of coherence and consistency

3. Amdahl's Law [30 points]

Consider the following three processors (X, Y, and Z) that are fabricated on a constant silicon area of $16A$. Assume that the single-thread performance of a core increases with the square root of its area.

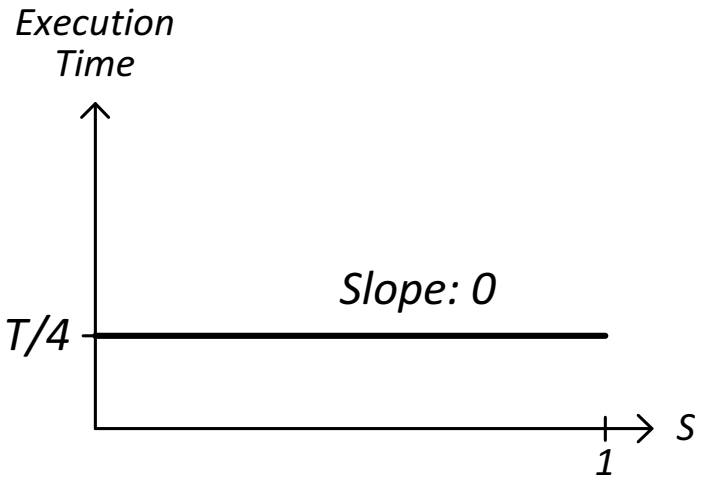
Processor X	**Processor Y**	**Processor Z**
1 *large* core of area $16A$	4 *medium* cores of area $4A$	16 *small* cores of area A



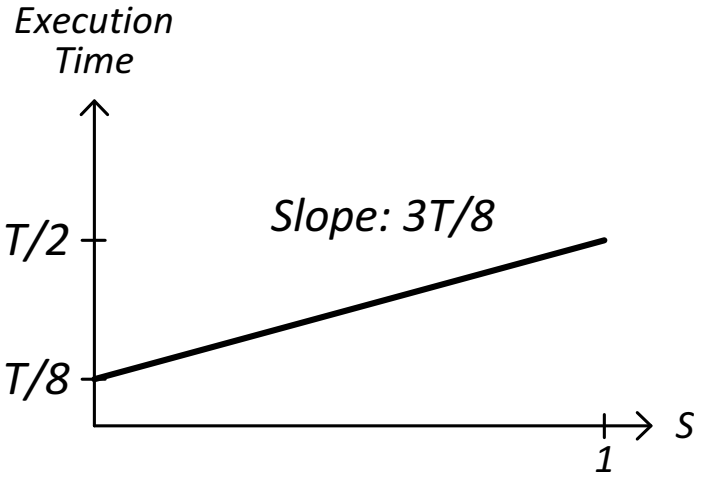
On each of the three processors, we will execute a workload where S fraction of its work is serial, and where $1 - S$ fraction of its work is **infinitely** parallelizable. As a function of S, plot the execution time of the workload on each of the three processors. Assume that it takes time T to execute the entire workload using only one of the small cores of Processor Z.

Please label the value of the **y-axis intercept** and the **slope**.

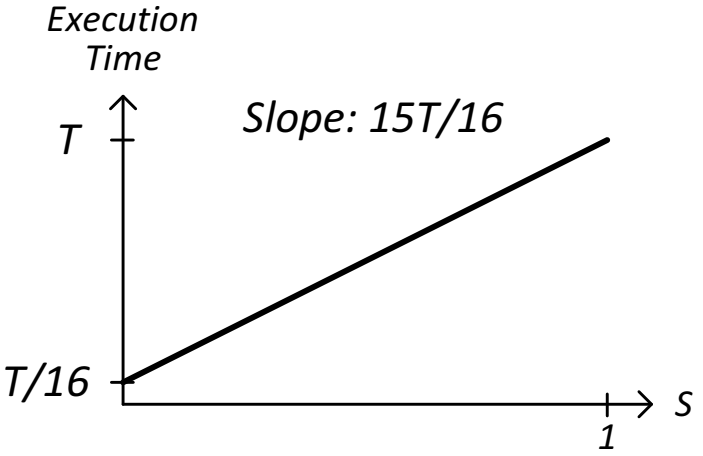
(a) Processor X



(b) Processor Y



(c) Processor Z



(d) Which processor has the lowest execution time for the widest range of S?

X

(e) Typically, for a realistic workload, the parallel fraction is not infinitely parallelizable. What are the three fundamental reasons why?

1. Synchronization
2. Load imbalance
3. Resource contention

4. Sequential Consistency [30 points]

Two threads (A and B) are concurrently running on a dual-core processor that implements a *sequentially consistent* memory model. Assume that the value at address 0x1000 is initialized to 0.

Thread A

```
X1: st 0x1, (0x1000)
X2: ld $r1, (0x1000)
X3: st 0x2, (0x1000)
X4: ld $r2, (0x1000)
```

Thread B

```
Y1: st 0x3, (0x1000)
Y2: ld $r3, (0x1000)
Y3: st 0x4, (0x1000)
Y4: ld $r4, (0x1000)
```

(a) List all possible values that can be stored in `$r3` after both threads have finished executing.

0x1, 0x2, 0x3

(b) After both threads have finished executing, you find that (`$r1`, `$r2`, `$r3`, `$r4`) = (1, 2, 3, 4). How many different *instruction interleavings* of the two threads produce this result?

X1/X2 have to execute "back-to-back". The same goes for X3/X4, Y1/Y2 and Y3/Y4. Let us call these instruction pairs as P, Q, R, and S.

Six possible interleavings:
P → Q → R → S
P → R → Q → S
R → P → Q → S
R → P → S → Q
R → S → P → Q
P → R → S → Q

(c) What is the total number of all possible instruction interleavings? You need not expand factorials.

Eight instructions in total are executed. Among them, you must choose the four that are from Thread A.

8-Choose-4 = 8!/4!/4!.

(d) On a *non-sequentially consistent* processor, is the total number of all possible instruction interleavings less than, equal to, or greater than your answer to question (c)?

The same. Since all the memory accesses are to the same address, they cannot be reordered more aggressively without affecting the correctness within even a single thread.

5. Coherent Caches [30 points]

You just got back a prototype of your latest processor design, which has two cores and uses the MESI cache coherence protocol for each core's private L1 caches.

(a) **Scenario 1** Let's say that you discover that there are some bugs in the design of your processor's coherence modules. Specifically, the `BusRead` and `BusWrite` signals on the module occasionally do *not* get asserted when they should have (but data still gets transferred correctly to the cache).

Fill in the table below with a ✔ if, for each MESI state, the missing signal has no effect on correctness. If correctness may be affected, fill in a ✖.

	`BusRead`	`BusWrite`
M	✖	✖
E	✖	✖
S	✔	✖
I	✔	✔

(b) **Scenario 2** Let's say that instead you discover that there are no bugs in the design of your processor's coherence modules. Instead, however, you find that occasionally cosmic rays strike the MESI state storage in your coherence modules, causing a state to instantaneously change to another.

Fill in a cell in the table below with a ✔ if, for a starting MESI state on the top, instantaneously changing the state to the state on the left affects neither correctness nor performance. Fill in a cell in the table with a ◯ if correctness is not affected but performance could be affected by the state change. If correctness may be affected, fill in a ✖.

Ending State (Current Status) \ Starting State (Real Status)	M	E	S	I
M	✔	◯	✖	✖
E	✖	✔	✖	✖
S	✖	◯	✔	✖
I	✖	◯	◯	✔

6. Programming a Systolic Array [35 points]

Figure 1 shows a systolic array processing element.

Each processing element takes in two inputs, M and N, and outputs P and Q. Each processing element also contains an "accumulator" R that can be read from and written to. The initial value of the "accumulator" is 0.

Figure 2 shows a systolic array composed of 9 processing elements. The smaller boxes are the inputs to the systolic array and the larger boxes are the processing elements. You will program this systolic array to perform the following calculation:

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix}$$

In each time cycle, each processing element will take in its two inputs, perform any necessary actions, and write on its outputs. The time cycle labels on the input boxes determine which time cycle the inputs will be fed into their corresponding processing elements. Any processing element input that is not driven will default to 0, and any processing element that has no output arrow will have its output ignored.

After all the calculations finish, each processing element's "accumulator" will hold one element of the final result matrix, arranged in the correct order.

(a) Please describe the operations that each individual processing element performs, using mathematical equations and the variables M, N, P, Q and R.

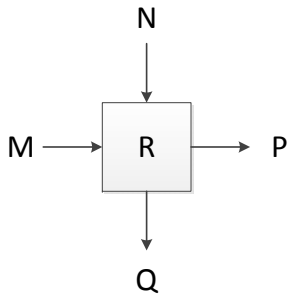


Figure 1: A systolic array processing element

P = M

Q = N

R = R + M × N

(b) Please fill in all 30 input boxes in Figure 2 so that the systolic array computes the correct matrix multiplication result described on the previous page. (Hint: Use a_{ij} and b_{ij}.)

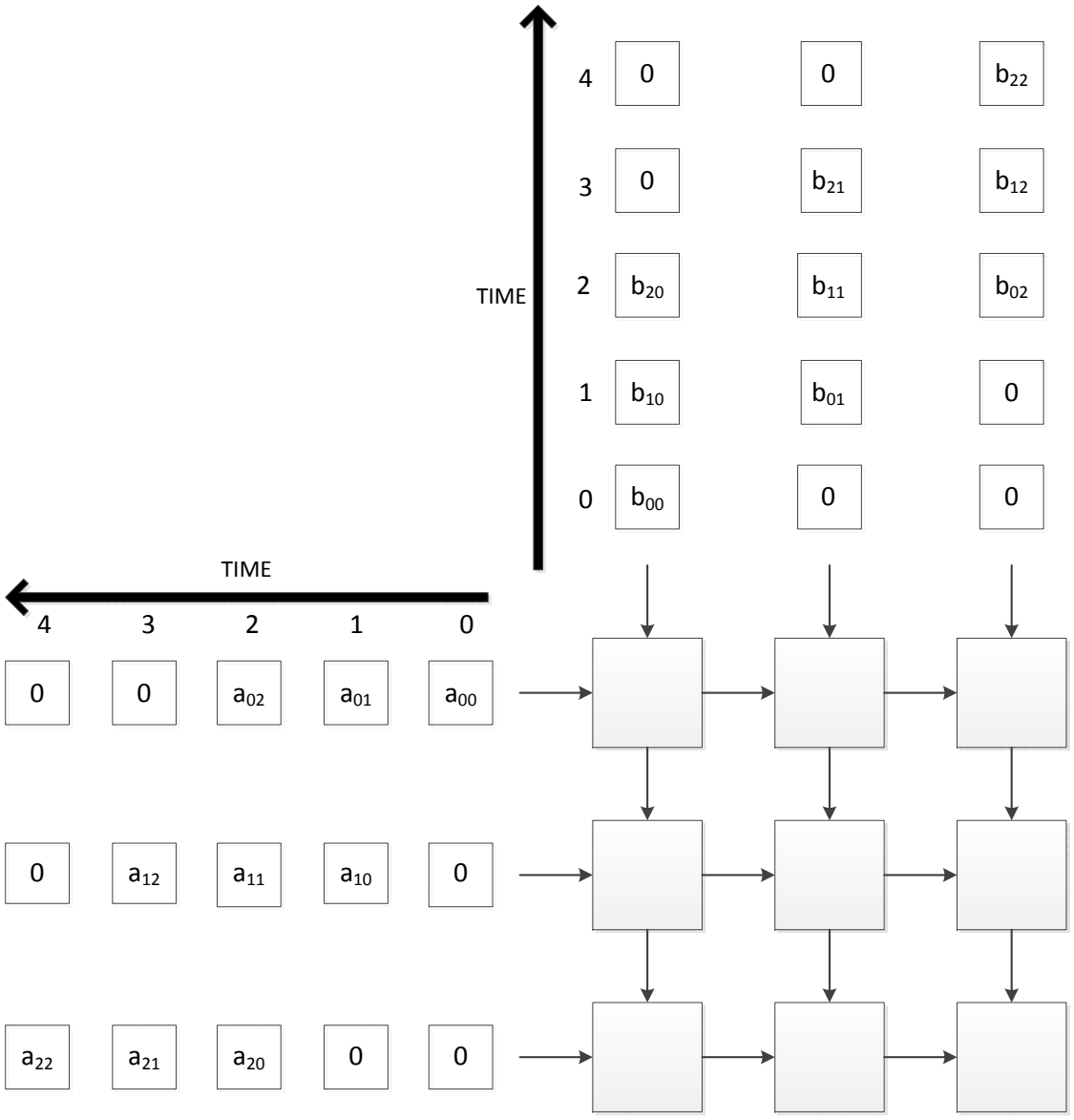
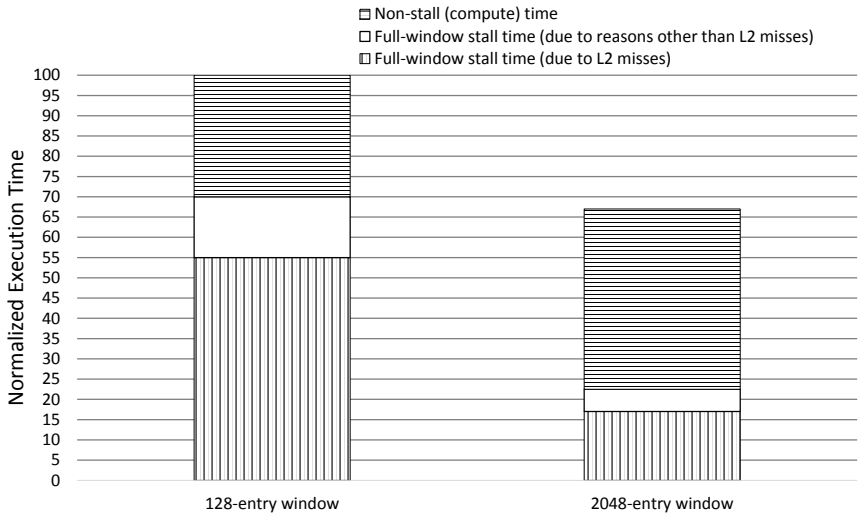


Figure 2: A systolic array

7. Analyzing the Window [55 points]



The figure above shows the execution time breakdown of a modern, Pentium 4-like out-of-order execution processor, averaged across 147 memory-intensive workloads. The processor is a 3-wide superscalar processor, the uncontended memory latency is 500 cycles, the memory has 16 banks, and an aggressive stream-based prefetcher is used.

This is the same figure you have seen in Lecture 28, Slide 24, redrawn.

Execution time is broken down into two categories:

(1) the time the processor spends on full window stalls, i.e., cycles in which in-flight instructions cannot be retired nor new instructions can be placed into the window (i.e., scheduled), due to:

   (i) L2 cache misses (i.e., the processor stalls because the oldest instruction is an L2 cache miss).

   (ii) reasons other than L2 cache misses.

(2) all the remaining time, designated as non-stall (compute) time.

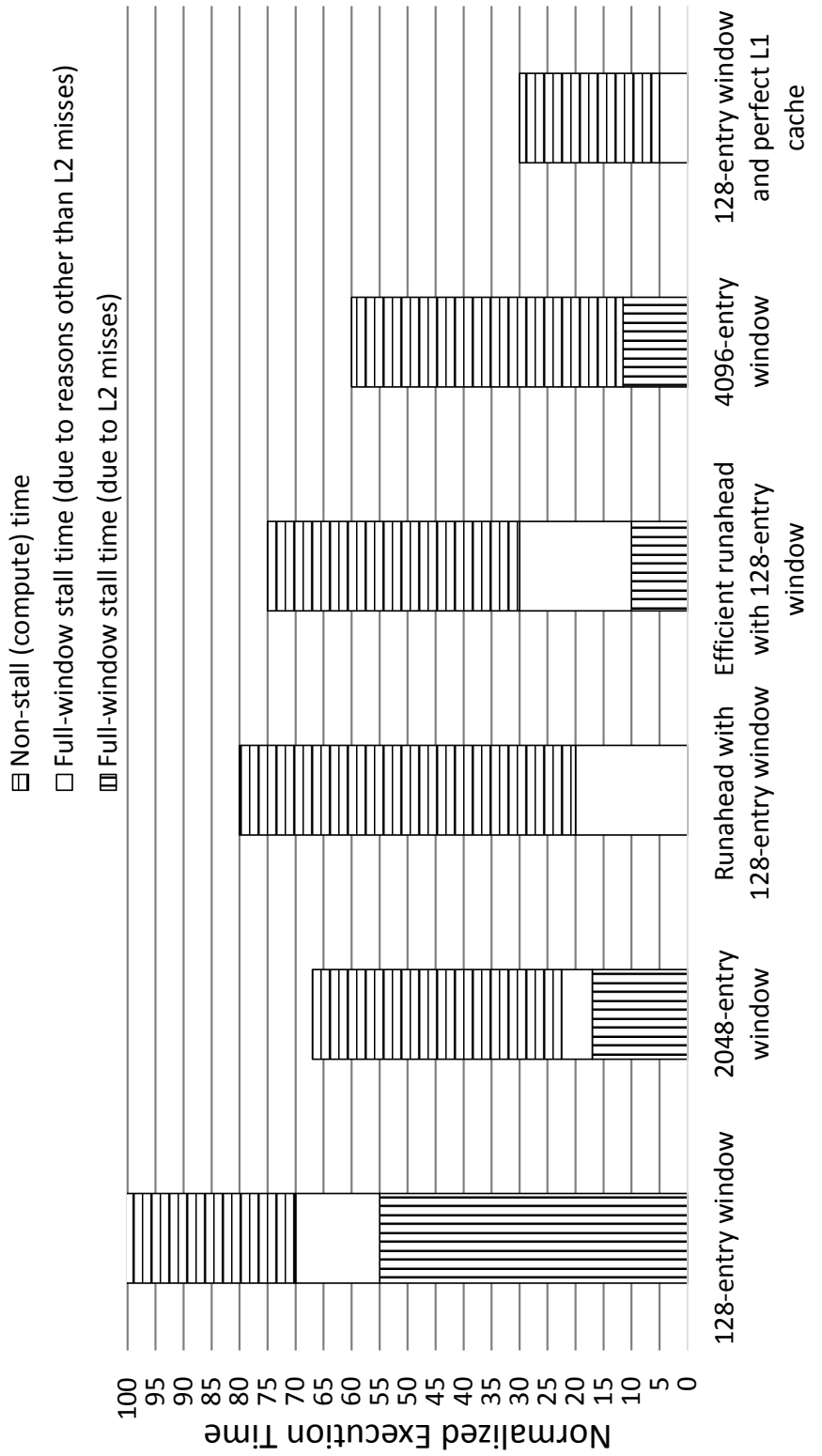
The stall-time is further broken down into the time the processor stalls because the oldest instruction is an L2 cache miss.

(a) Why does the full-window stall time decrease as the instruction window size increases from 128 (the baseline) to 2048 entries?

> A larger window reduces the number of full window stalls as it allows more instructions to be buffered.

(b) Why does the non-stall time increase as the instruction window size increases from 128 to 2048 entries?

> A larger window spends more time executing on the wrong path (due to reduced stalls).


Non-stall (compute) time
Full-window stall time (due to reasons other than L2 misses)
Full-window stall time (due to L2 misses)
Normalized Execution Time
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
128-entry window
2048-entry window
Runahead with 128-entry window
Efficient runahead with 128-entry window
4096-entry window
128-entry window and perfect L1 cache

(c) Given that memory is 500 cycles and the processor is 3-wide superscalar, why are there full-window stalls due to L2 cache misses in a 2048-entry window processor?

Some memory accesses incur bank conflicts, whose latencies cannot be tolerated by the 2048-entry window.

(d) Why are there full-window stalls *not* due to L2 cache misses in a 2048-entry window processor?

Due to other long latency operations (e.g., long floating point operation dependence chains).

(e) In the figure on page 14, draw a third bar on the right, above the x-axis label "Runahead with 128-entry window", showing what the execution time would look like on a runahead processor with a 128-entry instruction window. Assume execution time decreases by 20% with runahead execution compared to the baseline. Show the same categories as in other bars.

(f) In the figure on page 14, draw a fourth bar on the right, above the x-axis label "Efficient runahead with 128-entry window", showing what the execution time would look like on an *efficient* runahead processor (with a 128-entry instruction window) that employs mechanisms to reduce useless, overlapping, and short runahead periods, as discussed in class. Assume the average execution time decreases by 25% with this processor compared to the baseline.

(g) How would you reduce the non-stall time in the 2048-entry window processor?

Implement a more accurate branch predictor.

(h) In the figure on page 14, draw a fifth bar on the right, above the x-axis label "4096-entry window", showing what the execution time would look like on a 4096-entry window processor. Assume this processor reduces execution time by 40% over the baseline.

(i) Now, in the figure on page 14, draw a final bar, above the x-axis label "128-entry window and perfect L1 cache", showing what the execution time would look like on a processor with a 128-entry instruction window and a perfect L1 cache.

8. Cache Replacement and Prefetching [55 points]

You have been commissioned by Lightyear Electronics Group (LEG) to help improve the memory system of their next processor release. LEG engineers are notoriously indecisive, though, and so some of their memory system parameters have yet to be determined. Nevertheless—being the exceptional computer architect that you are—you have agreed to help answer their questions. You are to assume the following about the system:

- The system has a single level of cache before main memory, and all accesses get inserted into the cache.
- The cache has *Sets* sets, where *Sets* is a power of two.
- The cache has *Ways* ways, where *Ways* is a power of two.

The access pattern that is of particular interest to LEG is from one of their streaming benchmarks. It accesses the following N *cache block* addresses in succession, over and over again:

$$0, 1, 2, \ldots, N-3, N-2, N-1,$$

where $N > 2$ and N is a power of two.

For the following questions, we will be examining the steady-state behavior of the cache and prefetcher (i.e., we'll start tracking statistics after the access pattern has been executing for an arbitrarily-long amount of time).

Without a Prefetcher

(a) Assume $Sets = 1$ and $1 < Ways < N$. What is the cache hit rate if Bélády's optimal replacement policy (OPT) is used?

The $Ways - 1$ ways will fill up with the first $Ways - 1$ addresses in the stream, and the remaining $N - Ways - 1$ addresses in the stream will continually replace the other way in the set:

$$(Ways - 1)/N$$

(b) Assume $Sets = 1$ and $1 < Ways < N$. What is the cache hit rate if the replacement policy is true LRU?

Because $Ways < N$ and there is a streaming access pattern, no cache block will ever be in the cache by the time is is accessed again:

$$0$$

(c) Assume $Sets = 1$ and $1 < Ways < N$. What is the cache hit rate if the replacement policy is uniformly random (i.e., any cache block is equally likely to be replaced)?

The probability of a particular block *not* being evicted from the cache after an insertion is $1 - 1/Ways$. The probability of a particular block *not* being evicted in the $N - 1$ insertions in between its reuse is:

$$(1 - 1/Ways)^{N-1}$$

(d) **Bonus.** Assume $Sets = 1$ and $1 < Ways < N$. What is the cache hit rate if the bimodal insertion policy (BIP) is used? Remember that BIP inserts a cache block at the MRU position with probability α and that BIP inserts a cache block at the LRU position with probability $1 - \alpha$.

To remain in the cache after the $N - 1$ insertions in between its reuse, a block needs to be inserted in the MRU position (this happens with probability α), and then there need to be fewer than $Ways - 1$ subsequent MRU position insertions for the block to be at least in the LRU position by the time it is reused. The following expression covers all of the $Ways - 1$ cases where this can occur in all of $\binom{N-1}{n}$ positions in the access stream:

$$\alpha \sum_{n=0}^{Ways-1} \binom{N-1}{n} \alpha^n (1-\alpha)^{N-1-n}$$

With a Next-Line Prefetcher

For this part, assume:

- That the system employs a next-line prefetcher that issues prefetches whenever there is a demand cache miss and places prefetched blocks in the cache; on a hit in the cache, no prefetch is issued.
- Main memory access latency is so small that any time a block is prefetched, it is fetched from main memory and inserted into the cache before the next memory access can be issued.
- Accesses are always serviced before any prefetch they trigger is inserted into the cache.

(a) If $Ways = 1$, what range of values for $Sets$ will achieve a cache hit rate of exactly 50% in the steady state in this system?

From [1] to [$N - 1$], inclusive.

(b) What is the accuracy of the next-line prefetcher in this system?

[100] %

(c) What is the coverage of the next-line prefetcher in this system?

50	%

(d) Now, if $W = 2$, what range of values for *Sets* will achieve exactly a 50% cache hit rate in the system, assuming a least-recently–used (LRU) replacement policy and prefetches are treated no differently than demand requests for the purposes of replacement?

From | 1 | to | $N/2 - 1$ | , inclusive.

(e) A friend suggests changing the prefetcher such that prefetches are issued not only on cache misses, but also on cache hits for prefetched blocks. On one application you find that this improves system performance. For what type of access pattern might this be so?

A sequential access pattern: $0, 1, 2, \ldots, N-3, N-2, N-1$, over and over again.

(f) On another application, however, you find that this degrades system performance. For what type of access pattern might this be so?

A $+1$, $+2$ access pattern: $0, 1, 3, 4, 6, 7, \ldots, N-5, N-4, N-2, N-1$, over and over again.

With a Stride Prefetcher

For this question, assume that the system from the previous part now has $Ways = 1$ and $Sets = N/2$ and employs a stride prefetcher with distance D and degree 1. Assume the same access pattern as before.

(a) Assume that D is a power of two and $D < N$. Derive an expression for the cache hit rate in terms of D and N for this system.

50%

Prefetcher Design

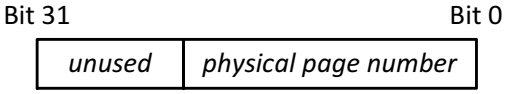
After a while, and many disappointing results, it finally hits you: LEG has been performing prefetching in an inferior manner! Describe why this is so as succinctly as possible:

LEG has been prefetching on cache misses only. It makes more sense to prefetch on cache *hits* as well.

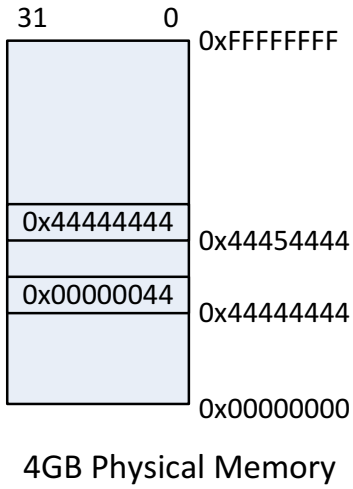
Bonus. 0x44444444 [44 points]

A 32-bit processor implements paging-based virtual memory using a single-level page table. The following are the assumptions about the processor's virtual memory.

- The number of bytes in a page is greater than four and is also a power of two.
- The base address of the page table is page-aligned.
- A page table entry (PTE) stores **only** the physical page number and has the following format. All of the unused bits in the PTE are set to 0.



The following figure shows the physical memory of the processor at a particular point in time.



At this point, when the processor executes the following piece of code, it turns out that the processor accesses the page table entry residing at the physical address of `0x44444444`.

```
char *ptr = 0x44444444;
char val = *ptr; // val == 0x44
```

What is the page size of the processor? Please show work for partial credit. (The answer box is continued on the next page.)

Let n be equal to $\log_2(pagesize)$.

- `Virtual Address (VA) = 0x44444444`
- `Virtual Page Number (VPN) = VA >>` n
- `Physical Address Of PTE (PTE_PA) = 0x44444444`
- `Size of PTE (PTE_SIZE) = 4`
- `Page Table Base Address (PTBA) = PTE_PA - VPN * PTE_SIZE`

Clue 1: PTBA is page-aligned.

```
PTBA & ∼(1<<n) = 0
∴ (PTE_PA - VPN * PTE_SIZE)         & ∼(1<<n) = 0
∴ (0x44444444-(0x44444444>>n)*4)    & ∼(1<<n) = 0
∴ (0x44444444-(0x44444444>>(n-2)) & ∼(1<<n) = 0
```

This means that n is of the form: $4k + 2$, where k is an integer.
Possible values of n: 6, 10, 14, 18, 22, 26, 30.
For these values of n, let us check whether the following equation is indeed equal to 0.

```
(0x44444444-(0x44444444>>(n-2)) & ∼(1<<n)
```

```
n=6:   (0x44444444-(0x44444444>>4))  & ∼(1<<6)  = 0x40000000 & 0x0000003F = 0
n=10:  (0x44444444-(0x44444444>>8))  & ∼(1<<10) = 0x44000000 & 0x000003FF = 0
n=14:  (0x44444444-(0x44444444>>12)) & ∼(1<<14) = 0x44400000 & 0x00003FFF = 0
n=18:  (0x44444444-(0x44444444>>16)) & ∼(1<<18) = 0x44440000 & 0x0003FFFF = 0
n=22:  (0x44444444-(0x44444444>>20)) & ∼(1<<22) = 0x44444000 & 0x003FFFFF != 0
n=26:  (0x44444444-(0x44444444>>24)) & ∼(1<<26) = 0x44444400 & 0x03FFFFFF != 0
n=30:  (0x44444444-(0x44444444>>28)) & ∼(1<<30) = 0x44444440 & 0x3FFFFFFF != 0
```

Possible values of n: 6, 10, 14, 18.

Clue 2: Physical address 0x44454444 is not a PTE

For the possible values of n, let us check whether the last PTE of the page table is stored at a lower physical address than 0x44454444.

```
PA of Last PTE (LPTE_PA) = PTBA + ((1<<(32-n)) - 1) * PTE_SIZE
∴ LPTE_PA = PTBA + ((1<<(34-n))-4)
```

```
n=6:   0x40000000+((1<<28)-4) = 0x40000000+0x0ffffffc = 0x4ffffffc > 0x44454444
n=10:  0x44000000+((1<<24)-4) = 0x44000000+0x00ffffffc = 0x44ffffffc > 0x44454444
n=14:  0x44400000+((1<<20)-4) = 0x44400000+0x000ffffc = 0x444ffffc > 0x44454444
n=18:  0x44440000+((1<<16)-4) = 0x44440000+0x0000fffc = 0x4444fffc < 0x44454444
```

The only possible value of n: 18.

Stratchpad

Stratchpad

Stratchpad