ECE459: Programming for Performance Winter 2013

Lecture 17 — March 12, 2013

Patrick Lam *version 1*

About Assignment 3

This year's Assignment 3 asks you to optimize a naïve implementation of Beier-Neely image morphing [BN92] that I found on the Internet[1].

> "an image processing technique typically used as an animation tool for the metamorphosis of one image to another."

Image morphing computes intermediate images between a source image and a destination image. It is more complicated than simply cross-dissolving between pictures (where one would just interpolate pixel colours).

$$\text{Morphing} = \text{warping} + \text{cross-dissolving}.$$

By warping, we mean converting the two images into comparable images (guided by user input). After the two images are warped, cross-dissolving gives a seamless transition between the images.

I'll include a more in-depth domain and implementation discussion on Thursday. General tips:

Algorithms. All of the C++ built-in algorithms work with anything that uses the standard C++ container interface.

- `max_element`—returns an iterator with the largest element; you can use your own comparator function.
- `min_element`—same as above, but the smallest element.
- `sort`—sorts a container; you can use your own comparator function.
- `upper_bound`—returns an iterator to the first element greater than the value; only works on a **sorted** container (if the default comparator isn't used, you have to use the same one used to sort the container).
- `random_shuffle`—does `n` random swaps of the elements in the container

[1] Rafael Robledo uploaded a code dump to Google Projects: `http://code.google.com/p/nm-morph/`. Thanks!

C++ Hurdles. If you've worked with C++ before, you probably know the awful compiler messages and pages of template expansions. You can use the `clang` compiler frontend if you have a compiler error; it will be easier to understand. (Use `-std=c++11`).

The following instructions are due to Paul Roukema. Thanks! To use clang temporarily use:

```
% make CXX=clang++
```

To use it permanently, add the following line to `test-harness.pro` (and perhaps `nm-morph.pro`):

```
QMAKE_CXX = clang++
```

To add additional compiler flags, use the line:

```
QMAKE_CXX_CFLAGS += <stuff>
```

Interpreting Names from the Profiler. Profiler messages might get pretty bad. Look for one of the main functions, or if it's a weird, mangled, name, look where it's called from. Google Perf Tools may also give you more fine-grained information.

A terrible example:

```
[32] std::_Hashtable<unsigned int, std::pair<unsigned int const, std::unordered_map<unsigned int, double,
     std::hash<unsigned int>, std::equal_to<unsigned int>, std::allocator<std::pair<unsigned int const,
     double> > >, std::allocator<std::pair<unsigned int const, std::unordered_map<unsigned int, double,
     std::hash<unsigned int>, std::equal_to<unsigned int>, std::allocator<std::pair<unsigned int const,
     double> > > >, std::_Select1st<std::pair<unsigned int const, std::unordered_map<unsigned int, double
     , std::hash<unsigned int>, std::equal_to<unsigned int>, std::allocator<std::pair<unsigned int const,
     double> > > >, std::equal_to<unsigned int>, std::hash<unsigned int>, std::__detail::
     _Mod_range_hashing, std::__detail::_Default_ranged_hash, std::__detail::_Prime_rehash_policy, false,
     false, true>::clear()
```

is actually `distance_map.clear()` (from last year's assignment), which is automatically called by the destructor.

Things You Can Do. Since I've provided you a random unoptimized implementation from the Internet, there should be a lot you can do to optimize it.

- You can introduce threads, using pthreads (or C++11 threads, if you're feeling lucky), OpenMP, or whatever you want.
- Play around with compiler options.
- Use better algorithms or data strutures—maybe Qt is inefficient for storing pixels!
- The list goes on and on.

Things You Need to Do. Profile!

- Keep your inputs constant between all profiling results so they're comparable.
- Baseline profile with no changes.

- You will pick your two best performance changes to add to the report.
  - You will include a profiling report before the change and just after the change (and only that change!)
  - More specific instructions in the handout.
- There may or may not be overlap between the baseline and the baseline for each change.
- My recommendation: use your initial baseline as the "before" for your first change, and the "after" of the first change for the baseline of your second change.
- Whatever you choose, it should be convincing.

Other tips. We'll run the submissions, polling from `ece459-1` and refreshing the results on a leaderboard; the earlier you submit, the better. There'll be something like a 10 second time limit.

A Word. This assignment should be **enjoyable**. Good luck!

Profiling

If you want to make your programs or systems fast, you need to find out what is currently slow and improve it. (duh!)

How profiling works:

- sampling-based (traditional): every so often (e.g. 2ns), query the system state; or,
- instrumentation-based, or probe-based/predicate-based (traditionally too expensive): query system state under certain conditions; like conditional breakpoints.

We'll talk about both per-process profiling and system-wide profiling.

If you need your system to run fast, you need to start profiling and benchmarking as soon as you can run the system. Benefits:

- establishes a baseline performance for the system;
- allows you to measure impacts of changes and further system development;
- allows you to re-design the system before it's too late;
- avoids the need for "perf spray" to make the system faster, since that spray is often made of "unobtainium"[2].

[2] `http://en.wikipedia.org/wiki/Unobtainium`

Tips for Leveraging Profiling. When writing large software projects:

- First, write clear and concise code.
  Don't do any premature optimizations—focus on correctness.
- Profile to get a baseline of your performance:
  - allows you to easily track any performance changes;
  - allows you to re-design your program before it's too late.

Focus your optimization efforts on the code that matters.

Look for abnormalities; in particular, you're looking for deviations from the following rules:

- time is spent in the right part of the system/program;
- time is not spent in error-handling, noncritical code, or exceptional cases; and
- time is not unnecessarily spent in the operating system.

For instance, "why is `ps` taking up all my cycles?"; see page 34 of Cantrill[3].

Development vs. production. You can always profile your systems in development, but that might not help with complexities in production. (You want separate dev and production systems, of course!) We'll talk a bit about DTrace, which is one way of profiling a production system. The constraints on profiling production systems are that the profiling must not affect the system's performance or reliability.

Userspace per-process profiling

Sometimes—or, in this course, often—you can get away with investigating just one process and get useful results about that process's behaviour. We'll first talk about `gprof`, the GNU profiler tool[4], and then continue with other tools.

`gprof` does sampling-based profiling for single processes: it requests that the operating system interrupt the process being profiled at regular time intervals and figures out which procedure is currently running. It also adds a bit of instrumentation to collect information about which procedures call other procedures.

"Flat" profile. The obvious thing to do with the profile information is to just print it out. You get a list of procedures called and the amount of time spent in each of these procedures.

The general limitation is that procedures that don't run for long enough won't show up in the profile. (There's a caveat: if the function was compiled for profiling, then it will show up anyway, but you won't find out about how long it executed for).

[3] `http://queue.acm.org/detail.cfm?id=1117401`
[4] `http://sourceware.org/binutils/docs/gprof/`

"Call graph". `gprof` can also print out its version of a call graph, which shows the amount of time that either a function runs (as in the "flat" profile) as well as the amount of time that the callees of the function run. Another term for such a call graph is a "dynamic call graph", since it tracks the dynamic behaviour of the program. Using the `gprof` call graph, you can find out who is responsible for calling the functions that take a long time.

Limitations of `gprof`. Beyond the usual limitations of a process-oriented profiler, `gprof` also suffers limitations from running completely in user-space. That is, it has no access to information about system calls, including time spent doing I/O. It also doesn't know anything about the CPU's built-in counters (e.g. cache miss counts, etc). Like the other profilers, it causes overhead when it's running, but the overhead isn't too large.

`gprof` usage guide

We'll give some details about using `gprof`. First, use the `-pg` flag with `gcc` when compiling and linking. Next, run your program as you normally would. Your program will now create `gmon.out`.

Use gprof to interpret the results: `gprof <executable>`.

Example. Consider a program with 100 million calls to two math functions.

```
int main() {
    int i,x1=10,y1=3,r1=0;
    float x2=10,y2=3,r2=0;

    for(i=0;i<100000000;i++) {
        r1 += int_math(x1,y1);
        r2 += float_math(y2,y2);
    }
}

int int_math(int x, int y){
    int r1;
    r1=int_power(x,y);
    r1=int_math_helper(x,y);
    return r1;
}

int int_math_helper(int x, int y){
    int r1;
    r1=x/y*int_power(y,x)/int_power(x,y);
    return r1;
}

int int_power(int x, int y){
    int i, r;
    r=x;
    for(i=1;i<y;i++){
        r=r*x;
    }
    return r;
}
```

```
float float_math(float x, float y) {
    float r1;
    r1=float_power(x,y);
    r1=float_math_helper(x,y);
    return r1;
}

float float_math_helper(float x, float y) {
    float r1;
    r1=x/y*float_power(y,x)/float_power(x,y);
    return r1;
}

float float_power(float x, float y){
    float i, r;
    r=x;
    for(i=1;i<y;i++) {
        r=r*x;
    }
    return r;
}
```

Looking at the code, we have no idea what takes longer. One might guess floating point math taking longer. This is admittedly a silly example, but it works well to illustrate our point.

Flat Profile Example. When we run the program and look at the flat profile, we see:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 32.58      4.69     4.69 300000000    15.64    15.64  int_power
 30.55      9.09     4.40 300000000    14.66    14.66  float_power
 16.95     11.53     2.44 100000000    24.41    55.68  int_math_helper
 11.43     13.18     1.65 100000000    16.46    45.78  float_math_helper
  4.05     13.76     0.58 100000000     5.84    77.16  int_math
  3.01     14.19     0.43 100000000     4.33    64.78  float_math
  2.10     14.50     0.30                             main
```

There is one function per line. Here are what the columns mean:

- **% time:** the percent of the total execution time in this function.
- **self:** seconds in this function.
- **cumulative:** sum of this function's time + any above it in table.
- **calls:** number of times this function was called.
- **self ns/call:** just self nanoseconds / calls.
- **total ns/call:** mean function execution time, including calls the function makes.

Call Graph Example. After the flat profile gives you a feel for which functions are costly, you can get a better story from the call graph.

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.30   14.19                 main [1]
                0.58    7.13 100000000/100000000     int_math [2]
                0.43    6.04 100000000/100000000     float_math [3]
-----------------------------------------------
                0.58    7.13 100000000/100000000     main [1]
[2]     53.2    0.58    7.13 100000000         int_math [2]
                2.44    3.13 100000000/100000000     int_math_helper [4]
                1.56    0.00 100000000/300000000     int_power [5]
-----------------------------------------------
                0.43    6.04 100000000/100000000     main [1]
[3]     44.7    0.43    6.04 100000000         float_math [3]
                1.65    2.93 100000000/100000000     float_math_helper [6]
                1.47    0.00 100000000/300000000     float_power [7]
-----------------------------------------------
                2.44    3.13 100000000/100000000     int_math [2]
[4]     38.4    2.44    3.13 100000000         int_math_helper [4]
                3.13    0.00 200000000/300000000     int_power [5]
-----------------------------------------------
                1.56    0.00 100000000/300000000     int_math [2]
                3.13    0.00 200000000/300000000     int_math_helper [4]
[5]     32.4    4.69    0.00 300000000         int_power [5]
-----------------------------------------------
                1.65    2.93 100000000/100000000     float_math [3]
[6]     31.6    1.65    2.93 100000000         float_math_helper [6]
                2.93    0.00 200000000/300000000     float_power [7]
-----------------------------------------------
                1.47    0.00 100000000/300000000     float_math [3]
                2.93    0.00 200000000/300000000     float_math_helper [6]
[7]     30.3    4.40    0.00 300000000         float_power [7]
```

To interpret the call graph, note that the line with the index [N] is the *primary line*, or the current function being considered.

- Lines above the primary line are the functions which called this function.
- Lines below the primary line are the functions which were called by this function (children).

For the primary line, the columns mean:

- **time:** total percentage of time spent in this function and its children.
- **self:** same as in flat profile.
- **children:** time spent in all calls made by the function;
  - should be equal to self + children of all functions below.

For callers (functions above the primary line):

- **self:** time spent in primary function, when called from current function.
- **children:** time spent in primary function's children, when called from current function.
- **called:** number of times primary function was called from current function / number of nonrecursive calls to primary function.

For callees (functions below the primary line):

- **self:** time spent in current function when called from primary.
- **children:** time spent in current function's children calls when called from primary.
  - self + children is an estimate of time spent in current function when called from primary function.
- **called:** number of times current function was called from primary function / number of nonrecursive calls to current function.

Based on this information, we can now see where most of the time comes from, and pinpoint any locations that make unexpected calls, etc. This example isn't too exciting; we could simplify the math and optimize the program that way.

Introduction to gperftools

Next, we'll talk about the Google Performance Tools.

`http://google-perftools.googlecode.com/svn/trunk/doc/cpuprofile.html`

They include:

- a CPU profiler
- a heap profiler
- a heap checker; and
- a faster `malloc`.

We'll mostly use the CPU profiler. Characteristics include:

- supposedly works for multithreaded programs;
- purely statistical sampling;
- no recompilation required (typically benefit from re-linking); and
- better output, including built-in graphical output.

You can use the profiler without any recompilation. But this is not recommended; you'll get worse data. Use `LD_PRELOAD`, which changes the dynamic libraries that an executable uses.

```
% LD_PRELOAD="/usr/lib/libprofiler.so" CPUPROFILE=test.prof ./test
```

The other (more-recommended) option is to link to the profiler with `-lprofiler`.

Both options read the `CPUPROFILE` environment variable, which specifies where profiling data goes.

You can use the profiling library directly as well:

```
#include <gperftools/profiler.h>
```

Then, bracket code you want profiled with:

```
ProfilerStart()
// ...
ProfilerEnd()
```

You can change the sampling frequency with the `CPUPROFILE_FREQUENCY` environment variable (default value 100 interrupts/second).

`pprof` usage. `pprof` is like `gprof` for Google Perf Tools. It analyzes profiling results. Here are some usage examples.

```
% pprof test test.prof
    Enters "interactive" mode
% pprof --text test test.prof
    Outputs one line per procedure
% pprof --gv test test.prof
     Displays annotated call-graph via 'gv'
% pprof --gv --focus=Mutex test test.prof
    Restricts to code paths including a .*Mutex.* entry
% pprof --gv --focus=Mutex --ignore=string test test.prof
    Code paths including Mutex but not string
% pprof --list=getdir test test.prof
    (Per-line) annotated source listing for getdir()
% pprof --disasm=getdir test test.prof
    (Per-PC) annotated disassembly for getdir()
```

Can also output `dot`, `ps`, `pdf` or `gif` instead of `gv`.

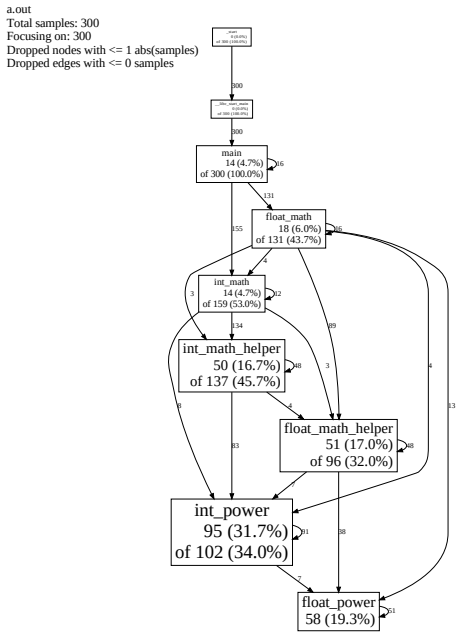
`gprof` text output. This is similar to the flat profile in `gprof`.

```
jon@riker examples master % pprof --text test test.prof
Using local file test.
Using local file test.prof.
Removing killpg from all stack traces.
Total: 300 samples
      95  31.7%  31.7%      102  34.0% int_power
      58  19.3%  51.0%       58  19.3% float_power
      51  17.0%  68.0%       96  32.0% float_math_helper
      50  16.7%  84.7%      137  45.7% int_math_helper
      18   6.0%  90.7%      131  43.7% float_math
      14   4.7%  95.3%      159  53.0% int_math
      14   4.7% 100.0%      300 100.0% main
       0   0.0% 100.0%      300 100.0% __libc_start_main
       0   0.0% 100.0%      300 100.0% _start
```

Columns, from left to right, denote:

- Number of samples in this function.
- Percentage of samples in this function (same as **time** in `gprof`).
- Percentage of checks in the functions printed so far (equivalent to **cumulative**, but in %).
- Number of checks in this function and its callees.
- Percentage of checks in this function and its callees.
- Function name.

Graphical Output. Google Perf Tools can also produce graphical output:



This shows the same numbers as the text output. Directed edges denote function calls. Note:

$$\#\text{ of samples in callees} = \#\text{ in ``this function + callees'' - }\#\text{ in ``this function''}.$$

For example, in `float_math_helper`, we have "51 (local) of 96 (cumulative)". Here,

$$96 - 51 = 45(\text{callees}).$$

- callee `int_power` = 7 (bogus)
- callee `float_power` = 38
- callees total = 45

Note that the call graph is not exact. In fact, it shows many bogus relations which clearly don't exist. For instance, we know that there are no cross-calls between `int` and `float` functions.

As with `gprof`, optimizations will change the graph.

You'll probably want to look at the text profile first, then use the `--focus` flag to look at individual functions.

References

[BN92] Thaddeus Beier and Shawn Neely. Feature-based image metamorphosis. In *Proceedings of SIGGRAPH 1992*, pages 35–42, 1992.