CSCE 4610: Computer Architecture

Review for Exam
Chapter 1
How to measure performance of a computer
CPI
Amdahl’s law
Averages, Harmonic means and Geometric means
Cost of making a chip, power consumption
Benchmarks

Appendix A
Selecting an instruction set
Instruction formats --Load-Store architecture; Stack architecture
Memory-Register architecture
Addressing modes, operations
branch and conditions, delayed branch
procedure call

CSCE 4610 Feb. 24, 2017

CSCE 4610: Computer Architecture

Appendix B and Chapter 2
Memory hierarchy
Cache design
associativity, cache blocks
Reads vs Writes
write through vs write back
3-C’s of cache misses
cold, capacity and conflict
Miss penalty and miss rate
Average memory access times
Techniques to improve cache performance
multi-level caches
global vs local miss rates
way prediction
non-blocking caches
prefetch: hardware and software
Compiler optimizations
CSCE 4610: Computer Architecture

Appendix B and Chapter 2
Virtual and physical address caches
Using different indexing schemes
   Exclusive Or Tag bits with index
   Prime modulo

DRAM
   latency vs bandwidth
   interleaving memory banks

Sample Problems from previous semesters

1 (20%) Consider the following frequencies for branch instructions

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional branches</td>
<td>25%</td>
</tr>
<tr>
<td>Unconditional branches</td>
<td>5%</td>
</tr>
<tr>
<td>Conditional branches taken</td>
<td>75%</td>
</tr>
</tbody>
</table>

Consider our 5-stage pipeline (as we have seen in class) and assume that branch is decoded in the second stage, but resolution (if branch is taken or not and the target address) is done in EX stage.

Calculate the average number of stalls per instruction.

Key.
Assume that the pipeline is stalled on a branch. We stall 2 cycles on all taken branches, and one cycle stall if the branch is not taken (the next instruction is already in the pipeline but stalled in IF). Unconditional branches are taken and thus there are 2 stalls.

(stalls due to unconditional) + (stalls due to taken) + (stalls due to not taken branches)

\[(5\% \times 2) + (25\% \times 75\% \times 2) + (25\% \times 25\% \times 1) = 0.5375\text{ stalls}\]
CSCE 4610: Computer Architecture

1.11 [10/10/20] <1.7> Availability is the most important consideration for designing servers, followed closely by scalability and throughput.

a. [10] <1.7> We have a single processor with a failures in time (FIT) of 100. What is the mean time to failure (MTTF) for this system?

b. [10] <1.7> If it takes 1 day to get the system running again, what is the availability of the system?

c. [20] <1.7> Imagine that the government, to cut costs, is going to build a supercomputer out of inexpensive computers rather than expensive, reliable computers. What is the MTTF for a system with 1000 processors? Assume that if one fails, they all fail.

Look at page 34 for a definition of FIT, which is the inverse of MTTF reports # failures in billions hours

a). MTTF = \(10^9/100 = 10^7\)

b). Note here if we have a failure once every \(10^7\) hours, and it takes another 24 hours before it can be fixed, essentially the system is available \(10^7\) hours out of \((10^7 + 24)\). Availability = \(10^7/(10^7+24)\)

c) Since there are a thousand processors with MTTF of \(10^7\) hours, the net MTTF is \(10^7/1000\) or \(10^4\) hours.

CSCE 4610 Feb. 24, 2017

1.15 [15/10] <1.9> Assume that we make an enhancement to a computer that improves some mode of execution by a factor of 10. Enhanced mode is used 50% of the time, measured as a percentage of the execution time when the enhanced mode is in use. Recall that Amdahl’s law depends on the fraction of the original, unenhanced execution time that could make use of enhanced mode. Thus, we cannot directly use this 50% measurement to compute speedup with Amdahl’s law.

a. [15] <1.9> What is the speedup we have obtained from fast mode?

b. [10] <1.9> What percentage of the original execution time has been converted to fast mode?

1.15 a. old execution time = 0.5 new + 0.5 \times 10\ new = 5.5\ new

speedup = (old time)/new = 5.5

1.15 b. In the original code, the unenhanced part is equal in time to the enhanced part sped up by 10, therefore:

\((1 - x) = x / 10\) or \(10 - 10x = x\)

or \(10 = 11x\)

\(x = 10/11 \approx 0.91 \approx 91\%\)

CSCE 4610 Feb. 24, 2017
Problems related to Appendix A (instruction sets)

How to compute CPI?

A.4 Compute CPI for MIPS (instruction frequencies given on page A-42 (Figure 1-28)
Cycles for different instruction types given in problem A-3 (page A-47)

- Loads 15.95% and loads take 1.4 cycles
- Stores 1.45% and stores take 1.4 cycles
- Add 30.5% and take 1 cycle,

So using such information, CPI is 1.76

A.7 Here you are asked to convert a C code segment into MIPS

```
for (i=0; i<=100, i++)
{A[i] = B[i] +c;}
```

A and B are 64 bit integer arrays, c and i are 64 bit integers
Starting addresses are

- A[0] = 1000, B[0] =3000, c = 5000, i = 7000

We can load c and i into integer registers. But we need keep loading B[i] and storing in C[i]

```
MIPS:
ex_a_7:
    DADD R1,R0,R0 ; R0 = 0, initialize i = 0
    SW 1000(R0),R1 ; store i

loop:
    LD R1,17000(R0) ; load word address of i
    DSLL R2,R1,#3 ; R2 = word offset of B[i]
    DADD R3,R2,#3000 ; add base address of B to R2
    LD R4,0(R3) ; load B[i]
    LD R5,5000(R0) ; load c
    DADD R6,R4,R5 ; B[i] + c
    LD R1,7000(R0) ; load base address of i
    DSLL R2,R1,#3 ; R2 = word offset of A[i]
    DADD R7,R2,#1000 ; add base address of A to R2
    SD 0(R7),R6 ; store i
    LD R1,7000(R0) ; load base address of i
    DADD R8,R1,#-101 ; increment i
    BNEZ R8,loop ; if not 101, repeat
```

Instructions Executed: 2 + (101 * 16) = 1618
Memory References: 1 + (101 * 8) = 809
Instruction Size = 4 bytes * 18 = 72 bytes
CSCE 4610: Computer Architecture

A8: Here we are asked to design an instruction format
how many bits for opcode, operands etc
Instruction length is 12 bits
there are 32 registers – need 5 bits to specify a register

a) is it possible to have 3 two operand instructions?
2 operands = 2*5 = 10 bits
with remaining 2 bits, we can allow 4 instructions
(opcode 00 should be left for no-op)

b). Is it possible to have 30 one operand instruction?
1 operand = 5 bits
We are left with 7 bits for opcode and can allow up to 128 instructions

A.18 Here we are comparing if zero (stack), one (accumulator), memory-memory, load/store type of instructions in terms of cycles, memory transfers

CSCE 4610: Computer Architecture

A18 We need to convert these statements into different instruction types
A= B+C;
B = A+C;
D = A-B;

a. Accumulator:

Load B ;Acc <- B
Add C ;Acc <- Acc + C
Store A ;Mem[A] <- Acc
Add C ;Acc <- "A" + C (Already loaded C)
Store B ;Mem[B] <- Acc
Negate ;Acc <- -Acc
Add A ;Acc <- -B + A
Store D ;Mem[D] <- Acc

Load-Store:

Load R1, B ;R1 <- Mem[B]
Load R2, C ;R2 <- Mem[C]
Add R3, R1, R2 ;R3 <- B + C
Add R1, R3, R2 ;R1 <- A + C
Add R4, R3, R1 ;R4 <- A - B
Store A,R3 ;Mem[A] <- R3
Store B,R1 ;Mem[B] <- R1
Store D,R4 ;Mem[D] <- R4

NOTE: only 7 of these instructions transfer data
Only 5 instructions involve data transfer

Memory-memory:

Sub D, A, B ;Mem[D] <- Mem[A] - Mem[B] (already loaded A, B loaded again but is updated)

Each instructions requires 3 memory accesses
CSCE 4610: Computer Architecture

A18 We need to convert these statements into different instruction types

\[
\begin{align*}
A &= B + C; \\
B &= A + C; \\
D &= A - B;
\end{align*}
\]

Stack:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push B</td>
<td>; stored on stack</td>
</tr>
<tr>
<td>Push C</td>
<td>; stored on stack</td>
</tr>
<tr>
<td>Add</td>
<td>; pop C, B push result</td>
</tr>
<tr>
<td>Pop A</td>
<td>; Mem[</td>
</tr>
<tr>
<td>Push A</td>
<td>; stored on stack</td>
</tr>
<tr>
<td>Push C</td>
<td>; stored on stack (already loaded C)</td>
</tr>
<tr>
<td>Add</td>
<td>; pop C, A push result</td>
</tr>
<tr>
<td>Pop B</td>
<td>; B &lt;- A + C</td>
</tr>
<tr>
<td>Push B</td>
<td>; stored on stack (reload updated B)</td>
</tr>
<tr>
<td>Push A</td>
<td>; stored on stack (already loaded A)</td>
</tr>
<tr>
<td>Sub</td>
<td>; pop A, B push result</td>
</tr>
<tr>
<td>Pop D</td>
<td>; D &lt;- A - B</td>
</tr>
</tbody>
</table>

Opcode = 8 bits (1 byte)
Addresses = 16 bits (2 bytes)
Registers = 16 (4 bits)

Memory requirements Data only

Memory in bytes:
- Assume word size = 16 bits
  - Accumulator = (7 * 16 bits)/8 = 14 bytes
  - Memory-memory = (9 * 16 bits)/8 = 18 bytes
  - Stack = (9 * 16 bits)/8 = 18 bytes
  - Load-Store = (5 * 16 bits)/8 = 10 bytes

We need to estimate the size of instructions also

Total:
- Accumulator = 22 + 14 = 36 bytes
- Memory-memory = 21 + 18 = 39 bytes
- Stack = 30 + 18 = 48 bytes
- Load-Store = 23 + 10 = 33 bytes

Accumulator is the most efficient.

CSCE 4610: Computer Architecture

Problems related to Appendix B and Chapter 2

Consider a processor with a 16 Kbyte 4-way set associative cache, with 8-byte lines. Show how you will translate 32-bit address to find the data in cache. Show the cache-set to which the following (hex) address falls: ABCED8F8. How many bits are needed for tags in this cache?

A 16K Byte cache with 8-byte line will contain 2048 lines and with 4-way set associativity, we will have 512 sets (each set with 4 lines). Thus we need 9-bits to locate a set, and 3-bits to locate a byte within a line (since each line has 8bytes). So we will divide a 32 bit address as follows.

```
| 20-bit Tag | 9-bit set index | 3-bit byte offset |
```

Address ABCED8F8 = 1010 1011 1100 1110 1101 1000 1111 1000
Yields 1000 1111 1 = 287 as the set number (and byte zero in the line)
This leaves the tag as ABCED = 1010 1011 1100 1110 1101
CSCE 4610: Computer Architecture

There are a total of 2048 lines and each line will have 20 bit tags. So the total number of bits for tags = 20 * 2048 = 40,960 bits

Consider a system with L-1 and L-2 caches, main memory and disk (or secondary memory) with the following parameters. It takes 10 ns to access L-1 cache, 100ns to access L-2 Cache, 500 ns to access RAM and 10ms to access disk. A program makes 250,000 memory references of which 215,000 are hits in L-1 cache, 235,000 are hit in L-2 cache, 245,000 are hit in RAM.

a). What are the L-1, L2 cache and RAM miss ratios?
b). Estimate the average access time assuming all memory accesses are reads.
c). If we are using write-back policy for L-1, 50% of all cache lines are dirty, what is the effective memory access time?

There are a total of 2048 lines and each line will have 20 bit tags. So the total number of bits for tags = 20 * 2048 = 40,960 bits

a). L-1 Miss rate = 35000/250000 = 0.14 (or 14%)
global L-2 miss rate = 15000/250000 = 0.06 (6%)
local L-2 miss rate = 15000/35000 = 0.4286 (42.86%)
global RAM miss rate = 5000/250000 = 0.02 (2%)
local RAM miss rate = 5000/15000 = 0.33 (33%)

b). Access time
   = (215000/250000)*10 + (20000/250000)*100 + (10000/250000)*500 + (5000/250000)*10,000,000
   = 0.86*10 + 0.08*100 + 0.04*500 + 0.02*10,000,000 = 200,036.6ns

c). We have a write back L-1, on a miss in L-1 we may have to write the dirty line back to memory 50% of the time. So, on a miss in L-1, we have to allow for 1.5 accesses to L2.

Access time = (215000/250000)*10 + 1.5* (20000/250000)*100 + (10000/250000)*500 + (5000/250000)*10,000,000
   = 0.86*10 + 1.5* 0.08*100 + 0.04*500 + 0.02*10,000,000 = 200,040.6ns
Consider the following program segment in C.

```c
for (k= 0; k < 50; k++)
{
    for (i =0; i < 100; i ++)
    {
        sum = 0;
        for (j =0; j < 100; j ++)
            if ( i !=j) sum = sum + a[i][j]*x[j];
        new_x[i] = (b[i] - sum)/a[i][ i];
    }
    for (i =0; i < 100; i ++) x[i] = new_x[i];
}
```

Let us only worry about memory accesses generated by the arrays `a, b, x, new_x` (ignoring accesses for scalar variables like `i, j, k, sum`) and each element is one word (32 bits).

a). What is the expected cache miss rate if each cache block is 16 bytes (and we will assume the cache is large enough so that any data brought into cache is not displaced). Initially the data is not in cache (cold start)
b). What is the expected cache miss if the cache block size is increased to 32 bytes each?

a). Each line of the cache is 16 bytes and will hold 4 array elements. So each time a line is brought into cache, 4 consecutive array elements are brought into cache. Matrices are stored row-major and thus the 4 elements belong to the same row.

Consider the loop

```c
for (j =0; j < 100; j ++)
    if ( i !=j) sum = sum + a[i][j]*x[j];
```

Here we are accessing the j-th column elements of the i-th row of `a` and the j elements of `x`. So we will have 100/4 = 25 misses for accessing `a[i][]`. Similarly there will be 100/4 = 25 misses for accessing `x[*]`.

The outer i-loop is repeated 100 times, we will have 25 misses for each `a[i][]`. Total misses on `a[*][*]= 100 * 25 = 2500`. However, the `x[*]` elements remain in the cache, for a total of 25 misses.

We will have to bring `new_x` and `b` arrays only once, and we will have 100/4 = 25 misses each for `new_x` and `b` arrays.
CSCE 4610: Computer Architecture

The next loop:
```c
for (i =0; i < 100; i ++)
    x[i] = new_x[i];
```
will not generate any additional misses since x and new_x are already in cache.

The k loop does not increase the miss count as everything is already loaded in the cache

So, the total number of misses = (a) 2500 + (x) 25 + (new_x) 25 + (b) 25 = 2575 misses.

Now that the number of misses has been determined, the total number of accesses is needed to compute the miss ratio.

Note that in
```c
for (j =0; j < 100; j ++)
    if ( i != j ) sum = sum + a[i][j] * x[j];
```
we have 100 accesses to ‘a’ and 100 accesses to ‘x’ in the j loop. (We’ll ignore the i != j )

Then if we consider the outer loop
```c
for (i =0; i < 100; i ++)
{
    sum = 0;
    for (j = 0; j < 100; j ++)
        if ( i != j ) sum = sum + a[i][j]*x[j];
    new_x[i] = (b[i] - sum) / a[i][i];
}
```
For each i iteration we have 100 accesses to ‘a’ and 100 accesses to ‘x’ from the j loop, 1 access to ‘new_x’ and one access to ‘b’. Since there are 100 i-loop iterations, the total number of accesses are
- 100 *100 accesses to ‘a’;
- 100 *100 accesses to ‘x’
- 100 accesses to ‘new_x’
- 100 accesses to ‘b’
- 100 accesses to a[i][i]

The total number of accesses = 20,300
CSCE 4610: Computer Architecture

```
for (i = 0; i < 100; i++)
    x[i] = new_x[i];
```

will add 100 accesses each to ‘x’ and ‘new_x’

The accesses for each k loop total 20,500
Since we have 50 values for k, the total number of accesses = 50 * 20,500 =1,025,000

Therefore, the miss rate = 2575 / 1,030,000 =0.25%

b). If the line size is doubled to 32 bytes, each line will have 8 elements. The total number of
misses will be halved for accessing each array
We will have 100/8 = 13 misses (note we will have 4 wasted elements in the last access) for
accessing each row of a, and 13 misses accessing each of the b, new_x and x arrays.

Total misses = 100 * 13 + 13 + 13 + 13 = 1339 misses

The miss rate = 1339 / 1,030,000 =0.13%

Consider a CPU with a cache. You are asked to compare the performance of two choices: write-
through cache and write-back cache.

You are given the following information.
On a hit, it takes 10ns to access the cache.
On a miss, it takes 100ns to bring the missing line to cache.
It takes 100ns to write one word of 32-bits (for write-through cache).
To write a full 16-byte cache block (for write-back cache), it takes 200ns.

Assume that 75% of all memory accesses are reads (for write-through cache design), and on
average 50% of all cache lines are dirty (for write-back cache design).

a). Initially we will assume that there is no write buffer. What are the access times for the two
cache designs assuming that the miss rate is 1%?

b). Assuming that we have a write buffer and that only 10% writes in a write-through cache cause
the CPU to stall for the 100ns needed to write 32-bit data to memory, what is the memory access
time for the write-through cache?
CSCE 4610: Computer Architecture

a). For Write-through cache, each write will have to be written to the next level of memory hierarchy and it will incur 100ns (no write buffer)
   Read hit will take 10ns
   Read miss will take 100ns (need to read entire 32 byte line)
   Write hit will take 100ns (since this is write through).

   Access time on hits (on reads and writes) = 75% * 10ns + 25% * 100ns = 32.5ns
   Access time on miss = 1% * 100ns (1% miss rate) = 1ns
   Effective access time = 33.5ns

   For write-back, on a miss, if the cache line to be replaced is dirty (50% probability), it will take 200ns to write the dirty line and 100ns to fetch the needed data. If the cache line to be replaced is clean (50% probability) it will take 100ns to fetch the needed data.
   Access time on hit (both for read and write) = 10ns
   Access time on a miss if replaced line is dirty = 1% * (200+100)ns * 0.5 = 1.5 ns
   Access time on a miss if replaced line is clean = 1% * 100ns * 0.5 = .5 ns

   Effective access time = 10ns + 1.5ns + 0.5ns = 12.0 ns
   So, write-back is better in this case.

CSCE 4610: Computer Architecture

b). If we use write-buffers, not every write will incur 100ns delay – in fact only 10% of writes incur the 100ns delay. Assuming 10 ns for the 90% of the writes:

   Effective access time =
   \[ 75\% * 10ns + (25\% * 10\%) * 100ns + (25\% * 90\%) * 10ns + 1\% * 100ns \]
   \[ = 7.5 \text{ ns} + 2.5 \text{ ns} + 22.5 \text{ ns} + 1 \text{ ns} \]
   \[ = 32.25 \text{ ns} \]

   Which is slightly slower than the write-back cache.
Consider the following program segment in C.

```c
for (i= 0; i < 100; i++)
    for (j= 0; j< 100; j++)
        sum += big[i][j];

for (i= 0; i < 100; i++)
    for (j= 0; j < 100; j++)
        big[j][i] = sum;
```

Let us only worry about memory accesses generated by the array big.

a) Assume that the cache line size is 32 bytes and each array element is 4-bytes. Compute the total number of memory accesses caused by the program segment and the number of misses caused by the program.

b) Assume that the cache is not large and can only hold 100 elements (approximately 400 bytes). How many misses are caused by the program segment above?

---

a) These loops generate 100*100 access to the array ‘big’. Since the j loop access elements of the array in row major order, the j loop only generates 100/8 = 13 cache misses. Each i loop brings in another row of the ‘big’ array. Therefore the total number of misses generated is 100 * 13 = 1300.

b) These loops also generate 100*100 accesses. Assuming the cache is large enough to hold all of the ‘big’ array elements, these loops should not cause any new misses.

Total number of accesses to big = 2 * 100 * 100 = 20,000
Total number of misses = 1300
Miss rate = 1300/20,000 = 6.5%
b) for (i= 0; i < 100; i++)
   for (j= 0; j< 100; j++)
       sum += big[i][j];

We still have 100*100 accesses for 'big' generated by these nested loops. Each j loop still
generates 100/8 = 13 misses. And since the j loop is still repeated 100 times, we have 1300 misses.

for (i= 0; i < 100; i++)
   for (j= 0; j < 100; j++)
       big[j][i] = sum;

These nested loops again generate 100*100 accesses. We will assume that cache is not large to keep
the array elements. The elements of big are not accessed along row major and only the last row is in cache. So for each
j loop, we are accessing elements of a column. For each value of j, we cause a miss (we cannot take
advantage of the line size being 32 bytes). So we generate 100 misses for j loop. And since j loop is
repeated 100 times, we generate 100 * 100 misses.

Total number of accesses = 2 * 100 * 100 = 20,000
Misses = 1300 + 100*100 = 11,300
Miss rate = 56.5%
CSCE 4610: Computer Architecture

c). A third way (used in PowerPC), is to add 16 bit immediate value to either the lower half or upper half of a register. In this case, we add the 16 bits to the either the upper or lower half of Rsrc (other bits are un-effected). So, we can first zero Rsrc, add 16 bit immediate to the upper half, and then add 16 bit to the lower half of Rsrc.

d). We can also add 16 bits always to the lower half of Rsrc. First zero Rsrc, add 16-bit immediate to the lower half; shift left Rsrc by 16 bit positions, and then add 16 bit immediate to the lower half.

We have studied the Amdahl’s law, which states that the speed-up that can be achieved using an enhancement is limited by the fraction (f) of the time this enhancement can be applied.

This can also be described in terms of the increased capacity resulting from the enhancement.

In other words, since the enhancement decreases the execution time, we can do more work in the same amount time.

Write an expression that shows the increased capacity when a fraction f is speeded up by a factor s. What is the upper bound on the increased capacity?

CSCE 4610: Computer Architecture

Key

Remember that we use Amdahl’s law to indicate speed up which is given by

\[ \text{Speed-Up} = \frac{1}{(1-f) + f/s} = \frac{s}{s - (s-1)*f} \]

In this problem we are rephrasing the equation. Let us look at the amount of time reduction using the enhancement. Time reduction is given by (original time – new time)

\[ \text{Time saved} = 1 - [(1-f) + f/s] = \frac{f(s-1)}{s} \]

To complete one unit of work in the enhanced mode we need [(1-f) + f*s]. So how much extra work can we do in f(s-1)/s time units?

Enhanced capacity = \[ \frac{f(s-1)/s}{(1-f)+f*s} = \frac{(s-1)*f}{s-f(s-1)} \]

The maximum enhanced capacity happens when f= 1 and is given by s-1. This makes sense since when f=1, speed up is s. So, we can do s-1 additional units of work in the same time.
Let us see what are the implications of phrasing Amdahl’s law in this manner. Consider the example from page 30 of our textbook, where we are given $f=0.4$ and $s=10$. The resulting speed up is 1.66.

But in terms of increased capacity we get 0.5625. That means we can do 56% more work in the same time. This phrasing makes us feel a bit more comfortable than saying that you can get only 1.66 times speed up with 10 times enhancement.

A variation of this increased capacity law was originally derived by John Gustavson of Ames Research Labs at Iowa State University and is the foundation of “Scalability” studies of parallel processing algorithms.