Midterm Sample Answer

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

Problem number	Maximum Score	Your Score
1	6	
2	20	
3	16	
4	30	
5	10	
6	18	
total	100	

Student Number: Name:

Problem 1. Basic Code Optimization Facts. (6 Points)

(a) (3 points) Name two code optimization blockers and briefly explain what they are.
Answer: memory aliasing, procedure side-effects.

Memory aliasing: two different memory references point to the same memory location. The compiler must assume that different pointers may designate a single place in memory.

Procedure side-effects: the function modifies some part of the global program state. Thus, for example, changing the number of times a function gets called may change the program behavior. The compiler has to assume the worst case and cannot optimize code containing function calls.

(b) (3 points) A sorting algorithm takes 1 second to execute. What advantages and disadvantages has using gprof as a profiling tool to find out the bottleneck of this algorithm ?
Answer: Gprof's sampling period is too coarse grained (10 ms). Since the execution time of this sorting algorithm is 1 second, gprof cannot provide sufficient profiling accuracy in this case (e.g., if this piece of code contains function calls, gprof may skip the entire execution of some of these functions during profiling). Gprof is simple to use and supported on all platforms, but does not provide the ability to insert arbitrary code in arbitrary places in the program, e.g., instruction counts, like other tools, hence its accuracy is low.

Problem 2. Performance Optimization. (20 Points)

The following problem concerns optimizing codes for maximum performance on an Intel Pentium III. Recall the following performance characteristics of the 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

Part A

Consider the following code segments:

Loop 1	Loop 2
`for (i = 0; i < n; i++)` `  x = y * a[i];`	`for (i = 0; i < n; i++)` `  x = x * a[i];`

When compiled with GCC, we obtain the following assembly code for the loop:

Loop 1	Loop 2

`.L21:`
`  movl %ecx,%eax`
`  imull (%esi,%edx,4),%eax`
`  incl %edx`
`  cmpl %ebx,%edx`
`  jl .L21`
	`.L27:`
	`  imull (%esi,%edx,4),%eax`
	`  incl %edx`
	`  cmpl %ebx,%edx`
	`  jl .L27`

Running on one of the cluster machines, we find that Loop 1 requires 3.0 clock cycles per iteration, while Loop 2 requires 4.0.

(a) (4 points) Explain how it is that Loop 1 is faster than Loop 2, even though it has one more instruction. Answer: Loop1 does not have any loop-carried dependence. It can therefore make better use of pipelining in the functional units.

(b) (4 points) We perform 4-way loop unrolling for the two loops. This speeds up Loop 1. Briefly explain why.
Answer: Loop unrolling reduces the loop overhead (incl, cmpl and jl) for each iteration. In other words, the loop overhead is amortized by combining several loop iterations in one.

(c) (4 points) Even with loop unrolling, we find that the performance of Loop 2 remains the same. Briefly explain why.
Answer: Performance is still limited by the latency of the integer multiply (4 cycles).

Part B

You've just joined a programming team that is trying to develop the world's fastest factorial routine. Starting with recursive factorial, they've converted the code to use iterations as follows:

```
int fact(int n) {
  int i;
  int result = 1;

  for (i = n; i > 0; i--)
    result = result * i;

  return result;
}
```

By doing so, they have reduced the number of cycles per element (CPE) for the function from around 63 to around 4 (really!). Still, they would like to do better.

One of the programmers heard about loop unrolling. He generated the following code:

```
int fact_u2(int n) {
  int i;
  int result = 1;

  for (i = n; i > 1; i-=2) {
    result = (result * i) * (i-1);
  }

  return result;
}
```

(a) (4 points) However, benchmarking `fact_u2` shows no improvement in performance. How would you explain that?

Answer: Performance is limited by the 4 cycle latency of integer multiplication.

(b) (4 points) You modify the line inside the loop to read:

```
result = result * (i * (i-1));
```

To everyone's astonishment, the measured performance now has a CPE of 2.5. How do you explain this performance improvement?

Answer: The multiplication i * (i-1) can overlap with the multiplication of the results from the previous iteration (can draw a picture similar to what we showed in class for x = x * (data[i] * data[i+1])).

Problem 3. Cache Miss Rate. (16 Points)

You are writing a new 3D game that you hope will earn you fame and fortune. You are currently working on a function to blank the screen buffer before drawing the next frame. The screen you are working with is a 640x480 array of pixels. The machine you are working on has a 64 KB direct mapped cache with 4 byte lines. The C structures you are using are:

```
struct pixel {
    char r;
    char g;
    char b;
    char a;
};

struct pixel buffer[480][640];
register int i, j;
register char *cptr;
register int *iptr;
```

Assume:

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

A. (4 points) What percentage of the writes in the following code will miss in the cache?

```
for (j=0; j < 640; j++) {
    for (i=0; i < 480; i++){
        buffer[i][j].r = 0;
        buffer[i][j].g = 0;
        buffer[i][j].b = 0;
        buffer[i][j].a = 0;
    }
}
```

Miss rate for writes to `buffer`: _____ %
Answer: 25

B. (4 points) What percentage of the writes in the following code will miss in the cache?

```
char *cptr;
cptr = (char *) buffer;
for (; cptr < (((char *) buffer) + 640 * 480 * 4); cptr++)
    *cptr = 0;
```

Miss rate for writes to `buffer`: _____ %
Answer: 25

C. (4 points) What percentage of the writes in the following code will miss in the cache?

```
int *iptr;
iptr = (int *) buffer;
for (; iptr < (buffer + 640 * 480); iptr++)
    *iptr = 0;
```

Miss rate for writes to `buffer`: _____ %
Answer: 100

D. (4 points) Which code (A, B, or C) should be the fastest? _____
Answer: C

Problem 4 . Cache Conflict Misses. (12 Points)

Consider the following matrix transpose routine

```
typedef int array[2][2];

void transpose(array dst, array src) {
  int i, j;

  for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
      dst[j][i] = src[i][j];
    }
  }
}
```

running on a hypothetical machine with the following properties:

- `sizeof(int) == 4`.
- The `src` array starts at address 0 and the `dst` array starts at address 16 (decimal).
- There is a single L1 cache that is direct mapped with a cache line size of 8 bytes. The address mapping to the cache line is as follows: cache line number = (address % cache size) / block size.
- Accesses to the `src` and `dst` arrays are the only sources of read and write misses, respectively.

Part A (6 points)

Suppose the cache has a total size of 16 data bytes (i.e., the block size times the number of sets is 16 bytes) and that the cache is initially empty. Then for each `row` and `col`, indicate whether each access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.

`dst` array	
dst[0][0]	m
dst[1][0]	
dst[0][1]	
dst[1][1]	

`src` array	
src[0][0]	m
src[0][1]	
src[1][0]	
src[1][1]	

Answer
dst
m m
m m

src
m m
m h

Part B (6 points)

Repeat part A for a cache with a total size of 32 data bytes.

`dst` array	
dst[0][0]	m
dst[1][0]	
dst[0][1]	
dst[1][1]	

`src` array	
src[0][0]	m
src[0][1]	
src[1][0]	
src[1][1]	

Answer

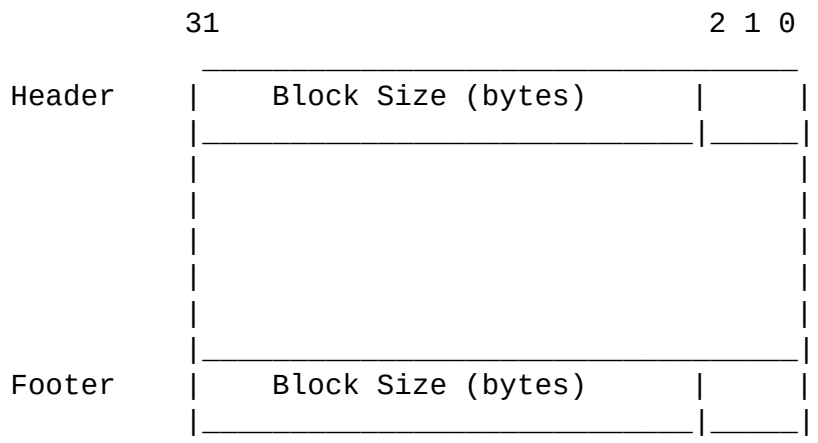
dst
m h
m h

src
m h
m h

Problem 4. Dynamic Memory Allocation. (30 Points)

Part A. (12 points)

Consider a memory allocator that uses an implicit free list. The layout of each allocated and 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. 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.

Note: The header and footer will always be present regardless of whether the block is allocated or not.

Student Number: Name:

Given the contents of the heap shown on the left in hex values (each hex digit translates into 4 bits e.g., 0x3 means 0011, 0xc means 1100), show the new contents of the heap (in the right table) after a call to `free(0x400b010)` is executed. Your answers should be given as hex values. Note that the address grows from bottom up. Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed. Please also briefly explain your answer.

Address	Content
0x400b028	0x00000012
0x400b024	0x400b611c
0x400b020	0x400b512c
0x400b01c	0x00000012
0x400b018	0x00000013
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	0x00000013
0x400b008	0x00000013
0x400b004	0x400b601c
0x400b000	0x400b511c
0x400affc	0x00000013

Address	Content
0x400b028	
0x400b024	0x400b611c
0x400b020	0x400b512c
0x400b01c	
0x400b018	
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	
0x400b008	
0x400b004	0x400b601c
0x400b000	0x400b511c
0x400affc	

Answer:
Note that the “free” call is given a pointer to the payload of the block (the same pointer returned by the corresponding “malloc” call which allocated this block). Hence, for free(0x400b010), the first task is to locate the header and footer of the corresponding block. Then, we need to look at the adjacent blocks (previous and next) to check whether they are free, hence if coalescing can occur. Note that the previous adjacent block is below the current block in the figure (because addresses are growing bottom up). After free(0x400b010), the freed block (0x400b00c-0x400b018) is merged (coalesced) with the adjacent free block (0x400b01c-0x400b028), and the new header/footer is 0x00000022. This hex representation shows that the size of this new block is 32 bytes and the fact that its previous adjacent block is used.

Student Number: Name:

Address	Content
0x400b028	0x00000012
0x400b024	0x400b611c
0x400b020	0x400b512c
0x400b01c	0x00000012
0x400b018	0x00000013
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	0x00000013
0x400b008	0x00000013
0x400b004	0x400b601c
0x400b000	0x400b511c
0x400affc	0x00000013

Address	Content
0x400b028	0x00000022
0x400b024	0x400b611c
0x400b020	0x400b512c
0x400b01c	0x00000012
0x400b018	0x00000013
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	0x00000022
0x400b008	0x00000013
0x400b004	0x400b601c
0x400b000	0x400b511c
0x400affc	0x00000013

Part B. (18 points)

Assume that you want to extend the previous implicit allocator to improve its performance. You would like to reduce allocation time by maintaining an explicit doubly-linked free list. In addition, you would like to improve its memory utilization by using the footer **only when a block is free**. You decide that a first-fit search algorithm is sufficient. You may assume that: `sizeof(void *) is 4`.

(a) (4 points) Given that the block size must be a multiple of 8 bytes, what is the minimum block size allowable under this scheme?
Answer: 16 bytes (4 for header, 4 for footer, 4 x 2 for next and prev pointers)

(b) (4 points) Given that the block size must be a multiple of 8 bytes, determine the amount of memory (in bytes), wasted due to internal fragmentation, after the following four allocation requests. Do not include the 4 bytes used for block headers in your count.

malloc(1)
Answer: 4 for header, 1 for data, 11 for padding
malloc(5)
Answer: 4 for header, 5 for data, 7 for padding
malloc(12)
Answer: 4 for header, 12 for data, no padding
malloc(13)
Answer: 4 for header, 13 for data, 7 padding
Internal fragmentation: 25 bytes (11 for malloc(1) + 7 for malloc(5) + 0 for malloc(12) + 7 for malloc(13))

In order to further improve the performance of your allocator, you decide to try to implement an explicit binary tree data structure to enable a fast best-fit search through the free blocks. Each free block within the tree must now maintain a pointer to each of its children, and to its parent.

(c) (4 points) Assuming that the block size must still be a multiple of 8 bytes, what is the minimum block size allowable under this new scheme?
Answer: 24 bytes (4 for header, 4 for footer, 4 x 3 for pointers + 4 for padding)

(d) (6 points) Comment on the effect that this allocator has on memory utilization when compared to the previous explicit linked list first-fit allocator. You should discuss any opposing tensions (trade-offs) that might exist.
Answer: The effect on memory utilization will depend on the allocation requests. This allocator will, on average, reduce the amount of external fragmentation because of the best-fit replacement policy; however, it suffers from higher internal fragmentation for small (less than 20 bytes) memory allocation requests.

Student Number: Name:

Problem 5. Profiling and Speedup. (10 Points)

Part A (5 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]
.....................
```

Answer:
Here is a basic explanation of the profile data. From the flat profile, we see that 96.33% of the application running time is spent in the function World::update. This translates to 1076.18 seconds. 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 we go over the time spent in each child and 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 the code for this function itself 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 speedup is 5.