ECE 454 – Computer Systems Programming

The Edward S. Rogers Sr. Department of Electrical and Computer Engineering

Midterm Examination Fall 2011

Name	
Student #	

Answer all questions. Write your answers on the exam paper. Show your work.
Each question has a different assigned value, as indicated.

The exam is open book (only simple calculators allowed, no cell phones or PDAs)

Total time available: **110 minutes**

Total marks available: **110** (roughly one mark per minute)

Verify that your exam has all the pages.

Part	Points	Mark
1	30	
2	15	
3	15	
4	15	
5	15	
6	10	
7	10	
Total	110	

PART 1) [30] Short Answer

a) **IBM-Visit Question**: Why did processor architects for machines programmed using punchcards favour CISC instructions over RISC instructions?

2 marks
Programming with punchcards was error-prone; CISC allowed more operations to be specified with fewer instructions, minimizing the amount of code to be written.

b) Why are array based codes easier for compilers to parallelize than pointer-based codes?

2 marks
Compilers have trouble disambiguating pointers, while array index functions can often be fully understood.

c) What is the main challenge for a deep pipeline, and what is the solution?

2 marks
stalls due to branches; the solution is branch prediction

d) Name advanatages of a superscalar processor architecture. For each indicate if the advantage is due to the wide-issue ability or the out-of-order-execution ability of the superscalar processor.

2 marks
* hide/tolerate miss/memory latency---ooo
* exploit instruction-level parallelism---both
* better utilize functional units---both

e) OptsRUs has only enough time to implement one of two optimizations for a customer. Which one provides the most performance? Compute the speedup expected for each, and state which option is the best.

1) Memory Optimization: make 40% of the program 3 times faster, but doing so makes the remaining 60% of the program 1.1 times slower.

3 marks
* speedup = 1/(0.4/3+0.6*1.1) = 1.26

2) Parallelization: 50% of the program is perfectly parallel, the rest is sequential. Half of the users have dual-core processors and the other half of the users have quad-core processors.

3 marks
* average speedup = (1/(0.5/2+0.5) + 1/(0.5/4+0.5))/2 = 1.47
(other reasonable methods of computing the average we accepted)

3) which optimization is best?

1 mark
* parallelization

f) What are pros & cons of profiling an application using a software processor simulator, compared to using an instrumentation approach like PIN.

1) Pros of simulation (vs PIN):

2 marks
* no observer effect
* can see all hardware details

2) Cons of simulation (vs PIN):

2 marks
* slow! can only measure small samples of execution
* difficult to use and program
* simulation may not be accurate

g) In homework2, loop unrolling did not improve performance much in most cases, and made performance worse in other cases. Why?

4 marks
* instruction cache performance is usually worse for unrolling
* in this case, any additional ILP and the savings of having fewer loop control isntructions were not enough to overcome the inst-cache cost.

h) * You are using gprof to analyze a long-running program as you try different gcc optimization levels. You notice that a function that was 20% of execution time for level -O2 is only 0.01% of execution time for level -O3. What most likely happened to cause this?

4 marks
* For -O3 the function was inlined to a "hot" call site and is hence no longer measured by gprof. There must be a non-inlined site as well, because of the 0.01%.

i) Give two reasons why this code:

```
for (i=n-1;i>=0;i--){
    sum += A[i];
}
```

would likely perform better than this code:

```
for (i=0;i<n;i++){
    sum += A[permute(i,n)];
}
```

assuming that

- the `x=permute(i,n)` operation converts each i into a unique pseudo-random number x such that 0<=x<n
- the `permute()` operation is "free" (has zero execution time)
- both codes calculate the identical result
- `A[]` is large
- the code is only exeucted once (not multiple times in a loop)

4 marks

* the first code would have cache block locality and likely fewer misses

* the first code is more amenable to hardware or compiler prefetching, the second is not since it is random access

PART 2) [15] Compiler Optimizations

Apply compiler optimizations to the following function foo (you probably want to do this on a scrap paper). Give a list of the optimizations that you do in order, and for each give the name of the optimization and the line number of the code that was modified. Write your final optimized version of foo in the space provided.

```
 1:  void foo(int n){
 2:    int x=10;
 3:    int y=20;
 4:    int z = x + y;
 5:    int w = 1;
 6:    int v = 0;
 7:
 8:
 9:
10:    for (int i=0;i<n;i++){
11:      x = y*z;
12:      if (x==600){
13:        v += i*(i + n);
14:      }
15:      w *= i + v + n;
16:    }
17:    printf("%d %d %d %d\n",v,w,x,z);
18:  }
```

4: CP
4: CF
11: CP
11: CF
11: LICM
12: CP
12: CF
12/14: DCE
13/14: CSE
17: CP
2: DCE
3: DCE
4: DCE

```
void foo(int n){
  int x;
  int y;
  int z;
  int w = 1;
  int v = 0;

  for (int i=0;i<n;i++){

    int tmp = i + n;
    v += i*tmp;

    w *= v+tmp;
  }
  printf("%d %d %d %d\n",v,w,600,30);
}
```

PART 3) [15] Alignment

```
struct mystruct {
  int x;   // 4
  char c;  // 1
  float y; // 4
  short z; // 2
  char k; // 1
  short p; // 2
} A[10];
```

a) How many bytes of memory will the array A occupy?

5 marks

x:4
c:1
pad:3
y:4
z:2
k:1
pad:1
p:2
pad:2
---
total:20B
array: 20B*10 = 200B

b) Give a space-optimized version of the array A.
5 marks

```
typedef struct mystruct {
  int x;   // 4
  float y; // 4
  short z; // 2
  short p; // 2
  char c;  // 1
  char k; // 1
} A[10];
```

c) How many bytes of memory will the optimized version occupy?
5 marks

x:4
y:4
z:2
p:2
c:1
k:1
pad:2
---
total:16B
array: 16B*10 = 160B

PART 4) [15] Tiling

Given this machine:

1st-level data cache:

- 128B cache blocks
- 8-way set associative
- 16 sets

Page size: 8KB

TLB:

- fully associative
- 128 entries

If you were to tile the following code

```
// assume D[] is already initialized to zeros
void mmmc(int A[], int B[], int C[], int D[], int n){
  for (i=0;i<n;i++){
    for (j=0;j<n;j++){
      for (k=0;k<n;k++)
        D[i*n+j] += A[i*n+k] * B[k*n+j] * C[i*n+j];
      }
    }
  }
}
```

Ignoring storage for instructions and small variables, and assuming that there are no unlucky conflict misses (only capacity misses), what is the largest tile size T (in number of elements, not bytes) that you can use to tile to minimize misses for the:

a) first level data cache
6 marks

capacity = 128*8*16 B = (2^7)*(2^3)*(2^4) B = (2^4)*(2^10) B = 16*1KB = 16KB

4*(4B*tile_size_in_elements**2) <= 16KB
4B*tile_size_in_elements**2 <= 4KB
tile_size_in_elements**2 <= 1024
tile_size_in_elements <= 32 elements

Tile size is 32 elements

b) TLB
6 marks

For a really large n, each row of each matrix would be on a different page.
Therefore rows_per_matrix = 128/4matrices = 32 rows.
Therefore tile size of 32 would fit in the TLB.

This line of reasoning was also accepted, although it assumes matrix size smaller than above:
128entries * 8KB/entry = (2^7)*(2^10)*(2^3) = 2^20 = 1MB total for all tiles
1MB/4tiles = 256KB per tile
4B*tile_size_in_elements = 256KB
tile_size_in_elements = 64 elements

c) given your answers to (a) and (b), what is a good overall tile size in elements to use (considering only 1st level data cache and TLB together)? Explain why the next power-of-two tile sizes that are bigger and smaller than your answer would be worse than your answer (eg., if your answer is 8 elements, discuss for 4 elements and 1 elements).

3 marks
* 32 elements is best

* 64 elements will have same/worse TLB behavior but more L1 misses
* 16 elements will under-use the L1 cache

PART 5) [15] Malloc

Consider an allocator that uses an implicit free list. Each memory block, either allocated or free, has a size that is a multiple of eight bytes. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer and is represented in units of bytes. The usage of the remaining 3 lower order bits is as follows:

- bit 0 indicates the use of the current block: 1 for allocated, 0 for free.
- bit 1 indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- bit 2 is unused and is always set to be 0.

Five helper routines are defined to facilitate the implementation of `free(void *p)`. The functionality of each routine is explained in the comment above the function definition. Fill in the body of the helper routines the code section label that implement the corresponding functionality correctly.

HINT1: all can be done in a single line of C code, but you are allowed to use more than one line if you want

HINT2: use lots of parentheses "()" to clarify your order of operations

```
/* given a pointer p to an allocated block, i.e., p is a
pointer returned by some previous malloc()/realloc() call;
returns the pointer to the header of the block */
void * header(void* p)
{
  void *ptr;

ptr = (void *)(((int *)p) - 1);
OR
ptr = p - 4;

  return ptr;
}
```

```
/* given a pointer to a valid block header or footer,   returns
the size of the block */
int size(void *hp)
{
  int result;

  result = (*((int *)hp)) & (~7);

  return result;
}

/* given a pointer p to an allocated block,i.e. p is a pointer
returned by some previous malloc()/realloc() call; returns the
pointer to the footer of the block*/
void * footer(void *p)
{
  void *ptr;

   ptr = p + size(header(p)) - 8;
  OR
   ptr = ((int *)p) + size(header(p))/4 - 2;

  return ptr;
}

/* given a pointer to a valid block header or footer,   returns
the usage of the currect block,
   1 for allocated, 0 for free */
int allocated(void *hp)
{
  int result;

  result = (*((int *)hp))&1;

  return result;
}

/* given a pointer to a valid block header,  returns the
pointer to the header of previous block in memory */
void * prev(void *hp)
{
  void *ptr;

   ptr = hp - size(hp-4);
   OR
   ptr = ((int *)hp) - size(((int *)hp)-1)/4;
  return ptr;
}
```

PART 6) [10] Cache Coherence

Assuming a 4-CPU multicore with MESI invalidation-based cache coherence with write-back caches, considering only the cache block for location X, in the table below are shown the order in time of loads and stores to X. Put a '1' or a checkmark in the column on the left next to each row for which a coherence message occurs that requests a copy of the cache block contents for the corresponding CPU's cache. HINT: a load-miss results in a copy request, while a write-hit does not.

Copy of cache block requested?	CPU 0	CPU 1	CPU 2	CPU 3
1	Load X			
	Store X			
1		Load X		
	Load X			
		Store X		
1			Load X	
1				Load X
			Load X	
		Store X		
1	Store X			
	Store X			
	Load X			
1		Load X		
1			Load X	

Minus two marks for any checkmark missing/extra

PART 7) [10] Consistency

Consider the following code for two threads, including an initial state:

```
Initialization: a=0,b=0,x=0,y=0;

T1:
x = 1;
b = y;

T2:
y = 1;
a = x;
```

a) What are the possible outcomes for (a,b) for all possible executions of the two threads given, assuming a modern out-of-order superscalar processor? Put a checkmark next to each possible valid outcome.

3 marks

(a,b) = (0,0) _____X

(a,b) = (0,1) _____X

(a,b) = (1,0) _____X

(a,b) = (1,1) _____X

b) Sequential consistency is a strict form of consistency for a parallel architecture where we assume that each thread/CPU executes operations in their original program order, but that instructions from different threads might be interleaved in any manner. What are the possible outcomes for (a,b) for all possible interleavings of the two threads given? Put a checkmark next to each possible valid outcome.

3 marks

(a,b) = (0,0) _____

(a,b) = (0,1) _____X

(a,b) = (1,0) _____X

(a,b) = (1,1) _____X

c) Assuming sequential consistency, will the following code for threads T1 and T2 properly synchronize the communication of the value of `a` from T1 to T2 via `x`? Explain briefly why or why not. Assume that flag is initialized to zero.

T1:
```
x = a;
flag = 1;
```

T2:
```
while(!flag){};
… = x;
```

3 marks
YES: in T1, "x=a" must execute before "flag=1", so in T2 the while loop cannot complete until after "flag=1" has executed, implying that "x=a" has already executed.