ECE459: Programming for Performance Winter 2014

Lecture 16 — March 6, 2014

Patrick Lam *version 1*

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 (multithreaded) `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`.

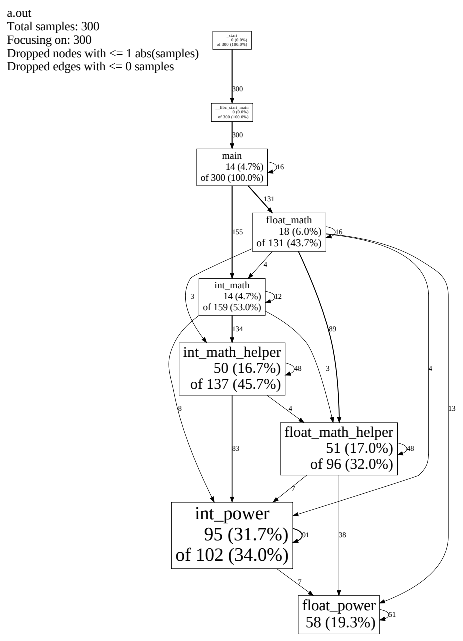
`gprof` **text output.** This is similar to the flat profile in `gprof`.

```
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:


a.out
Total samples: 300
Focusing on: 300
Dropped nodes with <= 1 abs(samples)
Dropped edges with <= 0 samples
_start 0 (0.0%) of 300 (100.0%)
300
__libc_start_main 0 (0.0%) of 300 (100.0%)
300
main 14 (4.7%) of 300 (100.0%)
16
131
155
float_math 18 (6.0%) of 131 (43.7%)
16
4
3
int_math 14 (4.7%) of 159 (53.0%)
12
134
89
int_math_helper 50 (16.7%) of 137 (45.7%)
48
3
4
13
8
4
float_math_helper 51 (17.0%) of 96 (32.0%)
48
83
7
int_power 95 (31.7%) of 102 (34.0%)
91
38
7
float_power 58 (19.3%)
51

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.

System-level profiling

Most profiling tools interrogate the CPU in more detail than `gprof` and friends. These tools are typically aware of the whole system, but may focus on one application, and may have both per-process and system-wide modes. We'll discuss a couple of these tools here, highlighting conceptual differences between these applications.

Solaris Studio Performance Analyzer. (Did not discuss in lecture.) This tool[1] supports `gprof`-style profiling ("clock-based profiling") as well as kernel-level profiling through DTrace (described later). At process level, it collects more process-level data than `gprof`, including page fault times and wait times. It also can read CPU performance counters (e.g. the number of executed floating point adds and multiplies). As a Sun application, it also works with Java programs.

Since locks and concurrency are important, modern tools, including the Studio Performance Analyzer, can track the amount of time spent waiting for locks, as well as statistics about MPI message passing. More on lock waits below, when we talk about WAIT.

VTune. (Did not discuss in lecture). Intel and AMD both provide profiling tools; Intel's VTune tool costs money, while AMD's CodeAnalyst tool is free software.

Intel uses the term "event-based sampling" to refer to sampling which fires after a certain number of CPU events occur, and "time-based sampling" to refer to the `gprof`-style sampling (e.g. every 100ms). VTune can also correlate the behaviour of the counters with other system events (like disk workload). Both of these sampling modes also include the behaviour of the operating system and I/O in their counts.

VTune also supports an instrumentation-based profiling approach, which measures time spent in each procedure (same type of data as `gprof`, but using a different collection scheme).

VTune will also tell you what it thinks the top problems with your software are. However, if you want to understand what it's saying, you do actually need to understand the architecture.

CodeAnalyst. (Discussed oprofile in lecture, but not CodeAnalyst specifics). AMD also provides a profiling tool. Unlike Intel's tool, AMD's tool is free software (the Linux version is released under the GPL), so that, for instance, Mozilla suggests that people include CodeAnalyst profiling data when reporting Firefox performance problems [2].

CodeAnalyst is a system-wide profiler. It supports drilling down into particular programs and libraries; the only disadvantage of being system-wide is that the process you're interested in has to execute often enough to show up in the profile. It also uses debug symbols to provide meaningful names; these symbols are potentially supplied over the Internet.

Like all profilers, it includes a sampling mode, which it calls "Time-based profiling" (TBP). This mode works on all processors. The other modes are "Event-based profiling" (EBP) and "Instruction-based sampling" (IBS); these modes use hardware performance counters.

AMD's CodeAnalyst documentation points out that your sampling interval needs to be sufficiently high to capture useful data, and that you need to take samples for enough time. The default sampling rate is once every millisecond, and they suggest that programs should run for at least 15 seconds to get meaningful data.

The EBP mode works like VTune's event-based sampling: after a certain number of CPU events occur, the profiler records the system state. That way, it knows where e.g. all the cache misses

[1] You can find a high-level description at `http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/oss-performance-tools-183986.pdf`

[2] `https://developer.mozilla.org/Profiling_with_AMD_CodeAnalyst`

are occuring. A caveat, though, is that EBP can't exactly identify the guilty statement, because of "skid": in the presence of out-of-order execution, guilt gets spread to the adjacent instructions.

To improve the accuracy of the profile information, CodeAnalyst uses AMD hardware features to watch specific x86 instructions and "ops", their associated backend instructions. This is the IBS mode[3] of CodeAnalyst. AMD provides an example[4] where IBS tracks down the exact instruction responsible for data translation lookaside buffer (DTLB) misses, while EBP indicates four potential guilty instructions.

oprofile. This free software is a sampling-based tool which uses the Linux Kernel Performance Events API to access CPU performance counters. It tracks the currently-running function (or even the line of code) and can, in system-wide mode, work across processes, recording data for every active application.

Webpage: `http://oprofile.sourceforge.net`.

You can run oprofile either in system-wide mode (as root) or per-process. To run it in system-wide mode:

```
% 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.
```

Or, per-process:

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

Profiling done.
```

Both of these invocations produce profiling output. You can read the profiling output by running `opreport` and giving it your executable.

```
% 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
```

[3]Available on AMD processors as of the K10 family—typically manufactured in 2007+; see `http://developer.amd.com/assets/AMD_IBS_paper_EN.pdf`. Thanks to Jonathan Thomas for pointing this out.

[4]`http://developer.amd.com/cpu/CodeAnalyst/assets/ISPASS2010_IBS_CA_abstract.pdf`

```
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 can also get better data:

```
% 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. This uses the same base data as oprofile, but provides a better (git-like) interface. Once again, it is an interface to the Linux kernel's built-in sample-based profiling using CPU counters. It works per-process, per-CPU, or system-wide. It can report the cost of each line of code.

Webpage: `https://perf.wiki.kernel.org/index.php/Tutorial`

Here's a usage example on the 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.

DTrace. DTrace[5][CSL04] is an instrumentation-based system-wide profiling tool designed to be used on production systems. It supports custom queries about system behaviour: when you are debugging system performance, you can collect all sorts of data about what the system is doing. The two primary design goals were in support of use in production: 1) avoid overhead when not tracing and 2) guarantee safety (i.e. DTrace can never cause crashes).

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

DTrace runs on Solaris and some BSDs. There is a Linux port, which may be usable. I'll try to install it on `ece459-1`.

Probe effect. "Wait! Don't 'instrumentation-based' and 'production systems' not go together[6]?"

Nope! DTrace was designed to have zero overhead when inactive. It does this by dynamically rewriting the code to insert instrumentation when requested. So, if you want to instrument all calls to the `open` system call, then DTrace is going to replace the instruction at the beginning of `open` with an unconditional branch to the instrumentation code, execute the profiling code, then return to your code. Otherwise, the code runs exactly as if you weren't looking.

Safety. As I've mentioned before, crashing a production system is a big no-no. DTrace is therefore designed to never cause a system crash. How? The instrumentation you write for DTrace must conform to fairly strict constraints.

DTrace system design. The DTrace framework supports instrumentation *providers*, which make *probes* (i.e. instrumentation points) available; and *consumers*, which enable probes as appropriate. Examples of probes include system calls, arbitrary kernel functions, and locking actions. Typically, probes apply at function entry or exit points. DTrace also supports typical sampling-based profiling in the form of timer-based probes; that is, it executes instrumentation every 100ms. This is tantamount to sampling.

You can specify a DTrace clause using probes, predicates, and a set of action statements. The action statements execute when the condition specified by the probe holds and the predicate evaluates to true. D programs consist of a sequence of clauses.

Example. Here's an example of a DTrace query from [CSL04].

```
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);
}
```

The first clause instruments all entries to the system call `read` and sets a thread-local variable `t` to the current time. The second clause instruments returns from `read` where the thread-local variable `t` is non-zero, calling `printf` to print out the relevant data.

The D (DTrace clause language) design ensures that clauses cannot loop indefinitely (since they can't loop at all), nor can they execute unsafe code; providers are responsible for providing safety guarantees. Probes might be unsafe because they might interrupt the system at a critical time. Or, action statements could perform illegal writes. DTrace won't execute unsafe code.

[6]For instance, Valgrind incurs a 100× slowdown.

Workflow. Both the USENIX article [CSL04] and the ACM Queue article referenced above contain example usages of DTrace. In high-level terms: first identify a problem; then, use standard system monitoring tools, plus custom DTrace queries, to collect data about the problem (and resolve it).

WAIT

Another approach which recently appeared in the research literature is the WAIT tool out of IBM. Unfortunately, this tool is not free and not generally available. Let's talk about it anyways.

Like DTrace, WAIT is suitable for use in production environments. It uses hooks built into modern Java Virtual Machines (JVMs) to analyze their idle time. It performs a sampling-based analysis of the behaviour of the Java VM. Note that its samples are quite infrequent; they suggest that taking samples once or twice a minute is enough. At each sample, WAIT records the state of each of the threads, which includes its call stack and participation in system locks. This data enables WAIT to compute (using expert rules) an abstract "wait state". The wait state indicates what the process is currently doing or waiting on, e.g. "disk", "GC", "network", or "blocked".

Workflow. You run your application, collect data (using a script or manually), and upload the data to the server. The server provides a report which you use to fix the performance problems. The report indicates processor utilization (idle, your application, GC, etc); runnable threads; waiting threads (and why they are waiting); thread states; and a stack viewer.

The paper presents six case studies where WAIT helped solve performance problems, including deadlocks, server underloads, memory leaks, database bottlenecks, and excess filesystem activity.

Other Applications of Profiling.

Profiling applies to languages beyond C/C++ and Java, of course. If you are profiling an interpreted language, you'll need a specific tool to get useful results. For Python, you can use `cProfile`; it is a standard implementation of profiling, from what I can see.

Here's a short tangent. Many of the concepts that we've seen for code also apply to web pages. Google's Page Speed tool[7], in conjunction with Firebug, helps profile web pages, and provides suggestions on how to make your web pages faster. Note that Page Speed includes improvements for the web page's design, e.g. not requiring multiple DNS lookups; leveraging browser caching; or combining images; as well as traditional profiling for the JavaScript on your pages.

[7] `http://code.google.com/speed/page-speed/`

Reentrancy versus Thread-Safety

On a different note, we're going to discuss the distinction between reentrant functions and thread-safe functions. There is overlap, but these terms are actually different.

A function is *reentrant* if it can be suspended in the middle and re-entered, or called again, before the previous execution returns.

Reentrant does not always mean **thread-safe** (although it usually is). Recall: **thread-safe** is essentially "no data races".

The distinction is moot if the function only modifies local data, e.g. `sin()`. Those functions are both reentrant and thread-safe.

Example. Courtesy of Wikipedia (with modifications), here's a program (to the left) and its trace (to right):

```
int t;

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

```
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
```

We can fix the example by storing the global variable in stack variable `s`, as follows.

```
int t;

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

    s = t;  // save global variable
    t = *x;
    *x = *y;
    // 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);
```

```
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
```

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

Let's see. Consider two calls to the reentrant swap:

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

```
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
```

To recap what we know so far: re-entrant does not always mean thread-safe. (But, for most sane implementations, reentrant is also thread-safe.)

But, 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[8].

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

To sum up, here's the 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.

If it helps, here's another definition of thread-safety.

> "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. Consider the following function.

```
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.

[8]I'm talking about lock implementations which don't maintain a lock counter; you can request Java locks multiple times in the same thread and everything will be fine, but not pthreads locks.

Good Programming 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.

OK, so inlining removes overhead. Sounds like better performance! Let's inline everything! There are two ways of 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. Or, we can be explicit:

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

The Other Side of Inlining. Inlining has 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.

Note also that 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.

Implications of inlining. Inlining can make your life worse in two ways. First, 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.

Second, it can be a problem for 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. (Congratulations, you made a non-binary-compatible change!)

This would not be a problem for non-inlined functions—programs execute the new function dynamically at runtime.

References

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 15–28, Berkeley, CA, USA, 2004. USENIX Association.