

CSCE 4610/5610: Solutions To Repeat Exam 2 (April 10, 2008)

1. (25%) 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?

Key

a). For Write-through cache, each write will have to be written to the next level of memory hierarchy and it will incur 100ns.

Read hit will take 10ns

Read miss will take 100ns

Read hit will take 100ns (since this is write through).

Access time on hits = $10\text{ns} * 75\% + 100\text{ns} * 25\%$

Access time on miss = $1\% * 100$ (1% miss rate)

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 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.

Access time on hit (both for read and write) = 10ns

Access time on a miss if replaced line is dirty = $1\% * 300\text{ns} * 0.5 = 1.5\text{ns}$

Access time on a miss if replaced line is clean = $1\% * 100\text{ns} * 0.5 = 0.5\text{ns}$

Effective access time = $10\text{ns} + 1.5 + 0.5 = 12.0\text{ ns}$

So, write-back is better in this case.

b). If we use write-buffers, not every write will incur 100ns delay – in fact we only 10% of 25% writes incur 100ns delay.

Effective access time = $10\text{ns} * 75\% + 25\% * 10\% * 100\text{ns} + 1\% * 100 = 11\text{ns}$.

In this case write-through is better.

2. (35%) Consider the following program segment in 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?

Key

a).

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

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. And the total number of misses generates $= 100 \times 13 = 1300$.

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

These loops also generate 100×100 accesses. Assuming cache is large enough to hold all array big elements, these loops should not cause any new misses.

Total number of accesses to big = $2 \times 100 \times 100 = 20,000$

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 access for big generated by these nested loops. However since we can only about 100 elements (or one row), each j loop still generates 100/8=13 misses. And since the j loop is 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 generates 100*100 misses. However, the elements of big are not accessed along row major. 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 access = 2*100*100 = 20,000

Misses = 1300+100*100 = 11300

Miss rate = 56.5%

3. (40%) You are given the following code. Note that floating-point instructions use floating point registers labeled F. Integer instructions use integer registers labeled R. We are given the following latencies for instructions (that is the dependent data must wait this many cycles to the data from the predecessor instruction).

Floating Point Add/Sub	2
Floating point Multiply	3
Load	1
Integer arithmetic (using data forwarding)	0

```

Loop:    LD      F0, 0(R1)
         MULDD F0, F0, F2
         LD      F4, 0(R2)
         ADDDD  F0, F0, F4
         SD      0(R2), F0
         SUBUI  R1, R1, #8
         SUBUI  R2, R2, #8
         BNEZ   R1, Loop

```

a). Assuming single-issue pipeline, unroll the loop 2 times and schedule the instructions to *minimize the number of cycles* needed to execute the code.

b). Repeat the problem by scheduling instructions to minimize the number cycles on a 3-wide VLIW where you have an integer instruction slot (which is used for integer instructions), a memory slot (which is used for LD, SD) and a floating-point slot (for all floating-point instructions).



Key.

a). Even though not asked, let us first find out how the loop executes with stalls.

			cycle
Loop:	LD	F0, 0(R1)	1
	stall		2
	MULD	F0, F0, F2	3
	LD	F4, 0(R2)	4
	stall		5
	stall		6
	ADDD	F0, F0, F4	7
	stall		8
	stall		9
	SD	0(R2), F0	10
	SUBUI	R1, R1, #8	11
	SUBUI	R2, R2, #8	12
	BNEZ	R1, Loop	13

Now let us think of unrolling the loop code (before scheduling)

Loop:	LD	F0, 0(R1)
	MULD	F0, F0, F2
	LD	F4, 0(R2)
	ADDD	F0, F0, F4
	SD	0(R2), F0
	LD	F6, -8(R1)
	MULD	F6, F6, F2
	LD	F8, -8(R2)
	ADDD	F6, F6, F8
	SD	-8(R2), F6
	SUBUI	R1, R1, #16
	SUBUI	R2, R2, #16
	BNEZ	R1, Loop

Remember that we need to adjust the displacements for load and store instructions to access the next array element. Also since we completed 2 iterations in one loop, we also decrement the index registers (R1 and R2) by 16 each.

If we do not reorder the code, but just schedule in the order the instructions appear, we have the following execution schedule

			cycle
Loop:	LD	F0, 0(R1)	1
	stall		2
	MULD	F0, F0, F2	3
	LD	F4, 0(R2)	4
	stall		5
	stall		6
	ADDD	F0, F0, F4	7
	stall		8
	stall		9
	SD	0(R2), F0	10
	LD	F6, -8(R1)	11
	stall		12
	MULD	F6, F6, F2	13
	LD	F8, -8(R2)	14
	stall		15
	stall		16
	ADDD	F6, F6, F8	17
	stall		18
	stall		19
	SD	-8(R2), F6	20
	SUBUI	R1, R1, #16	21
	SUBUI	R2, R2, #16	22
	BNEZ	R1, Loop	23

If we reorder the code, we can completely eliminate the stalls

			cycle
Loop:	LD	F0, 0(R1)	1
	LD	F6, -8(R1)	2
	MULD	F0, F0, F2	3
	MULD	F6, F6, F2	4
	LD	F4, 0(R2)	5
	LD	F8, -8(R2)	6
	ADDD	F0, F0, F4	7
	ADDD	F6, F6, F8	8
	SUBUI	R1, R1, #16	9
	SUBUI	R2, R2, #16	10
	SD	16(R2), F0	11
	SD	8(R2), F6	12
	BNEZ	R1, Loop	13

(You may be able to achieve the same number of cycles with a different reordering of instructions.)

b). Consider the following sequence of VLIW instructions.

Cycle	Integer operation	Memory operation	FP operation
1		LD F0, 0(R1)	
2		LD F6, -8(R1)	
3		LD F4, 0(R2)	MULD F0, F0, F2
4		LD F8, -8(R2)	MULD F6, F6, F2
5			
6			
7			ADDD F0, F0, F4
8	SUBUI R1, R1, #16		ADDD F6, F6, F8
9	SUBUI R2, R2, #16		
10		SD 16(R2), F0	
11	BNEZ R1, Loop	SD 8(R2), F0	

Note the 3-wide VLIW still needed 11 cycles. Since ideally we should be able to execute $11 \times 3 = 33$ instructions, and we executed only 13 instructions, the VLIW efficiency for this schedule is $13/33 = 39\%$. If we unroll the loop many more times (say 4 or 5 times), we may achieve better efficiency.