

# Lecture 25 — Using Google Performance Tools

Patrick Lam

`patrick.lam@uwaterloo.ca`

Department of Electrical and Computer Engineering  
University of Waterloo

April 4, 2019

# Part I

## gperftools

Google Performance Tools include:

- CPU profiler.
- Heap profiler.
- Heap checker.
- Faster (multithreaded) `malloc`.

We'll mostly use the CPU profiler:

- purely statistical sampling;
- no recompilation; at most linking; and
- built-in visual output.

You can use the profiler without any recompilation.

- Not recommended—worse data.

---

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

---

The other option is to link to the profiler:

- `-lprofiler`

Both options read the `CPUPROFILE` environment variable:

- states the location to write the profile data.

You can use the profiling library directly as well:

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

Bracket code you want profiled with:

- `ProfilerStart()`
- `ProfilerStop()`

You can change the sampling frequency with the `CPUPROFILE_FREQUENCY` environment variable.

- **Default value:** 100

Like gprof, it will analyze profiling results.

---

```
% 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.

## 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:

Number of checks (samples) in this function.

Percentage of checks 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.



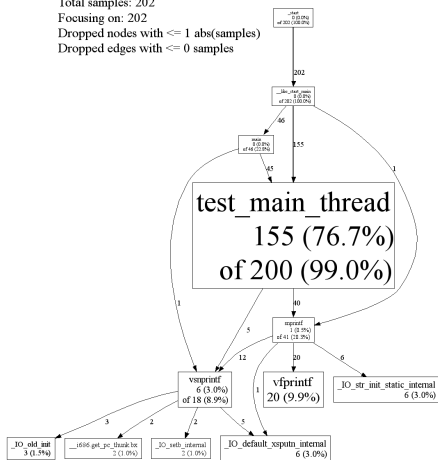
/tmp/profiler2\_unittest

Total samples: 202

Focusing on: 202

Dropped nodes with  $\leq 1$  abs(samples)

Dropped edges with  $\leq 0$  samples



Output was too small to read on the slide.

- Shows the same numbers as the text output.
- Directed edges denote function calls.
- Note: number of samples in callees =  
number in “this function + callees” -  
number in “this function”.
- **Example:**  
float\_math\_helper, 51 (local) of 96 (cumulative)  
96 - 51 = 45 (callees)
  - callee int\_power = 7 (bogus)
  - callee float\_power = 38
  - callees total = 45

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.

## Part II

System profiling:  
oprofile, perf, DTrace, WAIT

<http://oprofile.sourceforge.net>

Sampling-based tool.

Uses CPU performance counters.

Tracks currently-running function;  
records profiling data for every application run.

Can work system-wide (across processes).

Technology: Linux Kernel Performance Events  
(formerly a Linux kernel module).

Must run as root to use system-wide,  
otherwise can use per-process.

---

```
% sudo opcontrol \  
    --vmlinux=/usr/src/linux-3.2.7-1-ARCH/vmlinux  
% echo 0 | sudo tee /proc/sys/kernel/nmi_watchdog  
% sudo opcontrol --start  
Using default event: CPU_CLK_UNHALTED:100000:0:1:1  
Using 2.6+ OProfile kernel interface.  
Reading module info.  
Using log file /var/lib/oprofile/samples/oprofiled.log  
Daemon started.  
Profiler running.
```

---

Per-process:

---

```
[plam@lynch nm-morph]$ operf ./test_harness  
operf: Profiler started
```

Profiling done.

---

Pass your executable to opreport.

---

```
% sudo opreport -l ./test
CPU: Intel Core/i7 , speed 1595.78 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not
halted) with a unit mask of 0x00 (No unit mask) count 100000
samples  %          symbol name
7550      26.0749    int_math_helper
5982      20.6596    int_power
5859      20.2348    float_power
3605      12.4504    float_math
3198      11.0447    int_math
2601       8.9829    float_math_helper
160       0.5526     main
```

---

If you have debug symbols (-g) you could use:

---

```
% sudo opannotate --source \
--output-dir=/path/to/annotated--source /path/to/mybinary
```

---

Use `opreport` by itself for a whole-system view.  
You can also reset and stop the profiling.

---

```
% sudo opcontrol --reset  
Signalling daemon... done  
% sudo opcontrol --stop  
Stopping profiling.
```

---



`perf.wiki.kernel.org/index.php/Tutorial`

Interface to Linux kernel built-in sampling-based profiling.  
Per-process, per-CPU, or system-wide.  
Can even report the cost of each line of code.

## On previous Assignment 3 code:

---

```
[plam@lynch nm-morph]$ perf stat ./test_harness
```

```
Performance counter stats for './test_harness':
```

6562.501429	task-clock	#	0.997	CPUs utilized	
666	context-switches	#	0.101	K/sec	
0	cpu-migrations	#	0.000	K/sec	
3,791	page-faults	#	0.578	K/sec	
24,874,267,078	cycles	#	3.790	GHz	[83.32%]
12,565,457,337	stalled-cycles-frontend	#	50.52%	frontend cycles idle	[83.31%]
5,874,853,028	stalled-cycles-backend	#	23.62%	backend cycles idle	[66.63%]
33,787,408,650	instructions	#	1.36	insns per cycle	
		#	0.37	stalled cycles per insn	[83.32%]
5,271,501,213	branches	#	803.276	M/sec	[83.38%]
155,568,356	branch-misses	#	2.95%	of all branches	[83.36%]
6.580225847	seconds time elapsed				

---

perf can tell you which instructions are taking time, or which lines of code.

Compile with -ggdb to enable source code viewing.

---

```
% perf record ./test_harness  
% perf annotate
```

---

perf annotate is interactive. Play around with it.

<http://queue.acm.org/detail.cfm?id=1117401>  
<http://www.brendangregg.com/blog/2016-10-27/dtrace-for-linux-2016.html>

Intrumentation-based tool.

System-wide.

Meant to be used on production systems. (Eh?)

<http://queue.acm.org/detail.cfm?id=1117401>  
<http://www.brendangregg.com/blog/2016-10-27/dtrace-for-linux-2016.html>

Intrumentation-based tool.

System-wide.

Meant to be used on production systems. (Eh?)

(Typical instrumentation can have a slowdown of 100x (Valgrind).)

Design goals:

- 1 No overhead when not in use;
- 2 Guarantee safety—must not crash  
(strict limits on expressiveness of probes).

How does DTrace achieve 0 overhead?

- only when activated, dynamically rewrites code by placing a branch to instrumentation code.

Uninstrumented: runs as if nothing changed.

Most instrumentation: at function entry or exit points.  
You can also instrument kernel functions, locking,  
instrument-based on other events.

Can express sampling as instrumentation-based events also.

You write this:

---

```
syscall::read:entry {  
    self->t = timestamp;  
}  
  
syscall::read:return  
/self->t/ {  
    printf("%d/%d spent %d nsecs in read\n"  
        pid, tid, timestamp - self->t);  
}
```

---

`t` is a thread-local variable.

This code prints how long each call to read takes, along with context.

To ensure safety, DTrace limits expressiveness—no loops.

■ (Hence, no infinite loops!)

AMD CodeAnalyst—based on oprofile; leverages AMD processor features.

### WAIT

- IBM's tool tells you what operations your JVM is waiting on while idle.
- Non-free and not available.



Built for production environments.

Specialized for profiling JVMs,  
uses JVM hooks to analyze idle time.

Sampling-based analysis; infrequent samples  
(1–2 per minute!)

At each sample: records each thread's state,

- call stack;
- participation in system locks.

Enables WAIT to compute a “wait state”  
(using expert-written rules):  
what the process is currently doing or waiting on, e.g.

- disk;
- GC;
- network;
- blocked;
- etc.

You:

- run your application;
- collect data (using a script or manually); and
- upload the data to the server.

Server provides a report.

- You fix the performance problems.

Report indicates processor utilization (idle, your application, GC, etc); runnable threads; waiting threads (and why they are waiting); thread states; and a stack viewer.

Paper presents 6 case studies where WAIT identified performance problems: deadlocks, server underloads, memory leaks, database bottlenecks, and excess filesystem activity.

Profiling: Not limited to C/C++, or even code.

You can profile Python using `cProfile`; standard profiling technology.

Google's Page Speed Tool: profiling for web pages—how can you make your page faster?

- reducing number of DNS lookups;
- leveraging browser caching;
- combining images;
- plus, traditional JavaScript profiling.