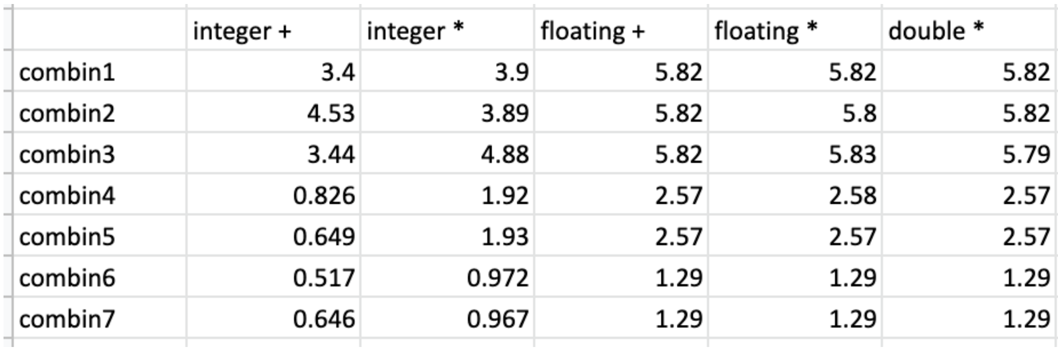
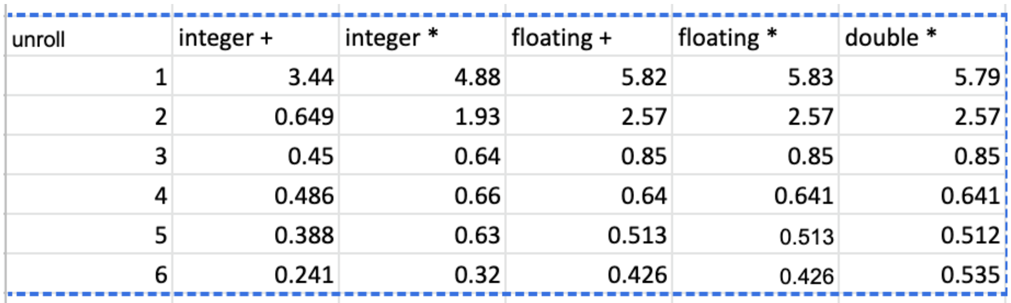
EC527
Lab2 report
Bin Xu
Seyed Reza Sajjadinasab

Part 1

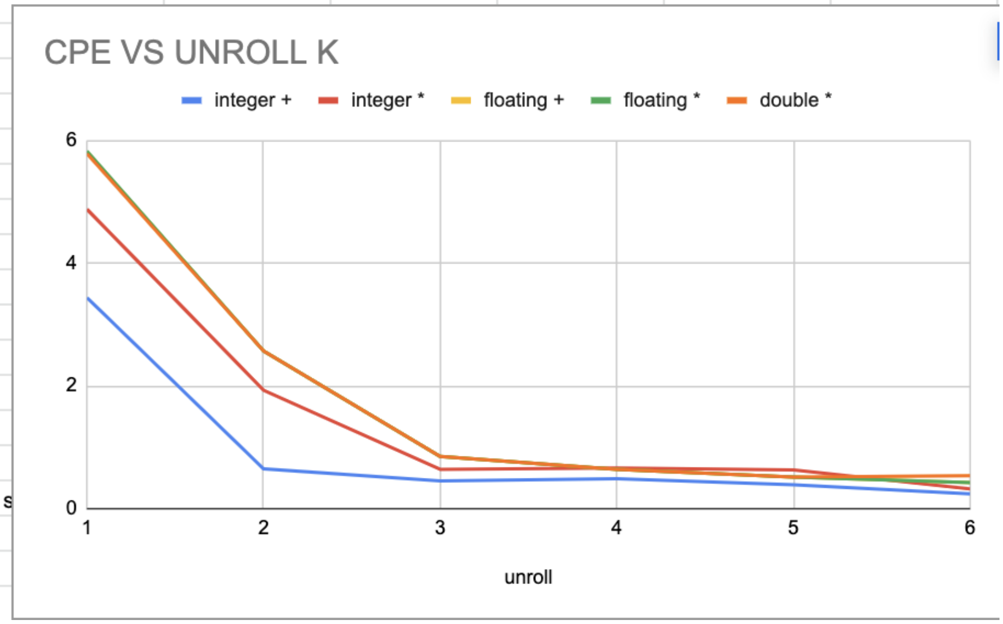
1a.



	integer +	integer *	floating +	floating *	double *
combin1	3.4	3.9	5.82	5.82	5.82
combin2	4.53	3.89	5.82	5.8	5.82
combin3	3.44	4.88	5.82	5.83	5.79
combin4	0.826	1.92	2.57	2.58	2.57
combin5	0.649	1.93	2.57	2.57	2.57
combin6	0.517	0.972	1.29	1.29	1.29
combin7	0.646	0.967	1.29	1.29	1.29



unroll	integer +	integer *	floating +	floating *	double *
1	3.44	4.88	5.82	5.83	5.79
2	0.649	1.93	2.57	2.57	2.57
3	0.45	0.64	0.85	0.85	0.85
4	0.486	0.66	0.64	0.641	0.641
5	0.388	0.63	0.513	0.513	0.512
6	0.241	0.32	0.426	0.426	0.535

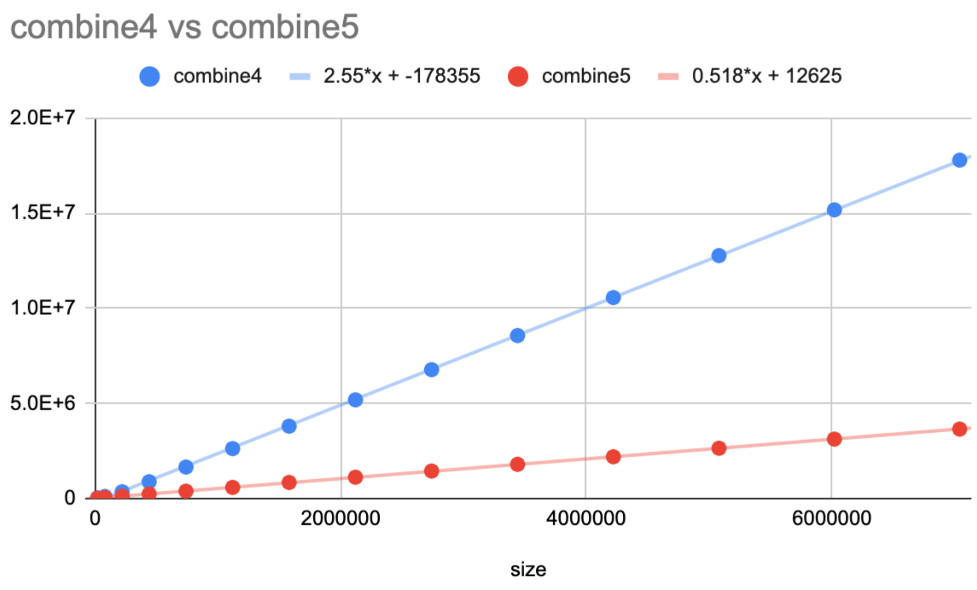


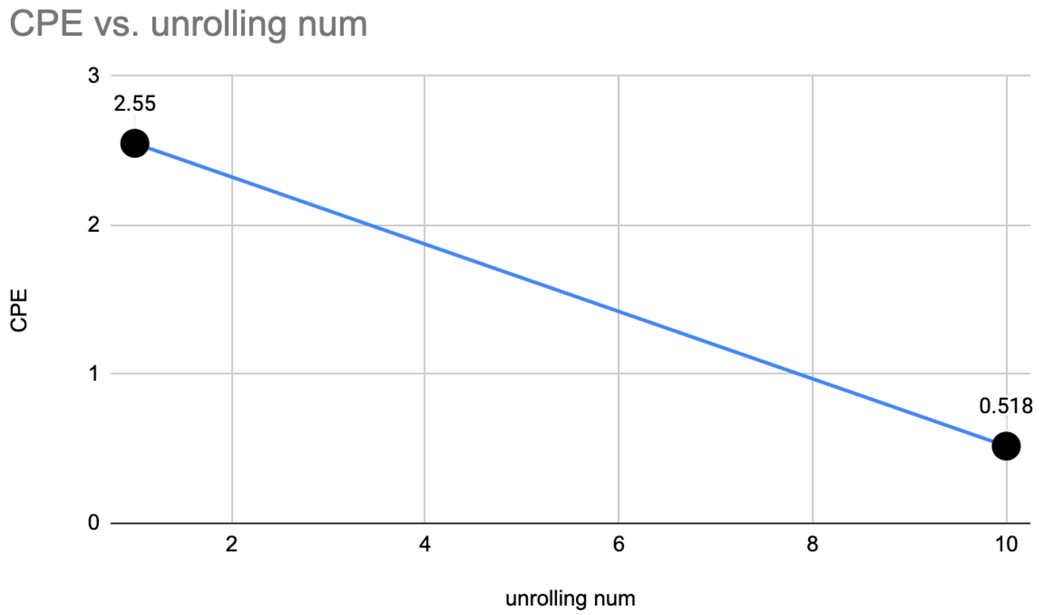
The overall trend of the graph is the same as the graph on the book. The CPE number has some differences as the CPE in the book. Because we use different CPU and GCC compiler.

1b.

Code in combine 5

```
for (i = 0; i < limit; i+=10) {
  //acc = (acc OP data[i]) OP data[i+1];
  acc = (acc OP (data[i] OP (data[i + 1] OP (data[i + 2] OP (data[i + 3] OP (data[i + 4] OP (data[i + 5]OP (data[i + 6]OP (data[i + 7]OP (data[i + 8])) )) )))))) OP data[i+9];
}
```





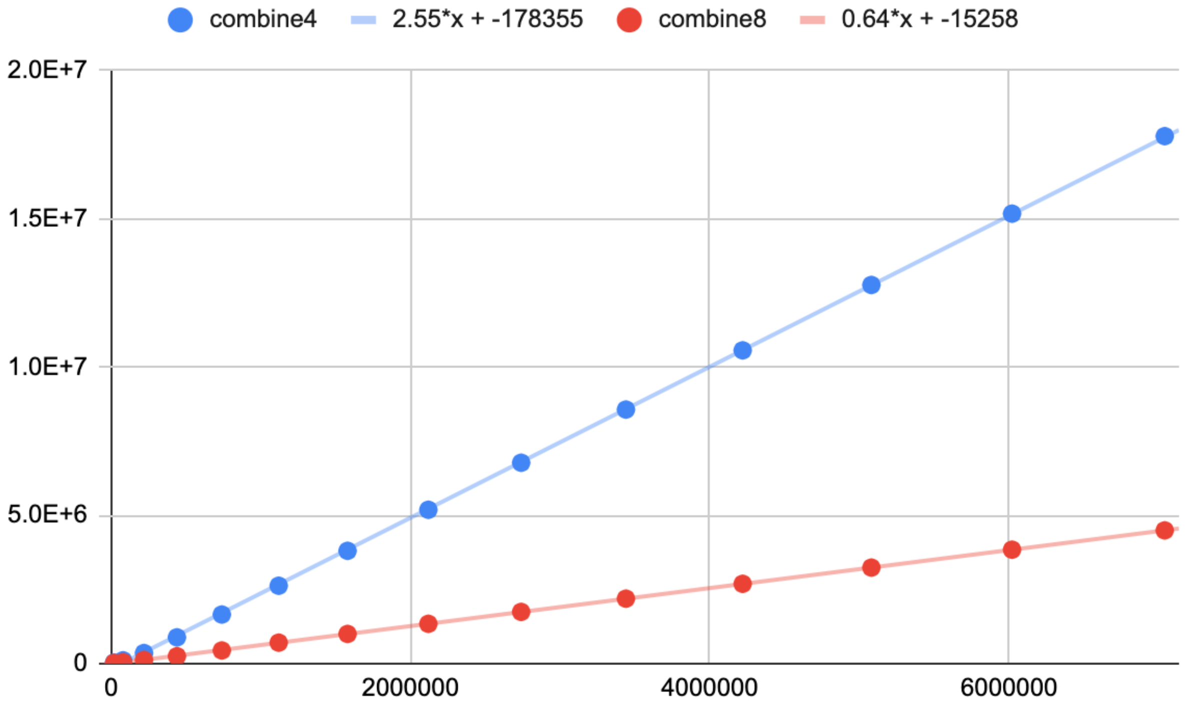
As we can see from the above chart, combine5 with 10 unrolling loops get much better performance than combine4 with 1 unroll loop. Unrolling a loop 10 times can increase instruction-level parallelism by exposing more opportunities for the processor to execute multiple instructions at the same time.

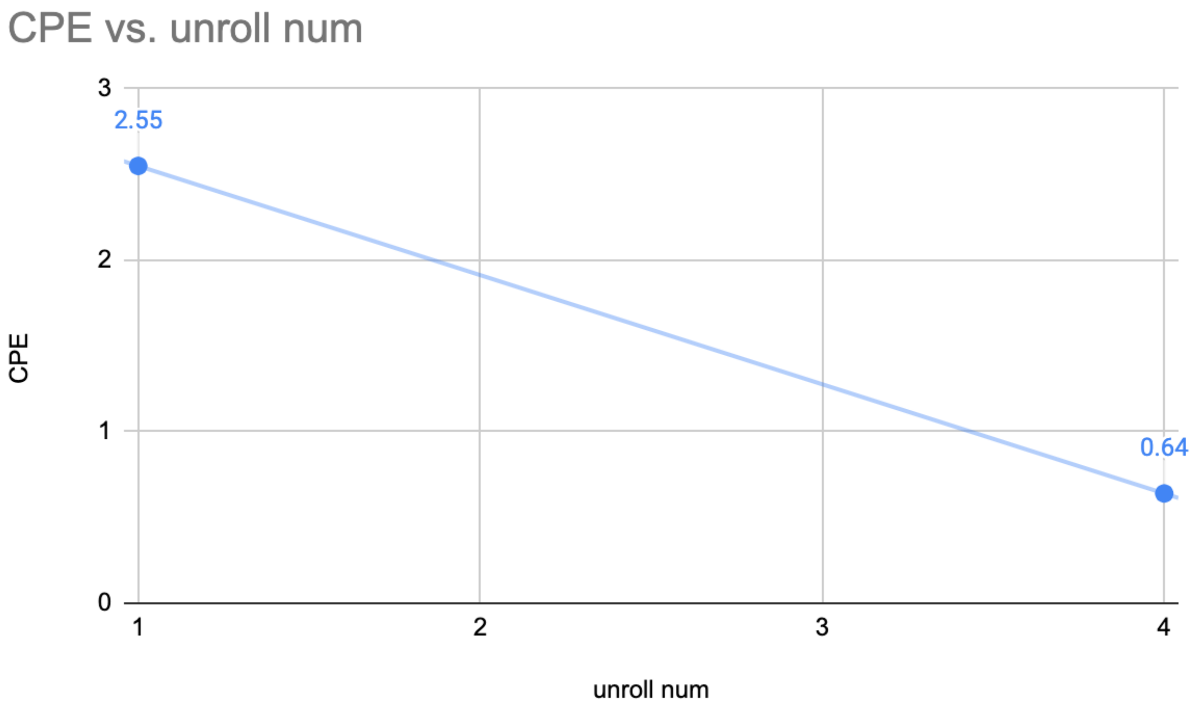
1c.

```
void combine8(array_ptr v, data_t *dest)
{
  long int i;
  long int length = get_array_length(v);
  long int limit = length - 1;
  data_t *data = get_array_start(v);
  data_t acc = IDENT;
  data_t acc0 = IDENT;
  data_t acc1 = IDENT;

  /* Combine two elements at a time */
  for (i = 0; i < limit; i+=4) {
     acc0 = acc0 OP (data[i] OP data[i + 1]);
     acc1 = acc1 OP (data[i+2] OP data[i + 3]);
  }

  /* Finish remaining elements */
  for (; i < length; i++) {
     acc = acc OP data[i];
  }
  acc = acc OP acc0 OP acc1;
  *dest = acc;
}
```

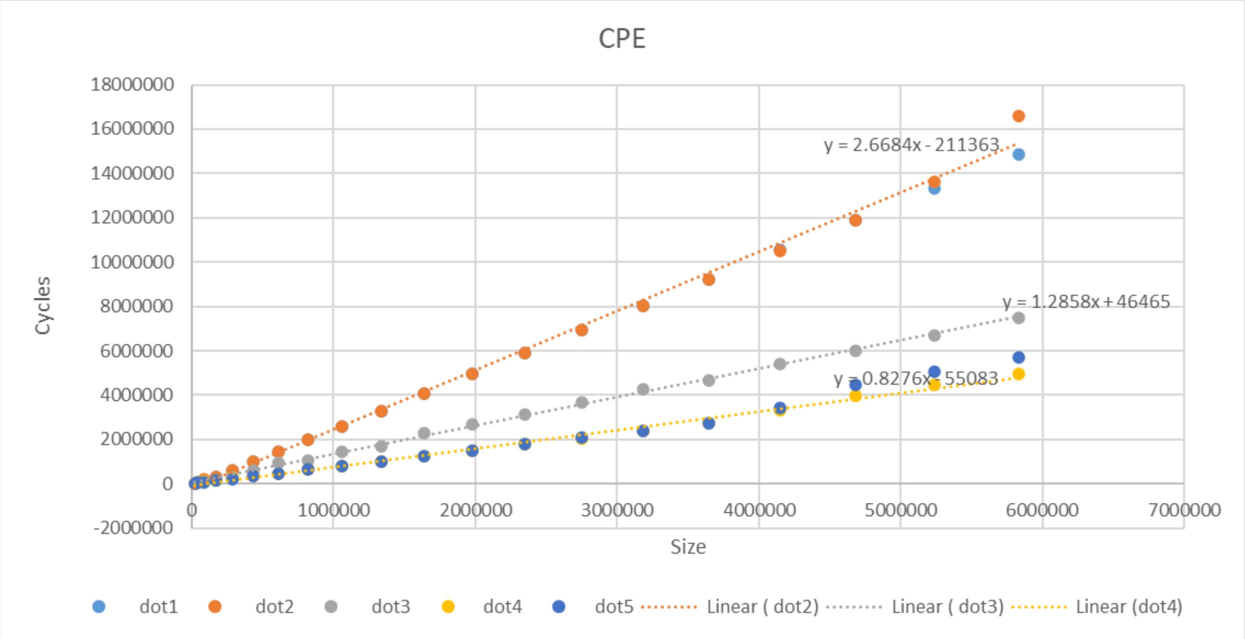




From the above chart, combine8, which is two parallelization methods, is much faster than combine 4.

Part 2

We used the combine4 method as the basis of our dot1 code. Then, based on the combine7 we enhance the result of our code (dot3). Also, we used the ivdep pragma to see if the compiler can enhance any vectorization, but we couldn’t see any significant change. The code having the pragma is in dot2. It can be due to the fact that the compiler is already aware of the fact that it can do the vectorization.
The optimizations that we used for dot3 include both parallelism by unrolling the loop.
Below is the graphical representation.
We also do the parallelism for an unrolling factor of 5 which results in the best CPE of 0.84.

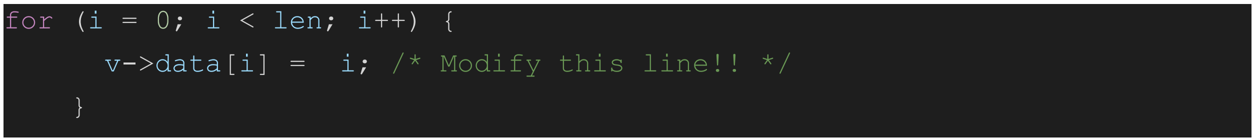


Note: dote5 version is just another version of the dot4 with two accumulators and an unrolling factor of 5 for each(overall 10). As you can see there is no significant difference between them.

Part 3

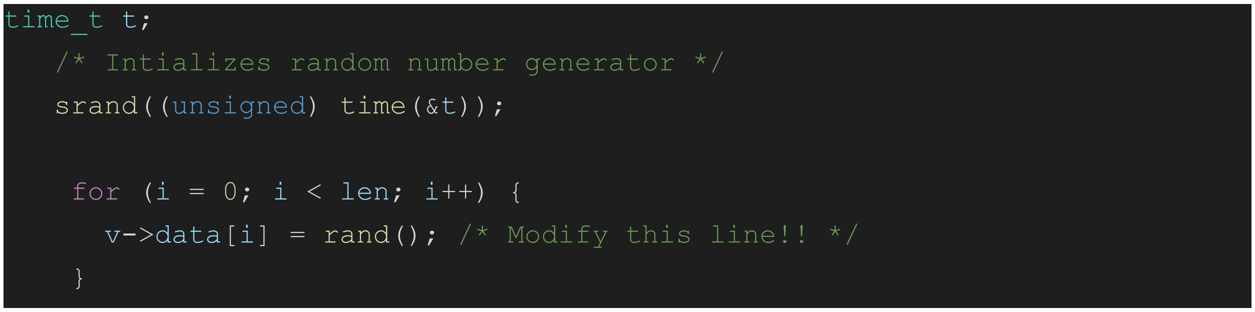
3a.

For generating predictable data we can use just an ascending number. So, the code can be like this



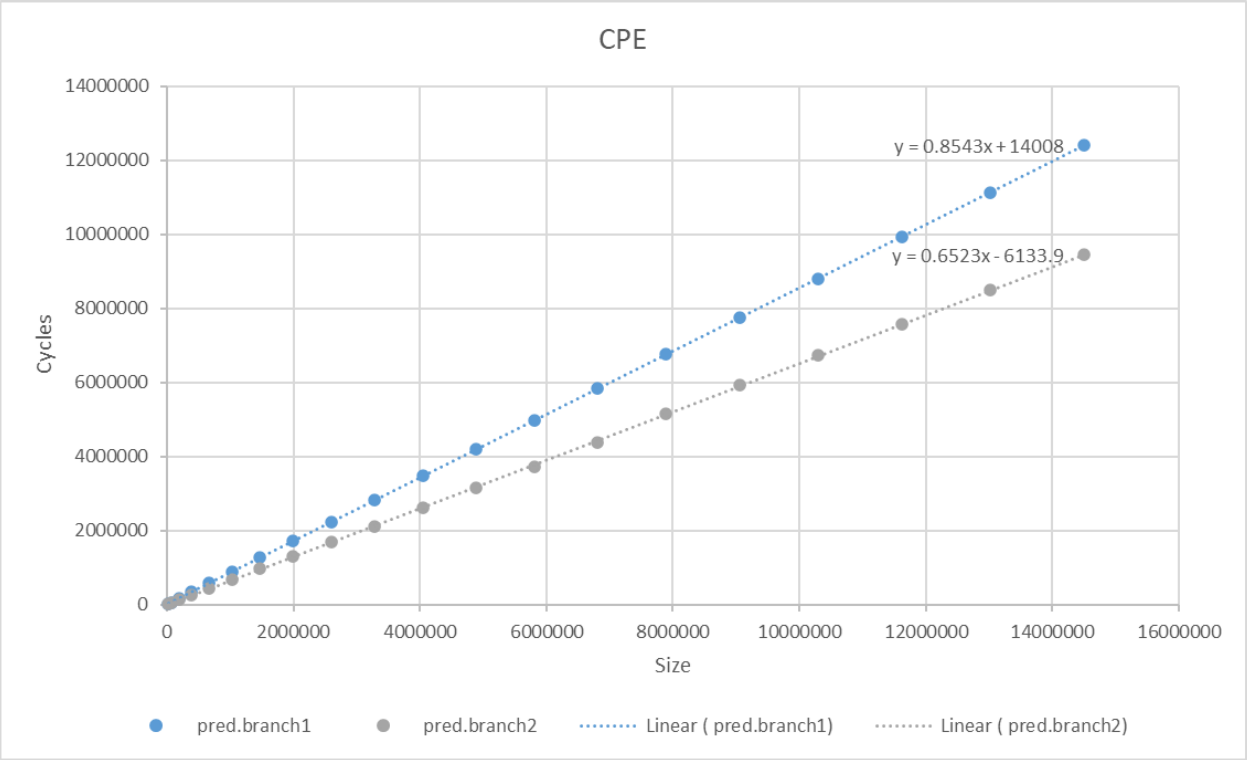
```
for (i = 0; i < len; i++) {
      v->data[i] =  i; /* Modify this line!! */
    }
```

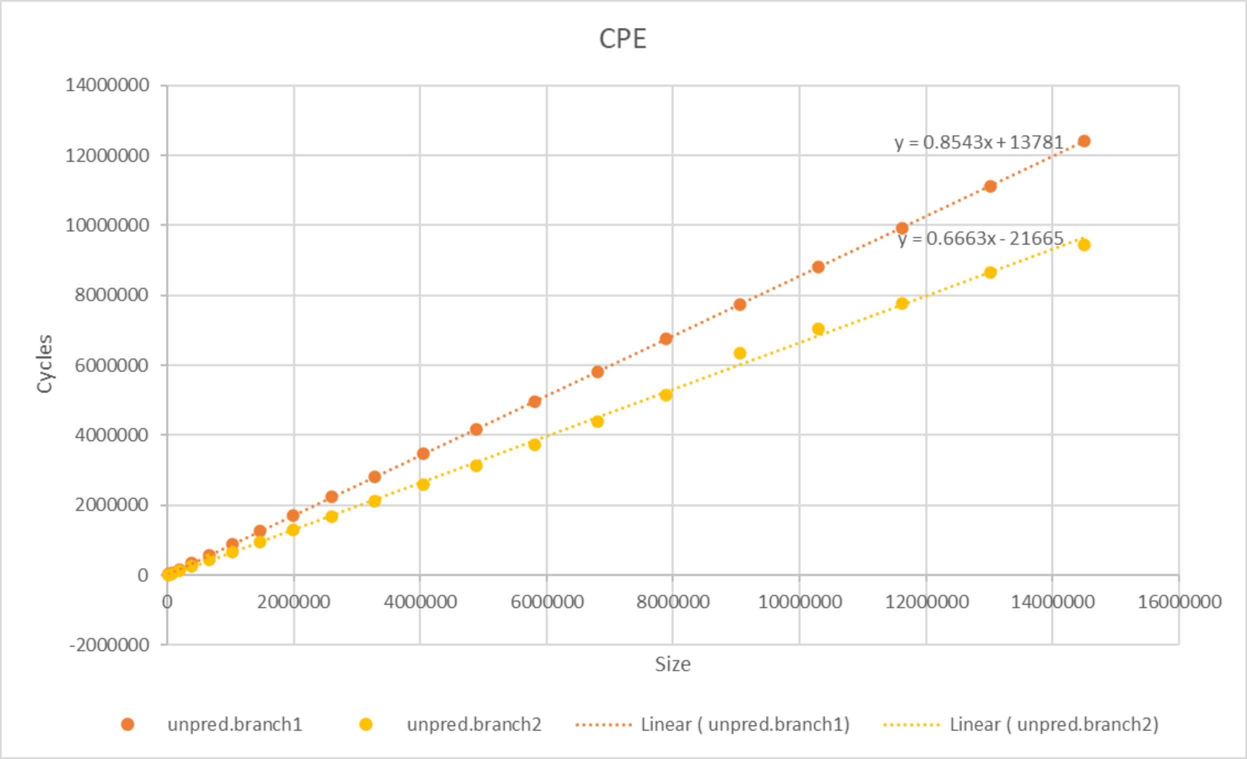
For producing unpredictable data we use the rand() method. So the code can be like this:



```
time_t t;
   /* Intializes random number generator */
   srand((unsigned) time(&t));

    for (i = 0; i < len; i++) {
      v->data[i] = rand(); /* Modify this line!! */
    }
```

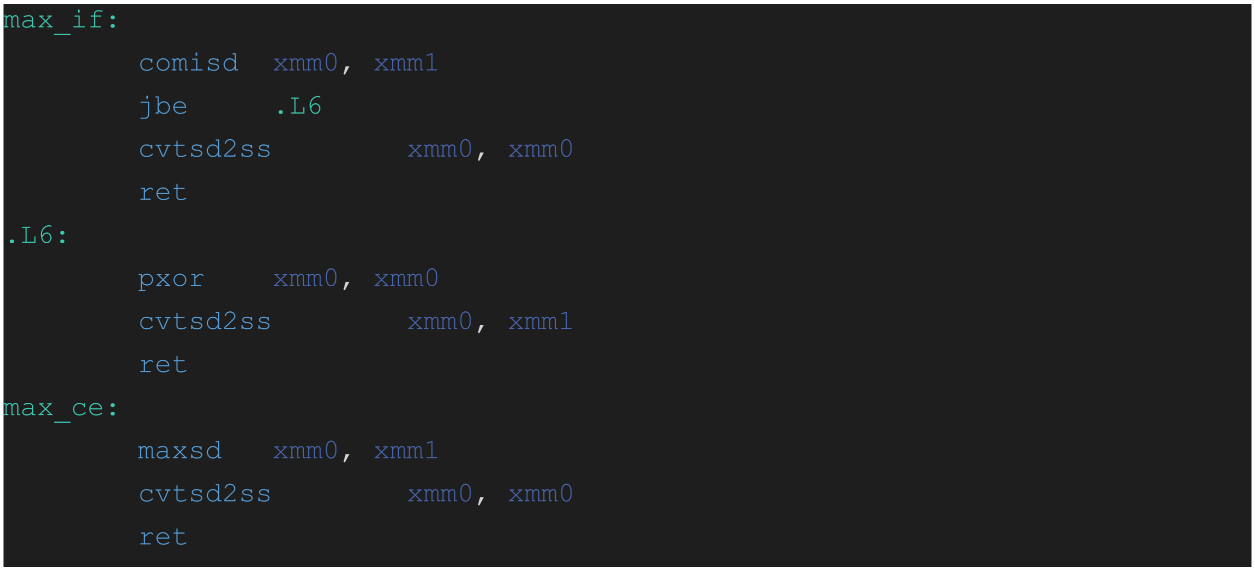




As we can see there is no obvious difference between the predicted version and non predicted one. Anyway, branch2 version is slightly better than the other.

3b.

This is the result for float:

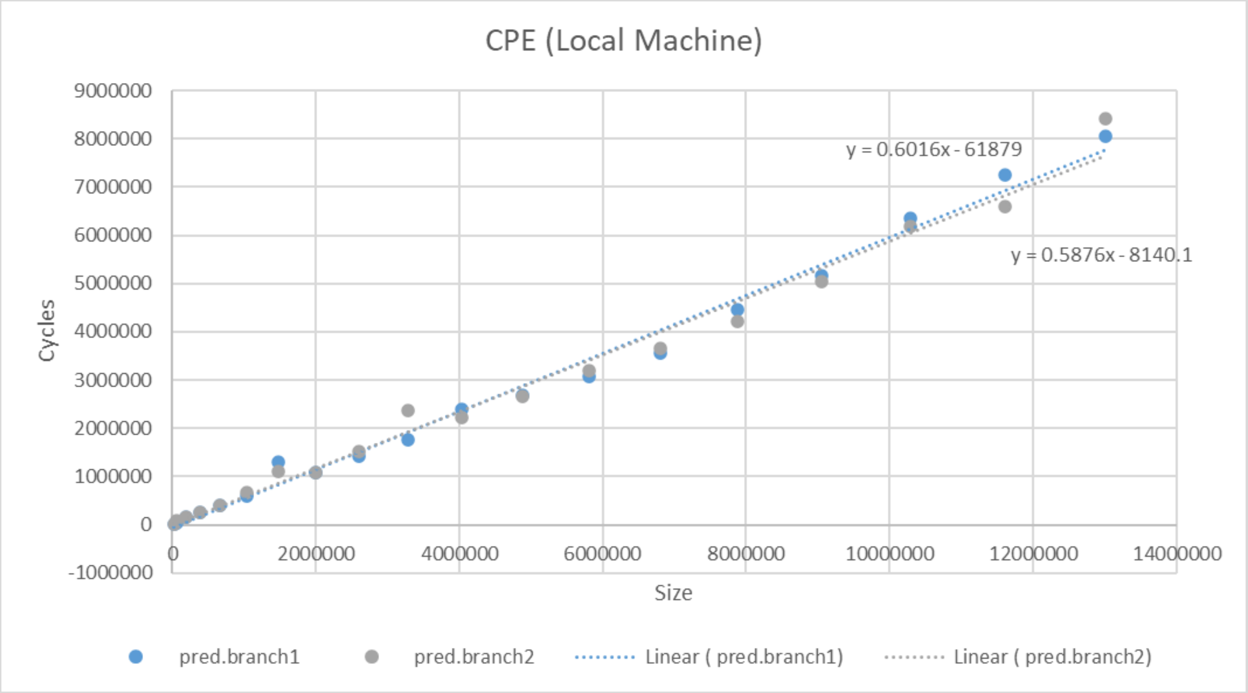


```
max_if:
        comisd  xmm0, xmm1
        jbe     .L6
        cvtsd2ss        xmm0, xmm0
        ret
.L6:
        pxor    xmm0, xmm0
        cvtsd2ss        xmm0, xmm1
        ret
max_ce:
        maxsd   xmm0, xmm1
        cvtsd2ss        xmm0, xmm0
        ret
```

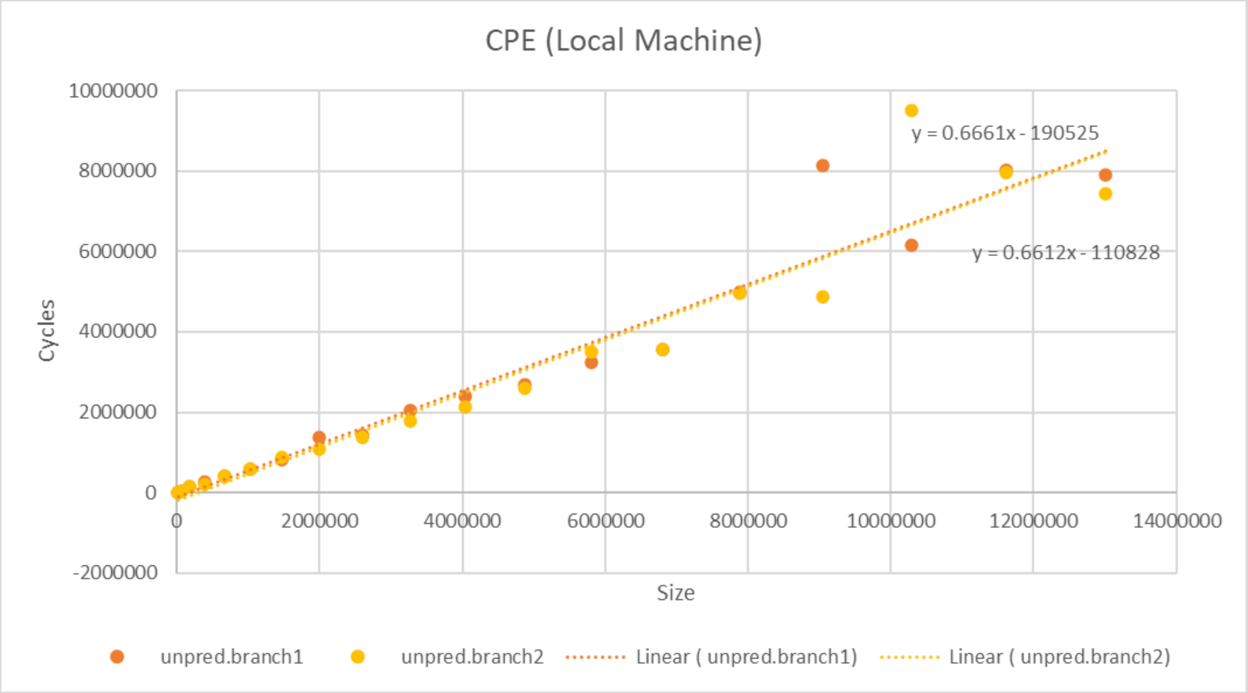
As we expected for max_if there is a jump function, but we don't have it for max_ce.

So, what is happening in the CPE, and why branch2 is better than branch1 is simply because it is a much simpler code than the other one with no need of branch prediction. One behavior that we can see here is that Only because the number of instructions to be performed is decreased the branch2 is better(We no longer need the branch). However, the prediction doesn't make any significant difference.

To see if this is always the case we also check the code locally, and it was the result



As we can see there is no significant difference when we compare locally.



Here, we can see locally there is no difference in all of the data types. This may be because the version of gcc on the local machine was a newer one (GCC 11.3.0), but on the lab machine we had GCC 4.8.5.

Using https://godbolt.org/ we obtained the assembly code for two different versions with GCC 4.8.5 Here you can see the main block that these two have differences in.

Branch1:

```
.L9:
        movss   xmm1, DWORD PTR [rdi+rdx*4]
        movss   xmm0, DWORD PTR [rcx+rdx*4]
        ucomiss xmm1, xmm0
        jbe     .L11
        movss   DWORD PTR [rsi+rdx*4], xmm1
        jmp     .L6
.L11:
        movss   DWORD PTR [rsi+rdx*4], xmm0
```

Branch2:

```
.L8:
        movss   xmm0, DWORD PTR [r8+rdx*4]
        movss   xmm1, DWORD PTR [rdi+rdx*4]
        maxss   xmm0, xmm1
        movss   DWORD PTR [rsi+rdx*4], xmm0
        add     rdx, 1
        cmp     rbx, rdx
        jg      .L8
```

As you can see in branch2 the, first we calculate the maximum in “maxss” instruction, but in Branch1, we use branch instruction to decide what to do based on a comparison made with “ucomiss” instruction.

Part 4

4a. 5
4b. No
4c. No
4d. No