Lecture 26 — Profilers, Profiler Guided Optimization

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

Department of Electrical and Computer Engineering
University of Waterloo

December 5, 2020

Part I

Profiling Tools

No-One Expects the Profiling Tools!

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

Profiling Tools

- Solaris Studio Performance Analyzer
- VTune (Intel)
- CodeAnalyst (AMD)

Plus a few more we'll consider in some more detail.

DTrace: Introduction

Intrumentation-based tool.
System-wide.
Meant to be used on production systems.

(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 expressiveness—no loops.

- (Hence, no infinite loops!)

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.

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.

Paper presents 6 case studies where WAIT identified performance problems.

Other Profiling Tools

Profiling: Not limited to regular compiled program 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.

I ran the command `nvprof target/release/nbody-cuda`.

```
==20734== Profiling application: target/release/nbody-cuda
==20734== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  10.7599s         1  10.7599s  10.7599s  10.7599s  calculate_forces
                    0.00%  234.72us         2  117.36us  100.80us  133.92us  [CUDA memcpy HtoD]
                    0.00%  94.241us         1  94.241us  94.241us  94.241us  [CUDA memcpy DtoH]
      API calls:   97.48%  10.7599s         1  10.7599s  10.7599s  10.7599s  cuStreamSynchronize
                    1.92%  211.87ms         1  211.87ms  211.87ms  211.87ms  cuCtxCreate
                    0.54%  59.648ms         1  59.648ms  59.648ms  59.648ms  cuCtxDestroy
                    0.04%  4.8704ms         1  4.8704ms  4.8704ms  4.8704ms  cuModuleLoadData
                    0.00%  404.72us         2  202.36us  194.51us  210.21us  cuMemAlloc
                    0.00%  400.58us         2  200.29us  158.08us  242.50us  cuMemcpyHtoD
                    0.00%  299.30us         2  149.65us  121.42us  177.88us  cuMemFree
                    0.00%  243.86us         1  243.86us  243.86us  243.86us  cuMemcpyDtoH
                    0.00%  85.000us         1  85.000us  85.000us  85.000us  cuModuleUnload
                    0.00%  41.356us         1  41.356us  41.356us  41.356us  cuLaunchKernel
                    0.00%  18.483us         1  18.483us  18.483us  18.483us  cuStreamCreateWithPriority
                    0.00%  9.0780us         1  9.0780us  9.0780us  9.0780us  cuStreamDestroy
                    0.00%  2.2080us         2  1.1040us     215ns  1.9930us  cuDeviceGetCount
                    0.00%  1.4600us         1  1.4600us  1.4600us  1.4600us  cuModuleGetFunction
                    0.00%  1.1810us         2     590ns     214ns     967ns  cuDeviceGet
                    0.00%     929ns         3     309ns     230ns     469ns  cuDeviceGetAttribute
```

And the better version...

Oh, and for comparison, here's the one where I make much better use of the GPU's capabilities (with better grid and block settings):

```
=22619== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.92%  417.53ms         1  417.53ms  417.53ms  417.53ms  calculate_forces
                    0.06%  236.03us         2  118.02us  101.44us  134.59us  [CUDA memcpy HtoD]
                    0.02%  93.057us         1  93.057us  93.057us  93.057us  [CUDA memcpy DtoH]
      API calls:   52.09%  417.54ms         1  417.54ms  417.54ms  417.54ms  cuStreamSynchronize
                   26.70%  214.00ms         1  214.00ms  214.00ms  214.00ms  cuCtxCreate
                   13.63%  109.26ms         1  109.26ms  109.26ms  109.26ms  cuModuleLoadData
                    7.42%  59.502ms         1  59.502ms  59.502ms  59.502ms  cuCtxDestroy
                    0.05%  364.08us         2  182.04us  147.65us  216.42us  cuMemcpyHtoD
                    0.04%  306.48us         2  153.24us  134.10us  172.37us  cuMemAlloc
                    0.04%  285.73us         2  142.86us  122.90us  162.83us  cuMemFree
                    0.03%  246.37us         1  246.37us  246.37us  246.37us  cuMemcpyDtoH
                    0.01%  61.916us         1  61.916us  61.916us  61.916us  cuModuleUnload
                    0.00%  26.218us         1  26.218us  26.218us  26.218us  cuLaunchKernel
                    0.00%  15.902us         1  15.902us  15.902us  15.902us  cuStreamCreateWithPriority
                    0.00%  9.0760us         1  9.0760us  9.0760us  9.0760us  cuStreamDestroy
                    0.00%  1.6720us         2     836ns     203ns  1.4690us  cuDeviceGetCount
                    0.00%  1.0950us         1  1.0950us  1.0950us  1.0950us  cuModuleGetFunction
                    0.00%     888ns         3     296ns     222ns     442ns  cuDeviceGetAttribute
                    0.00%     712ns         2     356ns     212ns     500ns  cuDeviceGet
```

Part II

Profiler Guided Optimization

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

Example: branch prediction.

```
fn which_branch(a: i32, b: i32) {
    if a < b {
        println!("Case one.");
    } else {
        println!("Case two.");
    }
}
```

A Virtual Call to Devirtualize

```
trait Polite {
    fn greet(&self) -> String;
}

struct Kenobi {
    /* Stuff */
}
impl Polite for Kenobi {
    fn greet(&self) -> String {
        return String::from("Hello
            there!");
    }
}
```

```
struct Grievous {
    /* Things */
}
impl Polite for Grievous {
    fn greet(&self) -> String {
        return String::from("General
            Kenobi.");
    }
}

fn devirtualization(thing: &Polite) {
    println!("{}", thing.greet());
}
```

```rust
fn match_thing(x: i32) -> i32 {
    match x {
        0..10 => 1,
        11..100 => 2,
        _ => 0
    }
}
```

Same thing with x: what is its typical value? If we know that, it is our prediction.

Actually, in a match block with many options, could we rank them in descending order of likelihood?

Adapting to an Uncertain World

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

Rust: 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

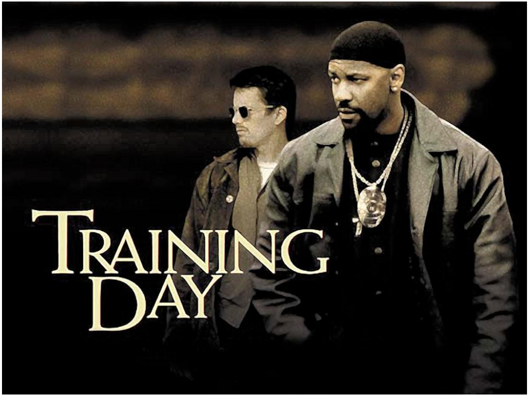
First, generate an executable with instrumentation.

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 (not SE 465/ECE 453 here!)

That would be counterproductive.

Usage data must match real world scenarios,
... or the compiler gets misinformed about what's important.

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

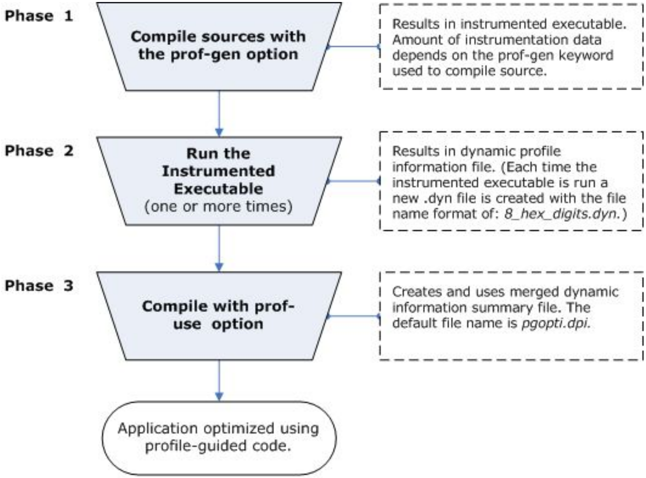
Step Three: Recompile

Finally, compile the program again.

Inputs: source plus training data.

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

Summary Graphic



Save Some Steps

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.

Combining Training Runs

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.

Full Test Scenario

```
# STEP 1: Compile the binary with instrumentation
rustc -Cprofile-generate=/tmp/pgo-data -O ./main.rs

# STEP 2: Run the binary a few times, maybe with common sets of args.
#         Each run will create or update ‘.profraw‘ files in /tmp/pgo-data
./main mydata1.csv
./main mydata2.csv
./main mydata3.csv

# STEP 3: Merge and post-process all the ‘.profraw‘ files in /tmp/pgo-data
llvm-profdata merge -o ./merged.profdata /tmp/pgo-data

# STEP 4: Use the merged ‘.profdata‘ file during optimization. All ‘rustc‘
#         flags have to be the same.
rustc -Cprofile-use=./merged.profdata -O ./main.rs
```

Behind the Scenes

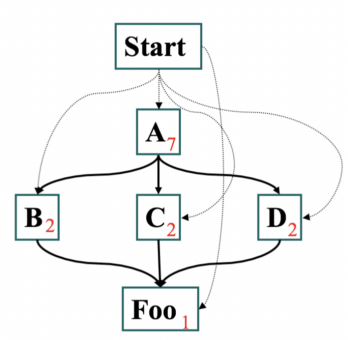
In the optimize phase, compiler uses the training data for:

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.

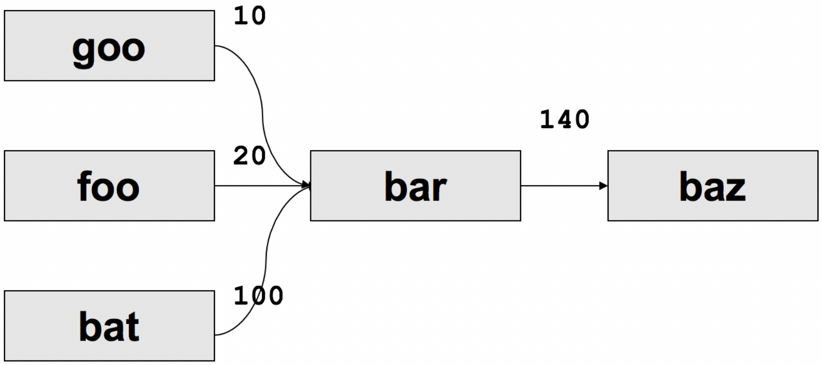
Decisions based on the call graph path profiling.

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



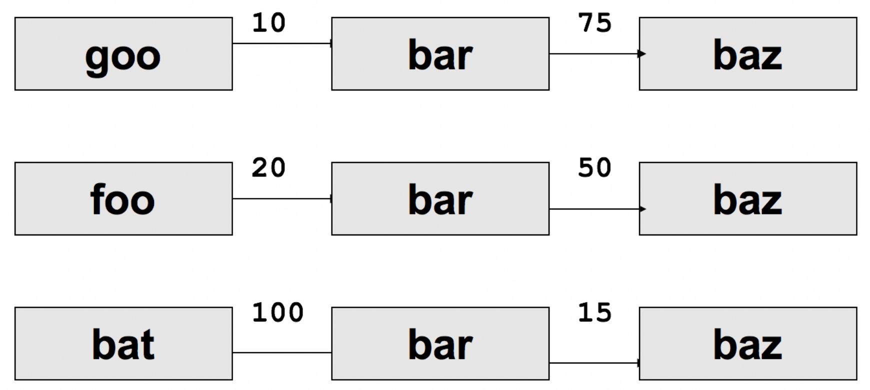
Another Call Graph

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

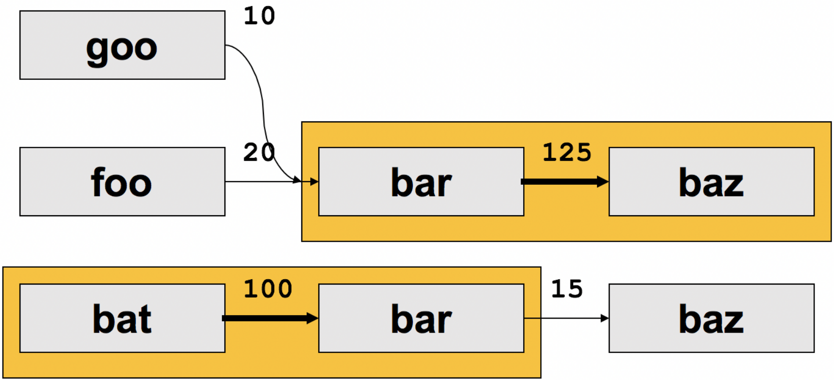


The POGO View of the World

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



The POGO View of the World



Page Locality

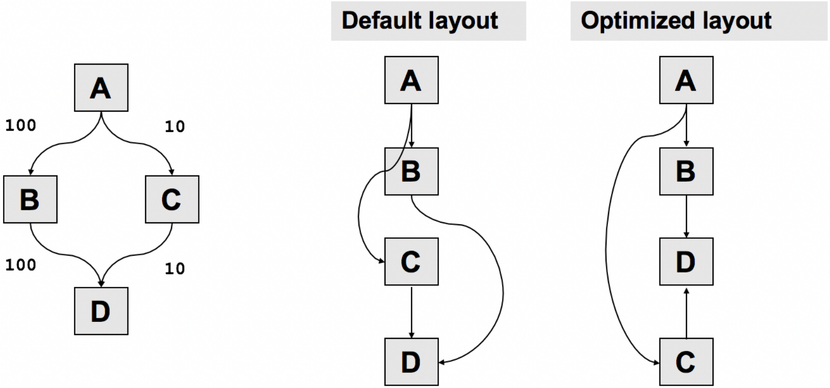
Call graph profiling data also good for packing blocks.

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”.

Block Layout



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…