# CS61C Spring 2014
# Review Session

March 9, 2014

## Mapreduce True / False

1. MapReduce programs running on a single core are usually faster than a simple serial implementation.
2. MapReduce works well on clusters with hundreds or thousands of machines.
3. MapReduce is the only framework used for writing large distributed programs.
4. MapReduce can sometimes give the wrong answer if a worker crashes.
5. A single Map task will usually have its map() method called many times.

## Mapreduce True / False

1. MapReduce programs running on a single core are usually faster than a simple serial implementation. (FALSE)
2. MapReduce works well on clusters with hundreds or thousands of machines. (TRUE)
3. MapReduce is the only framework used for writing large distributed programs. (FALSE)
4. MapReduce can sometimes give the wrong answer if a worker crashes. (FALSE)
5. A single Map task will usually have its map() method called many times. (TRUE)

## Caches

AMAT is influenced by three things - hit time, miss rate, and miss penalty. For each of the following changes, indicate which component will likely be improved:

1. using a second-level cache
2. using larger blocks
3. using a smaller L1$
4. using a larger L1$
5. using a more associative cache

## Caches

1. using a second-level cache

miss penalty

2. using larger blocks

miss rate

3. using a smaller L1$

hit time

4. using a larger L1$

miss rate

5. using a more associative cache

miss rate

## Cache bits

How many bits are needed for tag, index and offset in a direct mapped cache with $2^m$ bytes and $2^n$ lines, with $p$-bit addressing?

Suppose the above cache has $x$ maintenance bits (valid, dirty, etc.). How many bits does it have total?

If a cache has dirty bits, then it is almost definitely write-________, rather than write-________. (choose between through and back).

## Cache bits

How many bits are needed for tag, index and offset in a direct mapped cache with $2^m$ bytes and $2^n$ lines, with $p$-bit addressing?

Index = n, offset = m-n, tag = p - (n + m - n) = p - m

Suppose the above cache has $x$ maintenance bits (valid, dirty, etc.). How many bits does it have total?

$8 * 2^m + (x + tag) * 2^n = 2^{m+3} + (x + p - m) * 2^n$

If a cache has dirty bits, then it is almost definitely write-back, rather than write-through. (choose between through and back).

## AMAT Question

Suppose a MIPS program executes on a machine with a single data cache, and

- the data cache hit rate is 95%
- the cache has a miss penalty of 100 cycles
- the cache hit time is 1 cycle

a. Calculate the AMAT of the program.

b. Now suppose that we are concerned about the reliability of the bus between L1$ and the CPU. To deal with this we modify (in hardware) the procedure for performing a memory access. Instead of performing one load or store we perform three sequential accesses, and then take the most common result (assume that at least 2 will agree). What is the new AMAT, treating our redundant triple-load as a single access?

a. Calculate the AMAT of the program.

AMAT = L1_hit + P(L1_miss) * L1_Penalty

= 1 + .05 * 100 = 1 + 5 = 6 cycles

b. Now suppose that we are concerned about the reliability of the bus between L1$ and the CPU. To deal with this we modify (in hardware) the procedure for performing a memory access. Instead of performing one load or store we perform three sequential accesses, and then take the most common result (assume that at least 2 will agree). What is the new AMAT, treating our redundant triple-load as a single access?

Each access is now three accesses, but notice that the second 2 accesses will always be hits, so this is just

AMAT = AMAT_old + 2 * L1_hit = 8 cycles

## MIPS

Consider visit_in_order, then translate into MIPS.

```
typedef struct node{
    int value;
    struct node* left;
    struct node* right;
} node;
void visit_in_order(node *root, void (*visit)(node*)) {
    if (root) {
        visit_in_order(root->left, visit);
        visit(root);
        visit_in_order(root->right, visit);
    }
}
```

```
vio:
    addiu $sp $sp -12
    sw $s0 0($sp)
    sw $s1 4($sp)
    sw $ra 8($sp)

    #nullcheck
    beq $0 $a0 Exit
    move $s0 $a0
    move $s1 $a1

    #vist left
    lw $a0  4($s0)
    jal vio
    #visit myself
    move $a0 $s0
    jalr     $s1
    #visit right
    lw $a0 8($s0)
    move $a1 $s1
    jal vio

Exit:
    lw $s0 0($sp)
    lw $s1 4($sp)
    lw $ra 8($sp)
    addiu $sp $sp 12
    jr   $ra
```

## Number representation

Write the following 32-bit numbers in

1's complement :

-0 :

-179 :

2's complement :

-2^31 :

2^31 :

-134 :

134 :

## Number representation

Write the following 32-bit numbers in

1's complement :

-0 : 1111 1111 1111 1111 1111 1111 1111 1111

-179 : 1111 1111 1111 1111 1111 1111 0100 1100

2's complement :

-2^31 : 1000 0000 0000 0000 0000 0000 0000 0000

2^31 : Not possible

-134 : 1111 1111 1111 1111 1111 1111 0111 1010

134 : 0000 0000 0000 0000 0000 0000 1000 0110

# Floating Point

Convert the following decimal fractions to IEEE 754 32-bit floating point numbers (i.e. give the bit patterns). Assume rounding is always to the nearest bit, with ties rounding up.

a. 126.375/1

b. 23.6/0

c. -5/16

d. 0/0

# Floating Point

Convert the following decimal fractions to IEEE 754 32-bit floating point numbers (i.e. give the bit patterns). Assume rounding is always to the nearest bit, with ties rounding up.

a. 126.375/1 0 10000101 11111001100000000000000

+ $2^6$ * (1 + 0.5 + 0.25 + 0.125 + ...)

b. 23.6/0 0 11111111 00000000000000000000000 (+inf)

c. -5/16 1 01111101 01000000000000000000000

-0.3125 - $2^{-2}$ * (1 + 0.25)

d. 0/0 1 11111111 10000000000000000000000 (NaN)

(not unique)

## Floating Point (cont)

Convert the following bitstrings into their decimal value, interpereted as single precision floats

a. 0b0 00000000 1011...0

b. 0b1 00000001  011...0

c. 0b1 11111111    0...0

d. 0b1 11111111    1...1

e. 0b0 01111111    0...0

## Floating Point (cont)

Convert the following bitstrings into their decimal value, interpereted as single precision floats

a. 0b0 00000000 1011...0 = 11 x 2^-130

b. 0b1 00000001 011...0 = -11 x 2^-129

c. 0b1 11111111 0...0 = -inf

d. 0b1 11111111 1...1 = NaN

e. 0b0 01111111 0...0 = +1

## Floating Point Concepts

1) Why don’t we treat all floats as denormal (i.e. no implicit leading one)?

2) How does the number of floats between 0 and 1 compare to the number of floats between 1 and infinity?

3) If we move bits from the exponent field to the mantissa field, we will be able to represent more, fewer, or the same number of floats?

4) True or false: `(n != n + 1.0)` always evaluates to true, when `|n| != inf`.

## Floating Point Concepts (Answers)

1) To avoid duplicate representation of the same numbers

2) They are approximately equal.

3) Fewer -- we’ll be spending more bit patterns on NaN.

4) False -- We can’t represent every number when n is large.

# MIPS Assembly

Consider the following MIPS function foobar:

```
foobar:
    addu $v0 $0 $0
loop:
    andi $t0 $a0 1
    addu $v0 $v0 $t0
    srl $a0 $a0 1
    bne $a0 $0 loop
    jr $ra
```

**Give the output of foobar for the following calls:**

| | |
|---|---|
| foobar(0) | |
| foobar(0xC1001021) | |
| foobar(0xFFFF) | |
| foobar(0x8000) | |

**Briefly describe the behavior of foobar:**

# MIPS Assembly

Consider the following MIPS function foobar:

```
foobar:
    addu $v0 $0 $0
loop:
    andi $t0 $a0 1
    addu $v0 $v0 $t0
    srl $a0 $a0 1
    bne $a0 $0 loop
    jr $ra
```

**Give the output of foobar for the following calls:**

| | |
|---|---|
| foobar(0) | 0 |
| foobar(0xC1001021) | 6 |
| foobar(0xFFFF) | 16 |
| foobar(0x8000) | 1 |

**Briefly describe the behavior of foobar:**

It counts how many bits are set in $a0

## MIPS, C, and Pointers

Consider the following C function dot_product, which computes the dot product of two vectors of integers, a and b, of size n:

```
int dot_product(int *a, int *b, unsigned n)
{
    int result = 0;
    while(n != 0) {
        result += (*a) * (*b);
        a++;
        b++;
        n--;
    }
    return result;
}
```

**Implement dot_product in MIPS.**

## MIPS, C, and Pointers

Consider the following C function dot_product, which computes the dot product of two vectors of integers, a and b, of size n:

```
int dot_product(int *a, int *b, unsigned n)
{
    int result = 0;
    while(n != 0) {
        result += (*a) * (*b);
        a++;
        b++;
        n--;
    }
    return result;
}
```

```
dot_product:
        addu $v0 $0 $0 # result = 0
loop:
        beq $a2 $0 done # done looping?
        lw $t0 0($a0) # load a elem
        lw $t1 0($a1) # load b elem
        mul $t0 $t1 $t0 # assume this is 1 instr.
        addu $v0 $v0 $t0 # result += (*a) * (*b)
        addiu $a0 $a0 4
        addiu $a1 $a1 4
        addiu $a2 $a2 -1
        j loop
done:
        jr $ra
```

**Implement dot_product in MIPS.**

# MIPS, C, and Pointers

Consider the following C function **dot_product**, which computes the dot product of two vectors of integers, a and b, of size n:

```
float dot_product(int *a, int *b, unsigned n)
{
    int result = 0;
    while(n != 0) {
        result += (*a) * (*b);
        a++;
        b++;
        n--;
    }
    return result;
}
```

```
dot_product:
    addu $v0 $0 $0 # result = 0
loop:
    beq $a2 $0 done # done looping?
    lw $t0 0($a0) # load a elem
    lw $t1 0($a1) # load b elem
    mul $t0 $t1 $t0 # assume this is 1 instr.
    addu $v0 $v0 $t0 # result += (*a) * (*b)
    addiu $a0 $a0 4
    addiu $a1 $a1 4
    addiu $a2 $a2 -1
    j loop
done:
    jr $ra
```

**How many instructions are executed by dot_product (expressed as a function of argument n?**

## MIPS, C, and Pointers

Consider the following C function **dot_product**, which computes the dot product of two vectors of integers, a and b, of size n:

```
float dot_product(int *a, int *b, unsigned n)
{
    int result = 0;
    while(n != 0) {
        result += (*a) * (*b);
        a++;
        b++;
        n--;
    }
    return result;
}
```

```
dot_product:
    addu $v0 $0 $0 # result = 0
loop:
    beq $a2 $0 done # done looping?
    lw $t0 0($a0) # load a elem
    lw $t1 0($a1) # load b elem
    mul $t0 $t1 $t0 # assume this is 1 instr.
    addu $v0 $v0 $t0 # result += (*a) * (*b)
    addiu $a0 $a0 4
    addiu $a1 $a1 4
    addiu $a2 $a2 -1
    j loop
done:
    jr $ra
```

**How many instructions are executed by dot_product (expressed as a function of argument n? 3 + n * 9**

# Cache Locality

You have an array defined as follows:

```
#define N 1024
int matrix[N][N];
```

Your CPU has a byte addressed,16KB direct-mapped cache with 64-byte cache lines/blocks. To improve cache locality, you process the array **`matrix`** in blocks of size 5x5.

# offset bits =

# index bits =

Does a 5x5 block of **`matrix`** fit entirely in the cache? Explain.

Does the answer change if N = 1056?

# Cache Locality

You have an array defined as follows:

```
#define N 1024
int matrix[N][N];
```

Your CPU has a byte addressed,16KB direct-mapped cache with 64-byte cache lines/blocks. To improve cache locality, you process the array **`matrix`** in blocks of size 5x5.

# offset bits = **6** $2^6$ = 64 bytes/block

# index bits = **8** $2^{14}$ bytes/$2^6$ bytes/block = $2^8$ blocks

Does a 5x5 block of **`matrix`** fit entirely in the cache? Explain.

Does the answer change if N = 1056?

# Cache Locality

You have an array defined as follows:

```
#define N 1024
int matrix[N][N];
```

Your CPU has a byte addressed,16KB direct-mapped cache with 64-byte cache lines/blocks. To improve cache locality, you process the array `matrix` in blocks of size 5x5.

# offset bits = **6** $2^6$ = 64 bytes/block

# index bits = **8** $2^{14}$ bytes/$2^6$ bytes/block = $2^8$ blocks

Does a 5x5 block of `matrix` fit entirely in the cache? Explain.

No. Rows in the **matrix** are 4096 ($2^{12}$) bytes apart, so only the top two bits (13-14) of the index change moving across rows. Rows 1&5 have the same index bits, so they collide.

Does the answer change if N = 1056?

# Cache Locality

You have an array defined as follows:

```
#define N 1024
int matrix[N][N];
```

Your CPU has a byte addressed,16KB direct-mapped cache with 64-byte cache lines/blocks. To improve cache locality, you process the array `matrix` in blocks of size 5x5.

# offset bits = **6** $2^6$ = 64 bytes/block

# index bits = **8** $2^{14}$ bytes/$2^6$ bytes/block = $2^8$ blocks

Does a 5x5 block of `matrix` fit entirely in the cache? Explain.

No. Rows in the **matrix** are 4096 ($2^{12}$) bytes apart, so only the top two bits (13-14) of the index change moving across rows. Rows 1&5 have the same index bits, so they collide.

Does the answer change if N = 1056?

Yes. There are 4224 ($2^{12}$+$2^7$) bytes between rows, more index bits change so 5 adjacent rows map to 5 different cache blocks.

```
    la $a0 foobar
    li $a1 5
    jal baz
    j ELSEWHERE
baz:
    addiu   $sp $sp -4
    sw      $ra 0($sp)
    jal     hrm
    addiu   $t1 $v0 48
    addu    $t0 $0 $0
    addu    $t3 $a0 $0
L1:
    beq     $t0 $a1 L1done
    lw      $t2 0($a0)
    sw      $t2 0($t1)
    addiu   $t0 $t0 1
    addiu   $a0 $a0 4
    addiu   $t1 $t1 4
    j L1
```

**What value is returned by this call to baz?**

```
L1done:
    lw $a0 0($t3)
    sll $0 $0 0
    sll $0 $0 0
    sll $0 $0 0
    sll $0 $0 0
    sll $0 $0 0
    lw $ra 0($sp)
    addiu $sp $sp 4
    jr $ra
hrm:
    addiu $v0 $ra -4
    jr $ra
foobar:
    addu $v0 $0 $0
loop:
    andi $t0 $a0 1
    addu $v0 $v0 $t0
    srl $a0 $a0 1
    bne $a0 $0 loop
    jr $ra
    la $a0 foobar
    li $a1 5
    jal baz
    j ELSEWHERE
baz:
    addiu   $sp $sp -4
    sw      $ra 0($sp)
    jal     hrm
    addiu   $t1 $v0 48
    addu    $t0 $0 $0
    addu    $t3 $a0 $0
L1:
    beq     $t0 $a1 L1done
    lw      $t2 0($a0)
    sw      $t2 0($t1)
    addiu   $t0 $t0 1
    addiu   $a0 $a0 4
    addiu   $t1 $t1 4
    j L1
```

**What value is returned by this call to baz?**

3 =
# of bits that are set in encoding of
addu $v0 $0 $0

```
L1done:
    lw $a0 0($t3)
    sll $0 $0 0
    sll $0 $0 0
    sll $0 $0 0
    sll $0 $0 0
    sll $0 $0 0
    lw $ra 0($sp)
    addiu $sp $sp 4
    jr $ra
hrm:
    addiu $v0 $ra -4
    jr $ra
foobar:
    addu $v0 $0 $0
loop:
    andi $t0 $a0 1
    addu $v0 $v0 $t0
    srl $a0 $a0 1
    bne $a0 $0 loop
    jr $ra
```