Midterm Sample Solution
ECE 454F 2007: Computer Systems Programming
Date: Tuesday, Oct 23, 2007 3:10 p.m. - 5 p.m.

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

Problem number	Maximum Score	Your Score
1	16	
2	15	
3	24	
4	21	
5	24	
total	100	

This exam is open textbook and open lecture notes. You have one hour and 50 minutes 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.

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. Work independently.

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 profiling and optimization Facts. (16 Points)

Part A (8 points)

a)(2 points) State and briefly explain one significant limitation of gprof.

Gprof's sampling period is too coarse grained. Thus, gprof cannot provide accurate profiling for short running programs. For example, it may skip the entire execution of some functions during profiling.

b)(2 points) State and briefly explain one significant limitation of Pin.

Pin can run only on Intel architectures. However, any reasonable answer will be provided full or almost full marks. As an example - I will consider the following answer(s) correct too: Pin provides the ability to insert arbitrary code in arbitrary places in the program in the form of programmer plug-ins for profiling instruction counts, various code behaviors for simulated architectural features, etc. This implies both programmer burden for writting the plug-in and/or run-time overhead for running the plug-in.

c)(2 points) Briefly explain why most compilers don't do whole code (inter-procedural) optimizations.

Inter-procedural analysis is i) too expensive and ii) likely unable to determine optimization safety in cases of procedure side-effects.

d)(2 points) Briefly describe a code restructuring optimization that can improve cache locality for the following code assuming that arrays are stored in column-major order (i.e., elements are stored in memory contiguously by column instead of by row).

```
do i = 1,n
   do j = 1,n
     a(i,j) = a(i,j-1)
   end do
end do
```

Answer: Loop reordering (exchanging the i and j loops). People who answered blocking without any specifics would get partial points.

PART B Profiling with gprof (8 Points)

Given the gprof output bellow, answer the following questions:

```
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(int)
```

```
  ..................

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::calcTimeDeltaMilli() [27]
                0.01    0.00    5338/5538        World::resetForegroundElement() [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() [140]
```

(a) How much time is spent in each call of World::update?

Answer: 1076.18 sec / 278456 calls = 3.86 ms/call

(b) Which is the function that calls World::radiusForMass(float) the most?

Answer: World::update made 4052 of the 4287 calls to World :: radiusForMass (i.e., the most calls)

(c) How many times is World::radiusForMass called from other functions?

There are 4287 calls to World::radiusForMass from other functions, 4052 of which from World::update and 235 calls from functions other than World::update. Any correct answer (including mentioning just 4287) gets full points.

(d) Which is the function where most of the time is spent? How much time is spent in this function and how much time is spent in its children?

World::update. 1076.18 s is spent in this function. 0.09 s is spent in its children.

Problem 2. Machine Dependent Performance Optimization. (15 Points)

The following problem concerns optimizing codes for maximum performance on an Intel Pentium III, the same architecture we used in the lectures. Recall the following performance characteristics of the *pipelined* functional units for this machine:

Operation	Latency	Issue Time
Integer Add	1	1
Integer Multiply	4	1
Integer Divide	36	36
Floating Point Add	3	1
Floating Point Multiply	5	2
Floating Point Divide	38	38
Load or Store (Cache Hit)	1	1

Consider the following code segment:

```
int compute(int a[], int b[], int n)
{
    int i,x,y;

    for ( i = 0; i < n ; i++){
        y = a [i] + b [i];
        x *= y;
    }
    return x;
}
```

We express the performance of the function in terms of the number of cycles per element (CPE). This measure assumes the run time, measured in clock cycles, for an array of length n is a function of the form $Cn + K$, where C is the CPE. You are asked to compute the *theoretical* CPE, in which we will assume that we are not limited in the number of functional units executing operations in parallel (we have enough to accomodate all the pipelining scenarios below).

(a)(5 points) To improve the performance of the compute function, a student team restructured the code and unrolled the loop as follows. Please calculate the respective *theoretical* CPE's of: i) the original code (above) and ii) the new optimized code (below). Explain your CPE answers briefly.

```
// the student team introduced stores into an additional array (c)
// assume array c is declared/allocated before this code segment

for (i=0; i< n; i+=2){
   c[i] = a [i] + b[i];
   x *= c[i];
   c[i+1] = a [i+1] + b[i+1];
   x *= c[i+1];
}
```

Answer: The CPE is 4 in both cases since the dependencies on x exist and the multiplication latency (4 cycles) dominates in both code versions. In pipelining cases, you need to think of what cannot be overlapped. The loads, stores, adds, etc of one iteration can be overlapped with the loads, stores, adds and multiplies of the previous iteration(s). All multiplies need to execute sequentially because of dependencies.

(b)(5 points) Please write your own best attempt at the optimized version of the original code, but using the same loop unrolling factor of 2, and calculate your code's *theoretical* CPE.

Answer:

```
int x, y, z;
for (i=0; i< n; i+=2){
   y = a [i] + b[i];
   z = a [i+1] + b[i+1];
    x = x *(y*z);
}
```

Theoretical CPE is 4 / 2 = 2. We can perform the multiplications necessary for two elements in parallel.

(c)(5 points) In practice, besides the number of functional units, what are the limiting factors for increasing the loop unrolling factor for the code above ?

Answer: Limited instruction cache size (code bloat can cause more instruction cache misses) and limited number of registers.

Student Number: Name:

Problem 3. Machine Independent and Dependent Performance Optimization. (24 Points)

Part A (12 points)

The following code segment is from the SOR application discussed in the lectures.

```
for( i=1; i<n-1; i++ )
    for( j=1; j<n-1; j++ )
        temp[i][j] = 0.25 * (grid[i-1][j]+grid[i+1][j]+grid[i][j-1]+grid[i][j+1]);
```

(a)(3 points) Describe an optimization for the above code segment designed to improve the cache hit rate. Please briefly explain why your method helps.

Answer:

Traversing the array in blocks (technique called blocking/tiling) can make it highly probable that grid[i][j] is still in the cache on the second access to it (by reducing the number of accesses that go through the cache between the first and second access to it).

(b) (9 points) Write the optimized code after applying the method you proposed in (a) for this code segment.

Answer:

```
int block_size = 16;

for(ii=1; ii < n-1; ii+=block_size)
  for(jj=1; jj < n-1; jj+=block_size)
    for( i=ii; i<min(n-1, ii+block_size); i++ )
      for( j=jj; j<min(n-1,jj+block_size); j++ )
        temp[i][j] = 0.25 * (grid[i-1][j]+grid[i+1][j]+grid[i][j-1]+grid[i][j+1]);
```

Part B (12 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) (6 points) Assume that you first apply the same method you proposed in Part A 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) (6 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.

Problem 4. Cache Misses Rate Analysis. (21 Points)

In this problem, you need to determine the efficiency of the following algorithms on a machine with a 2048-byte direct-mapped data cache with 32 byte blocks.

You are given the following definitions:

```
struct point {
    int a;
    char b;
    char c;
    int d;
    int e;
};

struct point square[16][16];
register int i, j;
```

Assume:

- `sizeof(int) = 4`
- `sizeof(char) = 1`
- `square` begins at memory address 0
- The cache is initially empty.
- The only memory accesses are to the entries of the array `square`. Variables `i` and `j` are stored in registers.

A. What percentage of the writes in the following code will miss in the cache?

```
for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
        square[i][j].a = 0;
        square[i][j].b = 'a';
        square[i][j].c = 'a';
        square[i][j].d = 0;
        square[i][j].e = 0;
    }
}
```

Miss rate for writes to `square`: _____ %

Answer: 10.
Each structure point in square occupies 16 bytes due to padding, and each cache line can load 2 structure points. In this cache line, the first access of the structure member "a" is a miss, the other 9 accesses are all hits. So the miss rate is 1/10=10%.

B. What percentage of the writes in the following code will miss in the cache?

```
for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
        square[j][i].a = 0;
        square[j][i].b = 'a';
        square[j][i].c = 'a';
        square[j][i].d = 0;
        square[j][i].e = 0;
    }
}
```

Miss rate for writes to `square`: _____ %

Answer: 10 (20 also valid answer, but only if reasoning is correct and clear).
The data is accessed along the column here. As before, the access to structure member "a" is a miss, but then everything from the same element that fits inside the same cache line, i.e., "b","c","d","e" are all hits. We are bringing in the next element in the row-wise direction, but not accessing them. So far so good. People who reasoned clearly about this in such a way that I can check their reasoning and gave an answer of 20% got full marks. However, the problem has the additional subtlety of asking you to understand whether a column fits in the cache or not. In other words, how much data goes through the cache before I get to access the elements from the second column. When we get to the next column is it still there in the cache or not ? If we calculate the total data accessed in one column we get 16 x 16 bytes = 256 bytes, which fits in the cache (of 2048 bytes).

So the key is - after all data of one column is accessed, the data of the next column, which is brought into the cache with the previous column is still in the cache. So, the miss rate is the same as before 1/10=10%.

C. What percentage of the writes in the following code will miss in the cache?

```
Part 1:
for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
        square[i][j].a = 1;
    }
}
Part 2:
for (i=0; i<16; i++) {
    for (j=0; j<16; j++) {
        square[i][j].b = 'a';
        square[i][j].c = 'a';
        square[i][j].d = 0;
        square[i][j].e = 0;
    }
}
```

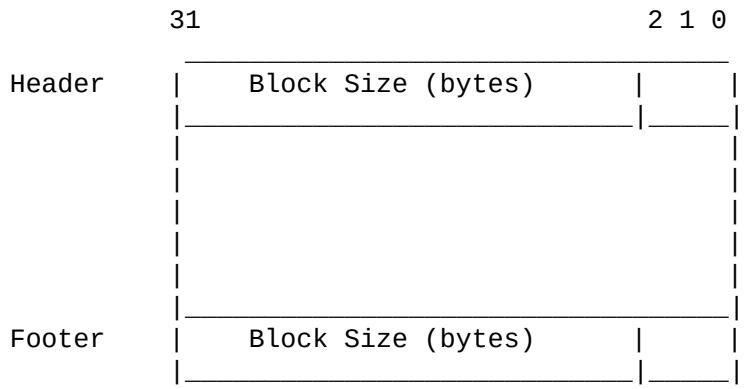
Miss rate for writes to `square`: _____ %

Answer: 20
Similar reasoning as Part A regarding alignment and what fits within a cache line.
One cache line contains two structure points, so the first part of the code has 128 misses, 128 hits. The cache size is only 2M, so it spills. Thus, the second part of the code has to load data into the cache. It has 128 misses and 7*128 hits. So, the total miss rate is (128+128)/(128+128+128+7*128)=2/10=20%
Note: A common mistake is to just add part 1's miss rate and part 2's miss rate.

Problem 5. Dynamic Memory Allocation. (24 Points)

Consider an allocator that uses an implicit free list.

The layout of each free memory block is as follows:



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 for free blocks). 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.

Part A. (12 points)

Part A.1.

Six 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 (A, B or C). Show your reasoning or explain your choices briefly.

```
/* 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;

  _______;
  return ptr;
}
```

```
A. ptr=p-1
B. ptr=(void *)((int *)p-1)
C. ptr=(void *)((int *)p-4)


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

  _______;
  return result;
}

A. result=(*hp)&(~7)
B. result=((*(char *)hp)&(~5))<<2
C. result=(*(int *)hp)&(~7)


/* 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;

  _______;
  return ptr;
}

A. ptr=p+size(header(p))-8
B. ptr=p+size(header(p))-4
C. ptr=(int *)p+size(header(p))-2



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

  ______;
  return result;
}

A. result=(*(int *)hp)&1
```

```
B. result=(*(int *hp)&0
C. result=(*(int *)hp)|1


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

  ______;
  return ptr;
}

A. ptr = hp - size(hp-4)
B. ptr = hp - size(hp-4) + 8
C. ptr = hp - size(hp-4) + 4


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

  ______;
  return ptr;
}

A. ptr = hp - size(hp)
B. ptr = hp - size(hp-4)
C. ptr = hp - size(hp-4) + 4
```

Answers: B, C, A, A, C, B

Part A.2.

From the functions implemented above, which is the least useful in an implementation of a dynamic memory allocator that uses implicit free lists? Explain your reasoning.

Answer: The prevpayload helper function is the least useful. We are generally interested in the header of the previous block (but not in the payload of the previous block) for the purpose of coallescing.

Part B. (12 points)

Consider the allocator that uses implicit free lists from part A. We make the following space optimization: *Allocated* blocks do not have a footer. *Free* blocks still have the footer and the rest of the block structure is the same as before.

(a) Please explain briefly whether or not we loose any functionality or performance in the allocator by making the above space optimization. Explain your answer.

Answer: We do not loose any functionality or performance. The footer is useful only for the purposes of coalescing. If we need to coalesce backwards and the previous block is free, then that block will have a footer, thus we know its size and we can coalesce (as per the description of the space optimization). So, the only thing we need to know is whether the previous block is allocated or free. Even without the footer in the current (allocated block) we can look at bit 1, which tells us whether the previous adjacent block is free (as per the meta-data given in the problem).

(b) Using the space optimization above and given the contents of the heap shown on the left, show the new contents of the heap (in the table on the right) after the following call: `malloc(4)` is executed. Your answers should be given as hex values. Update only the necessary boundary tags. Note that the address grows from bottom up. Assume that the allocator uses a *first fit* allocation policy.

Answer:

Student Number: Name:

Address	Content
0x400b028	0x00000012
0x400b024	0x400b621c
0x400b020	0x400b620c
0x400b01c	0x00000012
0x400b018	0x400b512c
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	0x00000011
0x400b008	0x00000012
0x400b004	0x400b601c
0x400b000	0x400b511c
0x400affc	0x00000012

Address	Content
0x400b028	0x00000012
0x400b024	0x400b621c
0x400b020	0x400b620c
0x400b01c	0x00000012
0x400b018	0x400b512c
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	0x00000011
0x400b008	0x0000000a
0x400b004	0x0000000a
0x400b000	0x400b511c
0x400affc	0x0000000b