Lab 05 - Profiling

Due: October 29, 2016

1 Introduction

After countless adventures together, Marlin has finally made up his mind to express his love for Dory. However, the clever Clownfish is having trouble putting into words how he feels about his whimsical companion. Marlin decides to use his CS skills to compile a list of bigrams to compliment Dory with. Unfortunately, the only way to generate the perfect compliments would be to analyze the best 10 bigram compliments from every rom-com ever made! Because Marlin is an acquatic creature, his programming skills leave much to be desired. His code is too slow to go through all these works of art! It is your job to optimize the program to make sure that Marlin can woo his best friend.

2 Assignment

In this lab assignment you will accomplish two things:

1. Learn to use a profiler, `google-pprof`, to see where time is spent during the execution of a program.
2. Implement some mid-level program optimizations to improve the performance of a provided executable program.

You are provided a program, *words*, which processes the contents of a file into a hashtable of bigram[1] counts. In this lab, you will use `google-pprof` to determine where the performance of this program can be improved, and then make those improvements. The Makefile that you are given builds the *words* executable as well as another executable *words_avg*, which executes the routines of *words* 10 times, printing no output.

Use the *words* program by executing the following command in a terminal:

```
./words <filename>
```

This command will cause *words* to compute bigram counts over each word in the file indicated by `filename`. *words* prints these counts to `stdout`, but you can send its output to a file instead by using

```
./words <filename> > <output_file>
```

2.1 Handout

Run the terminal command

[1] A *bigram* is a sequence of two adjacent words.

```
cs033_install lab05
```

to install the files for this lab to your home directory.

The lab handout contains the following files:

ttable.h: Header file for the hashtable implementation.

ttable.c: Implementation of the hashtable.

words.h: Header file for the bigram-count code.

words.c: Implementation of the bigram-count code.

file_io.h: Header file for code that reads the contents of a file into a character array.

file_io.c: Implementation of code that reads file contents into a character array.

main.c: Runs the bigram-counting code and prints its output to `stdout`.

main_avg.c: Runs the bigram-counting code 10 times.

words_base: The initial version of the *words* executable.

words_base_avg: The initial version of the *words_avg* executable.

words_fast: An optimized version of the *words* executable.

words_fast_avg: An optimized version of the *words_avg* executable.

Makefile: A makefile.

plays.txt: A test input file, containing the entire works of Shakespeare.

lab05.pdf: This document.

You will principally be working with the *words.c* file.

2.2 Optimizations

There are two main optimizations that you are expected to make in this lab. Each of these pertains to code contained within the *words.c* file. Feel free to examine the other C files, although doing so will not help with finding the optimizations in *words.c*.

Hint: take a look at how many times each character in the file is being iterated over.

3 Profiling With *google-pprof*

After using `make` to create the executable *words*, you will be able to begin profiling. To get a profile, run your program with the terminal command

```
CPUPROFILE=<profile_file> ./words plays.txt > plays.out
```

where <*profile_file*> is a file location of your choosing. The contents of this file will be completely overwritten with profiling information of the execution of *words*, or a new file will be created if it does not already exist.

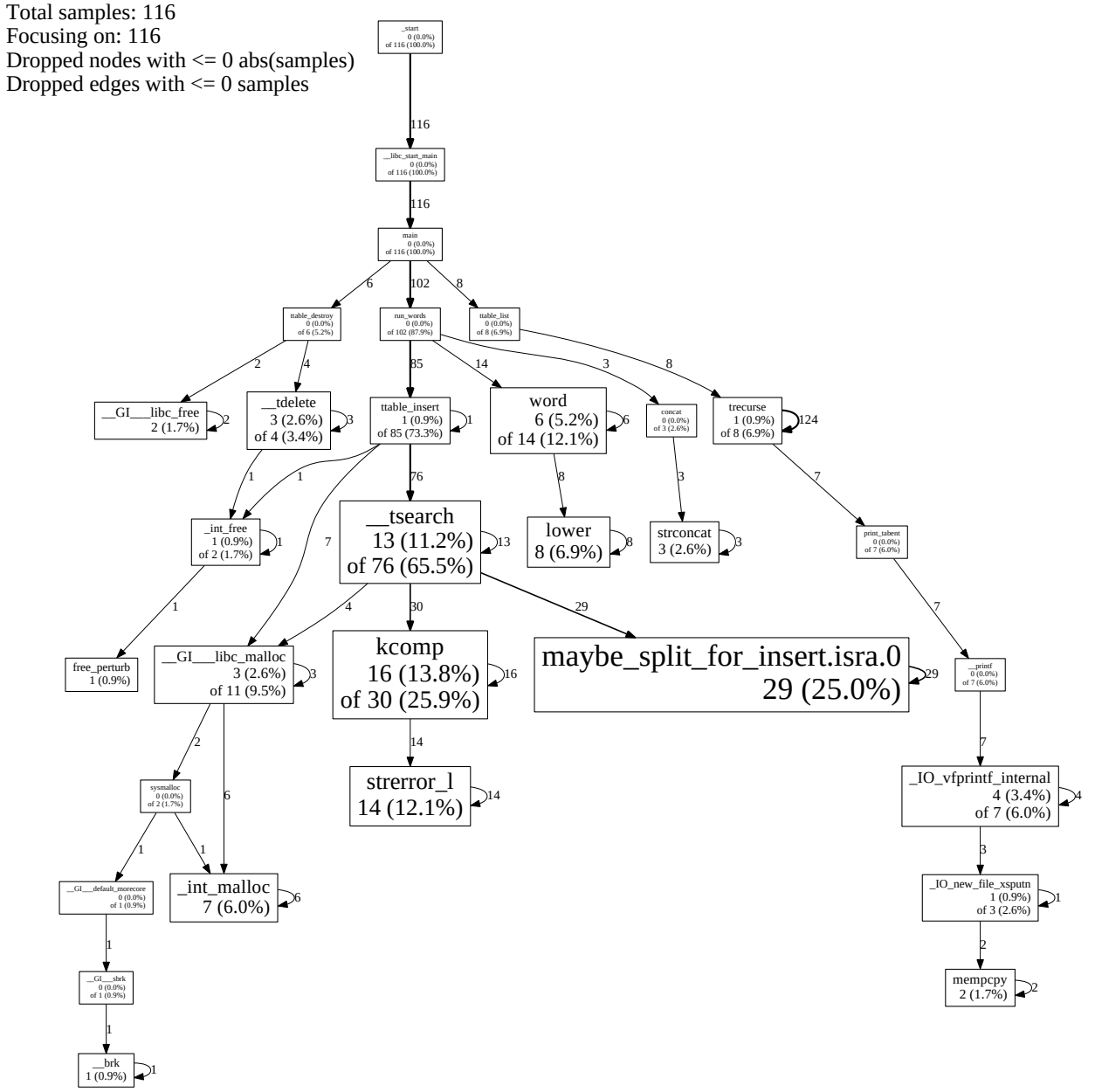
Once you have run the above and generated profile information, run the following command to generate a *callgraph*

```
google-pprof --pdf ./words <profile_file> > callgraph.pdf
```

This command will create a *callgraph*, i.e. a visual representation of the information contained within the profile file. Providing `google-pprof` the `--pdf` option instructs it to generate output in the *pdf* file format - you can view the resulting graph in any PDF viewer such as *evince* or *okular*.

4 Reading the Callgraph

A callgraph generated by `google-pprof` uses a graph to measure time spent in each part of your program. Functions are represented by boxes, which are sized according to their weight in the process (the amount of time spent within that function). A larger box will correspond to more time spent in a function. Edges between boxes are labeled with the number of samples that went between the two functions.



Above is the callgraph for the *words* program before any optimiziations are made. We can see that the program has spent a lot of time in `__tsearch` and `maybe_split_for_insert`. We can figure out some basic statistics about those functions from the callgraph.

`google-pprof` figures out how much time is spent in each function by taking snapshots, or samples, of the current call stack at certain points over the course of the execution of the program. This provides a kind of Monte Carlo estimation for what functions are the most expensive; functions which take up more time in the execution are highly likely to have more samples taken when they are running. The numbers in each box are related to the number of samples each function was executing in.

First, we can see that `maybe_split_for_insert` is a **leaf** function, in that it doesn't call any other functions. Thus, any time spent in the function will be executing that function's body. Because of this, the box for `maybe_split_for_insert` only has one number related to it: in this case, 29. This means that `pprof` took 29 samples while that function was executing. Additionally, this accounts for 25.0% of all samples taken.

`__tsearch` is a little more intricate, and has two numbers associated with it. We can see that there isn't just one sample number, but instead it is in terms of "X of Y". Because `__tsearch` calls other functions, `pprof` split out the number of samples where code from `__tsearch` was executing (13, which is the X), versus the number of samples in functions called by `__tsearch`, including `__tsearch` itself (76, which is the Y). Thus, while a significant fraction of the program is spent running code that `__tsearch` is responsible for, only some of that time is spent within `__tsearch` directly.

Finally, there is an edge from `__tsearch` to `maybe_split_for_insert`. This means that `maybe_split_for_insert` is called by `__tsearch`. The edge is also annotated with the number 29, meaning that 29 samples were taken where `__tsearch` called `maybe_split_for_insert`. In this way, the whole call graph can be presented at once, in a relatively straightforward manner.

If no sample enters or begins in a function, that function will not appear in the callgraph. `pprof` may also filter out functions which it considers to be inconsequential, usually because they are not called very often.

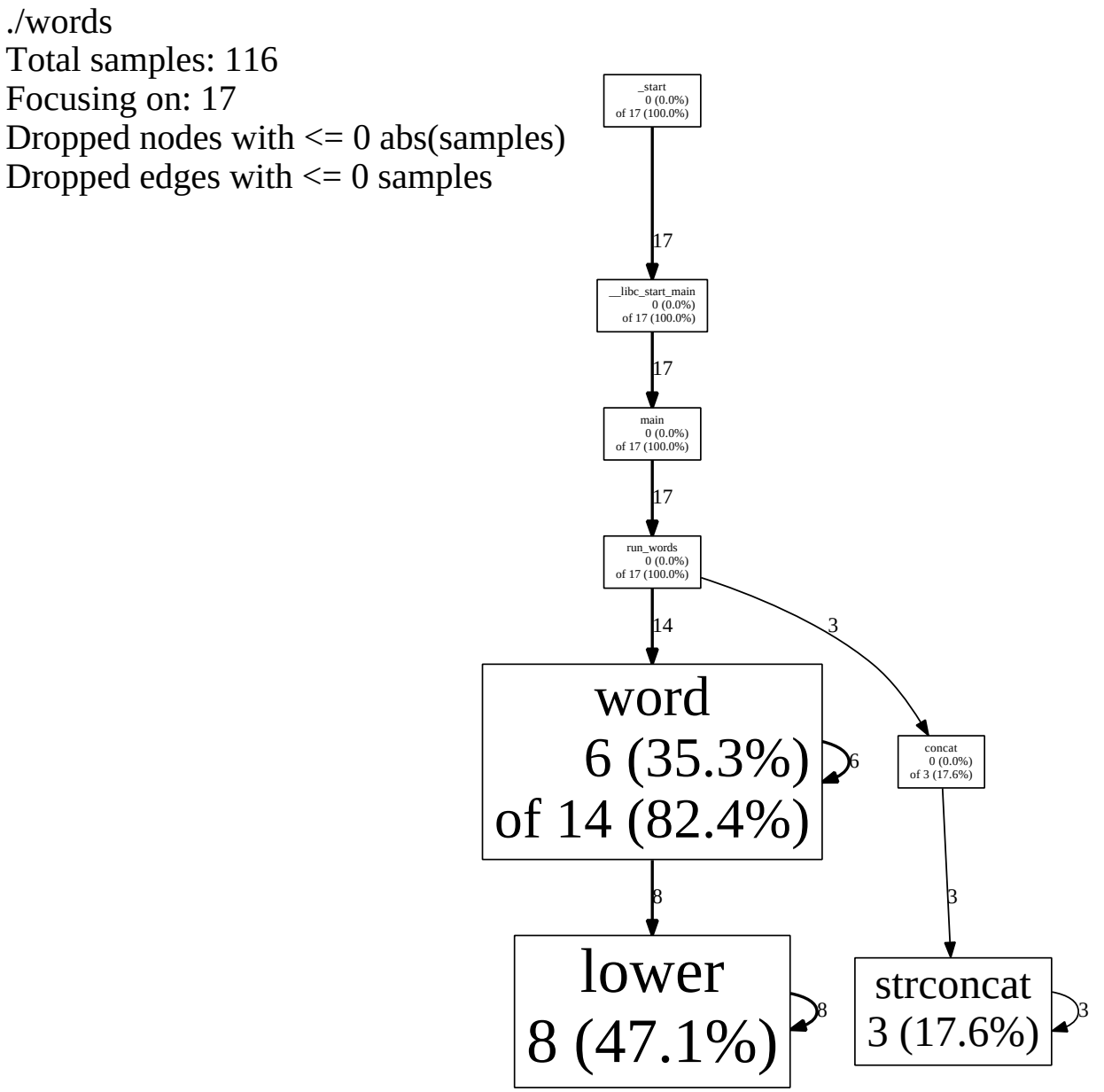
The callgraph above contains a lot of information, but not all of that information is relevant to your interests in this lab, or what you can do to improve the program. Fortunately, `google-pprof` has options to ignore functions that you are not interested in:

```
google-pprof --pdf --ignore=<regexp> <program> <profile_file> > callgraph.pdf
```

The `--ignore=<regexp>` option filters out all functions matching `<regexp>` from the callgraph. For this lab, the regular expression `'ttable'` will eliminate information that you don't need - run

```
google-pprof --pdf --ignore='ttable' ./words <profile_file> > callgraph.pdf
```

Doing this gives a new graph with only the most pertinent information:



5 Reading Text Output

To get instant feedback about how quickly `words` is running, instead of creating a callgraph each time, just generate some text mode statistics with the following command:

```
google-pprof --text ./words <profile_file>
```

This will return for you the following information in colomns[2]:

1. Number of profiling samples in this function
2. Percentage of profiling samples in this function
3. Percentage of profiling samples in the functions printed so far
4. Number of profiling samples in this function and its callees
5. Percentage of profiling samples in this function and its callees
6. Function name

6 Using `google-pprof` In Your Own Projects

`google-pprof` is a tool that you will likely find helpful on future projects both in and out of this course. Fortunately, it is quite simple to integrate it with a project: all you have to do is add the `-lprofiler` flag when you compile your project. In projects for this course you can do this by adding this option to the `CFLAGS` variable in the provided Makefile.

Programs compiled with `-lprofiler` will take samples of the CPU time and save information in the file indicated by the `CPUPROFILE` environment variable. Be sure to set this variable every time you want to profile your program.

7 Measuring Performance with `time`

`time` is a command that measures how long it takes for another program to run. You can use it in this lab to measure the performance of your *words_avg* program against *words_base_avg* and *words_fast_avg* to see how you're doing.

`time` is used as follows:

```
time <program> <arg1> <arg2> ... <argn>
```

For example, to time your *words* program, you would run

```
time ./words plays.txt
```

[2] http://google-perftools.googlecode.com/svn/trunk/doc/cpuprofile.html

During the lab, we recommend using `time` to measure the performances of *words_avg*, *words_base_avg*, and *words_fast_avg* instead of the executables that run only once. The code of *words* performs a lot of input and output operations, which dilutes the time contribution of the bigram-counting code. The *avg* executables perform these input and output operations only once, and then process the input ten times, so that more of the time spent executing them is spent executing the code that you are trying to optimize.

`time` lists the *real* time, time spent in *user* mode, and time spent in *sys*tem mode. *Real* time is the time from start to finish of your program, and is what you should use to measure the performance of your program. User and system time are related to operating systems concepts which will be covered later in the course.

8 Getting Checked Off

Once you have improved the performance of *words* to be near that of the provided faster version, submit your work using the handin script:

```
33lab_checkoff lab05 [--verbose]
```

This may take some time to run (20-30s), so don't worry if it hangs. It will test for both correctness of output and performance (your code should be within 10% of the fast version). If the `python3 /course/cs033/static/lab_tests/lab05/prof_compare.py` test fails, this is because your bigram did not match the expected bigram. If the `python3 /course/cs033/static/lab_tests/time_checker.py` test fails, this is because your code was not fast enough. Run with the –verbose flag for more detailed output.

Remember to read the course missive for information about course requirements and policies regarding labs and assignments.