CS840 Project 3:

Compiler Performance Measurement and

Line of Count Counting

Samuel Gluss

Jozo Dujmovic

3-28-2016

Abstract – The properties of a program’s code itself are important for many reasons. A programmer must write the code in the first place, and a compiler must parse it and produce executable machine code. To analyze this code, there must be some method to determine what requires time on the part of the programmer to produce, which are the actual physical lines of code (PLOC), and what the compiler must handle, the logical lines of code (LLOC). Once these standards are defined, and a way to parse code to count them is established, it is possible to begin analyzing code for relationships. Of particular interest in this paper will be the relationships between different line counters, as well as between LLOC, PLOC, produced executable size in bytes, and compile time. Lastly, performance analysis of GCC and VC++ in debug and release modes will be conducted to determine variation between compilers and compiler modes for numeric and nonnumeric algorithm execution times.

# I. Introduction

Programmer and compiler performance are extremely important considerations within the realm of computer science. Building a code base is a crucial step, but the line-count the source can be misleading, as this will often include lines that are blank, or have some trivial symbol on them that can often be automatically generated by an IDE. An example of this are the curly braces around the body of functions or conditional statements. The count of interest is the number of lines that require some thought and work on the part of the programmer, which means the line contains something meaningful, either a statement or comment. To determine the appropriate values for LLOC and PLOC for any given program, it will be necessary to parse the source code and increment counts for appropriate tokens.

Once these characteristics have been recorded, they can used to determine relationships between linecounts and properties such as executable size, or compile time. This information can be used to determine how much time a programmer can expect to wait for their code to compile, or how much space the default libraries and included files will occupy.

The last thing to test is the performance of code generated by different compilers in release and debug mode. The programs to be compiled will be numeric and nonnumeric, so any performance benefit for one or the other, or both, will be evident in the analysis. Ultimately, this will help the programmer to determine which tools to favor, and how to plan for their software development projects.

Table of Contents

[I. Introduction 1](#_Toc447072557)

[II. Code Counting and Compiler Performance 3](#_Toc447072558)

[III. LLOC Count Comparison: Benchmaker vs Test Parser 5](#_Toc447072559)

[IV. LLOC, PLOC, and Size of Compiled Executable in Bytes 6](#_Toc447072560)

[V. LLOC, PLOC, and Executable Size For Human Produced Code 9](#_Toc447072561)

[VI. Relationship between LLOC and Compile Time 12](#_Toc447072562)

[VII. Different Compilers, Computers, and Release vs Debug 13](#_Toc447072563)

[VIII. Conclusion 14](#_Toc447072564)

[IX. Appendix – Data 15](#_Toc447072565)

[X. Appendix - Code 16](#_Toc447072566)

# II. Code Counting and Compiler Performance

The first task was to produce a C++ parser and counter, which would be able to determine the LLOC and PLOC for any given piece of C++ code. The parser first counts physical lines of code by counting any line that has alphanumeric symbols on it, in addition to a newline. Such a line of code requires at least some work on the part of the programmer or IDE, and so should be counted by the PLOC counter.

The compiler will not process all of this code though, so the code is then sanitized to remove comments, preprocessor directives, and strings/chars. This is done using a state machine, which only retains code if it is not part of a comment, string, or preprocessor directive. This ensures that only normal code remains, which is easy to parse with regular expressions. This is then done to determine the number of functions, iterators, conditional statements, semicolons, and variable initialization assignments are present in the code. The LLOC is composed of the sum of these values, the number of lines of executable code.

The first test of this code will be on programs generated by Benchmaker, a C++ compiler benchmark generator which can produce code with varying LLOC, function count, and proportion of conditional and iterative elements. The LLOC reported by BM will be compared with the results of the test counter. This will give some insight into the relative behavior of the code counter. Once this is done, the relationships between LLOC, PLOC, and executable filesize will be determined. To present and analyze this, the outputs from Benchmaker will be analyzed first, and best fit curves for this data calculated. Then, 5 representative programs written for a programming languages class will be analyzed to determine if the trends observed in Benchmaker are consistent with realworld programs.

Compiler performance is measured in two different ways. First, large programs are produced with Benchmaker, and their compile times are measured by a Python script. By using very large source files, any error due to time measurement can be reduced to an acceptable level. The second test is to compile a numeric program that multiplies matrices together, and a nonnumeric program that sorts a very large array of random numbers using the quicksort algorithm. These programs are compiled in debug as well as release mode, with GCC and VC++. The resulting executables will be tested on two different machines, the specifications of which are as follows:

Thinkpad W541

-Intel i7-4710MQ 2.50 GHz quad-core processor and 16GB of Crucial DDR3 PC3-12800 RAM

Thinkpad T420s

-Intel Core i5-2540m 2.6Ghz twin-core processor and 16GB of Crucial DDR3 PC3-12800 RAM

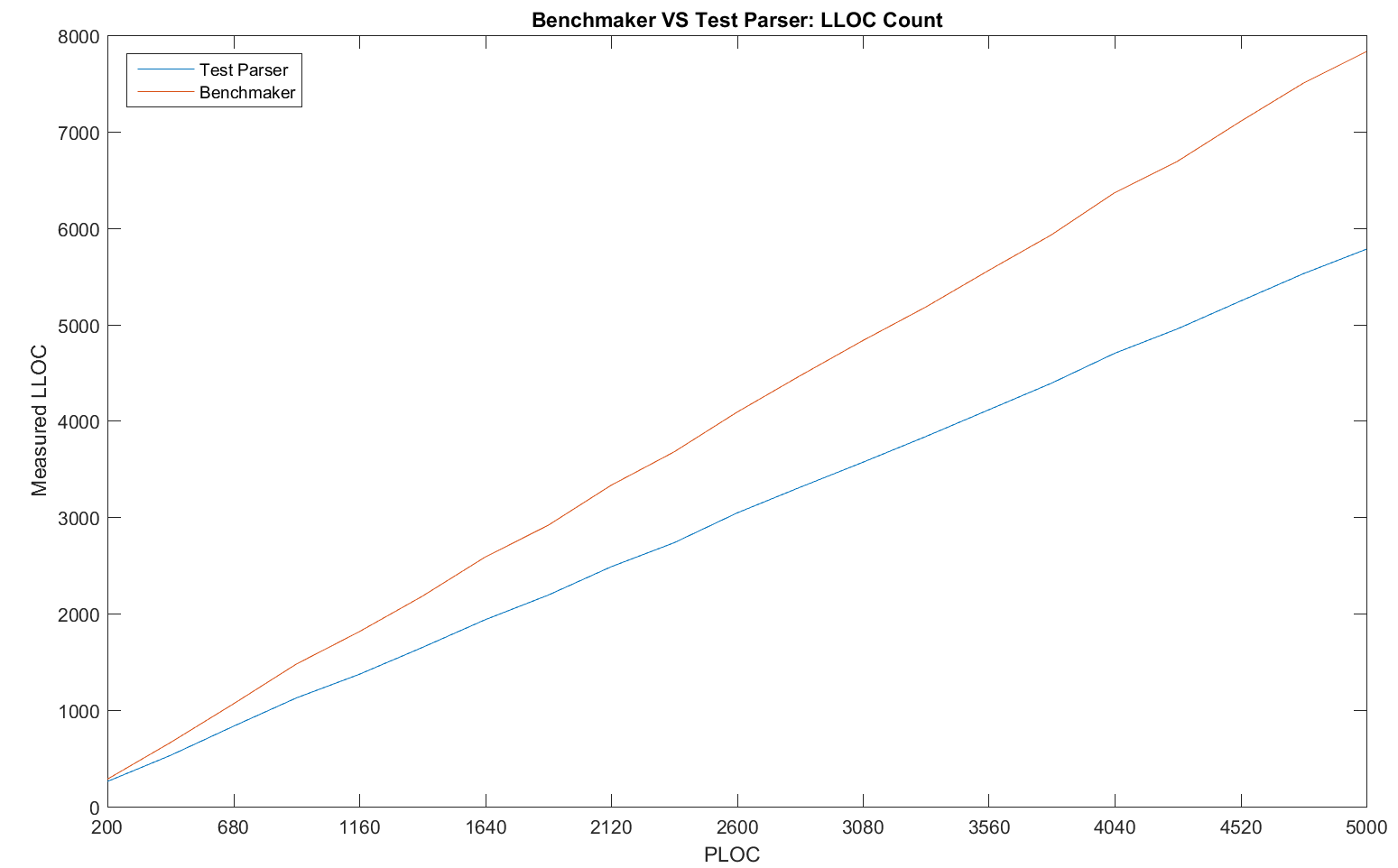
The compilers used are the following:

-Microsoft (R) C/C++ Optimizing Compiler Version 19.00.23506 for x86

- GCC version 4.9.3 (GCC)

The geometric mean of the ratios of performance for the two compilers, as well as the two computers, for numeric and nonnumeric programs, as well as the combined mean.

# III. LLOC Count Comparison: Benchmaker vs Test Parser

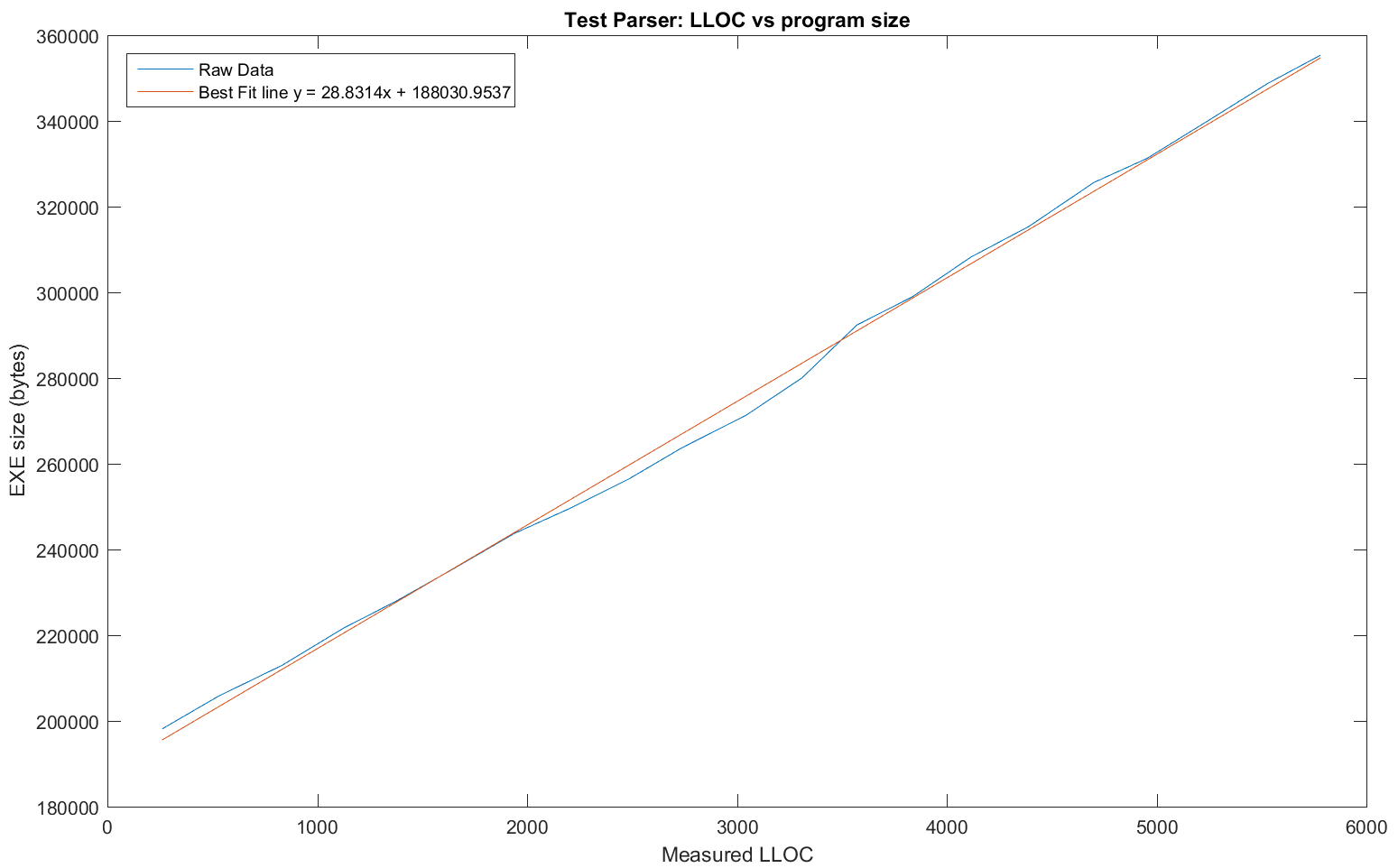


The first test of the parser is to compare the LLOC reported to the values in the Benchmaker output file. The LLOC is determined for ‘normal’ code that has been cleaned of strings, comments, and preprocessor directives. Once this is done, a sequence of regular expressions are used to match any for/while loops, if statements, semicolons, switches, and function definitions. Within each match for a variable declaration, a count of commas and equal signs are taken, and all these elements are combined to determine the LLOC for a source file.

The parser is run on each of 20 files produced by benchmaker, counting the occurences of features of interest. These counters are then written as comma separated values to an outfile, which becomes a matrix as more programs are parsed. This outfile is read in by matlab, which is then utilized to do the data analysis and graph production.

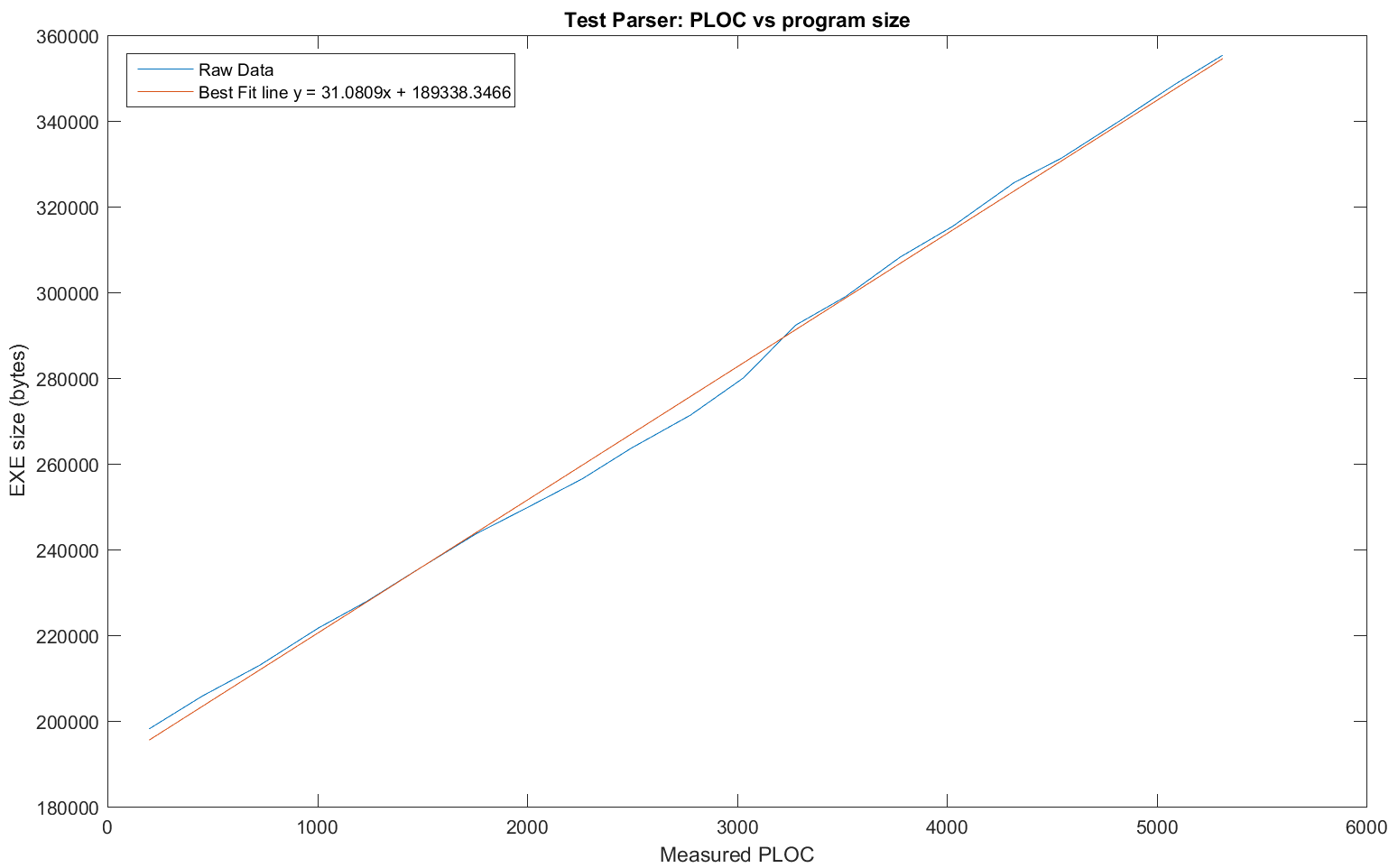
Interestingly, the parsers LLOC count lags behind that of Benchmaker by a relatively fixed ratio. This suggests that Benchmaker is counting something that the parser is not. Further testing, including inspection of the Benchmaker source code, would be able to reveal the reason for this relatively large ratio.

# IV. LLOC, PLOC, and Size of Compiled Executable in Bytes



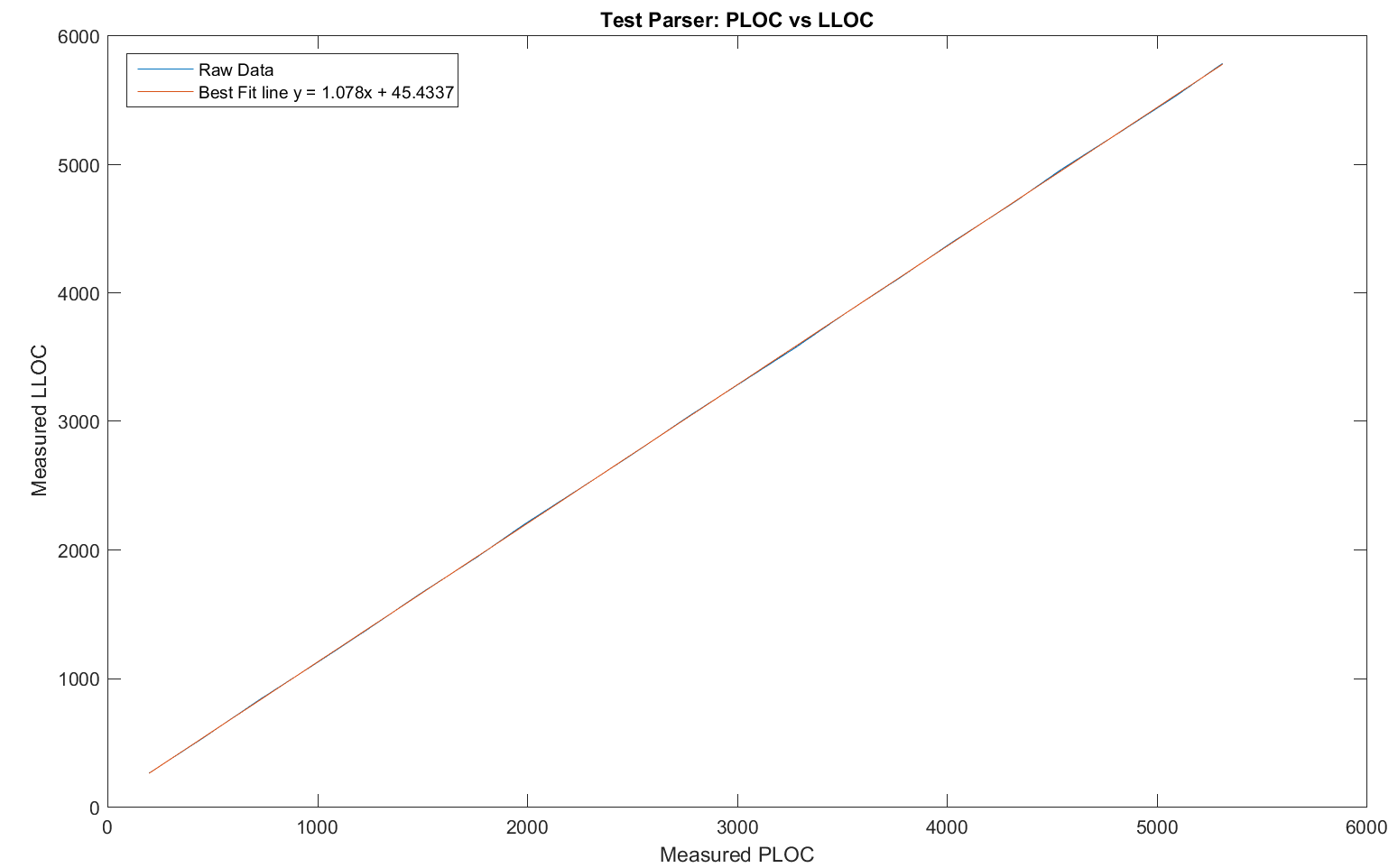
There is a strong correlation between LLOC and the size of the executable produced. The equation for the best fit line reveals that each addition LLOC is causes the executable to increase in size by approximately 29 bytes. Furthermore, the initial value at LLOC = 0 suggests a minimum executable size of about 188 kB. This is most likely due to the inclusion of various libraries for input/output and timing that are included in the benchmark files. A course of further testing would be to experiment with different compilers and optimization configurations to determine their effect on executable size.

The data conforms well to a linear best fit model, except for one peculiar dip between 2000 and 3500 LLOC. This could be explained by an optimization for executable size that becomes less efficient as executables grow larger. Testing with different optimization options might clarify this hypothesis. Another possibility is that the relationship between compiled size and LLOC is not a linear one, because while there is a dip in the middle, the data rides above the best fit line at the extremities.



In comparing PLOC to executable size, a pattern arises with PLOC. The shape of the data curve is approximately the same, again perhaps indicating a nonlinear relationship. Furthermore, the initial program size is similar, just over 189kB. The principal difference is that PLOC is related to executable size at a ratio of 1 to 31. This suggests that PLOC and LLOC are likely linearly related.

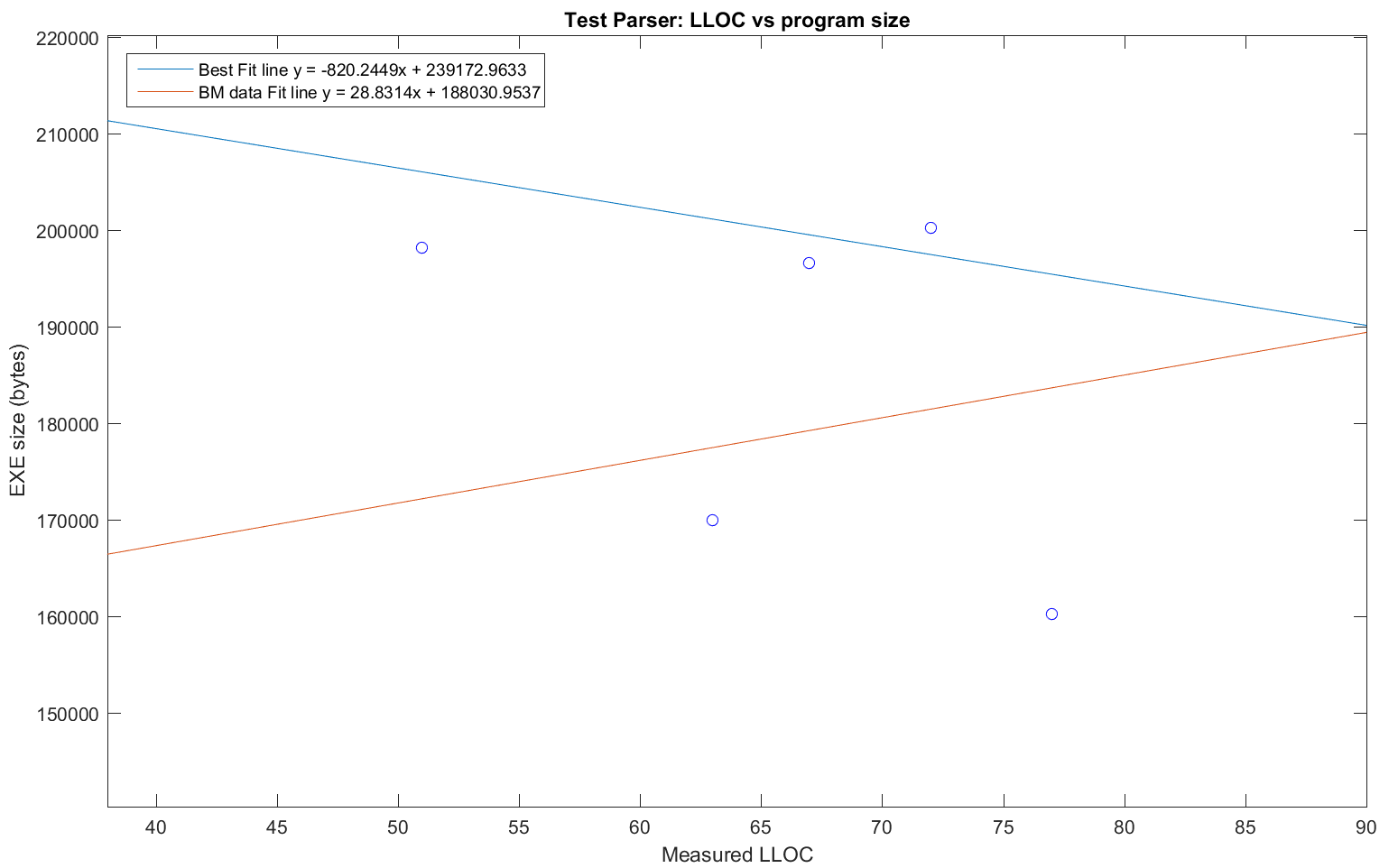
In the case of LLOC and PLOC, the data wavers around the best fit function, but does seem to converge at very large LOC values. This does seem to indicate that the data is unreliable at lower code due to some optimizing behavior of the compiler.



Any difference between the data and the best fit line is almost indiscernible, suggesting that the earlier hypothesis about PLOC and LLOC being linearly related is correct. This also means that Benchmaker operates very consistenty, as there is almost no deviation. The relationship between the two quantities is very close to 1-1, which suggests that most LLOC are also discrete PLOC.

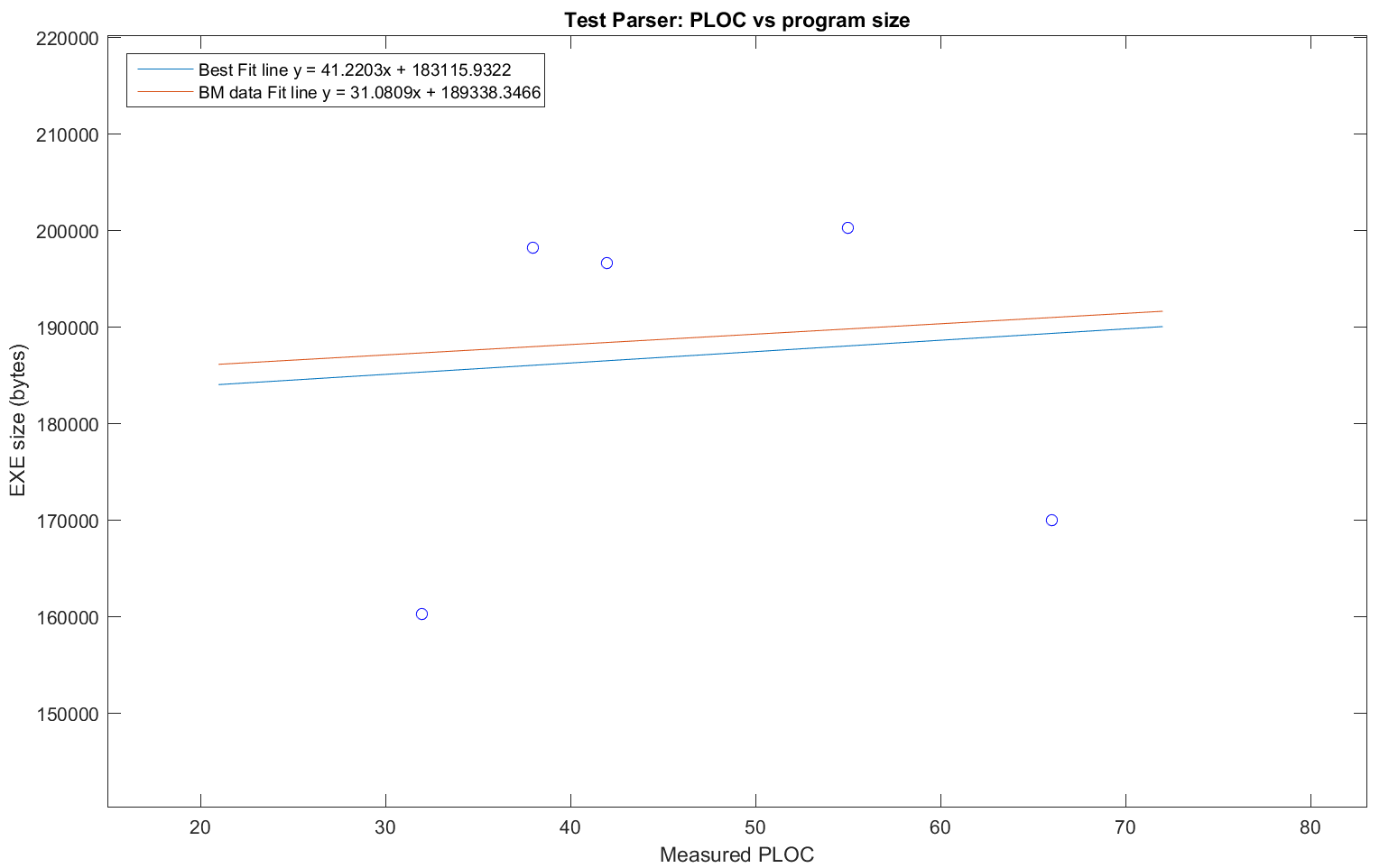
The unusual thing about the best fit line is its initial value of LLOC = 45, suggesting that there would be logical code without any physical lines present. This could be due to how Benchmaker initializes many variables on one line commonly, giving LLOC a significant offset from PLOC for even relatively small programs.

# V. LLOC, PLOC, and Executable Size For Human Produced Code

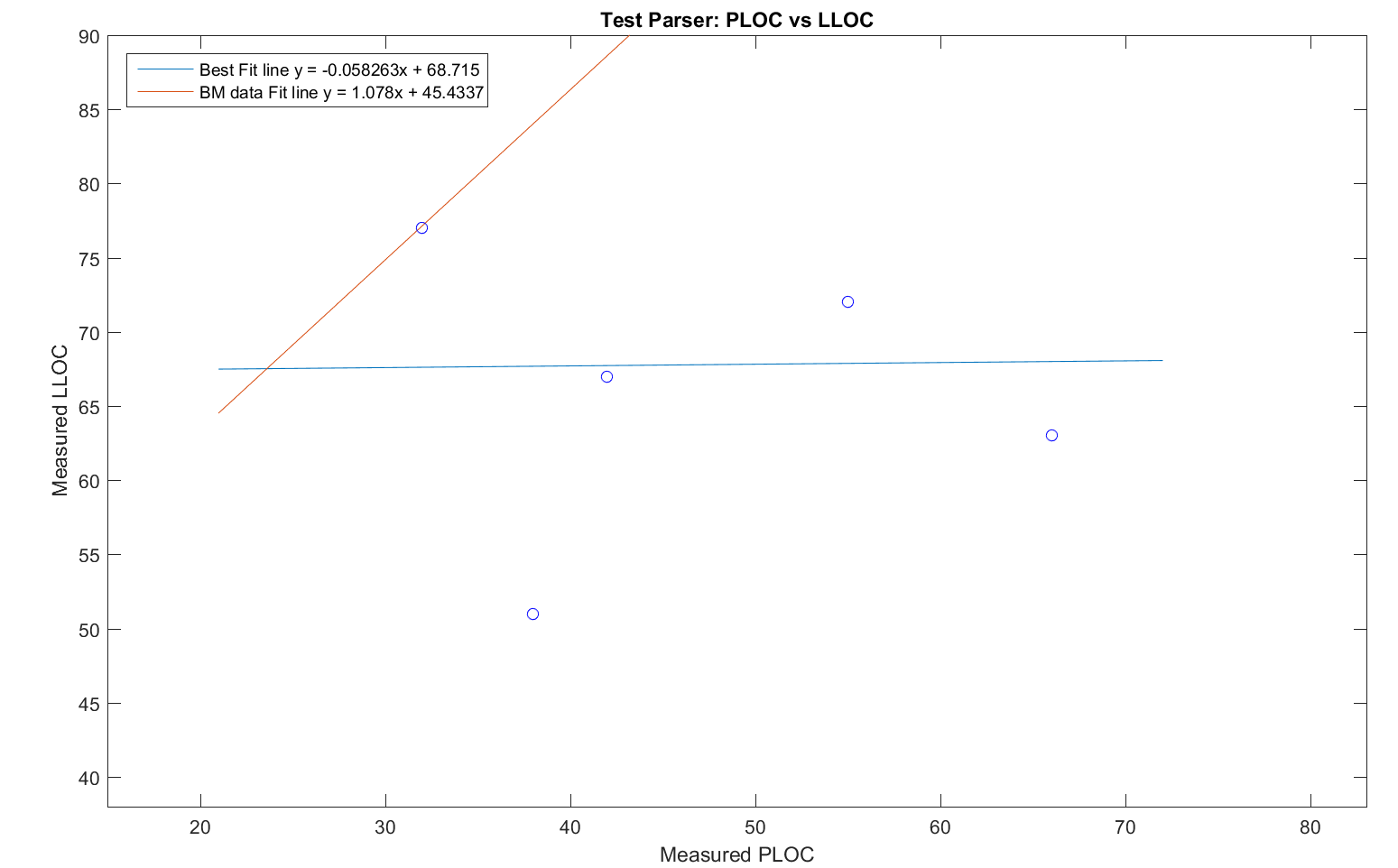


The regular code samples used for this test came from a programming languages class, where multiple problems were solved with C++ programs. As a result, some may be somewhat irregular in their structure compared the relatively homogenous benchmaker programs. In this case, there is one program with very high LLOC and very low executable size, which runs counter to the hypothesis that higher LLOC causes larger sized executables. This could be due to a larger number of strings present in the other programs, a library which was perhaps not included, or some other anomaly.

The human-produced programs could vary wildly given the range of syntax that C++ compilers are able to parse. Covering every possible edge case in a language as complex as C++ is quite challenging, so an improvement for the parser would be to build it with a generator such as Bison, using a lexical analyzer like Flex to produce input tokens. This would make it easier to produce a more flexible parser than the one currently being used.



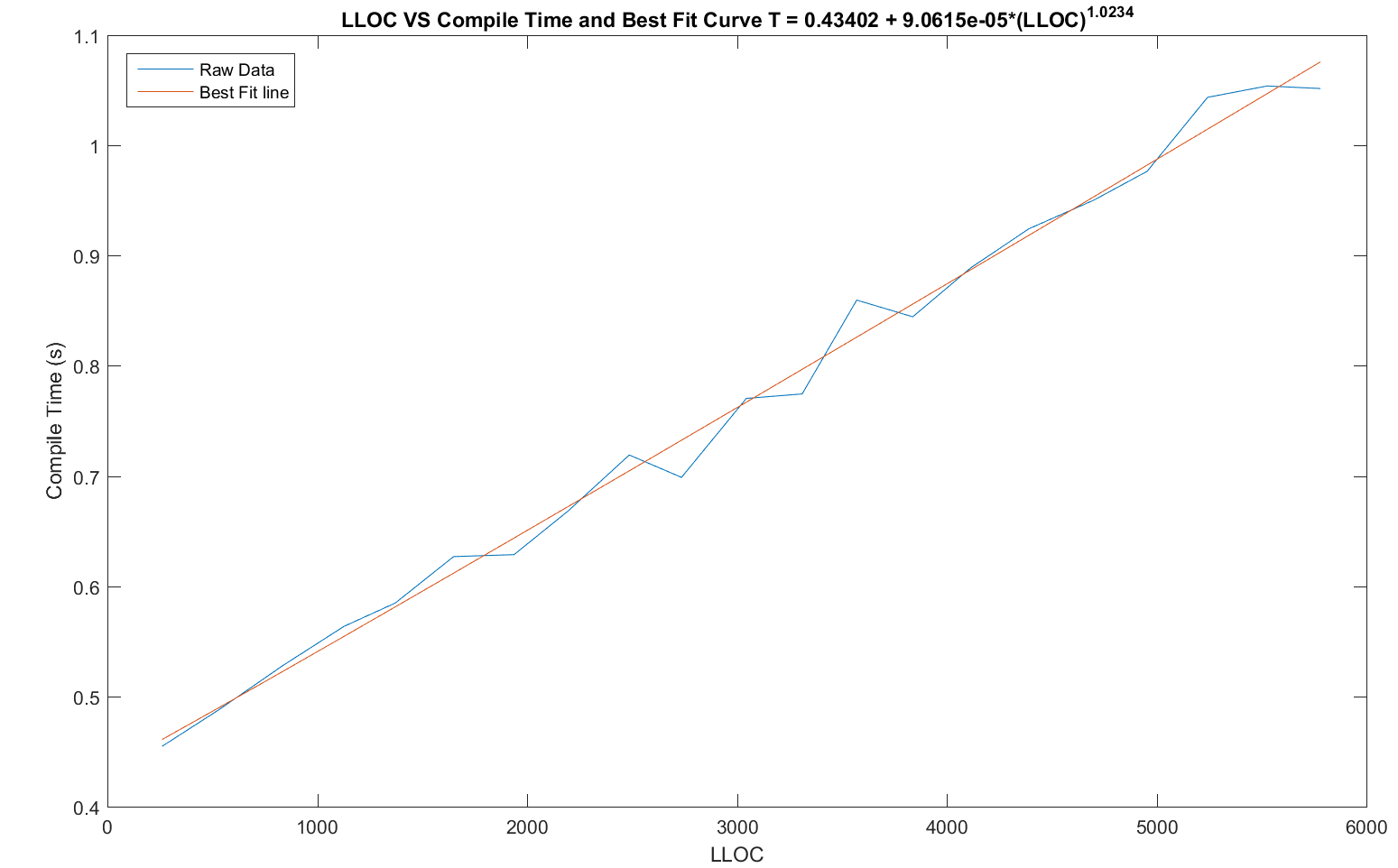
PLOC seems to be much more reliably related to program size for the test programs, but this may be coincidence as the data points are all significantly displaced from the best fit line. There is still a relatively dramatic (>25%) difference between the slopes of the best fit lines. Despite that, the predicted initial value for size of a program is approximately in line with the results from benchmaker. This suggests that PLOC may have a more direct correlation with executable size, even for programs of unusual composition.



This data shows how different Benchmaker programs can be in composition compared to their human-generated cousins. The human-made programs each solve a discrete problem, so their LLOC do not vary much, even if the program itself may change in size dramatically. For Benchmaker, LLOC and PLOC are very much linearly related, so an increase in one will result in an increase in the other. This causes the dramatic difference in the best fit lines.

This unusual best fit line could be explained by the absence of larger, more complex C++ programs. These programs would be larger in terms of both LLOC and PLOC. Even so, it’s evident that the Benchmaker best fit line does not pass through the middle of the cluster of human datapoints. This is most likely due to the fact that human-generated code will have additional documentation, and use more than one line of code to do variable initializations and assignments.

# VI. Relationship between LLOC and Compile Time



Compilation time is a key performance metric for any compiler or source program, since it determines how long rebuilding will take after a change is made. Sometimes compiler performance can vary dramatically, so being able to model it effectively can add value to the planning process. In this case, Benchmaker programs were used with the VC++ compiler to provide a dataset for determining compiler performance. The desired relationship is LLOC to compile time, so the test parser is used to determine the amount of code the compiler is actually processing. Once this is done, the programs are programmatically compiled by a Python script, which also measures the build time. Python has been previously established to utilize the Windows QueryPerformanceCounter library which offers sub-microsecond precision, so this is a satisfactory way to measure performance. A previous attempt to leverage the ‘/Bt’ for build time duration measuring proved unsuccessful, but measuring the duration of the entire build execution still gives a suitable result. Objectively, the experience of waiting for a build to complete is what is being measured here.

The data shows that compile time can be effectively modeled by a linear function. The data oscillates evenly around the fit line, with random offshoots that can be explained by random testing errors. A continuation of this test would be to compile each program multiple times, to acquire a more accurate overall average compile time for each program.

# VII. Different Compilers, Computers, and Release vs Debug

While the code itself makes a big difference on the ultimate performance of compiled executables, the compiler itself also has a large role in determining the efficacy of the resulting machine code. For the purposes of this experiment, two programs were written, one which sorts an array of ten million random numbers with the quicksort algorithm, and another which multiplies two 500x500 matrices together. These two workloads offer some variety to the test, as the sorting load is nonnumeric and recursive, whereas the matrix multiplication is numeric and iterative. In both cases, the tests are repeated 10 times and averaged to produce reliable results. These results may be referenced in the appendix.

Both programs were compiled in VC++ and GCC, in release as well as debug modes. VC++ compiler settings were set to x86 debug/release from within Visual Studio 2015. The command line parameters for GCC were as follows:

mingw32-c++.exe qsmark.cpp -std=c++11 -O2 -s -D\_\_NO\_INLINE\_\_ -DNDEBUG -o qsmarkRelease.exe

mingw32-c++.exe qsmark.cpp -std=c++11 -O0 -g -o qsmarkDebug.exe

Once this data was recorded, MatLab was used to compute the geometric ratio for the performance of GCC vs VC++, as well as the difference between results on two computers, a W541 with an Intel i7 CPU, and a T420s with a i5.

The release versions of each program was generally 1.5-5 times more performant than its debug counterpart. This is consistently true for both machines. Matrix multiplication benefited dramatically more from release mode than Quicksort, which is logical as any debug code attached the multiplication step will be executed O(n^3) times. A ratio was not produced for debug vs release, since this varied so much between the workloads.

Between the two compilers, VC++ runs the matrix test 2% slower than GCC, but runs sorting 12% faster. These are geometrics means of the performance, but the data itself shows that VC++ consistently produces faster sorting executables, especially in release mode.

The W541 runs matrix multiplication 20% faster than the T420, and is 15% faster in sorting. Since the T420 is actually clocked slightly faster, this is a bit surprising. The improvement is most likely due to the improved instruction set and other slight improvements in the heavy-duty workstation laptop, which cause it to be 17% faster overall. As to why the W541 is slightly faster at matrix multiplication, it is likely that the larger and faster CPU cache in the newer i7 gives it a slight edge for handling cache misses in the cumbersome O(n^3) matrix multiplication algorithm. By contrast, cache misses are less likely to create a problem in the faster quicksort algorithm which operates in O(nlogn).

# VIII. Conclusion

In the process of counting LLOC and PLOC for various programs, it is evident that there are many different approaches to this problem. Some are as simple as counting newline characters and semicolons, and some are as complex as using tools like Bison and Flex to create a true lexical analyzer. The approach taken in this paper does a satisfactory job of cleaning the code for counting with regular expressions, but more testing for edge cases is certainly needed, especially for more complex object oriented C++ with extra libraries and data types.

The benchmaker results were all generated with equal distributions of each feature type to be generated. A good direction for future research would be repeating the experiments with different configurations for benchmaker to produce, either focusing on conditional statements or loops, and observe the effect this has on properties such as executable size and LLOC.

Expanding the parser to accommodate more realistic and complex C++ programs would allow a large test set of programs to be analyzed. A major issue with the set tested in this paper is that the programs are of limited scope, and there are only five of them. This causes the resulting data set to be somewhat malformed, since the selected programs were themselves each written to solve a small discrete problem. Future tests with additional programs will alleviate this issue.

# IX. Appendix – Data

**Benchmaker CPPparse output:**

1,31,1,6,15,7,196,3,6,0,0,199,261,34

10,304,1,90,56,88,2129,44,102,0,0,2498,2736,336

11,343,1,100,61,98,2367,49,113,0,0,2779,3045,372

12,369,1,108,65,107,2578,54,125,0,0,3032,3311,405

13,395,1,116,69,114,2787,58,139,0,0,3280,3571,438

14,434,1,124,73,122,2989,62,149,0,0,3518,3838,468

15,460,1,134,78,132,3208,66,159,0,0,3778,4115,503

16,499,1,142,82,140,3419,71,170,0,0,4028,4392,534

17,525,1,152,88,152,3666,76,183,0,0,4322,4701,575

18,564,1,160,92,160,3855,80,191,0,0,4545,4955,603

19,590,1,170,96,169,4087,85,205,0,0,4823,5243,640

2,57,1,16,20,16,408,8,16,0,0,453,531,68

20,616,1,180,101,178,4313,90,216,0,0,5092,5527,675

21,655,1,188,105,186,4502,94,224,0,0,5315,5781,703

3,96,1,26,25,26,639,13,26,0,0,726,833,103

4,122,1,36,30,36,874,18,38,0,0,1006,1127,140

5,148,1,46,34,44,1065,22,46,0,0,1234,1372,170

6,187,1,54,38,52,1278,26,58,0,0,1485,1650,202

7,213,1,64,43,62,1507,31,70,0,0,1759,1938,239

8,252,1,70,47,70,1699,35,80,0,0,1984,2193,267

9,278,1,82,52,80,1933,40,91,0,0,2264,2487,305

CPPparse.py parses all cpp files in a directory, and outputs the results in the following format:

File#,commas in variable init,equals in variable init, ifCount, forCount, whileCount, semiCount, switchCount, caseCount, ANDCount, ORCount, PLOC, LLOC, cyclomaticComplexity

CPPparse is forward compatible to cyclomatic complexity analysis

CS600 C++ source code CPPparse output:

1,0,9,3,3,0,34,0,0,0,0,38,51,6

2,32,17,2,0,0,24,0,0,0,0,32,77,2

3,4,13,7,5,0,37,0,0,0,0,42,67,12

4,0,4,0,2,2,51,1,10,0,0,66,63,14

5,0,13,5,3,1,47,0,0,0,0,55,72,9

These files are parsed by Matlab into matrices for further analysis

**Compiler Test Outfile:**

1,8974,198144,0.454779724199

10,109516,263680,0.69876303981

11,121654,271360,0.770423801528

12,133436,280064,0.774477894417

13,146006,292352,0.85963478268

14,156596,299008,0.844487460024

15,168360,308224,0.888990931148

16,179661,315392,0.924386959574

17,192688,325632,0.950351628411

18,202755,331264,0.976459165534

19,215014,339968,1.04362080786

2,20456,205824,0.487866869748

20,227140,348672,1.05391512474

21,237172,355328,1.05159967321

3,32192,212992,0.527643371199

4,44358,221696,0.563523016984

5,54509,227840,0.584762768853

6,65482,235520,0.626903165682

7,77279,243712,0.628670955452

8,86861,249344,0.667859562117

9,99355,256512,0.719051978603

CompilerMetrics.py compiles all cpp files in a directory, and output the following properties for each source file:

File#, size of cpp source file (bytes), size of executable (bytes), compile time (seconds).

This file is parsed into a matrix in Matlab for further analysis.

Benchmaker Raw Output:

BM1benchmaker1.cpp 200 195 285

BM1benchmaker2.cpp 440 434 664

BM1benchmaker3.cpp 680 692 1065

BM1benchmaker4.cpp 920 955 1478

BM1benchmaker5.cpp 1160 1170 1814

BM1benchmaker6.cpp 1400 1405 2181

BM1benchmaker7.cpp 1640 1662 2588

BM1benchmaker8.cpp 1880 1874 2917

BM1benchmaker9.cpp 2120 2137 3332

BM1benchmaker10.cpp 2360 2356 3678

BM1benchmaker11.cpp 2600 2621 4090

BM1benchmaker12.cpp 2840 2857 4467

BM1benchmaker13.cpp 3080 3087 4833

BM1benchmaker14.cpp 3320 3311 5181

BM1benchmaker15.cpp 3560 3557 5561

BM1benchmaker16.cpp 3800 3791 5931

BM1benchmaker17.cpp 4040 4068 6366

BM1benchmaker18.cpp 4280 4279 6693

BM1benchmaker19.cpp 4520 4537 7106

BM1benchmaker20.cpp 4760 4790 7503

BM1benchmaker21.cpp 5000 5001 7830

This file must be sanitized so that matlab can parse it. This is done by cleanBMoutput.py, which produces the following result:

1,200,195,285

2,440,434,664

3,680,692,1065

4,920,955,1478

5,1160,1170,1814

6,1400,1405,2181

7,1640,1662,2588

8,1880,1874,2917

9,2120,2137,3332

10,2360,2356,3678

11,2600,2621,4090

12,2840,2857,4467

13,3080,3087,4833

14,3320,3311,5181

15,3560,3557,5561

16,3800,3791,5931

17,4040,4068,6366

18,4280,4279,6693

19,4520,4537,7106

20,4760,4790,7503

21,5000,5001,7830

This maintains the format from Benchmaker, but can be parsed into a matrix by Matlab.

# X. Appendix - Code