#1. This problem is related to Amdahl’s law. Amdahl’s law says that, “the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used”. Consider an extension. Suppose we have two modes of a processor that are improved by $s_1$ times and $s_2$ times respectively; and these modes are used $f_1$ and $f_2$ fractions of times respectively.

a). Write an expression to show the overall performance gain (or speedup) due to both improvements.

b). As a concrete example, we speeded up the first mode by a factor 10 and the second mode by a factor 5; the first mode is used 50% of the time and the second mode is used 25% of the time. What is the speedup?

c). Now consider generalizing the above case with $n$ modes of a system that are improved: mode $i$ is speeded up by $s_i$ and used $f_i$ fraction of the time. Write an expression for speedup in this general case.

Key: (a). New execution time = $(1-f_1-f_2)/s_1 + f_1/s_1 + f_2/s_2$
   Speedup = $1/(1-f_1-f_2)/s_1 + f_1/s_1 + f_2/s_2$

   (b). New execution time = $0.25 + 0.5/10 + 0.25/5 = 0.35$
   Speedup = $1/0.35 = 2.857$

   (c). New execution time = $[(1-f_1-f_2-\ldots-f_n)/s_1 + (f_1/s_1) + (f_2/s_2) + \ldots + (f_n/s_n)]$
   Speedup = $1/(new\ execution\ time)$

#2. Consider a processor with only L1 cache with 32 byte blocks. Assume write-back cache and assume that at any given time 25% of cache lines are dirty (which means that on a cache miss dirty lines must be written back). Assume 2% miss rate to the cache, and on a miss you fetch 32 bytes from memory. Assume processor generates $10^9$ memory accesses per second. If the memory can support $10^9$ bytes per second bandwidth, what fraction of this bandwidth is utilized by the processor?

Key: CPU generates $10^9$ memory requests per second, of this 2% are misses.
   So we will go to memory 0.02*10$^9$ times per second.
   However, 25% of the time we have to write back data.

   Thus the actual number of request to memory = $1.25*0.02*10^9 = 25*10^6$.
   But on each request we transfer 32 bytes.

   So the total number of bytes transferred to/from memory = $32*25*10^6$ bytes = $800*10^6 = 0.8*10^9$ bytes.

   Memory can support $1.0*10^9$.
   So 80% of this bandwidth is used by the CPU, leaving only 20% for I/O request.

   In most case this is not sufficient for I/O.
#3. int i, j, b[100], c[100], a[100][100];
    for (i=0; i<100; i++)
        c[i]=0.0;
    for (j=0; j<100; j=j+1)
        c[i] = a[i][j]*b[j];

Key: a) 16-byte cache lines \( \rightarrow \) each line contains 4 array elements

Total Misses

For a, we cause \( 100 \times \frac{100}{4} \) = 2500 misses
For b and c we cause \( 100/4 = 25 \) misses each
Total misses = 2550 misses

Total number of access:
We access element of a only once for a total of \( 100 \times 100 \) accesses for a
We access each c[i] 101 times for a total of \( 101 \times 100 \) accesses for c
We access each b[j] 100 times for a total of \( 100 \times 100 \) accesses for b
Total accesses = 30,100

Miss rate = \( \frac{2550}{30,100} = 0.0847 \) or 8.47%

b). With 32 byte cache lines \( \rightarrow \) 8 array elements per line

Total Misses

For a, we cause \( 100 \times \frac{100}{8} \) = 1250 misses
Or if we assume that inside j loop 100 a[*][j] elements are accessed
Causing \( 100/8 = 12.5 \), rounded to 13 misses
Accesses to a cause \( 100 \times 13 \) misses = 1300

For b and c we cause \( 100/8 = 12.5 \) but rounded off to 13 misses each
Total misses = 1276 misses or 1326 misses

Total number of accesses do not change.
Miss rate = \( \frac{1276}{30100} = 0.0424 \) or 4.24%
Or \( \frac{1326}{30100} = 4.4 \) %
CSCE 5610: Computer Architecture

#4. As we saw in class, way-prediction can lead to energy savings, but may increase execution times, since on a mis-prediction when using way-prediction requires one additional cycle if the data is in one of the other ways. On the other hand, higher associativity may slow down the clock speed since we have to compare multiple tags.

Consider the following data.

The miss rate when using 4-way associative cache is 0.001.
When using way-prediction, we have a 90% success and it costs 1 ns. However on a mis-prediction, but if the data is in one of the other ways, it will cost 2ns.
If we use conventional 4-way associative cache, the clock speed will be slower by 20% (compared with way-prediction). So, the hit time is 1.2ns.
Miss penalty is 10ns (with or without way-prediction).

i). What is average memory access time when using way-prediction?

ii). What is the average memory access time when using conventional 4-way associative cache (with slower clock)?

CSCE 5610 Oct. 23, 2018

Key:
i). Note on a hit we have two cases: if way-prediction is correct it costs 1ns; if way-prediction is wrong it costs 2ns. On a miss(0.1%), we take 10ns.

Average Memory access time = (hit-time) + (miss-rate)*(miss-penalty)
= 1*[1*0.9+2*0.1] + 0.001*10 = 1.11

ii). Here the clock is slow, but we spend 1.2ns on hit and 10ns on miss.

Average memory access time = (hit-time) + (miss-rate)*(miss-penalty)
= 1.2ns + 0.001*10 = 1.21 ns

OR
If you used: AMAT = (hit rate)*(hit time) + (miss rate)*(miss penalty)

AMAT = 0.999*[1*0.9+2*0.1] + 0.001*10 ns
AMAT = 0.999*1.2 + 0.001*10

CSCE 5610 Oct. 23, 2018
Dynamic Branch Prediction

2-bit predictor

4 states:

- **00**: predict taken
- **01**: predict taken (but 1-misprediction)
- **11**: predict not taken
- **10**: predict not taken (but 1-misprediction)

Sequential circuit -- 2 flip flops.

- the input is the decision of the branch (taken or not taken – correct prediction or wrong prediction)
- and the current values of the flip flops.

We use a portion of PC (instruction address) as an index into prediction memory.

Based on the 2 bits at that entry determines if the branch is taken or not.

Note two different branch instructions may use the same entry if they LSB 12 bits of both instructions are the same.
Correlation branch predictors

Consider the following program:
```
if (aa==2) aa=0;
if (bb==2) bb=0;
if (aa !=bb) {...
```
The third branch depends on the previous two branches -- branches correlate.

Branch correlation

If we know that the previous branch is taken (or not taken) we may have a better predictions for the current branch instruction.
So, we can use two different prediction tables:
- One if the previous branch is taken
- One if the previous branch is not taken

For current branch we have two 2-bit predictors based on previous branch decision –
- one 2-bit predictor if previous branch is taken and another if the previous branch is not taken

We can use 1-bit flip flop to indicate if previous branch is taken or not

Use this 1 bit to select one of two 2-bit predictions for current branch
CSCE 5610: Computer Architecture

Let us look at a comparison of correlation branch prediction with 2 bit predictor

The correlation predictor performs better particularly for Integer benchmarks.

![Comparison of correlation branch prediction with 2 bit predictor](image)

(2,2) mean we have 2 bits for previous branches or remember 2 branches and 2-bit predictors (or four 1024 entry (2bit) predictors)

More complex predictions – Global share

Keep the history of last k branches (say 0 means not taken, 1 mean taken) use this bit vector as an index into pattern history table (PHT).

Use the k-bit history along with PC bits to index
Branch History Table

- Gselect – concatenate
- Gshare – XOR (better)

More complex predictions –

For some branches local BHT performed better
For other using global history performed better

So switch between these using tournament style selection
Tournament using Saturation counters
If the predictor we used is correct and the predictor not used is incorrect
that is, we selected the right predictor

Increment the counter (stays at the maximum – saturation – normally a small value)
If the opposite is true (the selected predictor is wrong and the unselected
predictor is correct), decrement the counter
again saturation means -- do not decrement below 0

Tournament Predictor in Alpha 21264
- Two predictors, local (based on current branch) and global based history
- Each predictor uses a 4K 2-bit prediction
  - Need 12 bits to find an entry in these tables
  - Can be 12 bits from PC for local branch predictor
  - History of the last 12 branches (taken=1, not taken =0)
- Use counters to indicate if you should use local or global

PROJECT IDEA. Read about various branch prediction techniques.
Use if they are already implemented in simplescalar and collect performance data
Or implement new branch prediction techniques in simplescalar
Collect benchmark data and write a report.
Branch prediction is only half the story.
We still need to compute the address of target of the branch
PC+4 if not taken
PC + displacement if branch is taken

Along with prediction (say 2-bit predictor), save predicted next instruction address

Branch Target Buffer (pre-computed addresses of all branch destinations)
ADDRESSED USING PC – Any problems?

Once again, we do not want to use all 32 bits of PC to address the branch target buffers.

Suppose we only have 4K entries, we need 12 bits to address this buffer
We are only using the least significant 12 bits of PC.

It is possible for two different branch instructions to have the same 12 LSB bits but they may have
two different destinations

We need to make sure that the entry in the BTB clearly indicates to which branch the data belongs
CSCE 5610: Computer Architecture

While fetching an instruction, we also look up this BTB.

**BTB should contain a valid entry only if the instruction fetched is a branch instruction**

Otherwise there should not be a valid entry – the tag part will not match

Let us look at the example. We are given:
- prediction correctness = 90%
- hit rate (found in BTB) = 90%
- Penalty on mis-prediction = 2 cycles
- Penalty if not found in BTB and branch is taken = 2 cycles
- If not found in the BTB, and branch is not taken = 0 cycle penalty
- 60% branches are taken and 40% not taken

10% mis-prediction for 90% found in BTB, penalty = 0.9*0.1*2= 0.18 cycles
10% miss in BTB and 60% taken, penalty = 0.1*0.6*2 =0.12 cycles
10% miss in BTB and 40% not taken, penalty = 0.1*0.4*0 =0 cycles

Total penalty = 0.3 cycles

CSCE 5610: Computer Architecture

Another issue to deal with is the case when the EX stage takes different amounts of time for different instruction types.

Consider the following example of an architecture

![Diagram of instruction pipeline](Fig C.47, page C-48)
Two important terms to remember

**Delay** – how soon another instruction of the same kind can be issued
**Latency** – how soon can the results of an instruction can be made available
(for data forwarding)

FP divide is not pipelined and needs 24 cycles to complete
but using forwarding, we can forward data from the last
stage back to itself
So latency = 23 and inter-instruction initiation delay = 24

FP multiply --- pipelined and has 7 stages
data can be forwarded from M7 to M1
latency = 6; delay = 1
FP add -- pipelined with 4 stages and data from A4 can be forwarded to A1
latency = 3; delay = 1

Let us now see the impact of the instructions with different delays and latencies

Consider the following code

```plaintext
Foo:
eld  f2, 0(x1) : load floating point value
fnul f4, f2, f0 : Floating multiply
fla f6, 0(x2) : load floating point value
fadd f6, f4, f6 : Floating Add
fzd f6, 0(x2) : Floating store
addi x1, x1, #8 : Integer add immediate
addi x2, x2, #8 : add immediate
blt x1, x5, foo : check if done
```

Let us use the latency and delay information to see how many cycles are
need for each loop iteration

**We will use latencies as follows:**
- Float ADD = 3; Float MULT = 6; Integer add= 0
Also we need 1 stall if we have a data dependency between LD and an arithmetic operation
Consider unrolling the loop twice. That is each new iteration completes operations of two original iterations

```
Foo:       Foo:
fld       fld       : load floating point value
           f2, 0(x1)   : Floating multiply
stall     stall
fmul      fmul      : load floating point value
           f4, f2, f0 : Floating multiply
fadd      fadd      : load floating point value
           f6, 0(x2) : Floating Add
fst        fst       : Floating store
addi      addi      : Integer add immediate
x1, x1, #8 x2, x2, #8
blt        blt       : add immediate
x1, x5, foo
(stall)   (branch taken)
-------------
```

We do not need to increment x1 and x2 two times or check BEQZ twice
CSCE 5610: Computer Architecture

Foo:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fld f2, 0(x1)</td>
<td>load floating point value</td>
</tr>
<tr>
<td>fmul f4, f2, f0</td>
<td>Floating multiply</td>
</tr>
<tr>
<td>fld f6, 0(x2)</td>
<td>load floating point value</td>
</tr>
<tr>
<td>fadd f6, f4, f6</td>
<td>Floating Add</td>
</tr>
<tr>
<td>fadd f6, f4, f6</td>
<td>Floating Add</td>
</tr>
<tr>
<td>fsd f6, 0(x2)</td>
<td>Floating store</td>
</tr>
<tr>
<td>fld f2, 8(x1)</td>
<td>load floating point value</td>
</tr>
<tr>
<td>fmul f4, f2, f0</td>
<td>Floating multiply</td>
</tr>
<tr>
<td>fld f6, 8(x2)</td>
<td>load floating point value</td>
</tr>
<tr>
<td>fadd f6, f4, f6</td>
<td>Floating Add</td>
</tr>
<tr>
<td>fadd f6, f4, f6</td>
<td>Floating Add</td>
</tr>
<tr>
<td>fsd f6, 8(x2)</td>
<td>Floating store</td>
</tr>
<tr>
<td>addi x1, x1, #16</td>
<td>Integer add immediate</td>
</tr>
<tr>
<td>addi x2, x2, #16</td>
<td>add immediate</td>
</tr>
<tr>
<td>blt x1, x5, foo</td>
<td>check if done</td>
</tr>
</tbody>
</table>

Since we are completing 2 iterations once, we need to load i and i+1 values of the array.

We can reorder the instructions. But we need to use additional registers.

Note we are completing 2 original iterations.

16 cycles to complete two iterations.

But we can rearrange instructions.
CSCE 5610: Computer Architecture

Foo: 
fld f2, 0(x1) Foo: 
fld f2, 0(x1) 
fld f8, 0(x1) fml f4, f2, f0 
fmul f10, f8, f0 
fmul f4, f2, f0 
fmul f10, f8, f0 
fdl f6, 0(x2) fdl f6, 0(x2) 
fdl f12, 8(x2) fdl f12, 8(x2) 
addi x1, x1, #16 addi x1, x1, #16 
fadd f6, f4, f6 add x1, x1, #16 
fadd f12, f10, f12 fadd f12, f10, f12 
addi x2, x2, #16 addi x2, x2, #16 
fsd f6, -16(x2) fadd f6, -16(x2) 
fsd f12, -8(x2) fadd f12, -8(x2) 
fsd f12, -8(x2) fadd f12, -8(x2) 
stall fblt x1, x5, foo 
blt x1, x5, foo 

Since addi instructions are moved ahead of sd, we need adjust offsets with store Now we have 13 cycles for two iterations Or 6.5 cycles per iteration (down from 16)

14 cycles to complete two iterations Or, we can unroll 3 times

If we have delayed branch?

See more examples from text book on pages 178-180

CSCE 5610 Oct. 23, 2018 25

CSCE 5610: Computer Architecture

Instruction Level Parallelism vs Thread Level Parallelism
vs Task level parallelism vs Data parallelism

ILP: parallel execution of instructions within a basic block
(Mostly from Chapter 3)

What is a basic block?
A sequence of instructions with only one entry at the top and one exit at the bottom of the sequence

An example below

Foo: 
 fld f2, 0(x1) : load floating point value 
 fml f4, f2, f0 : Floating multiply 
 ld f6, 0(x2) : load floating point value 
 fadd f6, f4, f6 : Floating Add 
 fsd f6, 0(x2) : Floating store 
 addi x1, x1, #8 : Integer add immediate 
 addi x2, x2, #8 : add immediate 
 blt x1, x5, foo : check if done

You can only jump to Foo
And you can exit after blt
You cannot jump to fadd from somewhere else

CSCE 5610 Oct. 23, 2018 26