# Profiling with Gprof

## Brief

Profiling is a mechanism for identifying program 'hotspots'; areas where your program spends a good portion of it's time. These regions are prime candidates for optimizations, where concentration on efficiency will give you the most bang for your buck. Gprof is a GNU profiling tool available for a host of platforms and tightly integrated with the ever-so-popular Gcc compiler. This paper will walk through some basic information to utilize Gprof to identify program hotspots, optimize these regions and with-luck note the performance gains.

## Introduction

Delivery of high-performance applications requires a critical step of evaluating the performance of the application. While care in design and implementation with concentration on optimization is essential in delivering high-performance applications profiling the source after implementation may identify regions where code optimization may show the greatest benefits. This paper will demonstrate the procedure for profiling an application, identify program 'hotspots' and optimize these hotspots with intent to show performance gains. The example application is an example produced for an independent course with no artificially introduced optimization candidates. With luck, at papers end the application will be optimized and demonstrate a performance gain.

## Discussion

Profiling with Gprof consists of 3 phases: 1) compiling your application with profiling enabled, 2) executing the application to gather profiling metrics, and 3) evaluation of the collected metrics.

### *Compiling*

Enabling profiling requires compiling your application with Gcc much as you'd normally do. The difference however is the introduction of two flags that you may not normally provide (*-pg* and *-g*). The *-pg* option enables profiling, the *-g* option introduces debugging symbols. The *-g* option is not too uncommon, but has particular significance when profiling as it provides the required debugging symbols for line-by-line profiling. The absence of the *-g* flag will result in the profiler gathering and reporting measures on a per function basis. Since we are interested in profiling on a line-by-line basis, we will provide both flags; as follows:

```
$ gcc -std=c99 -g -pg demo.c -ltheora -lm -o demo
```

### *Execution*

Execution of your application takes the same form as your normally run it. You should specify the same command line arguments, your inputs and outputs are identical as that while running without profiling enabled. The only possibly observed difference is that your program may run slower as it acquires and tallies your profiling measures. It is worth noting that your application must terminate in a normal fashion, to include returning from the *main()* function, or via calling *exit(),* abrupt termination in a non-typical fashion will interrupt profiling and result in no profiling information collected.

The normal termination of your program will result in the profiler generating an output file *(gmon.out)* just prior to exiting. The location of this output file is generally in the current working directory, however if your application makes a habit of changing directories you may file this output file in the present working directory of your application prior to terminating. The absence of this output file hints at the lack of providing the *-pg* option during compilation, or the possibility of your application terminating in a non-normal manner.

```
$ ./demo
```

### *Evaluation*

Now that your application has been compiled with profiling enabled, executed your application in a nominal fashion, and profiling metrics have been collected. The final step in profiling is the evaluation of the collected metrics. This is explicitly where the *gprof* utility is used. The default report style is done simply by executing *gprof* while specifying the executable as follows:

```
$ gprof demo
```

This command will evaluate the profiler metrics and generate a default profiler report. We have yet to see the example application source, as it becomes relevant it is listed as follows:

```
// example used for Theora quick survey paper
#include <stdio.h>
#include <stdlib.h>
#include <ogg/ogg.h>
#include <theora/theora.h>
#include <string.h>
#include <math.h>
#include <assert.h>

// this method encodes the specified frame and writes to the ogg file
void writeFrame( FILE *fp, theora_state *theoraState,
                 ogg_stream_state *ogg_os,
                 unsigned w, unsigned h, unsigned char *yuv) {
  // write the frame
  yuv_buffer yuv_buf;
  ogg_packet op;
  ogg_page og;

  unsigned long yuv_w;
  unsigned long yuv_h;

  unsigned char *yuv_y;
  unsigned char *yuv_u;
  unsigned char *yuv_v;

  unsigned int x;
  unsigned int y;

  /* Must hold: yuv_w >= w */
  yuv_w = (w + 15) & ~15;

  /* Must hold: yuv_h >= h */
```

```
  yuv_h = (h + 15) & ~15;

  yuv_y = (unsigned char*)malloc(yuv_w * yuv_h);
  yuv_u = (unsigned char*)malloc(yuv_w * yuv_h / 4);
  yuv_v = (unsigned char*)malloc(yuv_w * yuv_h / 4);
  yuv_buf.y_width = yuv_w;
  yuv_buf.y_height = yuv_h;
  yuv_buf.y_stride = yuv_w;
  yuv_buf.uv_width = yuv_w >> 1;
  yuv_buf.uv_height = yuv_h >> 1;
  yuv_buf.uv_stride = yuv_w >> 1;
  yuv_buf.y = yuv_y;
  yuv_buf.u = yuv_u;
  yuv_buf.v = yuv_v;

  for(y = 0; y < yuv_h; y++) {
    for(x = 0; x < yuv_w; x++) {
      yuv_y[x + y * yuv_w] = 0;
    }
  }
  for(y = 0; y < yuv_h; y += 2) {
    for(x = 0; x < yuv_w; x += 2) {
      yuv_u[(x >> 1) + (y >> 1) * (yuv_w >> 1)] = 0;
      yuv_v[(x >> 1) + (y >> 1) * (yuv_w >> 1)] = 0;
    }
  }

  for(y = 0; y < h; y++) {
    for(x = 0; x < w; x++) {
      yuv_y[x + y * yuv_w] = yuv[3 * (x + y * w) + 0];
    }
  }

  for(y = 0; y < h; y += 2) {
    for(x = 0; x < w; x += 2) {
      yuv_u[(x >> 1) + (y >> 1) * (yuv_w >> 1)] =
        yuv[3 * (x + y * w) + 1];
      yuv_v[(x >> 1) + (y >> 1) * (yuv_w >> 1)] =
        yuv[3 * (x + y * w) + 2];
    }
  }

  assert(0 == theora_encode_YUVin(theoraState, &yuv_buf));
  assert(0 != theora_encode_packetout(theoraState, 0, &op));

  ogg_stream_packetin(ogg_os, &op);
  if(ogg_stream_pageout(ogg_os, &og)) {
    fwrite(og.header, og.header_len, 1, fp);
    fwrite(og.body, og.body_len, 1, fp);
  }

```

```
  free(yuv_y);
  free(yuv_u);
  free(yuv_v);
}

int main() {
  // frame dimensions
    const unsigned Width = 200;
    const unsigned Height = 200;
  // frames per second
    const unsigned Fps = 10;
  // video quality 0-63 (63-highest quality)
    const int VideoQuality = 63;
    const unsigned NumFrames = 30 * Fps;  // 30-sec video

  // this is the codec's internal state and context
    theora_state theoraState;

  // Info Header
    theora_info encoderInfo;
    theora_info_init(&encoderInfo);

    encoderInfo.width = ((Width + 15) >>4)<<4;  // encoder frame must
                                                // be defined
    encoderInfo.height = ((Height + 15)>>4)<<4; //  in multiples of 16
    encoderInfo.frame_width = Width;
    encoderInfo.frame_height = Height;
    encoderInfo.offset_x = 0;
    encoderInfo.offset_y = 0;
    encoderInfo.fps_numerator = Fps;
    encoderInfo.fps_denominator = 1;
    encoderInfo.aspect_numerator = Width;
    encoderInfo.aspect_denominator = Height;
    encoderInfo.colorspace = OC_CS_UNSPECIFIED;
    encoderInfo.pixelformat = OC_PF_420;
    encoderInfo.target_bitrate = 0;
    encoderInfo.quality = VideoQuality;
    encoderInfo.dropframes_p = 0;
    encoderInfo.quick_p = 1;
    encoderInfo.keyframe_auto_p = 1;
    encoderInfo.keyframe_frequency = 64;
    encoderInfo.keyframe_frequency_force = 64;
    encoderInfo.keyframe_data_target_bitrate = 0;
    encoderInfo.keyframe_mindistance = 8;
    encoderInfo.noise_sensitivity = 1;

  // initialize the codecs state
    assert(0 == theora_encode_init(&theoraState, &encoderInfo));

  // Comment Header
    theora_comment tc;
```

```
    theora_comment_init(&tc);
    theora_comment_add(&tc,"AUTHOR=me");
    theora_comment_add(&tc,"BRIEF=demo generated file");

    FILE *fp = fopen("demo.ogg", "wb");
    {
      ogg_stream_state ogg_os;
      ogg_packet op;
      ogg_page og;

      assert(0 == ogg_stream_init(&ogg_os, rand()));

      // encode info header in data packet
      theora_encode_header(&theoraState, &op);
      ogg_stream_packetin(&ogg_os, &op);
      if(ogg_stream_pageout(&ogg_os, &og)) {
        fwrite(og.header, og.header_len, 1, fp);
        fwrite(og.body, og.body_len, 1, fp);
      }

      // encode comment header in data packet
      theora_encode_comment(&tc, &op);
      ogg_stream_packetin(&ogg_os, &op);
      if(ogg_stream_pageout(&ogg_os, &og)) {
        fwrite(og.header, og.header_len, 1, fp);
        fwrite(og.body, og.body_len, 1, fp);
      }

      // now encode the tables
      theora_encode_tables(&theoraState, &op);
      ogg_stream_packetin(&ogg_os, &op);
      if(ogg_stream_pageout(&ogg_os, &og)) {
        fwrite(og.header, og.header_len, 1, fp);
        fwrite(og.body, og.body_len, 1, fp);
      }

      // flush all headers to packet
      if(ogg_stream_flush(&ogg_os, &og)) {
        fwrite(og.header, og.header_len, 1, fp);
        fwrite(og.body, og.body_len, 1, fp);
      }

      for (int i=0; i<NumFrames; ++i) {
        unsigned char yuv[Width*Height*3];
        memset(yuv,0,Width*Height*3);
        const double R = 255.0;
        const double G = 255.0;
        const double B = 255.0;
        const double dY = (0.257 * R) + (0.504 * G) + (0.098 * B) + 16;
        const double dU = -(0.148 * R) - (0.291 * G) + (0.439 * B) + 128;
        const double dV = (0.439 * R) - (0.368 * G) - (0.071 * B) + 128;
```

```
      const unsigned char Y = (unsigned char)dY;
      const unsigned char U = (unsigned char)dU;
      const unsigned char V = (unsigned char)dV;

      // generate the frame background with specified color
      {
       unsigned i;
       for(i = 0; i < Width*Height*3; i+=3) {
         yuv[i] = Y;
         yuv[i+1] = U;
         yuv[i+2] = V;
       }
      }

     // -- generate spinning dot of a foreground color --
     {
      const double R = 0.0;
      const double G = 0.0;
      const double B = 0.0;
      const double dY = (0.257 * R) + (0.504 * G) + (0.098 * B) + 16;
      const double dU = -(0.148 * R) - (0.291 * G) + (0.439 * B) + 128;
      const double dV = (0.439 * R) - (0.368 * G) - (0.071 * B) + 128;
      const unsigned char Y = (unsigned char)dY;
      const unsigned char U = (unsigned char)dU;
      const unsigned char V = (unsigned char)dV;
      const unsigned cX = Width/2;
      const unsigned cY = Height/2;
      const unsigned Radius = 50;
      static double theta = 0.0;
      const unsigned x = (unsigned)(Radius * sin(theta) + cX);
      const unsigned y = (unsigned)(Radius * cos(theta) + cY);
      const unsigned k=3*(x + (Width*y));
      theta -= 5.0 * 3.14159/180.0;
      yuv[k] = Y;
      yuv[k+1] = G;
      yuv[k+2] = B;
     }

     writeFrame(fp,&theoraState,&ogg_os,Width, Height, yuv);
    }

    // prepare and close up the packet
    theora_encode_packetout(&theoraState, 1, &op);
    if(ogg_stream_pageout(&ogg_os, &og)) {
     fwrite(og.header, og.header_len, 1, fp);
     fwrite(og.body, og.body_len, 1, fp);
    }

    theora_info_clear(&encoderInfo);
    theora_clear(&theoraState);

```

```
235      fflush(fp);
236      fclose(fp);
237      }
238    return EXIT_SUCCESS;
239  }
```

Since we are interested in line-by-line profiling, we will forgo the default report and generate a line-by-line profile report as follows:

```
$ gprof -l -b demo
```

The *-b* argument is to surpress some default output and generate a *brief* report; the output is as follows:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 17.81      0.26     0.26                             writeFrame (demo.c:61 @ 8048d54)
 13.36      0.46     0.20                             main (demo.c:191 @ 80495d5)
 13.01      0.65     0.19                             main (demo.c:177 @ 804943b)
 11.64      0.81     0.17                             writeFrame (demo.c:48 @ 8048cc4)
  9.59      0.95     0.14                             writeFrame (demo.c:60 @ 8048d7d)
  6.16      1.04     0.09                             writeFrame (demo.c:67 @ 8048da8)
  5.48      1.12     0.08                             main (demo.c:194 @ 80495c2)
  5.48      1.21     0.08                             writeFrame (demo.c:69 @ 8048de1)
  4.11      1.26     0.06                             main (demo.c:193 @ 80495b1)
  3.42      1.31     0.05                             main (demo.c:192 @ 80495a1)
  3.42      1.36     0.05                             writeFrame (demo.c:55 @ 8048d0b)
  3.42      1.42     0.05                             writeFrame (demo.c:53 @ 8048d2a)
  1.37      1.44     0.02                             writeFrame (demo.c:54 @ 8048cec)
  0.68      1.45     0.01                             writeFrame (demo.c:49 @ 8048cb4)
  0.68      1.46     0.01                             writeFrame (demo.c:52 @ 8048d36)
  0.34      1.46     0.01                             main (demo.c:200 @ 80495f3)
  0.00      1.46     0.00      300     0.00     0.00  writeFrame (demo.c:12 @ 8048c04)


                        Call graph


granularity: each sample hit covers 4 byte(s) for 0.68% of 1.46 seconds

index % time    self  children    called     name
                0.00    0.00     300/300         main (demo.c:222 @ 8049892) [121]
[17]     0.0    0.00    0.00     300         writeFrame (demo.c:12 @ 8048c04) [17]
-----------------------------------------------


Index by function name

   [3] main (demo.c:177 @ 804943b) [17] writeFrame (demo.c:12 @ 8048c04) [15] writeFrame (demo.c:52 @ 8048d36)
  [10] main (demo.c:192 @ 80495a1) [14] writeFrame (demo.c:49 @ 8048cb4) [1] writeFrame (demo.c:61 @ 8048d54)
   [9] main (demo.c:193 @ 80495b1) [4] writeFrame (demo.c:48 @ 8048cc4) [5] writeFrame (demo.c:60 @ 8048d7d)
   [7] main (demo.c:194 @ 80495c2) [13] writeFrame (demo.c:54 @ 8048cec) [6] writeFrame (demo.c:67 @ 8048da8)
   [2] main (demo.c:191 @ 80495d5) [11] writeFrame (demo.c:55 @ 8048d0b) [8] writeFrame (demo.c:69 @ 8048de1)
  [16] main (demo.c:200 @ 80495f3) [12] writeFrame (demo.c:53 @ 8048d2a)
```

The report summarizes a good deal of information about the application run but doesn't give a total execution time to determine overall performance gains. We can time the execution of the application for a few samples to calculate baseline performance statistics.

```
$ time ./demo
```

A series of 3 runs shows an average real application run-time of 1.6553 seconds; we'll define this as our base performance metric.

The report is ordered with respect to decreasing run time spent on each line. If we focus our attention on the heavy-hitters in decreasing order we'll focus our attention in one the following:

| **Run-Time Percentage** | **Line** |
|---|---|
| 17.81 | 61 |
| 13.36 | 191 |
| 13.01 | 177 |
| 11.64 | 48 |
| 9.59 | 60 |
| 6.16 | 67 |
| 5.48 | 194 |
| 5.48 | 69 |
| 4.11 | 193 |
| 3.42 | 192 |
| 3.42 | 55 |
| 3.42 | 53 |
| 1.37 | 54 |
| 0.68 | 49 |
| 0.68 | 52 |
| 0.34 | 200 |

Working our way down the candidate list we find lines 61, 191, 177 and so on we find optimization of these lines to be of a tough breed. Lines 48-49 and 52 however can readily be replaced with calls to *memset*. Lines 47-57 become

Timing our application once again shows an average of 1.481 seconds, showing a performance improvement of approximately 11%.

The default application encodes 300 frames, resulting in profiler metrics being collected for 300 calls. Profiler sample counts of 0.01 seconds implies that cumulative profiler measurements less than the sample count cannot be trusted. Increasing the number of iterations allows for cumulative results that exceed this sample count value and therefore increases our candidate optimization regions. Increasing the iterations to 30000 and rerunning profiler collection show a significantly different report on our updated source code.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 16.77     18.20    18.20                             writeFrame (demo.c:53 @ 8048cfa)
 16.25     35.84    17.64                             main (demo.c:183 @ 804957e)
 14.96     52.08    16.24                             main (demo.c:169 @ 80493e4)
 11.15     64.18    12.11                             writeFrame (demo.c:52 @ 8048d23)
  7.59     72.42     8.24                             writeFrame (demo.c:61 @ 8048d87)
  6.84     79.84     7.42                             writeFrame (demo.c:59 @ 8048d4e)
  6.22     86.59     6.75                             main (demo.c:186 @ 804956b)
```

```
  4.99     92.02     5.42                             writeFrame (demo.c:47 @ 8048ca3)
  4.95     97.39     5.38                             main (demo.c:185 @ 804955a)
  2.47    100.07     2.68                             main (demo.c:184 @ 804954a)
  2.31    102.58     2.50                             writeFrame (demo.c:58 @ 8048dc2)
  1.32    104.00     1.43                             writeFrame (demo.c:48 @ 8048cb8)
  1.13    105.23     1.23                             writeFrame (demo.c:49 @ 8048cd0)
  0.82    106.12     0.89                             writeFrame (demo.c:51 @ 8048d2e)
  0.79    106.98     0.86                             main (demo.c:183 @ 8049541)
  0.74    107.79     0.81                             writeFrame (demo.c:52 @ 8048cf1)
  0.29    108.11     0.32                             writeFrame (demo.c:57 @ 8048dce)
  0.19    108.31     0.20                             writeFrame (demo.c:58 @ 8048d45)
  0.04    108.35     0.04                             main (demo.c:167 @ 8049881)
  0.03    108.38     0.03                             writeFrame (demo.c:67 @ 8048e18)
  0.02    108.41     0.03                             main (demo.c:205 @ 80496b5)
  0.02    108.42     0.02                             writeFrame (demo.c:66 @ 8048dde)
  0.01    108.44     0.01                             main (demo.c:206 @ 804972e)
  0.01    108.45     0.01    30000   333.33   333.33  writeFrame (demo.c:12 @ 8048c04)
  0.01    108.46     0.01                             main (demo.c:167 @ 804937f)
  0.01    108.47     0.01                             main (demo.c:170 @ 8049409)
  0.01    108.48     0.01                             main (demo.c:174 @ 8049460)
  0.01    108.49     0.01                             main (demo.c:178 @ 8049522)
  0.01    108.50     0.01                             main (demo.c:192 @ 804959c)
  0.01    108.51     0.01                             main (demo.c:196 @ 80495db)
  0.01    108.52     0.01                             writeFrame (demo.c:70 @ 8048e6c)
  0.00    108.53     0.01                             main (demo.c:176 @ 80494cc)
  0.00    108.53     0.01                             main (demo.c:177 @ 8049503)
  0.00    108.53     0.01                             main (demo.c:207 @ 80497a7)
  0.00    108.54     0.01                             main (demo.c:208 @ 80497bf)


                        Call graph


granularity: each sample hit covers 4 byte(s) for 0.01% of 108.54 seconds

index % time    self  children    called     name
                0.01    0.00   30000/30000       main (demo.c:214 @ 804983b) [31]
[24]     0.0    0.01    0.00   30000         writeFrame (demo.c:12 @ 8048c04) [24]
-----------------------------------------------


Index by function name

  [25] main (demo.c:167 @ 804937f) [29] main (demo.c:192 @ 804959c) [1] writeFrame (demo.c:53 @ 
8048cfa)
   [3] main (demo.c:169 @ 80493e4) [30] main (demo.c:196 @ 80495db) [4] writeFrame (demo.c:52 @ 
8048d23)
  [26] main (demo.c:170 @ 8049409) [21] main (demo.c:205 @ 80496b5) [14] writeFrame (demo.c:51 @ 
8048d2e)
  [27] main (demo.c:174 @ 8049460) [23] main (demo.c:206 @ 804972e) [18] writeFrame (demo.c:58 @ 
8048d45)
  [33] main (demo.c:176 @ 80494cc) [35] main (demo.c:207 @ 80497a7) [6] writeFrame (demo.c:59 @ 
8048d4e)
  [34] main (demo.c:177 @ 8049503) [36] main (demo.c:208 @ 80497bf) [5] writeFrame (demo.c:61 @ 
8048d87)
  [28] main (demo.c:178 @ 8049522) [19] main (demo.c:167 @ 8049881) [11] writeFrame (demo.c:58 @ 
8048dc2)
  [15] main (demo.c:183 @ 8049541) [24] writeFrame (demo.c:12 @ 8048c04) [17] writeFrame (demo.c:57 @ 
8048dce)
  [10] main (demo.c:184 @ 804954a) [8] writeFrame (demo.c:47 @ 8048ca3) [22] writeFrame (demo.c:66 @ 
8048dde)
   [9] main (demo.c:185 @ 804955a) [12] writeFrame (demo.c:48 @ 8048cb8) [20] writeFrame (demo.c:67 @ 
8048e18)
   [7] main (demo.c:186 @ 804956b) [13] writeFrame (demo.c:49 @ 8048cd0) [32] writeFrame (demo.c:70 @ 
```

```
8048e6c)
   [2] main (demo.c:183 @ 804957e) [16] writeFrame (demo.c:52 @ 8048cf1)
```

We can now concentrate our efforts on some new regions. Lines 170-178 are responsible for converting RGB values to YUV colorspace. A quick search for an alternative integer approximation may prove to increase our performance; namely:

```
unsigned char Y = (abs(R * 2104 +
                       G * 4130 +
                       B * 802 + 4096 + 131072) >> 13);
 unsigned char U = (abs(R * -1214 +
                       G * -2384 +
                       B * 3598 + 4096 + 1048576) >> 13);
 unsigned char V = (abs(R * 3598 +
                       G * -3013 +
                       B * -585 + 4096 + 1048576) >> 13);
```

Since we use this conversion in two areas in our code, the replacement is two-fold. Our new time measurements show an average execution time of 0.9403; a performance improvement of ~43% over our original application. This proves worthy of our hours worth of effort. We can repeat this cycle and wring out candidate regions until we are satisfied with our applications performance.

## Conclusion

Profiling our application and focusing our optimization on program hotspots demonstrated a 43% performance increase for approximately an hours worth of effort. Application optimization should begin at the crucial stage of application design. A poorly designed application will never yield an optimized application, regardless of how many cycles you perform the profile/optimize cycle. A well-designed application can be profiled by using the Gprof utility and identified candidate regions can be concentrated on for optimization opportunities.

## References

1. http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_chapter/gprof_toc.html
2. http://en.wikipedia.org/wiki/Performance_analysis