CS33 Homework Assignment 6 Solutions

Fall 2019

1. Consider the following function:

```
void minmax1(long *a, long *b, long n) {
  long i;
  for (i=0; i<n; i++) {
    if (a[i] > b[i]) {
     long t = a[i];
     a[i] = b[i];
     b[i] = t;
   }
  }
}
```

It compares corresponding elements of its input arrays *a* and *b*, and puts into *a* the smaller elements and into *b* the larger elements. This code is provided to you in minmax.c, along with some code that runs it through various timed tests. The function *minmax1* performs much worse for random data than it does for predictable data. This has to do with the processor's ability to do branch prediction. For random data, prediction is not possible. In this problem we compute what the penalty is for branch misprediction on the Sunlab computers. You should examine the code in the file carefully — part of the exercise is figuring out what it does. Rather than expressing time in clock cycles as is done in the slides and in the textbook, we express time in nanoseconds. So as to get reasonably accurate results, we use arrays of size 2^{22} (roughly 4 million). You will not need to modify the code, but you should compile it using gcc's -O2 flag so as to turn on a fair amount of optimization, i.e.,

```
gcc -o minmax minmax.c -O2
```

The program prints out various figures for time/iteration for calls to *minmax1* using different arguments. For parts a and b, you are to explain what these values refer to.

a. Which value printed by the program is the time per iteration of the loop if the data is such that the branch is never taken? Which value corresponds to the case in which the data is such that the branch is always taken? Explain.

   Answer: The time printed out for "small/large" is the time per iteration for the case in which the branch is never taken. The time printed out for "large/small" is the time per iteration in which the branch is always taken. Note that, in both cases, branch prediction can be assumed to be perfect — while it might take a couple iterations for the processor to determine that the branch is never/always being taken, for the bulk of the four million iterations, it predicts the branch correctly.

b. Which value printed by the program is the time per iteration of the loop if it's completely random whether the branch is taken? Explain.

   Answer: This is the case printed out for "random". Note that the processor is predicting the branch result correctly half the time.

c. What is the penalty, in nanoseconds, for a branch misprediction? (Assume the penalty is constant, regardless of whether the misprediction is for a taken branch or a non-taken branch. Also assume that, in the random case, half the time the branch is taken, half the time the branch is not taken, and misprediction occurs half the time.) Explain.

   Answer: To determine the penalty, we look at the random case, in which misprediction occurs half the time. Thus if T_r is the time taken per iteration for the random case, T_t is the time taken per iteration if the branch is taken and correctly predicted, T_n is the time taken per iteration if the branch is not taken and correctly predicted, and P is the misprediction penalty, then

   $$T_r = .5*P + .5*T_n + .5*T_t$$

   and thus

   $$P = 2*(T_r - .5*(T_n+T_t))$$

   Given the values $T_r = 3.894$ nsecs, $T_t = 2.117$ nsecs, and $T_n = 1.493$ nsecs, this computes P to be 4.167 nsecs.

d. Look at the assembler code produced by gcc (use the –S flag to get gcc to produce a *.s* file containing assembler code). The compiler made a structural change to the program to speed things up a bit, what did it do? (Hint: look at the compiled code for *main*.)

   Answer: The compiler has "inlined" the function *minmax1*, i.e., rather than placing calls to it within *main*, the code was copied into *main* and executed directly.