Midterm Examination Sample Solution
ECE 454F 2009: Computer Systems Programming
Date: Nov 10, 2009, 3-5 p.m.

Instructor: Cristiana Amza
Department of Electrical and Computer Engineering
University of Toronto

Problem number	Maximum Score	Your Score
1	12	
2	12	
3	15	
4	18	
5	15	
6	12	
7	16	
total	100	

This exam is open textbook and open lecture notes. You have two hours to complete the exam. Use of computing and/or communicating devices is NOT permitted. You should not need any such devices. You can use a basic calculator if you feel it is absolutely necessary.

Work independently. Do not remove any sheets from this test book. Answer all questions in the space provided. No additional sheets are permitted. Scratch space is available at the end of the exam.

Write your name and student number in the space below. Do the same on the top of each sheet of this exam book.

Your Student Number	
Your First Name	
Your Last Name	

Student Number: Name:

Problem 1. Basic Facts. (12 Points)

Part A (6 points)

a)(2 points) State and briefly explain one significant limitation of gprof.
Answer: gprof's sampling period is too coarse grained. For execution times of a few seconds, gprof cannot provide sufficient profiling accuracy in this case (e.g., if a piece of code contains function calls, gprof may skip the entire execution of some of these functions during profiling).

b)(2 points) State and briefly explain one significant limitation of Pin.
Answer: Pin can run only on Intel architectures.

c)(2 points) Briefly explain why most compilers don't do whole code (inter-procedural) optimizations.
Answer: Inter-procedural analysis is i) too expensive and ii) likely unable to determine optimization safety in cases of procedure side-effects.

Part B (6 points)

```
typedef struct
{
  char c[3];
} Obj;
```

Assume a program that you are writing is supposed to access N objects in sequence e.g., in a loop. The objects can either be *statically* allocated (as an array of objects) or dynamically allocated (as an array of pointers to objects). Furthermore, the dynamic allocation can use either an *implicit* list allocator or an *explicit* list allocator. Assume an empty heap initially and a first-fit policy for the dynamic allocators. Please order the three approaches above from best to worst in terms of estimated cache behavior of the program. Justify your answer briefly by stating any assumptions about the cache line size and sketching or stating the structure of allocated blocks (you can assume and use any variation of the techniques above learned in class for block allocation).

Answer:
Best: Statically allocated. 2nd: Implict list allocator. Worst: Explicit list allocator.

Student Number: Name:

Problem 2. Profiling and Speedup. (12 Points)

Part A (7 points)

Explain the following profile data obtained using gprof on the n-body simulation application. Note: You have to explain the meaning of the numbers and the organization of the output. What conclusions can you draw from this profile data? Think of why we used gprof in the first place when drawing conclusions.

```
Flat profile sample: ..................
%      cumulative self               self     total
time   seconds    seconds    calls   s/call   s/call  name
96.33   1076.18   1076.18   278456   0.00     0.00  World::update()
 0.96   1086.85    10.67  212849617  0.00     0.00  ThreadManagerSerial::doSerialWork()
 0.63   1093.86     7.01        1    7.01    39.30  OpenGLWindow::runWindowLoop
 (char const*, int, int, bool (*)(), void (*)(SDL_KeyboardEvent*))
..................

Call graph sample:

index % time    self  children    called     name
..................
             1076.18    0.09  278456/278456   worldUpdate() [5]
[6]     96.3 1076.18    0.09  278456        World::update() [6]
                0.07    0.01  278240/278241   World::calculateTimeDeltaMilliseconds()[27]
                0.01    0.00    5338/5538
World::resetForegroundElement(World::ForegroundElement*) [38]
                0.00    0.00     149/149      World::updateCallsPerSecond(int) [47]
                0.00    0.00    4052/4287     World::radiusForMass(float) [139]
                0.00    0.00     642/642
World::swapForegroundElement(World::ForegroundElement*,World::ForegroundElement*)[140]
..................
```

Please also use the space on the next page if needed.

Answer:
You should provide a basic explanation of the profile data. For example, from the flat profile we see that 96.33% of the application running time is spent in the function World::update. In seconds this translates to 1076.18. This function is being called 278456 times and the time per call is in terms of milliseconds or less. Similarly for the other 2 functions in the flat profile.

The call graph profile is for the function World::update. This function is being called from worldUpdate (the parent) and it calls a number of other functions, such as World::calculateTimeDeltaMilliseconds, World::updateCallsPerSeconds, World::radiusForMass, World::swapForegroundElement (these are the children). As in the previous case, the application spends 96.33% in World::update, meaning 1076.18 s. Next you should say how much time is spent in each child. You should also explain the called column. From the called column we can see that worldUpdate is the only parent of World::update and there are 278456 calls to this function. The same for the children.

Conclusions: From the flat profile we see that World::update is the bottleneck of the application. From the call graph profile we notice that the bottleneck is actually inside this function and not in its children.

Part B (5 Points)

Assume that by profiling the code of our game application with gprof you get that 80% of the time is spent in a loop which is updating the physics of the game world, while 20% of the time is spent in frame rendering. Therefore you focus all your efforts only on improving the update world function through code optimizations such as loop unrolling, use of blocking for the update loop for cache hit improvements and loop parallelization. What is the best speedup you can get for the game application as a whole ?
Answer:
By Amdahl's Law, if $1/s$ of the program is sequential, then you can never get a speedup better than s.
For this question, $1/s = 0.2$, so the best speed up is 5.

Student Number: Name:

Problem 3. Performance (CPE). (15 Points)

Consider the following function for computing the dot product of two arrays of n integers each. We have unrolled the loop by a factor of 3.

```
int dotprod(int a[], int b[], int n)
{
  int i, x1, y1, x2, y2, x3, y3;
  int r = 0;
  for (i = 0; i < n-2; i += 3) {
    x1 = a[i]; x2 = a[i+1]; x3 = a[i+2];
    y1 = b[i]; y2 = b[i+1]; y3 = b[i+2];
    r = r + x1 * y1 + x2 * y2 + x3 * y3; // Core computation
  }
  for (; i < n; i++)
     r += a[i] * b[i];
   return r;
}
```

Assume that we run this code on a machine in which multiplication requires 7 cycles, while addition requires 5. Further, assume that these latencies are the only factors constraining the performance of the program. Don't worry about the cost of memory references or integer operations, resource limitations, etc.

Please compute the ideal cycles per element (CPE) for each of the following associations for the core computation of this function and explain its performance. *Note: 1. If your answer is a fraction, leave it as such, you don't need to compute the result as a decimal number. 2. You need to explicitly state your assumptions, if any, and/or explicitly show your work.*

a. ((r + x1 * y1) + x2 * y2) + x3 * y3
Answer: CPE = 15/3

b. (r + (x1 * y1 + x2 * y2)) + x3 * y3
Answer: CPE = 10/3

c. r + ((x1 * y1 + x2 * y2) + x3 * y3)
Answer: CPE = ~~7~~ 5/3

Student Number: Name:

Problem 4. Cache Miss Rates. (18 Points)

After a stressful semester you suddenly realize that you haven't bought a single Christmas present yet. Fortunately, you see that one of the big electronic stores has CD's on sale. You don't have much time to decide which CD will make the best presents for which friend, so you decide to automatize the decision process. For that, you use a database containing an entry for each of your friends. It is is implemented as an 8×8 matrix using a data structure `person`. You add to this data structure a field for each CD that you consider:

```
struct person{
  char name[16];
  int age;
  int male;
  short nsync;
  short britney_spears;
  short avril_lavigne;
  short garth_brooks;
}
struct person db[8][8];
register int i, j;
```

Part 1

After thinking for a while you come up with the following smart routine that finds the ideal present for everyone.

```
void generate_presents(){
  for (j=0; j<8; j++){
      for (i=0; i<8; i++) {
          db[i][j].nsync=0;
          db[i][j].britney_spears=0;
          db[i][j].garth_brooks=0;
          db[i][j].avril_lavigne=0;
      }
   }
  for (j=0; j<8; j++){
      for (i=0; i<8; i++) {
          if(db[i][j].age < 30){
             if(db[i][j].male)
                   db[i][j].britney_spears = 1;
             else  db[i][j].nsync = 1;
          }
          else{
             if(db[i][j].male)
                   db[i][j].avril_lavigne =1;
             else  db[i][j].garth_brooks = 1;
          }
      }
   }
```

```
}
```

Of course, runtime is important in this time-critical application, so you decide to analyze the cache performance of your routine. You assume that

- your machine has a 512-byte direct-mapped data cache with 64 byte blocks.
- `db` begins at memory address 0
- The cache is initially empty.
- The only memory accesses are to the entries of the array `db`. Variables `i`, and `j` are stored in registers.

Answer the following questions: *Note: You can write the answer for A and B as a set of multiplications e.g., $a * b * c$, and the answer for C as a fraction.*

A. What is the total number of read and write accesses? _____.
Answer: 4*8*8 + 3*8*8 = 448

B. What is the total number of read and write accesses that miss in the cache? _____ .
Answer: 1*8*8 + 1*8*8 = 128

C. So the fraction of all accesses that miss in the cache is: _____.
Answer: 2/7

Part 2

After testing, you found that the performance of the above implementation (Part 1) is quite poor. Could you think of an alternative implementation which can greatly reduce the cache misses for the same application? Please write and/or explain in plain words your code, and with the same assumptions as in Part 1 compute the answer to the following questions for your implementation.
Answer:

```
void generate_presents(){

  for (i=0; i<8; i++){
      for (j=0; j<8; j++) {

          if(db[i][j].age < 30)
             if(db[i][j].male) {
                db[i][j].nsync=0;
                db[i][j].britney_spears=1;
                db[i][j].garth_brooks=0;
                db[i][j].avril_lavigne=0;
             }
```

```
            else
               db[i][j].nsync=1;
               db[i][j].britney_spears=0;
               db[i][j].garth_brooks=0;
               db[i][j].avril_lavigne=0;
            }
         }

         else{
            if(db[i][j].male) {
               db[i][j].nsync=0;
               db[i][j].britney_spears=0;
               db[i][j].garth_brooks=0;
               db[i][j].avril_lavigne=1;
            }

            else{
               db[i][j].nsync=0;
               db[i][j].britney_spears=0;
               db[i][j].garth_brooks=1;
               db[i][j].avril_lavigne=0;
            }
         }


      }
   }

}
```

A. What is the total number of read and write accesses? _____
Answer: 8*8*6

B. What is the total number of read and write accesses that miss in the cache? _____
Answer: 8*4

C. So the fraction of all accesses that miss in the cache is: _____.
Answer: 1/12

Problem 5. Dynamic Memory Allocation (15 Points)

Consider an allocator with the following specification:

- Uses a single explicit free list.
- All memory blocks have a size that is a multiple of 8 bytes and is at least 16 bytes.
- All headers, footers, and pointers are 4 bytes in size
- Headers consist of a size in the upper 29 bits, a bit indicating if the block is allocated in the lowest bit, and a bit indicating if the previous block is allocated in the second lowest bit.
- Allocated blocks consist of a header and a payload (no footer)
- Free blocks consist of a header, two pointers for the next and previous free blocks in the free list, and a footer at the end of the block.
- All freed blocks are immediately coalesced.
- The heap starts with 0 bytes, never shrinks, and only grows large enough to satisfy memory requests.
- The heap contains only allocated blocks and free blocks. There is no space used for other data or special blocks to mark the beginning and end of the heap.
- When a block is split, the lower part of the block becomes the allocated part and the upper part becomes the new free block.
- Any newly created free block (whether it comes from a call to free, the upper part of a split block, or the coalescing of several free blocks) is inserted at the beginning of the free list.
- All searches for free blocks start at the head of the list and walk through the list in order.
- If a request can be fulfilled by using a free block, that free block is used. Otherwise the heap is extended only enough to fulfill the request. If there is a free block at the end of the heap, this can be used along with the new heap space to fulfill the request.

Student Number: Name:

Simulating Malloc

Below you are given a series of memory requests. You are asked to show what the heap looks like after each request is completed using a first fit placement policy. The heap is represented as a series of boxes, where each box is a single block on the heap, and the bottom of the heap is the left most box. In each block, you should write the total size (including headers and footers) of the block in bytes and either 'f' or 'a' to mark it as free or allocated, respectively. For example, the following heap contains an allocated block of size 16, followed by a free block of size 32.

16a	32f

Assume that the heap is empty before each of the sequences is run. You do not necessarily have to use all the boxes provided for the heap. Some of the boxes are already filled in to help you.

```
ptr1 = malloc(32);
ptr2 = malloc(16);
ptr3 = malloc(16);
ptr4 = malloc(40);
free(ptr3);
free(ptr1);
ptr5 = malloc(16);
free(ptr4);
ptr6 = malloc(48);
free(ptr2);
```

40a					
40a	24a				
40a	24a	24a			
40a	24a	24a	48a		
40a	24a	24f	48a		
40f	24a	24f	48a		
24a	16f	24a	24f	48a	
24a	16f	24a	72f		
24a	16f	24a	56a	16f	
24a	40f	56a	16f		

Student Number: Name:

Problem 6. Parallel Programming. (12 Points)

Identify the dependences in the following for loop nest, written in pseudocode, specifying: the type of the dependence, whether it is loop-dependent or loop-independent and the iterations/statements/variables involved.

Part A. (6 points)

```
for( I = 1; I <= 100; I++){
   for( J = I; J <= I + 19; J++) {
S1:   A [I + 20][J] = A [I][J] + B;
   }
}
```

Answer: There are no dependences here. While it may appear that there is a dependence, with the distance vector being (20, 0), this is not the case, because the iteration ranges for the J-loop at the source and sink of the dependence are non-intersecting: when I = 1, statement S1 stores into A(21, 1:20), while at the sink, statement S1 reads from A(21, 21:40). Since there are no common memory locations in the two slices, there can be no dependence.

Part B. (6 points)

```
for(J = 1; J <= M; J++){
   for(I = 1; I <= N; I++){
      A [I][J + 1] = A [I + 1][J] + B;
   }
}
```

Is it legal to perform a loop interchange between the two loops in the above code ? Explain. If the transformation is legal, explain in your own words how the code can be parallelized or parallelize the restructured code using the parallel for OpenMP pragma.
Answer: It is not legal. The transformation reorders the endpoints of the dependence, therefore it is not legal. This can be verified by the following example: A (2, 2)

Initially: first: J = 1, I = 2: A (2, 2) = ... later: J = 2, I = 1: ... = A (2, 2)

So the value in A (2, 2) is read (in the second iteration of the outer loop) AFTER it is assigned to in the first iteration of the outer loop.

After loop interchange: first: I = 1, J = 2: ... = A (2, 2) later: I = 2, J = 1: A (2, 2) = ...

So now the value in A (2, 2) is used BEFORE being assigned to, which is clearly different from the original code, hence wrong.

Student Number: Name:

Problem 7. Performance Optimization. (16 Points)

SOR is a kernel (core) code used in many image processing functions, such as the smoothing function for image processing discussed below.

We will consider an image to be represented as a two-dimensional matrix M, where $M_{i,j}$ denotes the value of (i, j)th pixel of M. Pixel values are triples of red, green, and blue (RGB) values. We will only consider square images. Let N denote the number of rows (or columns) of an image. Rows and columns are numbered, in C-style, from 0 to $N - 1$.

Given this representation, the `smooth` operation is implemented by replacing every pixel value with the average of all the pixels around it (in a maximum of 3×3 window centered at that pixel).

The smoothing function takes as input a source image `src` and returns the smoothed result in the destination image `dst`. The core data structure deals with image representation. A `pixel` is a struct as shown below:

```
typedef struct {
   unsigned short red;   /* R value */
   unsigned short green; /* G value */
   unsigned short blue;  /* B value */
} pixel;
```

As can be seen, RGB values have 16-bit representations ("16-bit color"). An image `I` is represented as a one-dimensional array of `pixel`s, where the (i, j)th pixel is `I[RIDX(i,j,n)]`. Here `n` is the dimension of the image matrix, and `RIDX` is a macro defined as follows:

```
#define RIDX(i,j,n) ((i)*(n)+(j))
```

Here is part of a the implementation for the smooth function in the naive_smooth code version below:

```
/*
 * accumulate_sum - Accumulates field values of p in corresponding
 * fields of sum
 */
static void accumulate_sum(pixel_sum *sum, pixel p)
{
    sum->red += (int) p.red;
    sum->green += (int) p.green;
    sum->blue += (int) p.blue;
    sum->num++;
    return;
}


/*
 * assign_sum_to_pixel - Computes averaged pixel value in current_pixel
 */
static void assign_sum_to_pixel(pixel *current_pixel, pixel_sum sum)
{
    current_pixel->red = (unsigned short) (sum.red/sum.num);
    current_pixel->green = (unsigned short) (sum.green/sum.num);
    current_pixel->blue = (unsigned short) (sum.blue/sum.num);
```

```
    return;
}


/*
 * avg - Returns averaged pixel value at (i,j)
 */
static pixel avg(int dim, int i, int j, pixel *src)
{
    int ii, jj;
    pixel_sum sum;
    pixel current_pixel;

    initialize_pixel_sum(&sum);
    for(jj=max(j-1, 0); jj <= min(j+1, dim-1); jj++)
        for(ii=max(i-1, 0); ii <= min(i+1, dim-1); ii++)
            accumulate_sum(&sum, src[RIDX(ii,jj,dim)]);

    assign_sum_to_pixel(&current_pixel, sum);

    return current_pixel;
}


void naive_smooth(int dim, pixel *src, pixel *dst) {
  int i, j;

  for(i=0; i < dim; i++)
    for(j=0; j < dim; j++)
      dst[RIDX(i,j,dim)] = avg(dim, i, j, src); /* Smooth the (i,j)th pixel */

  return;
}
```

Our *only* goal is to optimize the `smooth` function to run as fast as possible.

(a) (8 points) Assume that you first apply blocking to the `smooth` function, but you find that this optimization does not reduce its CPE. Please explain why this would happen in the specific case of the given `smooth` function.
Answer:
Blocking/tiling addressed the likely bottleneck in the first case, which was memory accesses (there was not much else happening in the SOR code). The smooth code has many function calls inside double nested loops. The avg functions is called in a double nested loop, calling in its own turn the accumulate_sum function in another double nested loop (min is also called on every iteration, which is unlikely to be optimized by the compiler) and also the assign_sum_to_pixel function.

As mentioned in the lectures, function calls withing loops are one of the most expensive things your code can have (we usually try to move them out of the loop or just replace them with plain computation).

People that mentioned as reason that accesses for accumulation of points are in column order instead of row order got partial points. Nobody is forcing you to do column-wise sweeps when blocking. In fact, for the SOR code, you likely did row-wise sweeps inside each block. If I changed the order of the summation of the 4 neighbors in SOR, would that have automatically made you scan the grid array by column as well ? The touching order for the 9 points in the

naive smooth function (or the 4 points in SOR) when accumulating them is likely not of any substantial importance.

(b) (8 points) Please describe in sufficient detail a different method which you think would be more successful in reducing the CPE for the `smooth` function (no need to write any code) and briefly explain its expected effects.

Answer:
Use plain computation for the point accumulation (for each color), rather than calling accumulate. Avoid all function calls by replacing avg with the actual computation, the function call for the index calculation (RIDX) with actual computation or pointer arithmetic (as exercised in Lab 2). Specialize the code for corner cases. Introducing some `if` statements is not ideal, but intuitively much better than having a doubly nested loop calling two functions for adding 9 point values or less together. In the avg function, as a secondary factor compared to calling the accumulate function, the two loops themselves introduce overhead for such a simple computation requiring 3 iterations or less - so we should just write all cases of pixel accumulation using additions of all pixel values involved.

After all of this, we can apply blocking and, at that point, it may indeed improve performance.