Lecture 25 — System-Level Profiling, Profiler Guided Optimization

Patrick Lam & Jeff Zarnett
patrick.lam@uwaterloo.ca, jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

October 14, 2020

System Profiling

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).

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

`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.

DTrace: Introduction

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.

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!)

No-One Expects the Profiling Tools!

AMONGST our profiling tools are such diverse elements AS...

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.

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 I

Profiler Guided Optimization

Using static analysis,
the compiler makes its best predictions about runtime behaviour.

Example: branch prediction.

```
void whichBranchIsTaken(int a, int b)
{
    if (a < b) {
        puts("a is less than b.");
    } else {
        puts("b is >= a.");
    }
}
```

A Virtual Call to Devirtualize

```
void devirtualization(int count)
{
    for (int i = 0; i < count; i++)
    {
        (*p) (x, y);
    }
}
```

```c
void switchCaseExpansion(int i)
{
    switch (i)
    {
    case 1:
        puts("I took case 1.");
        break;
    case 2:
        puts("I took case 2.");
        break;
    }
}
```

How can we know where we go?

- could provide hints…

Java HotSpot virtual machine:
updates predictions on the fly.

So, just guess.
If wrong, the Just-in-Time compiler adjusts & recompiles.

The compiler runs and it does its job and that's it; the program is never updated with newer predictions if more data becomes known.

Profiling Mitigates Uncertainty

C: usually no adaptive runtime system.

POGO:

- observe actual runs;
- predict the future.

So, we need multi-step compilation:

- compile with profiling;
- run to collect data;
- recompile with profiling data to optimize.

Step One: Measure

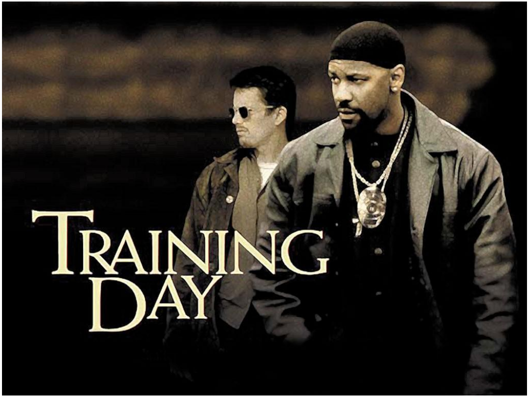
First, generate an executable with instrumentation.

The compiler inserts a bunch of probes into the generated code to record data.

- Function entry probes;
- Edge probes;
- Value probes.

Result: instrumented executable
plus empty database file (for profiling data).

Step Two: Training Day



Step Two: Training Day

Second, run the instrumented executable.

Real-world scenarios are best.

Ideally, spend training time on perf-critical sections.

Use as many runs as you can stand.

Don’t exercise every part of the program
(ain’t SE 465 here!)

That would be counterproductive.

Usage data must match real world scenarios,
or compiler gets misfacts about what’s important.

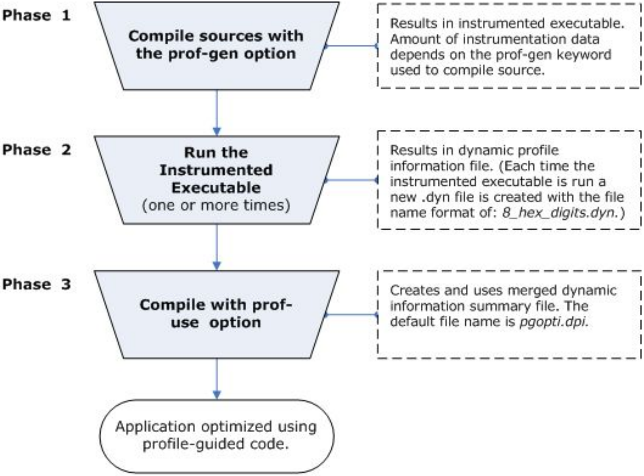
Or you might end up teaching it that almost nothing is important…(“everything’s on the exam!”)

Step Three: Recompile

Finally, compile the program again.

Inputs: source plus training data.

Outputs: (you hope) a better output executable than from static analysis alone.


Phase 1
Compile sources with the prof-gen option
Results in instrumented executable. Amount of instrumentation data depends on the prof-gen keyword used to compile source.
Phase 2
Run the Instrumented Executable
(one or more times)
Results in dynamic profile information file. (Each time the instrumented executable is run a new .dyn file is created with the file name format of: 8_hex_digits.dyn.)
Phase 3
Compile with prof-use option
Creates and uses merged dynamic information summary file. The default file name is pgopti.dpi.
Application optimized using profile-guided code.

Not necessary to do all three steps for every build.

Re-use training data while it's still valid.

Recommended dev workflow:

- dev A performs these steps,
  checks the training data into source control
- whole team can use profiling information for their compiles.

Not fixing all the problems in the world

What does it mean for it to be better?

The algorithms will aim for
speed in areas that are “hot”.

The algorithms will aim for
minimal code size in areas that are “cold” .

Less than 5% of methods compiled for speed.

Can combine multiple training runs and manually give suggestions about important scenarios.

The more a scenario runs in the training data,
the more important it will be,
from POGO's point of view.

Can merge multiple runs
with user-assigned weightings.

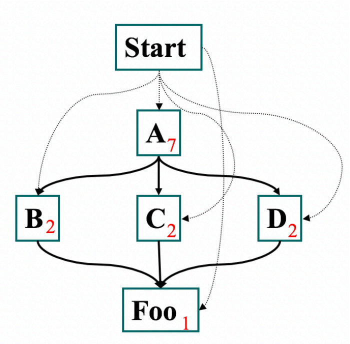
In the optimize phase, compiler uses the training data for:

1. Full and partial inlining
2. Function layout
3. Speed and size decision
4. Basic block layout
5. Code separation
6. Virtual call speculation
7. Switch expansion
8. Data separation
9. Loop unrolling

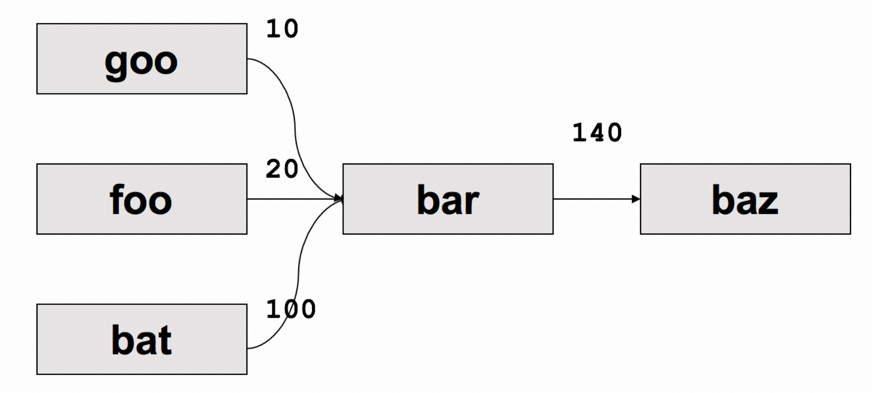
Most performance gains from inlining.

Decisions based on the call graph path profiling.

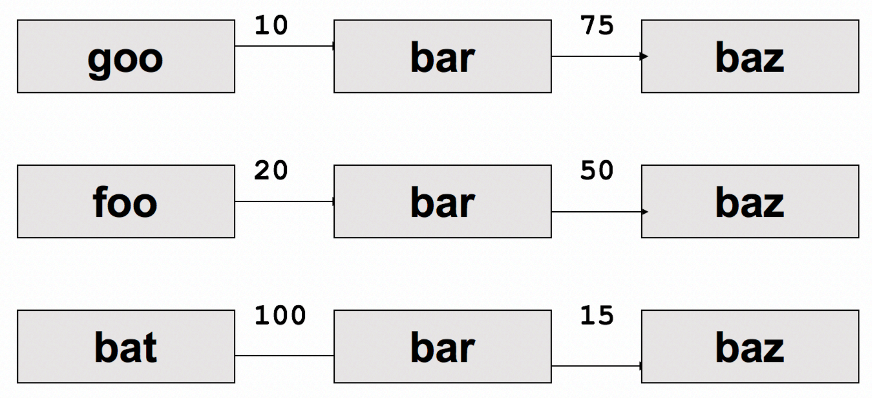
But: behaviour of function `foo` may be very different when called from B than when called from D.

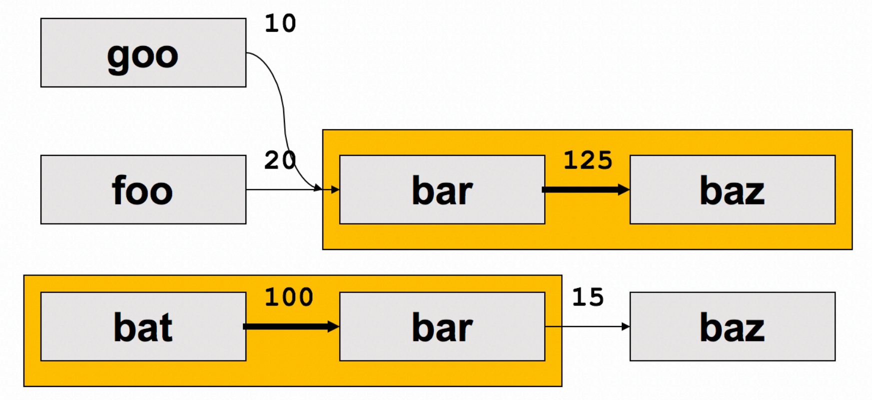


Example 2 of relationships between functions.
Numbers on edges represent the number of invocations:



When considering what to do here, POGO takes the view like this:



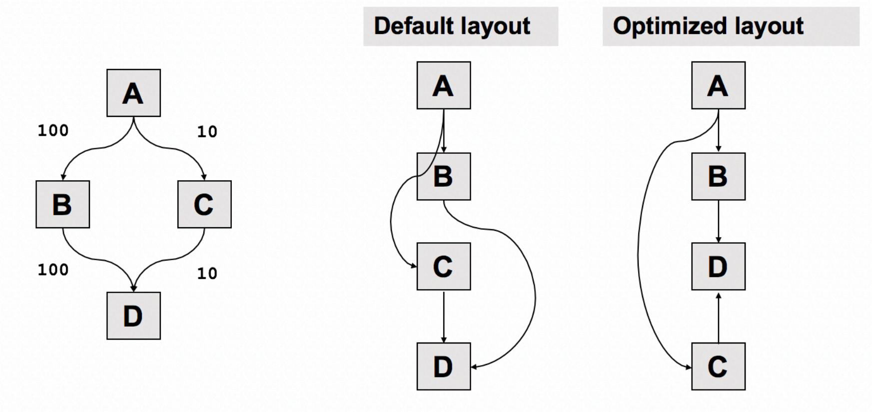

goo
10
foo
20
bar
125
baz
bat
100
bar
15
baz

Call graph profiling data also good for packing blocks.

Put most common cases nearby.
Put successors after their predecessors.

Packing related code = fewer page faults (cache misses).

Calling a function in same page as caller = “page locality”.


A
100
10
B
C
100
10
D
Default layout
A
B
C
D
Optimized layout
A
B
D
C

Dead Code?

According to the author, “dead” code goes in its own special block.

Probably not truly dead code (compile-time unreachable).

Instead: code that never gets invoked in training.

OK, how well does POGO work?

The application under test is a standard benchmark suite (Spec2K):

Spec2k:	**sjeng**	**gobmk**	**perl**	**povray**	**gcc**
App Size:	Small	Medium	Medium	Medium	Large
Inlined Edge Count	50%	53%	25%	79%	65%
Page Locality	97%	75%	85%	98%	80%
Speed Gain	8.5%	6.6%	14.9%	36.9%	7.9%

We can speculate about how well synthetic benchmarks results translate to real-world application performance…