# CMPT 295

Name: Junchen Li
Student ID: 301385486

Date: 2020/4/6

Assignment 9 - out of 30 marks

Objectives:

- Pipelined Execution Analysis – Uniform and non-uniform stage lengths
- Benchmarking "branch reduction' optimization
- Microbenchmarking array search versus linked list search
- "Loop unrolling" optimization

---

Submission:

- Submit your document called **Assignment_9.pdf**, which must include your answers to all of the questions in Assignment 9.
    - Add your full name and student number at the top of the first page of your document **Assignment_9.pdf**.
- When creating your assignment, first include the question itself and its number then include your answer, keeping the questions in its original numerical order.
- **If you hand-write your answers (as opposed to using a computer application to write them):** When putting your assignment together, do not take photos (no .jpg) of your assignment sheets! Scan them instead! Better quality -> easier to read -> easier to mark!
- Submit your assignment **Assignment_9.pdf** electronically on CourSys.

---

Due:

- Thursday, April 9 at 3pm.
- Late assignments will receive a grade of 0, but they will be marked in order to provide the student with feedback.

---

Marking scheme:

This assignment will be marked as follows:

- All questions will be marked for correctness.

The amount of marks for each question is indicated as part of the question.

A solution will be posted after the due date.

1. [5 marks] Pipelined Execution Analysis – Uniform and non-uniform stage lengths

   Part 1 - Following on from our Assn 8 Question 4 a. and b. ...

   Imagine we built a processor. The propagation delay of its entire combinational logic circuit is 600 ps and the propagation delay to load the clocked register we used is 30 ps.

   c. What is the minimum clock cycle time (in picoseconds), the latency (in picoseconds) and throughput (in GIPS) of our processor if it performs pipelined execution using 8 equal stages? Answer: 600/8 = 75 minimum clock cycle = 75+30 = 105 ps
   Latency: (75+30)8 = 840 ps CPU Throughput = 8/840 = 9.52 GIPS

   Part 2 - Now, imagine we built a second processor. For the purpose of this question, we shall called this processor the **original processor**.

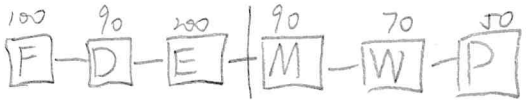
   The propagation delay of its entire combinational logic circuit is 600 ps and the propagation delay to load the clocked register we used is 30 ps.

   However, the entire combinational logic circuit of this processor has been divided into 6 stages:

   - stage F with a propagation delay of 100 ps,
   - stage D with a propagation delay of 90 ps,
   - stage E with a propagation delay of 200 ps,
   - stage M with a propagation delay of 90 ps,
   - stage W with a propagation delay of 70 ps, and
   - stage P with a propagation delay of 50 ps.

   So far, no clocked registers have been placed between any of these 6 stages.

   a. If we were to place one clocked register between a pair of adjacent stages of the original processor in order to form a 2-stage pipeline, where would we place this clocked register? Compute the minimum clock cycle time and the maximum CPU throughput of this processor.




the most take time stage will influence the clock efficeny.

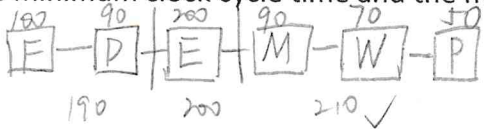
100 ✓ 300 ✓
190 ✓ 410 ✓
390 ✓ 210
480 ✓ 140

clocked register put between FDE and MWP. and form a 2 stage pipeline.

minimum clock cycle: 100+90+200+30 = 420 ps

max CPU throughput 1/420 = 2.38 GIPS

b. If we were to place two clocked registers between the stages of the original processor in order to form a 3-stage pipeline processor, where would we place these two clocked registers? Compute the minimum clock cycle time and the maximum CPU throughput of this processor.




100 290 210
190 200 210 ✓

minimum clock cycle time = 90+70+50+30 = 240 ps

maximum CPU throughput = 1/240 = 4.16 GIPS

c. If we were to place more and more clocked registers between the stages of the original processor in order to form greater staged-pipeline processors, which stage would become a limiting factor on the minimum clock cycle time?

Hint: Form a 4-stage pipeline processor then a 5-stage pipeline processor in order to answer this question.

100 90 | 200 | 90 70 | 50
190 200 ✓ 160 50

100 | 90 | 200 | 90 | 70 50
200 ✓

when we add three clocked registers and form four staged-pipelines the biggest propagation delay 200 limits the clock cycle time and three clocked registers can obstruct it form bigger than 200.

d. If we were to fully pipeline our original processor (create a 6-stage pipeline processor), what would its minimum clock cycle time and its maximum CPU throughput be?

100 | 90 | 200 | 90 | 70 | 50

minimum clock cycle time = 200 + 30 = 230 ps

maximum CPU throughput = 1/230 = 4.34 GIPS

---

2. [10 marks] Benchmarking "branch reduction" optimization (using the benchmarking tool introduced in Lab 5)

In this question, we shall optimize the performance of the *linear search* algorithm, which as its name implied, has a time complexity of O(n).

Download `Assn9-files.zip` and extract its files. Have a look at the files in the `Q2` folder. We see that `main.c` benchmarks a pair of linear searches on a randomized array. The first of these linear search functions is found at the end of `main.c`. As we can see, it is a very straightforward implementation of the linear search algorithm. In this question, we

are asked to implement an alternative linear search and to demonstrate that it performs better (faster) than the first implementation.

a. Even though this second implementation of the linear search will also have a time complexity of O(n), it can be optimized to reduce the cost of each loop. In other words, we are expecting this second implementation of the linear search to produce a smaller CPE (clock cycles – cost - per element).

   A good first attempt at optimizing this function is to replace its expensive operations: the comparisons. A comparison in our C code will generate a corresponding branch in the assembly code. And as we know, a branch has the potential to be mispredicted by the CPU (i.e., the CPU's instruction pipeline). The standard algorithm does two comparisons per loop: the comparison between `A[i]` and `target` and the comparison between `i` and `n`. The number of comparisons can be roughly cut in half using the following algorithm (expressed in pseudocode), which makes use of `target` as a sentinel put in the last position of the array:

```
search(A[n], target)
    if n <= 0 then return -1

    tmp <- A[n-1]
    A[n-1] <- target

    i <- 0
    while A[i] != target do
        i <- i+1

    A[n-1] <- tmp

    if i < n-1 then return i
    else if A[n-1] == target then return n-1
    else return -1
```

   Code this algorithm in `lsearch_2.c`.

   Submit `lsearch_2.c` electronically via CourSys (make sure it is the version that compiles and executes successfully) and add a hardcopy of the same version in this assignment.

```
// Filename: lsearch_2.c
// Description: A more efficient implementation of the linear search.

int lsearch_2(int *A, int n, int target) {

   // Question 2 a.  Put your code below
   if (n<=0)
      return -1;
   int temp=A[n-1];
   A[n-1]=target;
   int i=0;
   while (A[i]!=target)
      i=i+1;
   A[n-1]=temp;
   if (i<n-1)
      return i;
   else if (A[n-1]==target)
      return n-1;
   else
      return -1;

} // lsearch_2
```

b. Benchmark `lsearch_1()` and `lsearch_2()` for `N` = {5000000, 10000000, 15000000, 20000000}, and `NTESTS = 400`. Collect three samples for each `N`, and present your raw data and the average times in a table format and plot the average times on a graph (with labels) as we did in Lab 5.

What conclusion de we draw from our results?

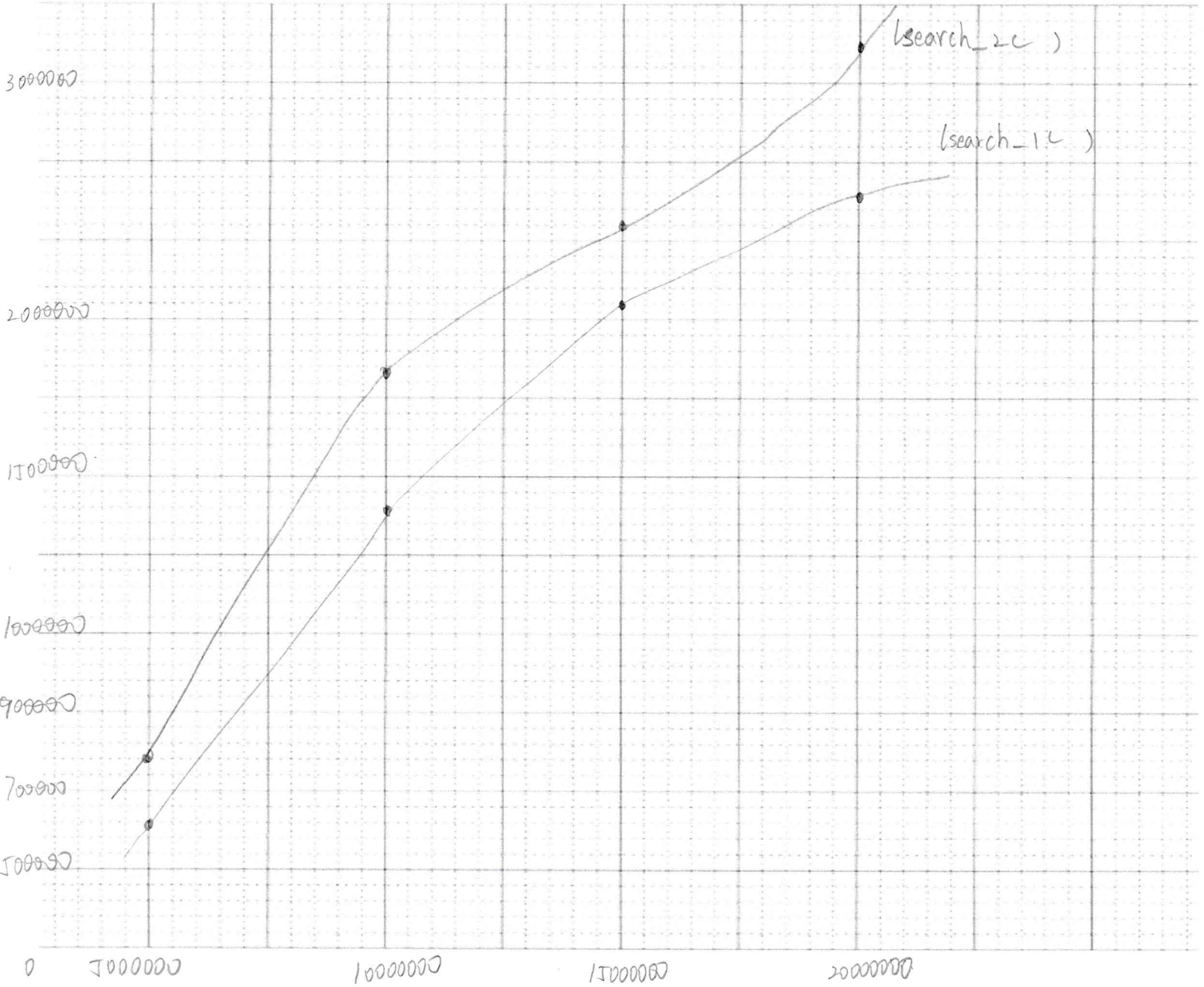
Include all datasets and graph in this assignment.

`lsearch_1()`:

| N | T1 | T2 | T3 | μ |
|---|---|---|---|---|
| 5000000 | 615605 | 594182 | 594740 | 601509 |
| 10000000 | 1337824 | 1411564 | 1381991 | 1377126.33 |
| 15000000 | 2081455 | 2040063 | 2064093 | 2061870.33 |
| 20000000 | 2740431 | 2235609 | 2737440 | 2571160 |

lsearch_2():

| N | T1 | T2 | T3 | μ |
|---|---|---|---|---|
| 5000000 | 778269 | 780184 | 782542 | 780331.66 |
| 10000000 | 1599381 | 1584365 | 1577954 | 1587233.33 |
| 15000000 | 2394680 | 2361302 | 2384896 | 2380292.67 |
| 20000000 | 3252209 | 3232135 | 3152979 | 3212574.33 |



c. Open `lsearch_2.s` and have a look at the assembly code `gcc` has created on our behalf. Then do the following in the file `lsearch_2.s`:
   - Above the line "`lsearch_2:`", ...
     - add the function prototype as a comment
     - add the list of parameter-register mapping as a comment
   - Remove all the directives that can be removed,
   - Replace the labels used by the jump instructions with more descriptive labels, and
   - Comment the assembly code so that it would help the reader understand the code. In the comments, make sure you indicate which variable is held in which register (as well as its value).

Include `lsearch_2.s` in this assignment. You do not have to submit it to CourSys.

```
// Filename : lsearch_2.s

// Description: A more efficient implementation of the linear search.

//int lsearch_2(int *A, int n, int target)

// A is %rdi  n is %rsi  target is %rdx

_lsearch_2:


        pushq  %rbp                    // push rbp into stack


        movq   %rsp, %rbp

        testl    %esi, %esi            // check if n < = 0

        jle      LBB0_1                // if it is less than and equal zero then jump


        movslq %esi, %rax          // let return value is  n

        movl   -4(%rdi,%rax,4), %r8d     // temp=A[n-1] A + 4*n-4 because int is 4 byte and n-1
is move back a element

        movl   %edx, -4(%rdi,%rax,4)     // move A[n-1] into target
```

```
        leaq    -1(%rax), %rcx          // i=0
        movl    $-1, %esi               // n=n-1
        movq    %rdi, %rax              // return value is A
LBB0_3:
        incl    %esi                    // i = i+1
        cmpl    %edx, (%rax)            // test if A[i] equal to the target
        leaq    4(%rax), %rax           // A=address of next element
        jne     LBB0_3                  // if it is not equal do LBB0_3 loop again

        cmpl    %edx, %r8d              // compare the temp and target if they are equal
        movl    $-1, %eax               // put -1 into return value(n): n-1
        cmovel  %ecx, %eax              // move when less or equal , return value is i
        movl    %r8d, (%rdi,%rcx,4)     // temp = A+ 4* i--> temp=A[n-1]
        cmpl    %esi, %ecx              // check if i <n-1
        cmovgl  %esi, %eax              // move i to retur nvalue when greater and less than
        popq    %rbp
        retq
LBB0_1:
        movl    $-1, %eax               // let return value is -1 and return it.
        popq    %rbp
        retq
```

d. Because compiler optimization was turned on (-O2), `gcc` has made some adjustments to our original algorithm (original algorithm given above). Figure out what these adjustments to our original algorithm are and write this algorithm with these new adjustments, in pseudocode, below.

The adjustment in the order in which the program is run. When the overall logic is not affected, the operation on the values that are no longer used is executed first.

If (n<=0) do result← -1

N← n-1

temp ←A[n-1]

target← temp

n← n-1

while (A[i]!=target) do A[i]←A[i+1] i←i+1

if (A[n-1]==target) result← n-1

if (i<n-1) result← i

---

3. [8 marks] Microbenchmarking array search versus linked list search (using the microbenchmarking tool introduced in Lab 6)

We will be able to fully answer this question after having done our Lab 6 on Monday April 6.

In this question, we shall investigate the performance of the *linear search* algorithm as it searches an array and a linked list. But as we saw in the previous question, not all linear time algorithms are created equal, so to identify the most efficient algorithm, we shall measure the clock cycles (cost) per element (CPE) using microbenchmarking.

a. If we have not yet done so, let's download `Assn9-files.zip` and extract its files. Have a look at the files in the `Q3` folder. The `main.c` sets up two data structures — an array `int A[N]` and a linked list `List *L` — with the values 0 . . . N – 1 in a random order. The microbenchmarking code follows.

We can choose between `lsearch()` for the array by entering 1 at the command line (`./x 1`) or `LLsearch()` for the linked list by entering 0 at the command line (`./x 0`).

The code for these linear search functions is in `LL.c`. Have a look. Compile the code and open `LL.s`. Comparing the main loops of the two functions:

```
LLsearch:                          lsearch:
   . . .                              . . .
.L42:                              .L57:
   cmpl %esi, (%rdx)                  addq $1, %rcx
   je .L41                            cmpl %edx, -4(%rdi,%rcx,4)
.L35:                                 je .L54
   movq 8(%rdx), %rdx              .L56:
   addl $1, %eax                      cmpq %rsi, %rcx
   testq %rdx, %rdx                   movl %ecx, %eax
   jne .L42                           jne .L57
   ...                                ...
```

it looks like there are the same number of instructions per element on both sides, but the cost of `LLsearch()` will be higher because of the von Neumann bottleneck, i.e., there are two memory references per element for `LLsearch()`, but only one memory reference per element for `lsearch()`. To even out the situation, let's add a memory reference instruction to `lsearch` as follows:

```
                                   lsearch:
                                      . . .
                                   .L57:
                                      addq $1, %rcx
Add this line and recompile ->        movl -8(%rdi), %r8d
                                      cmpl %edx, -4(%rdi,%rcx,4)
                                      je .L54
                                   .L56:
                                      cmpq %rsi, %rcx
                                      movl %ecx, %eax
                                      jne .L57
                                      ...
```

Re-build the code and microbenchmark each function for `N` = {150, 200, 250, 300}. For each, run the code 11 times and throw out the shortest 3 and longest 3 times. Let's tabulate our results, one quintet of measurements per row. Compute the average of each set of 5 times. Then, plot both data sets on a graph (with labels) and draw a straight line through each set as we did in Lab 6.

Compute the slope of each line. The slope = the clock cycles (cost) per element (CPE) for linearly searching each data structure (the array and the linked list).

What conclusion de we draw from our results?

Answer: we can see the lsearch which means array takes less time than LLsearch( ) means Linked List. After the size of 250, list has a sharp increasing.

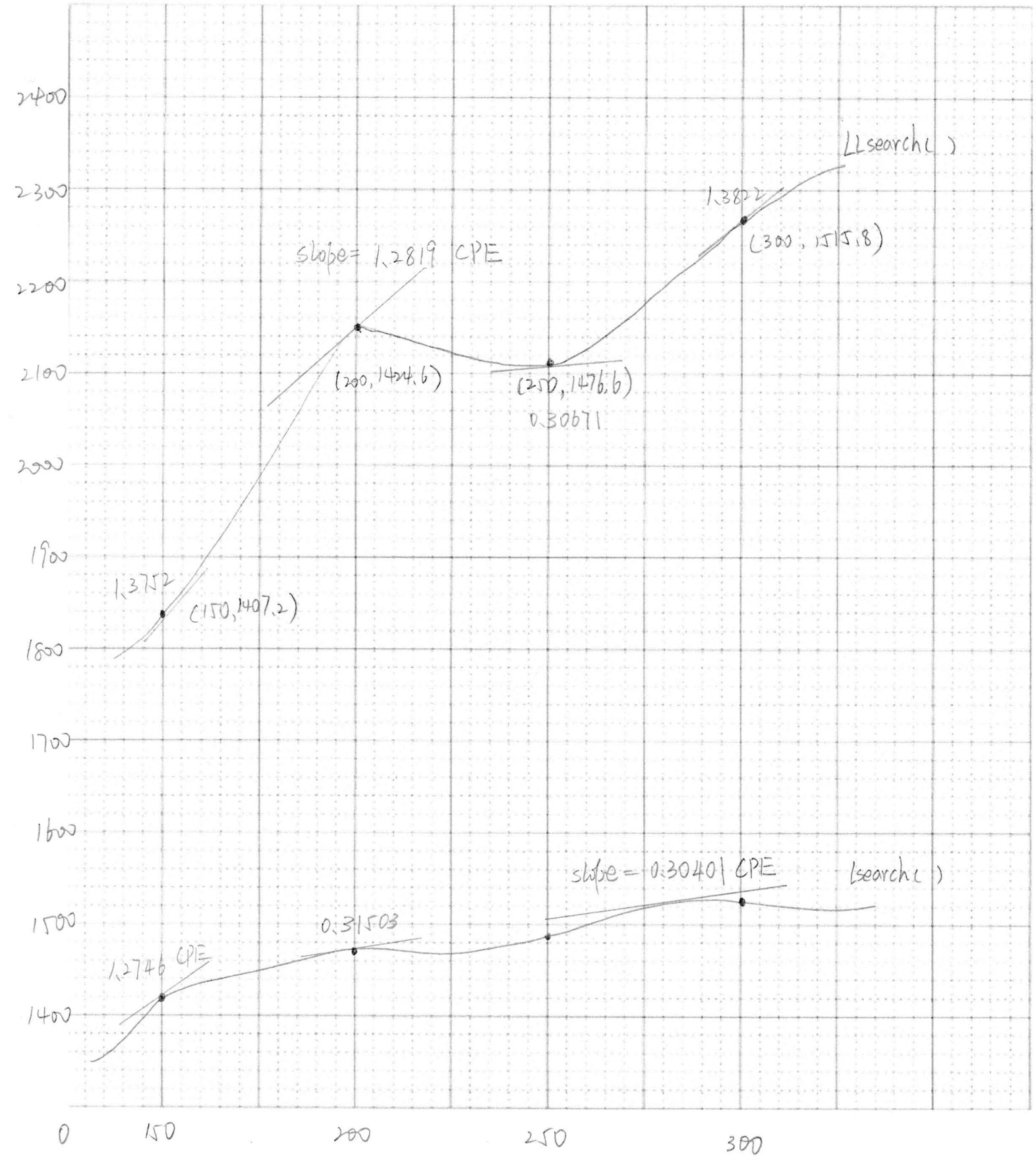
Submit all datasets and graph with this assignment.

lsearch(): for array

| N | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 | μ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 150 | 1620 | ~~1631~~ | ~~1644~~ | 1354 | 1376 | ~~1779~~ | 1342 | 1344 | ~~1336~~ | ~~1639~~ | ~~1170~~ | 1407.2 |
| 200 | 1412 | 1435 | ~~1408~~ | 1417 | ~~1686~~ | ~~1684~~ | 1418 | ~~1776~~ | ~~1440~~ | 1441 | ~~1462~~ | 1424.6 |
| 250 | 1489 | 1470 | 1488 | 1464 | ~~1461~~ | 1487 | ~~1489~~ | ~~1487~~ | ~~1481~~ | ~~1581~~ | ~~1457~~ | 1476.6 |
| 300 | 1507 | ~~1502~~ | 1504 | 1522 | ~~1288~~ | ~~1534~~ | 1514 | ~~1521~~ | 1532 | ~~1535~~ | ~~1534~~ | 1515.8 |

LLsearch() : for linked list

| N | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 | μ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 150 | 1979 | 1734 | ~~1986~~ | ~~1994~~ | ~~1434~~ | ~~2091~~ | 1982 | ~~1702~~ | 1733 | ~~1702~~ | 1718 | 1829.2 |
| 200 | 2178 | ~~2241~~ | ~~2362~~ | 2181 | ~~1908~~ | ~~1988~~ | 2218 | ~~1989~~ | ~~2226~~ | 2196 | 1987 | 2152 |
| 250 | ~~2124~~ | ~~1987~~ | 2077 | ~~1776~~ | 2110 | 2099 | ~~2276~~ | 2110 | ~~2314~~ | 2118 | ~~2177~~ | 2102.8 |
| 300 | ~~2378~~ | ~~2291~~ | 2277 | 2249 | ~~2057~~ | ~~1948~~ | ~~2251~~ | ~~2261~~ | 2283 | 2262 | 2291 | 2272.4 |


2400
2300
2200
2100
2000
1900
1800
1700
1600
1500
1400
LLsearch( )
1,3802
(300, 1515.8)
slope= 1.2819 CPE
(200, 1424.6)
(250, 1476.6)
0.30671
1,3752
(150, 1407.2)
slope = 0.30401 CPE
lsearch( )
0.31503
1.2746 CPE
0
150
200
250
300

b. The main reason for the difference in the CPE is a data dependency in the linked list search that isn't present in the array search. Explain the sequence of instructions that causes the delay. Feel free to include a diagram as part of your explanation.

```
while (curr && curr->el != target) {

    curr = curr->next;

    pos++;}
```

The totally time of LLsearch is more than lsearch which means the linked list using more time to handle data than array. Because of linked list have more data dependency than array.

```
movq   8(%rcx), %rcx

testq   %rcx, %rcx
```

```
movq   8(%rdi), %rcx

movl   $-1, %eax

testq   %rcx, %rcx
```

The data dependency more easy to cause the data hazard. So that why linked list need more time than array.

---

4. [7 marks] "Loop unrolling" optimization

   a. In a file called `unrolling.c`, write a version of the inner product function described in Homework Problem 5.13 (at page 570 in our textbook) that uses 6 x 1 loop unrolling and call this function `inner6x1unrolling(...)`.

   b. In the same file, write a second version of the inner product function described in Homework Problem 5.13 that uses 6 x 6 loop unrolling and call this function `inner6x6unrolling(...)`.

   Add a header comment block and comments to your functions and submit `unrolling.c` electronically via CourSys as well as adding a hardcopy of this file in this assignment.

```
// Filename : unrolling.c
// purpose: change original function into inner6x1unrolling and inner6x6unrolling
// Junchen Li
// 301385486

void inner6x1unrolling(vec_ptr u, vec_ptr v, data_t *dest)
{
        long i;
        long length= vec_length (u);
        data_t *udata = get_vec_start(u);
        data_t *vdata = get_vec_start(v);
        data_t sum = (data_t) 0;

        for (i = 0; i < length-6; i+=6)
        {
                sum = sum + (udata[i]*vdata[i] + udata[i+1]*vdata[i+1]+ udata[i+2]*vdata[i+2]+
udata[i+3]*vdata[i+3]+ udata[i+4]*vdata[i+4] +udata[i+5]*vdata[i+5]);
                //add current value and next five values into sum
        }
        for (; i < length ; i++)
                sum+= udata[i]*vdata[i];       // add the rest element into result if it has more

        *dest = sum;                    // return back to the destination
}

void inner6x6unrolling(vec_ptr u, vec_ptr v, data_t *dest)
{
        long i;
        long length= vec_length (u);
        data_t *udata = get_vec_start(u);
        data_t *vdata = get_vec_start(v);
        data_t sum1 = (data_t) 0;         // introduce more variables for doing 6x6
         data_t sum2 = (data_t) 0;
        data_t sum3 = (data_t) 0;
        data_t sum4 = (data_t) 0;
        data_t sum5 = (data_t) 0;
```

```
        data_t sum6 = (data_t) 0;
        data_t sum_result = (data_t) 0;

        for (i = 0; i< length-6 ; i+=6)
        {
                sum1 += udata[i]*vdata[i];        // add current value into sum1
                sum2 += udata[i+1]*vdata[i+1];
        sum3 += udata[i+2]*vdata[i+2];
        sum4 += udata[i+3]*vdata[i+3];
        sum5 += udata[i+4]*vdata[i+4];
        sum6 += udata[i+5]*vdata[i+5];// add next value into sum2
        }
        sum_result= sum1+sum2+sum3+sum4+sum5+sum6;        // merge two results into
sum_result (the final result)

        for (; i< length; i++)
                sum_result += udata[i]*vdata[i];      //add the rest element into result if it has
more

        *dest = sum_result;                  // return back to the destination
}
```