ECE297 Performance Profiling Guide

“If you can not measure it, you can not improve it.”
— Lord Kelvin

1 Introduction

This guide describes how to profile (or measure) a program to determine which parts consume large amounts of CPU time. This is key to identifying where to invest effort in improving a program’s performance.

Programmers often believe they know where the performance problems are located in their programs, but they are often wrong. This leads to pre-mature optimization — wasted effort spent ‘optimizing’ code which is not performance critical. As a result, it is important to actually measure which parts of a program are slow, a process known as profiling. This also has the important benefit of allowing you to re-measure after you have made a modification to see if you have actually made performance better, or worse.

This guide describes how to generate and analyze profiling data to identify the performance critical hot-spots in your program. Once these have been identified you can focus your time improving performance where it really matters.

If you just want to know how to run profiling on your project see Section 4.

For a tutorial on how to use Gprof to find hot-spots in your code see Section 3. In particular Section 3.2 describes how to interpret the call graph. Finally Section 5 provides pointers to some other resources for profiling.

2 Configuring Gprof

Gprof (the GNU Profiler) is a profiler provided with the GCC compiler. When using Gprof, your program automatically records additional information about how many times functions are called$^1$, and how much time is spent in them$^2$.

There are several steps to using Gprof:

Compiling with profiling: We must re-compile and link the program with the -pg option.

This causes g++ to include the extra code required to track function calls and other profiling information. In ECE297, you can simply select the Profiling configuration in NetBeans for your project and rebuild; this build configuration is set up to pass the correct options to g++ for profiling.

Running the executable: Next, we run the executable as usual.

---

$^1$ Gprof does this by modifying function entry and exit points to record how often this occurs.

$^2$ This is done by sampling — periodically stopping program execution and checking which function was running.
On exit the profiling information will be written to the file `gmon.out`.

### Processing the profiling data

Finally, we convert `gmon.out` into a human-readable form using the `gprof` utility.

To illustrate the usage of Gprof, we will use a simple example program.

## 3 Tutorial: Identifying Hot-spots with Gprof

The example code used in this section is available for download to try yourself.

Let’s consider a simple function which extracts the negative values from an input vector and returns them. One implementation is shown in Listing 1.

```cpp
std::vector<int> extract_negatives_vector(std::vector<int>& numbers) {
    std::vector<int> negatives;
    std::vector<int>::iterator iter = numbers.begin();
    while(iter != numbers.end()) {
        if(*iter < 0) {
            negatives.push_back(*iter);
            iter = numbers.erase(iter); //Returns next iter
        } else {
            iter++; //Next element
        }
    }
    return negatives;
}
```

Listing 1: The `extract_negatives_vector()` method in `src/extract_negatives.cpp`

This implementation loops through the `numbers` copying negative values into `negatives` (line 6) and removing them from `numbers` (line 7) using the `.erase()` method.

We can compile our program with `make` and provide the argument `vector` to the executable. This will run the `extract_negatives_vector()` function on a vector of 800000 random numbers.

```bash
# In the demo program folder
> ls
Makefile src

# Build the demo program
> make
...

# Test the vector method
> ./extract_negatives_vector
Extracting Negatives using ‘vector’ method ....
Took 32165 ms

# Few... that took a long time!
```

Listing 2: Testing the vector method.
3.1 Generating Profiling Data

This guide uses `make CONF=profile` to disable function inlining to generate a clearer call graph. This can make some operations (e.g. iterator operations, vector accesses) appear slow as usually in a release configuration the compiler would inline (paste in) the code of these short functions into the calling function to avoid the overhead of a function call. While inlining speeds up program that contain many short functions, it makes it harder to interpret profiling results as some functions have disappeared; hence we turn function inlining off in the profile configuration.

As Listing 2 shows the `extract_negatives_vector()` function is very slow, taking over 30 seconds to run.

To figure out what is making the function slow we can use Gprof. We first compile with profiling enabled and then run the executable as usual (Listing 3).

```
# Create a profiling build (i.e. g++ ... -pg)
> make CONF=profile

# Run the new executable
> ./extract_negatives vector
Extracting Negatives using 'vector' method ....
Took 32414 ms

# Note the new 'gmon.out' file
> ls
build extract_negatives gmon.out Makefile src
```

Listing 3: Testing the vector method.

This will generate a new file `gmon.out` which contains all the profiling information.

One of the best ways to interpret Gprof’s output is to visualize it as a call graph. The command in Listing 4 will generate an interactive call graph like the one shown in Figure 1.

```
gprof extract_negatives gmon.out | gprof2dot.py -s | xdot -
```

Listing 4: Visualizing gprof data. Note that `extract_negatives` is the name of the executable.

3.2 The Call Graph

The call graph represents the relationship between functions in your program. Nodes represent functions, and directed edges indicate that the parent (caller) has called the child (callee).

Figure 2 is used to illustrate the profiling information annotated on the call graph.

Nodes are annotated with:

1. The total-time spent in the function – including its descendants (e.g. 100.00% in `main()`). This also determines the node’s colour.
2. The self-time, the time spent executing statements in the function – excluding descendants (e.g. 9.09% in `main()`).
Figure 1: Call graph from testing `extract_negatives_vector()`.
3. The number of times the function was called (e.g. `extract_negatives_vector()` was called 1×).

**Directed edges** are annotated with:

1. The percentage of total-time incurred in the child when called by the parent (e.g. `std::vector::push_back()` accounted for 20.20% of total-time when called from `main()`).
2. The number of times the parent called the child (e.g. `main()` called `std::vector::push_back()` 800000×).

3.3 Identifying the bottle-neck in `extract_negatives_vector()`

Since `extract_negatives_vector()` accounted for most of the run time in `main()` we will look at it in more detail. Figure 3 shows that most of the total-time in `extract_negatives_vector()` was spent in the `std::vector::erase()` method. Why is the `std::vector::erase()` method so slow?

Looking at the STL documentation shows that `std::vector::erase()` takes $O(n)$ time\(^3\). This means `extract_negatives_vector()` takes $O(n^2)$ time. No wonder it was slow!

3.4 A faster method to extract negatives

Now that we’ve identified `std::vector::erase()` as the problem, can we come up with a faster alternative?

To make the extract negatives operation fast (i.e. $O(n)$) we need a fast erase operation. There are a variety of data structures that support fast insertion and deletions. One such data structure

\(^3\)Erasing an element requires shifting all subsequent elements in the vector.
Figure 3: Cropped view of call graph in Figure 1 around `extract_negatives_vector()`.

is a linked-list. Sure enough the STL documentation shows that `std::list::erase()` is an $O(1)$ operation.

It is relatively straightforward to implement `extract_negatives` with `std::list` instead of `std::vector`:

```cpp
1 std::list<int> extract_negatives_list(std::list<int>& numbers) {
2   std::list<int> negatives;
3   std::list<int>::iterator iter = numbers.begin();
4   while(iter != numbers.end()) {
5       if(*iter < 0) {
6           negatives.push_back(*iter);
7           iter = numbers.erase(iter); //Returns next iter
8       } else {
9           iter++; //Next element
10       }
11   }
12   return negatives;
13 }
```

Listing 5: The `extract_negatives_list()` function in `src/extract_negatives.cpp`

We can test this method by providing `list` as the argument to the `extract_negatives` executable:

```bash
1 #Test the `extract_negatives_list()` method
2 > ./extract_negatives list
3 Extracting Negatives using 'list' method ....
4 Took 154 ms
5 #Now thats fast!
```
Listing 6: Testing the list method.

Using lists instead of vectors has sped our code up from 32165 ms to 154ms - that’s a speed-up of over $382 \times$! Clearly for any moderately sized dataset the difference between an $O(n)$ and $O(n^2)$ can be drastic.

While we might be happy with our new implementation, we can still profile to see where time is spent. Figure 4 shows much of the time is spent in low level STL functions (e.g. `_M_create_node`, `allocate`). From their names, we can infer they are related to creating and destroying nodes in the linked list. Since STL is generally pretty efficient and well implemented there doesn’t seem to be much we can do.

3.5 But can we go any faster?

What if 154 ms is not fast enough? At this point we need to re-consider the algorithmic approach we are using. The example code includes one more method for extracting negatives called `extract_negatives_fast()`, which uses a slightly different algorithm to performing negative extraction. You can check out the code in `src/extract_negatives.cpp` to see how it works!

We can test the new approach with the `fast` option.

```
$ ./extract_negatives fast
Extracting Negatives using ‘fast’ method ....
Took 155 ms
#No faster!
```

Listing 7: Testing the fast method.

The new algorithm wasn’t any faster – it took 155 ms, about the same as the list method. We should never decide what algorithm is fastest without fully enabling compiler optimizations, including function inlining, however.

We can can fully enable compiler optimizations (including function inlining) by using a `release` build.

```
$ make CONF=release
...
$ ./extract_negatives list
Extracting Negatives using ’list’ method ....
Took 21 ms
# Really fast!
$ ./extract_negatives fast
Extracting Negatives using ’fast’ method ....
Took 5 ms
#Blazingly fast, and now faster than list!
```

Listing 8: Testing the fast method.

With optimization it takes only 5ms, and the fast method now outperforms the list method. That’s a speed-up of over $6000 \times$ from our initial implementation!
Figure 4: Call graph from testing `extract_negatives_list()`.
3.6 Is that it?

5 ms is pretty good, particularly compared to our first implementation which took over 30 seconds. However, could we do any better? Almost certainly the answer is yes. Performance optimization is a deep rabbit hole, and there are many techniques that we still haven’t applied. However the fast method is performant and remains clear and readable. Further optimizations would likely make the code increasingly difficult to read and maintain, for (likely) smaller performance improvements.

4 Profiling your Project

This section describes how you can perform performance profiling on your project, using either Unit Tests (Section 4.1) or the main executable (Section 4.2).

4.1 Profiling Unit-Tests

Profiling the unit tests in your project can be done in a manner similar to Section 3:

```
# In the project directory
> ls
Makefile libstreetmap libstreetsdatabase main test_libstreetmap
build libstreetmap.a libstreetsdatabase.a nbproject

# Builds the unit test executable ‘test_libstreetmap’
# with profiling enabled and runs it
> make CONF=profile test
...

# Note the new file ‘gmon.out’
> ls
Makefile gmon.out libstreetmap.a libstreetsdatabase.a nbproject
build libstreetmap libstreetsdatabase main test_libstreetmap
```

Listing 9: Profiling Unit Tests

You can then visualize the call graph as shown in Listing 10.

```
gprof test_libstreetmap gmon.out | gprof2dot.py -s | xdot -
```

Listing 10: Visualizing unit test Gprof data

Alternately you can write the gprof output to a text file as shown in Listing 11.

```
gprof test_libstreetmap gmon.out > gprof.log
```

Listing 11: Writing Gprof unit test data to a text file ‘gprof.log’

4.2 Profiling Your Executable

First build your executable in profiling mode, and then run it:

```
# In the project directory
> ls
Makefile libstreetmap libstreetsdatabase main test_libstreetmap
```

Page 9 of 10
```plaintext
build libstreetmap.a libstreetsdatabase.a nbproject

#Builds the main executable `mapper'
> make CONF=profile
...

#Run the executable to generate `gmon.out'
> ./mapper

Listing 12: Profiling the mapper executable

You can then visualize the call graph as shown in Listing 13.

gprof mapper gmon.out | gprof2dot.py -s | xdot -

Listing 13: Visualizing mapper Gprof data

Alternately you can write the gprof output to a text file as shown in Listing 14.

gprof mapper gmon.out > gprof.log

Listing 14: Writing mapper Gprof data to a the text file `gprof.log'

5 Additional Resources

- The Gprof manual provides more detail on using Gprof, and how it works.
- gprof2dot.py -h lists the various options available for customizing the displayed call graph.
- An alternative to Gprof is Valgrind’s profiler Callgrind.

The main advantage of Callgrind is that it provides an exact (rather than sampled) call graph. If some functions are not showing up in Gprof (usually because they are very fast and don’t get sampled) callgrind may help. However Callgrind is slower than Gprof, and only reports performance in number of instructions (instead of actual execution time). It does not require a specialized build like Gprof.

It is run much like valgrind for memory checking:

valgrind --tool=callgrind ./test_libstreetmap

Listing 15: Profiling with unit tests with Callgrind

It will generate a file callgrind.out.XXXX, where XXXX is the PID (Process ID) of the program. You can then visualize the call graph using:

gprof2dot -f callgrind callgrind.out.XXXX | xdot -

Listing 16: Visualizing Callgrind’s profiling results