Lab Report

ECPE 170 – Computer Systems and Networks – Fall 2012

Name: Erich Viebrock

Lab Topic: Performance Optimization (Lab #: 5)

Pre-Lab

I am booting Linux directly instead of inside a virtual machine.

Question #6:

Boot Linux. With no applications running in Linux, how much RAM is available *inside* the virtual machine? The "System Monitor" program should report that information. This is the space that is actually available for our test application.

Answer:

Not sure if there's a “System Monitor” program for Xubuntu, but by using the command

```
cat /proc/meminfo
```

I see I have 96 MB available RAM out of 500 MB total RAM, while idle (yikes!).

Question #7:

What is the code doing? (Describe the algorithm in a paragraph, focusing on the `combine1()` function.)

Answer:

The code essentially allows the user to enter an amount of elements into a vector. The vector is stored in dynamically allocated memory, then deleted upon exiting the program.

Question #8:

What is the largest number of elements that the vector can hold WITHOUT using swap storage (virtual memory), and how much memory does it take?

Answer:

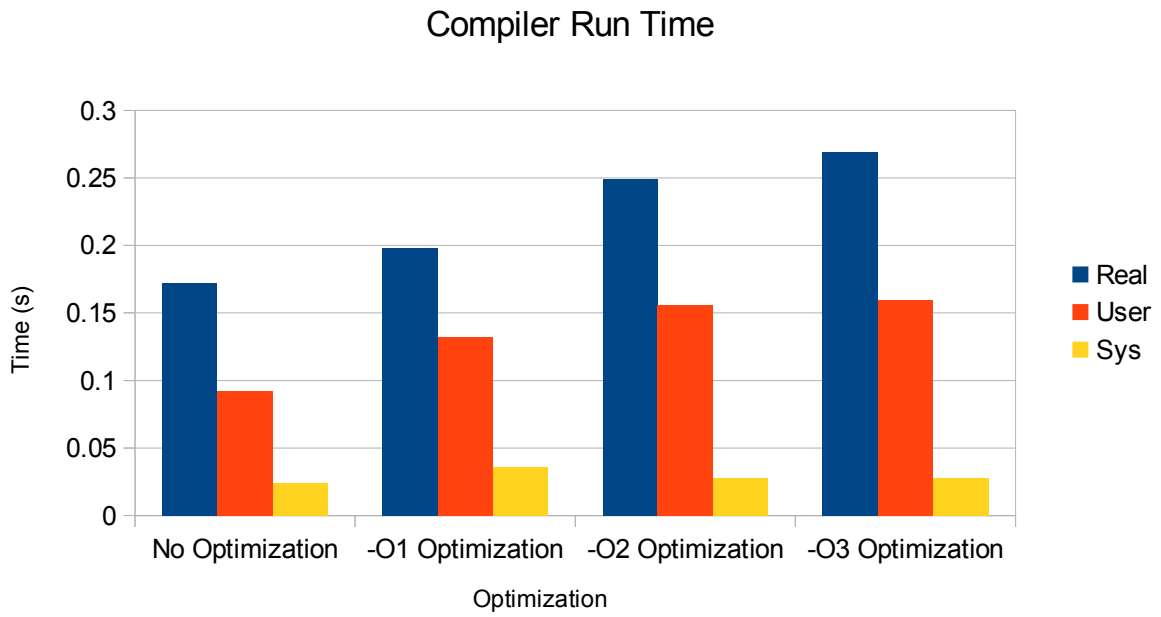
When entering 100,000,000 elements, the program allocates 381.47 MB of RAM for the vector. This is about 100 MB less of total memory, while around 300 MB more than available RAM. Needless to say, entering 100,000,000 elements caused the machine to crash for around a minute. When using 10,000,000 elements, however, only 38.15 MB was allocated, allowing the machine to run flawlessly. Therefore, the largest number of elements that the vector can hold without using swap storage should be somewhere in between these numbers.

Lab

Question #1:
What vector size are you using for all experiments in this lab?
Answer:
10,000,000 elements.

Question #2:
How much time does the **compiler** take to finish with (a) no optimization, (b) with -O1 optimization, (c) with -O2 optimization, and (d) with -O3 optimization? Create both a table and a graph in LibreOffice Calc.
Answer:

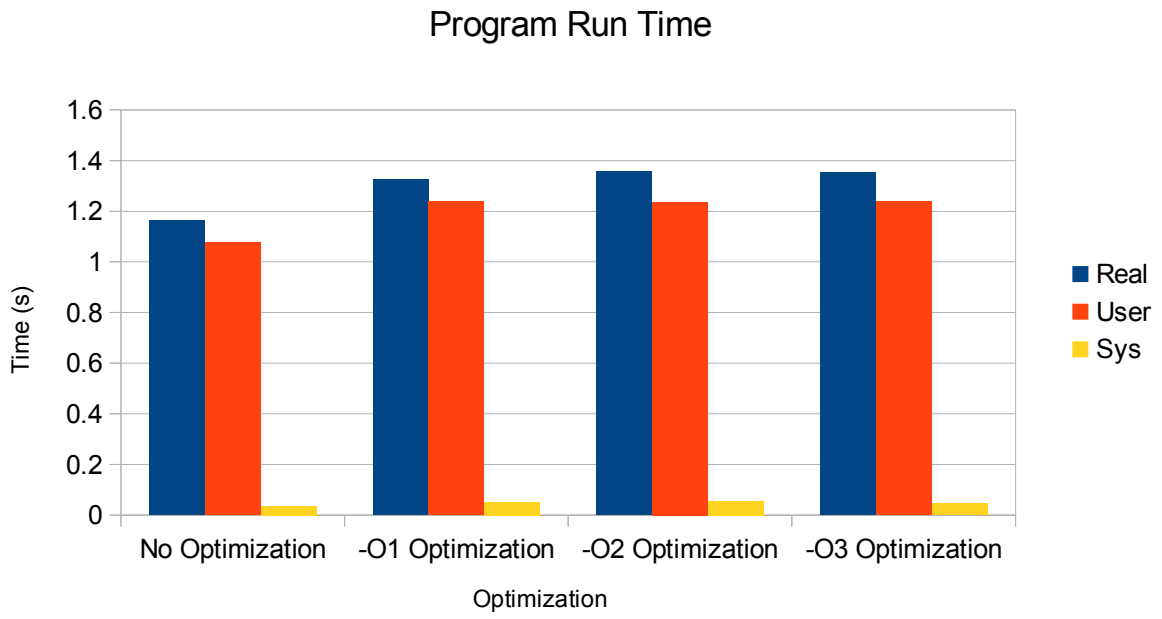
Compiler Run Time (s)				
	No Optimization	**-O1 Optimization**	**-O2 Optimization**	**-O3 Optimization**
Real	0.172	0.198	0.249	0.269
User	0.092	0.132	0.156	0.16
Sys	0.024	0.036	0.028	0.028



Question #3:
How much time does the **program** take to finish with (a) no optimization, (b) with -O1 optimization, (c) with -O2 optimization, and (d) with -O3 optimization? Create both a table and a graph in LibreOffice Calc.

Answer:
Although program run time should decrease with optimization tweaks, my computer does not. Dr. Shafer and I tested this in class, and verified the run time was longer with optimized code vs. non-optimized code.

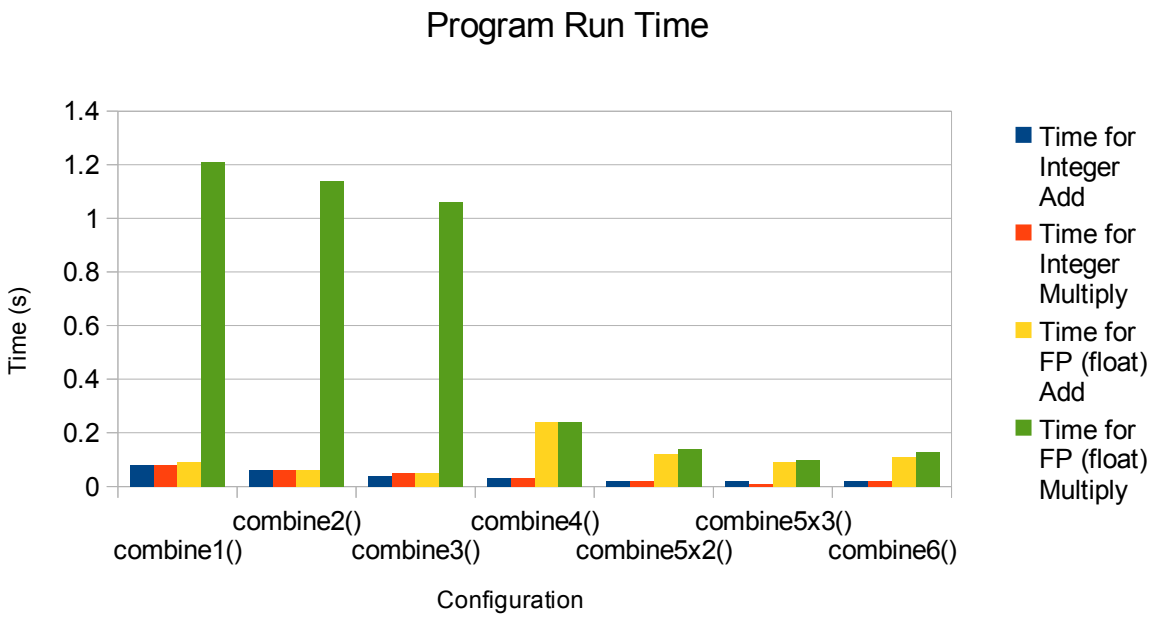
Program Run Time (s)				
	No Optimization	**-O1 Optimization**	**-O2 Optimization**	**-O3 Optimization**
Real	1.165	1.326	1.356	1.353
User	1.076	1.24	1.236	1.24
Sys	0.036	0.052	0.056	0.048



Question #4:
After implementing each function, benchmark it for a variety of data types and mathematical operations. Fill in the table below as you write each function.
Answer:

Function Benchmarking						
Configuration	**Vector Size (elements)**	**Vector Size (MB)**	**Time for Integer Add**	**Time for Integer Multiply**	**Time for FP (float) Add**	**Time for FP (float) Multiply**
combine1()	10000000	38.15	0.08	0.08	0.09	1.21
combine2()	10000000	38.15	0.06	0.06	0.06	1.14
combine3()	10000000	38.15	0.04	0.05	0.05	1.06
combine4()	10000000	38.15	0.03	0.03	0.24	0.24
combine5x2()	10000000	38.15	0.02	0.02	0.12	0.14
combine5x3()	10000000	38.15	0.02	0.01	0.09	0.1
combine6()	10000000	38.15	0.02	0.02	0.11	0.13

Question #5:
Using LibreOffice Calc, create a *single* graph that shows the data in the table created, specifically the four time columns. (You don't need to plot vector size).
Note: No credit will be given for a sloppy graph that lack X and Y axis labels, a legend, and a title.
Answer:



Combine.c Source Code:

```
#include "config.h"
#include "vec.h"
#include "combine.h"

#include <stdio.h>

// ORIGINAL function.
// This combiner function uses the greater amount
// of abstraction to operate, but has the slowest
// performance.
void combine1(vec_ptr v, data_t *dest)
{
  printf("Running combine1() - No code-level optimizations\n");

  long int i;

  *dest = IDENT;

  for(i=0; i < vec_length(v); i++)
    {
      data_t val;
      get_vec_element(v, i, &val);
      *dest = *dest OP val;
    }
}


// CODE MOTION OPTIMIZATION:

// Move the call to vec_length() out of the loop
// because we (the programmer) know that the vector length will
// not change in the middle of the combine() function.
// The compiler, though, doesn't know that!
void combine2(vec_ptr v, data_t *dest)
{
  printf("Running combine2()\n");
  printf("Added optimization: Code motion\n");

  long int i;

  *dest = IDENT;

  int tmp_length = vec_length(v);

  for(i=0; i < tmp_length; i++)
    {
      data_t val;
```

```
      get_vec_element(v, i, &val);
      *dest = *dest OP val;
    }

}


// REDUCING PROCEDURE CALLS OPTIMIZATION:

// This optimization eliminates the function call to
// get_vec_element() and accesses the data directly,
// trading off higher performance versus some loss
// of program modularity.
void combine3(vec_ptr v, data_t *dest)
{
  printf("Running combine3()\n");
  printf("Added optimization: Reducing procedure calls\n");

  long int i;

  *dest = IDENT;

  int tmp_length = vec_length(v);

  data_t *ptr;
  ptr = get_vec_start(v);

  for(i=0; i < tmp_length; i++)
    {
      *dest = *dest OP ptr[i];
    }

}


// ELIMINATING UNNEEEDED MEMORY ACCESSES OPTIMIZATION:

// This optimization eliminates the trip to memory
// to store the result of each operation (and retrieve it
// the next time). Instead, it is saved in a local variable
// (i.e. a register in the processor)
// and only written to memory at the very end.
void combine4(vec_ptr v, data_t *dest)
{
  printf("Running combine4()\n");
  printf("Added optimization: Eliminating unneeded memory
accesses\n");

  long int i;
```

```
  int tmp = IDENT;

  int tmp_length = vec_length(v);

  data_t *ptr;
  ptr = get_vec_start(v);

  for(i=0; i < tmp_length; i++)
    {
      tmp = *dest OP ptr[i];
    }
  *dest = tmp;

}


// LOOP UNROLLING x2
// (i.e. process TWO vector elements per loop iteration)
void combine5x2(vec_ptr v, data_t *dest)
{
  printf("Running combine5x2()\n");
  printf("Added optimization: Loop unrolling x2\n");

  long int i;

  int tmp = IDENT;

  int tmp_length = vec_length(v);

  data_t *ptr;
  ptr = get_vec_start(v);

  for(i=0; i<tmp_length; i=i+2)
    {
      tmp = (*dest OP ptr[i]) OP ptr[i+1];
    }
  if(i == tmp_length+1)
    {
      tmp = *dest OP ptr[i-1];
    }
  *dest = tmp;

}

// LOOP UNROLLING x3
// (i.e. process THREE vector elements per loop iteration)
void combine5x3(vec_ptr v, data_t *dest)
{
```

```
  printf("Running combine5x3()\n");
  printf("Added optimization: Loop unrolling x3\n");

  long int i;

  int tmp = IDENT;

  int tmp_length = vec_length(v);

  data_t *ptr;
  ptr = get_vec_start(v);

  for(i=0; i<tmp_length; i=i+3)
    {
      tmp = (*dest OP ptr[i]) OP ptr[i+1] OP ptr[i+2];
    }
  if(i == tmp_length+1)
    {
      tmp = *dest OP ptr[i-1];
    }
  else if(i == tmp_length+2)
    {
      tmp = *dest OP ptr[i-2];
    }
  *dest = tmp;

}


// LOOP UNROLLING x2 + 2-way parallelism
void combine6(vec_ptr v, data_t *dest)
{
  printf("Running combine6()\n");
  printf("Added optimization: Loop unrolling x2, Parallelism x2\n");

  long int i;

  int tmp0 = IDENT;
  int tmp1 = IDENT;

  int tmp_length = vec_length(v);

  data_t *ptr;
  ptr = get_vec_start(v);

  for(i=0; i<tmp_length; i=i+2)
    {
      tmp0 = *dest OP ptr[i];
      tmp1 = *dest OP ptr[i+1];
```

```
    }
  if(i == tmp_length+1)
    {
      tmp1 = *dest OP ptr[i-1];
    }
  *dest = tmp0;
  *dest = tmp1;

}
```

Vec.c Source Code:

```
// http://csapp.cs.cmu.edu/public/ics2/code/opt/vec.c
// Computer Systems: A Programmer's Perspective
// Chapter 5: Optimizing Program Performance

#include <stdlib.h>
#include <stdio.h>
#include "config.h"
#include "vec.h"

// Create vector of specified length
vec_ptr new_vec(long int len)
{
  double total_bytes_to_alloc = sizeof(vec_rec) + sizeof(data_t)*len;
  printf("Allocating %.2f MB for vector storage\n",
total_bytes_to_alloc/1024/1024);

  // Allocate header structure
  vec_ptr result = (vec_ptr) malloc(sizeof(vec_rec));
  if (!result)
    return NULL;  /* Couldn't allocate storage */
  result->len = len;

  result->allocated_len = len;

  // Allocate array
  if (len > 0) {
    data_t *data = (data_t *)calloc(len, sizeof(data_t));
    if (!data) {
      free((void *) result);
      return NULL; /* Couldn't allocate storage */
    }
    result->data = data;
  }
  else
    result->data = NULL;
  return result;
}

// Fill vector with initial values
void init_vec(vec_ptr v)
{
  long int i;
  data_t value=1;
  long int length = vec_length(v);

  for(i=0; i < length; i++)
    {
```

```
      v->data[i] = value;
      value++;
    }
}

// Free vector
void free_vec(vec_ptr ptr)
{
  printf("Freeing vector from memory\n");
  free(ptr->data);
  free(ptr);
  ptr = NULL;
}


// Retrieve vector element and store at dest.
// Return 0 (out of bounds) or 1 (successful)
int get_vec_element(vec_ptr v, long int index, data_t *dest)
{
  if (index < 0 || index >= v->len)
    return 0;
  *dest = v->data[index];
  return 1;
}

// Return length of vector
long int vec_length(vec_ptr v)
{
    return v->len;
}

// Return first address of vector
data_t *get_vec_start(vec_ptr v)
{
     return &(v->data[0]);
}
```

Post-Lab

Question #1:

Measure the execution time for `moby.txt` (for n=2).

Answer:

The execution time for `moby.txt` for bigrams is 43.870000 seconds.

Question #2:

Copy the program output for `moby.txt` (for n=2) into your report.

Answer:

```
Running analysis program...

Options used when running program:
ngram     2
details   10
hash-table-size     1024
N-gram size 2

Running analysis... (This can take several minutes or more!)
 Initializing hash table...
 Inserting all n-grams into hash table in lowercase form...
 Sorting all hash table elements according to frequency...

Analysis Details:
(Top 10 list of n-grams)
1840 'of the'
1142 'in the'
714  'to the'
435  'from the'
375  'the whale'
367  'of his'
362  'and the'
350  'on the'
```

```
328  'at the'
323  'to be'

Analysis Summary:
214365 total n-grams
114421 unique n-grams
91775 singleton n-grams (occur only once)
Most common n-gram (with 1840 occurrences) is 'of the'
Longest n-gram (4 have length 29) is 'phrenological characteristics'
Total time = 43.870000 seconds

real 0m44.903s
user 0m43.859s
sys  0m0.020s
```

Question #3:

Measure the execution time for `shakespeare.txt` (for n=2).

Answer:

The execution time for `shakespeare.txt` for bigrams is 763.220000 seconds.

Question #4:

Copy the program output for `shakespeare.txt` (for n=2) into your report.

Answer:

```
Running analysis program...

Options used when running program:
ngram     2
details   10
hash-table-size     1024
N-gram size 2

Running analysis... (This can take several minutes or more!)
 Initializing hash table...
 Inserting all n-grams into hash table in lowercase form...
```

```
 Sorting all hash table elements according to frequency...

Analysis Details:
(Top 10 list of n-grams)
1892 'i am'
1804 'i ll'
1753 'in the'
1709 'my lord'
1681 'to the'
1660 'i have'
1603 'i will'
1554 'of the'
1118 'it is'
1020 'to be'

Analysis Summary:
965028 total n-grams
363039 unique n-grams
266018 singleton n-grams (occur only once)
Most common n-gram (with 1892 occurrences) is 'i am'
Longest n-gram (1 have length 32) is 'honorificabilitudinitatibus
thou'
Total time = 763.220000 seconds

real 13m21.954s
user 12m42.928s
sys  0m0.304s
```

Question #5:

Copy the Valgrind *memcheck* report for your **optimized program**, clearly showing **no errors**.

Answer:

Below is the memcheck for an unoptimized program; couldn't figure out how to implement `qsort()` or reformat the hash table.

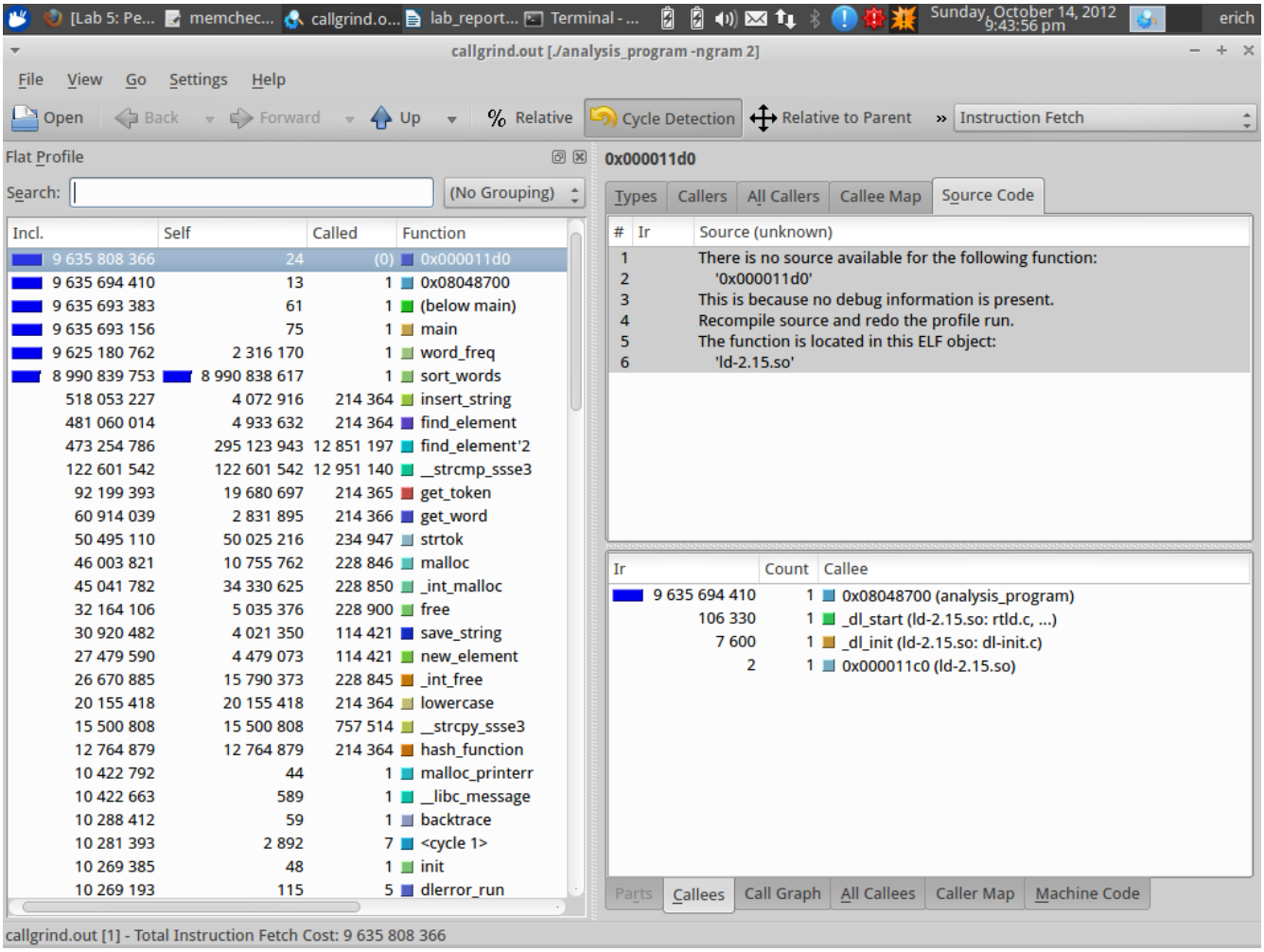
```
==2497== Memcheck, a memory error detector
==2497== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et
```

```
al.
==2497== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright
info
==2497== Command: ./analysis_program -ngram 2
==2497== Parent PID: 1721
==2497==
==2497== Invalid free() / delete / delete[] / realloc()
==2497==    at 0x402B06C: free (in
/usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==2497==    by 0x8049224: main (analysis.c:366)
==2497==  Address 0x80487c0 is in the Text segment of
/home/erich/bitbucket/2012_fall_ecpe170/lab05/postlab/analysis_progra
m
==2497==
==2497==
==2497== HEAP SUMMARY:
==2497==     in use at exit: 0 bytes in 0 blocks
==2497==   total heap usage: 228,845 allocs, 228,846 frees, 3,224,755
bytes allocated
==2497==
==2497== All heap blocks were freed -- no leaks are possible
==2497==
==2497== For counts of detected and suppressed errors, rerun with: -v
==2497== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from
0)
```

Question #6:

Take two screenshots of the Valgrind kcachegrind tool: one for the **unoptimized program**, and one for the final **optimized program**.

Answer:



Question #7:

Measure the execution time for `moby.txt` (for n=2) for your **optimized program**. What is the speedup compared to the original program?
Include the command used to run your program, since you will (almost certainly) want to modify some of the arguments.
Copy the program output into your report.

Answer:

Below is the execution time for an unoptimized program:

```
real 0m1.927s
user 0m0.960s
sys  0m0.016s
```

Below is the command for the execution time of the program:

```
time ./analysis_program -ngram 2 < moby.txt
```

Question #8:

Measure the execution time for `shakespeare.txt` (for n=2) for your **optimized program**. What is the speedup compared to the original program?
Include the command used to run your program, since you will (almost certainly) want to modify some of the arguments.
Copy the program output into your report.

Answer:

Below is the execution time for an unoptimized program:

```
real 0m22.973s
user 0m20.653s
sys  0m0.080s
```

Below is the command for the execution time of the program:

```
time ./analysis_program -ngram 2 < shakespeare.txt
```

Question #9:

Provide a `diff` between the original program and your final optimized program, so I can clearly see all changes that were made to all files. Version control can be helpful here!

Answer:

Because I could not implement the quick sort function, nor reformat the hash table, a `diff` would be meaningless.

Question #10:

Write a half-page narrative describing the optimization process you followed. Be sure to mention any false starts and unsuccessful changes too!

Answer:

Well, I began the attempt of optimization with `man qsort` via unix. Those man pages didn't really help me understand how to implement the qsort function through the STL library, so I then went online to make a quick sort by scratch. Again, I didn't really make much sense of it being the things sorted were pointers (I think), rather than integers. Because I couldn't understand how to use qsort in the program, I attempted to reformat the hash table. Unfortunately, I found myself more confused than I was before.

Post-Lab Wrapup

Question #1:

What was the best aspect of this lab?

Answer:

I enjoyed seeing how different implementations of code changed the performance of the program.

Question #2:

What was the worst aspect of this lab?

Answer:

OpenOffice Calc was quite the rascal for formatting tables and graphs.

Question #3:

How would you suggest improving this lab in future semesters?

Answer:

Can't think of anything, well designed lab!