Lecture 16—More Profiling: gperftools, systemwide tools: oprofile, perf, DTrace, etc.

ECE 459: Programming for Performance

March 6, 2014

Part I

gperftools

Introduction to 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.

Google Perf Tools profiler usage

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.

Other Usage

You can use the profiling library directly as well:

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

Bracket code you want profiled with:

- `ProfilerStart()`
- `ProfilerEnd()`

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

- **Default value:** 100

pprof Usage

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.

Text Output

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
```

Text Output Explained

Columns, from left to right:

Number of checks (samples) in this function.
Percentage of checks in this function.

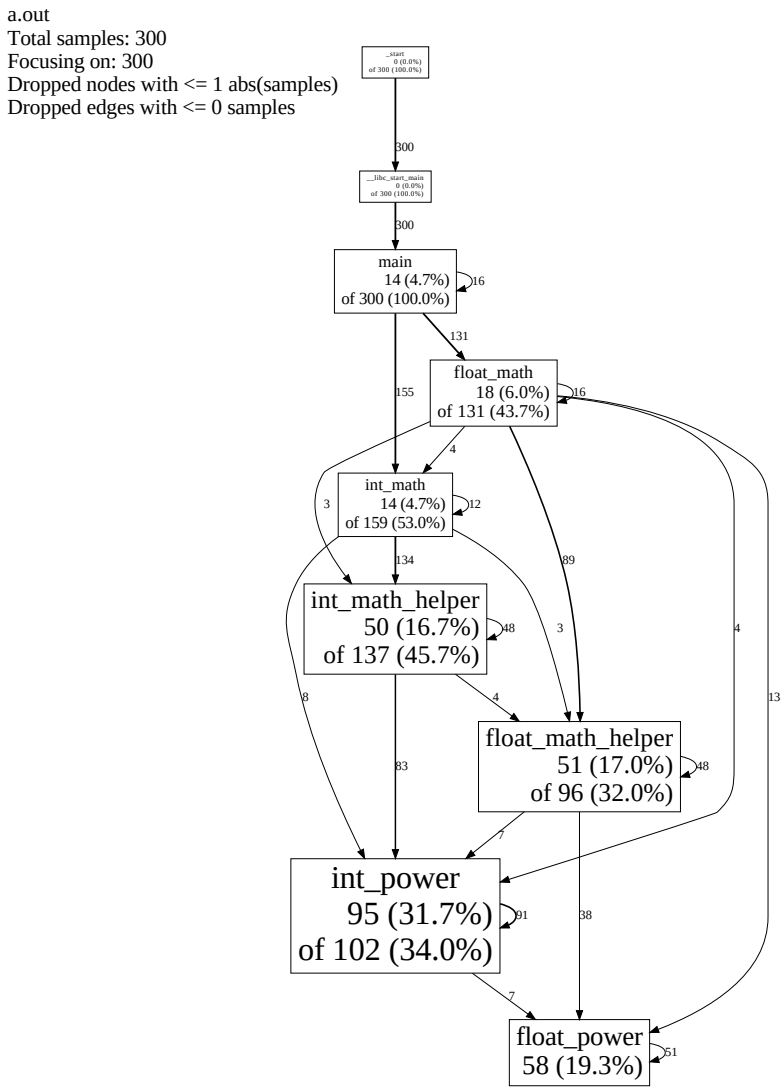
- Same as **time** in `gprof`.

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.

Graphical Output



Graphical Output Explained

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

Things You May Notice

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

Introduction: oprofile

`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).

Setting up oprofile

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.
```

oprofile Usage (1)

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
```

oprofile Usage (2)

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: Introduction

`https://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.

Perf: Usage Example

On last year's 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: Source-level Analysis

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.

DTrace: Introduction

`http://queue.acm.org/detail.cfm?id=1117401`

Intrumentation-based tool.
System-wide.
Meant to be used on production systems. (Eh?)

DTrace: Introduction

`http://queue.acm.org/detail.cfm?id=1117401`

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).

DTrace: Operation

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.

DTrace Example

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 what you write; e.g. no loops.

- (Hence, no infinite loops!)

Other Tools

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.

Other Tools

AMD CodeAnalyst—based on oprofile.

WAIT

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

WAIT: Introduction

Built for production environments.

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

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

WAIT: Operation

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.

WAIT: Workflow

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.

Other Profiling Tools

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.

Part III

Reentrancy

Reentrancy

$\Rightarrow$ A function can be suspended in the middle and **re-entered** (called again) before the previous execution returns.

Does not always mean **thread-safe** (although it usually is).

- Recall: **thread-safe** is essentially "no data races".

Moot point if the function only modifies local data, e.g. `sin()`.

Reentrancy Example

Courtesy of Wikipedia (with modifications):

```
int t;

void swap(int *x, int *y) {
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
}

void isr() {
    int x = 1, y = 2;
    swap(&x, &y);
}
...
int a = 3, b = 4;
...
    swap(&a, &b);
```

Reentrancy Example—Explained (a trace)

```
call swap(&a, &b);
 t = *x;                          // t = 3 (a)
 *x = *y;                         // a = 4 (b)
 call isr();
    x = 1; y = 2;
    call swap(&x, &y)
    t = *x;                       // t = 1 (x)
    *x = *y;                      // x = 2 (y)
    *y = t;                       // y = 1
 *y = t;                          // b = 1

Final values:
a = 4, b = 1

Expected values:
a = 4, b = 3
```

Reentrancy Example, Fixed

```
int t;

void swap(int *x, int *y) {
    int s;

    s = t;  // save global variable
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
    t = s;  // restore global variable
}

void isr() {
    int x = 1, y = 2;
    swap(&x, &y);
}
...
int a = 3, b = 4;
...
    swap(&a, &b);
```

Reentrancy Example, Fixed—Explained (a trace)

```
call swap(&a, &b);
s = t;                          // s = UNDEFINED
t = *x;                         // t = 3 (a)
*x = *y;                        // a = 4 (b)
call isr();
    x = 1; y = 2;
    call swap(&x, &y)
    s = t;                      // s = 3
    t = *x;                     // t = 1 (x)
    *x = *y;                    // x = 2 (y)
    *y = t;                     // y = 1
    t = s;                      // t = 3
*y = t;                         // b = 3
t = s;                          // t = UNDEFINED

Final values:
a = 4, b = 3

Expected values:
a = 4, b = 3
```

Previous Example: thread-safety

Is the previous reentrant code also thread-safe?
(This is more what we're concerned about in this course.)

Let's see:

```
int t;

void swap(int *x, int *y) {
    int s;

    s = t;  // save global variable
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
    t = s;  // restore global variable
}
```

Consider two calls: `swap(a, b)`, `swap(c, d)` with
`a = 1, b = 2, c = 3, d = 4`.

Previous Example: thread-safety trace

```
global: t

/* thread 1 */                                  /* thread 2 */
a = 1, b = 2;
s = t;      // s = UNDEFINED
t = a;      // t = 1
                                                c = 3, d = 4;
                                                s = t;      // s = 1
                                                t = c;      // t = 3
                                                c = d;      // c = 4
                                                d = t;      // d = 3
a = b;      // a = 2
b = t;      // b = 3
t = s;      // t = UNDEFINED
                                                t = s;      // t = 1

Final values:
a = 2, b = 3, c = 4, d = 3, t = 1

Expected values:
a = 2, b = 1, c = 4, d = 3, t = UNDEFINED
```

Reentrancy vs Thread-Safety (1)

- Re-entrant does not always mean thread-safe (as we saw)
  - But, for most sane implementations, it is thread-safe

Ok, but are **thread-safe** functions reentrant?

Reentrancy vs Thread-Safety (2)

Are **thread-safe** functions reentrant? Nope. Consider:

```
int f() {
    lock();
    // protected code
    unlock();
}
```

Recall: **Reentrant functions can be suspended in the middle of execution and called again before the previous execution completes.**

`f()` obviously isn't reentrant. Plus, it will deadlock.

Interrupt handling is more for systems programming, so the topic of reentrancy may or may not come up again.

Summary of Reentrancy vs Thread-Safety

Difference between reentrant and thread-safe functions:

Reentrancy

- Has nothing to do with threads—assumes a **single thread**.
- Reentrant means the execution can context switch at any point in in a function, call the **same function**, and **complete** before returning to the original function call.
- Function's result does not depend on where the context switch happens.

Thread-safety

- Result does not depend on any interleaving of threads from concurrency or parallelism.
- No unexpected results from multiple concurrent executions of the function.

Another Definition of Thread-Safe Functions

"A function whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after another, in an undefined order, even if the actual execution is interleaved."

Good Example of an Exam Question

```
void swap(int *x, int *y) {
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
```

- Is the above code thread-safe?
- Write some expected results for running two calls in parallel.
- Argue these expected results always hold, or show an example where they do not.

Part IV

Good Practices

Inlining

We have seen the notion of inlining:

- Instructs the compiler to just insert the function code in-place, instead of calling the function.
- Hence, no function call overhead!
- Compilers can also do better—context-sensitive—operations they couldn't have done before.

No overhead... sounds like better performance...
let's inline everything!

Inlining in C++

Implicit inlining (defining a function inside a class definition):

```
class P {
public:
    int get_x() const { return x; }
...
private:
    int x;
};
```

Explicit inlining:

```
inline max(const int& x, const int& y) {
    return x < y ? y : x;
}
```

The Other Side of Inlining

One big downside:

- Your program size is going to increase.

This is worse than you think:

- Fewer cache hits.
- More trips to memory.

Some inlines can grow very rapidly (C++ extended constructors).

Just from this your performance may go down easily.

Compilers on Inlining

Inlining is merely a suggestion to compilers.
They may ignore you.

For example:

- taking the address of an "inline" function and using it; or
- virtual functions (in C++),

will get you ignored quite fast.

From a Usability Point-of-View

Debugging is more difficult (e.g. you can't set a breakpoint in a function that doesn't actually exist).

- Most compilers simply won't inline code with debugging symbols on.
- Some do, but typically it's more of a pain.

Library design:

- If you change any inline function in your library, any users of that library have to **recompile** their program if the library updates. (non-binary-compatible change!)

Not a problem for non-inlined functions—programs execute the new function dynamically at runtime.

Summary

- System profiling tools:
  oprofile, DTrace, WAIT, perf
- Reentrancy vs thread-safety.
- Inlining: limit your inlining to trivial functions:
  - makes debugging easier and improves usability;
  - won't slow down your program before you even start optimizing it.
- Tell the compiler high-level information but think low-level.