

Lecture 25—Profiling

ECE 459: Programming for Performance

March 9, 2015

Part I

Profiling

Introduction to Profiling

So far we've been looking at small problems.

Must **profile** to see what takes time in a large program.

Two main outputs:

- flat;
- call-graph.

Two main data gathering methods:

- statistical;
- instrumentation.

Profiler Outputs

Flat Profiler:

- Only computes the average time in a particular function.
- Does not include other (useful) information, like callees.

Call-graph Profiler:

- Computes call times.
- Reports frequency of function calls.
- Gives a call graph: who called what function?

Data Gathering Methods

Statistical:

Mostly, take samples of the system state, that is:

- every 100ms, check the system state.
- will cause some slowdown, but not much.

Instrumentation:

Add additional instructions at specified program points:

- can do this at compile time or run time (expensive);
- can instrument either manually or automatically;
- like conditional breakpoints.

Guide to 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.

Things to Look For

Good signs:

- Time is spent in the right part of the system.
- Most time should not be spent handling errors; in non-critical code; or in exceptional cases.
- Time is not unnecessarily spent in the operating system.

gprof introduction

Statistical profiler, plus some instrumentation for calls.

Runs completely in user-space.

Only requires a compiler.

gprof usage

Use the `-pg` flag with `gcc` when compiling and linking.

Run your program as you normally would.

- Your program will now create a `gmon.out` file.

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

gprof example

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);  
    }  
}
```

- Looking at the code, we have no idea what takes longer.
- Probably would guess floating point math taking longer.
- (Overall, silly example.)

Example (Integer Math)

```
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;
}
```

Example (Float Math)

```
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;  
}
```

Flat Profile

When we run the program and look at the profile, we see:

Flat profile :

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ns/call	total 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

- One function per line.
- **% 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.

Flat Profile

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ns/call	total 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

- **calls:** number of times this function was called
- **self ns/call:** just self nanoseconds / calls
- **total ns/call:** average time for function execution, including any other calls the function makes

Call Graph Example (1)

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]
<hr/>					
[2]	53.2	0.58	7.13	100000000/100000000	main [1]
		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]
<hr/>					
[3]	44.7	0.43	6.04	100000000/100000000	main [1]
		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]

Reading the Call Graph

The line with the index is the current function being looked at
(primary line).

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

Primary Line

- **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

Reading Callers from Call Graph

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.

Reading Callees from Call Graph

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.
 - ▶ $\text{self} + \text{children}$ 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.

Call Graph Example (2)

index	% time	self	children	called	name
[4]	38.4	2.44	3.13	100000000/100000000	int_math [2]
		2.44	3.13	100000000	int_math_helper [4]
		3.13	0.00	200000000/300000000	int_power [5]
[5]	32.4	1.56	0.00	100000000/300000000	int_math [2]
		3.13	0.00	200000000/300000000	int_math_helper [4]
		4.69	0.00	300000000	int_power [5]
[6]	31.6	1.65	2.93	100000000/100000000	float_math [3]
		1.65	2.93	100000000	float_math_helper [6]
		2.93	0.00	200000000/300000000	float_power [7]
[7]	30.3	1.47	0.00	100000000/300000000	float_math [3]
		2.93	0.00	200000000/300000000	float_math_helper [6]
		4.40	0.00	300000000	float_power [7]

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.

Summary (Profiling)

- Saw how to use gprof
(one option for Assignment 3).
- Profile early and often.
- Make sure your profiling shows what you expect.