1.

What is loop unrolling? How can it make your program more efficient? How can it make your program less efficient?

Loop unrolling is multiple iterations of a loop at once. It makes program more efficient by taking advantage of parallelism. But it makes program less efficient by using more variables, possibly causing registers running out and producing cache misses.

2.

What affects the data stored on the stack? Think registers and instructions!

Pop, push, ret, call instructions on %rsp.

3.

The following table gives the parameters for a different number of caches. For each cache, fill in the missing fields in the table.

* *m* is the number of physical address bits
* *C* is the cache size
* *B* is the block size
* *E* is the associativity
* *S* is the number of the cache sets
* *t* is the number of tag bits
* *s* is the number of set index bits
* *b* is the number of block offset bits

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Cache | *m* | *C* | *B* | *E* | *S* | *t* | *s* | *b* |
| 1 | 32 | 1024 | 4 | 4 |  |  |  |  |
| 2 | 32 | 2048 |  |  | 128 | 23 | 7 |  |
| 3 | 32 | 1024 | 8 | 1 |  |  |  |  |
| 4 | 32 | 1024 | 8 | 128 |  |  |  |  |
| 5 | 32 | 1024 |  |  |  | 25 | 4 | 3 |
| 6 | 32 | 1024 | 32 | 4 |  |  |  |  |

4.

The provided function func\_one takes as input two pointers, that are actually each individually pointing to the first element in a N by M array of integers.

int func\_one(char\* one, char\* two, int N, int M) {

int i, j, k;

int sum = 0;

char\* ptr1 = one;

char\* ptr2 = two;

for (k = 0; k < 4; k++) {

for (j = 0; j < M; j++) {

for (i = 0; i < N; i++) {

char one = \*(ptr1 + k + j\*4 + i\*4\*M);

int masked = one & 0xFF;

int shift = k << 3;

int shifted = masked << shift;

\*(ptr2 + k + j\*4 + i\*4\*M) = masked;

sum += shifted;

}

}

}

return sum;

}

In what ways can we optimize the above function?

Loop unrolling

5.

The provided code below is an optimization of the previous problem. Fill in the blanks.

int func\_two(char\* one, char\* two, int N, int M) {

int i, j, k;

int sum = 0;

int temp = 0;

char\* ptr1 = one;

char\* ptr2 = two;

for (i = 0; i < N; i++) {

for (j = 0; j < M; j++) {

temp = (0xFF & \*ptr1);

sum += temp

\*ptr2 = temp

ptr1++;

ptr2++;

temp = 0xFF & \*ptr1

sum += temp << 8

\*ptr2 = temp

ptr1++;

ptr2++;

temp = 0xFF & \*ptr1

sum += temp << 16

\*ptr2 = temp

ptr1++;

ptr2++;

temp = 0xFF & \*ptr1

sum += temp << 24

\*ptr2 = temp

ptr1++;

ptr2++;

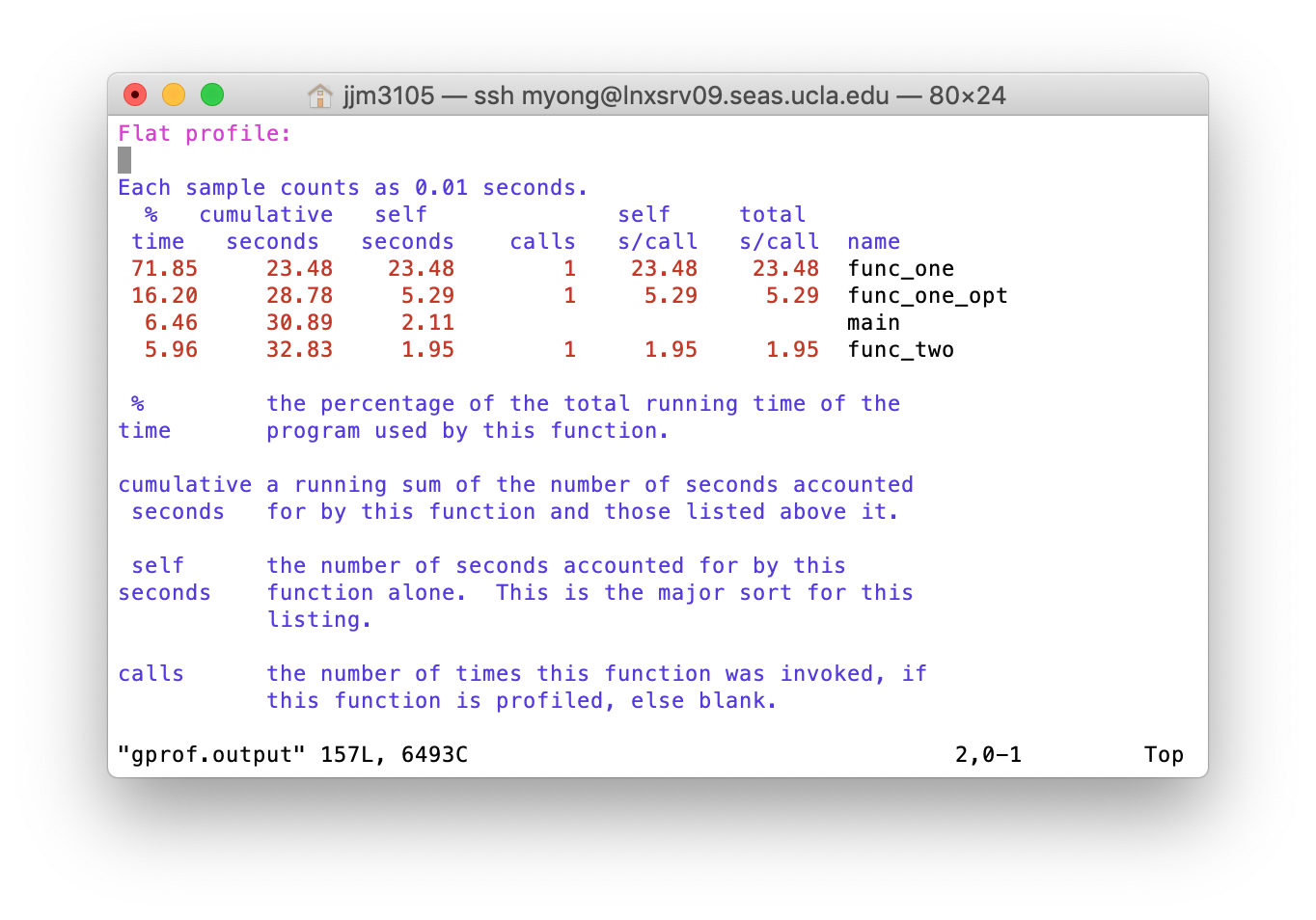
}

}

return sum;

}

FYI.



The above shows the running time of the functions discussed in problems 5 and 6. func\_one is the code from problem 5 as is, func\_one\_opt is one optimization of func\_one, and func\_two is the completed code from problem 6.

The results were generated using gprof

**OPTIONAL**

6.

Assume the following:

* The memory is byte addressable.
* Memory accesses are to 1-byte words (not to 4-byte words).
* Addresses are 13 bits wide.
* The cache is two-way set associative (E = 2), with a 4-byte block size (B = 4) and eight sets (S = 8).

The contents of the cache are as follows, with all numbers given in hexadecimal notation.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Line 0 | | | | | | Line 1 | | | | | |
| Set Index | Tag | Valid | Byte0 | Byte1 | Byte2 | Byte3 | Tag | Valid | Byte0 | Byte1 | Byte2 | Byte3 |
| 0 | 09 | 1 | 86 | 30 | 3F | 10 | 00 | 0 |  |  |  |  |
| 1 | 45 | 1 | 60 | 4F | E0 | 23 | 38 | 1 | 00 | BC | 0B | 37 |
| 2 | EB | 0 |  |  |  |  | 0B | 0 |  |  |  |  |
| 3 | 06 | 0 |  |  |  |  | 32 | 1 | 12 | 08 | 7B | AD |
| 4 | C7 | 1 | 06 | 78 | 07 | C5 | 05 | 1 | 40 | 67 | C2 | 3B |
| 5 | 71 | 1 | 0B | DE | 18 | 4B | 6E | 0 |  |  |  |  |
| 6 | 91 | 1 | A0 | B7 | 26 | 2D | F0 | 0 |  |  |  |  |
| 7 | 46 | 0 |  |  |  |  | DE | 1 | 12 | C0 | 88 | 37 |

Suppose a program running on a machine with such a cache references the 1 byte word at the address 0x0E34.

What is the resulting of the following?

Cache block offset:

Cache set index:

Cache tag:

Cache hit?:

Byte returned: