

Lecture 25 — Profiling

Patrick Lam & Jeff Zarnett

`patrick.lam@uwaterloo.ca, jzarnett@uwaterloo.ca`

Department of Electrical and Computer Engineering
University of Waterloo

November 20, 2022

Part I

Profiling

Think back: what operations are fast and what operations are not?

Takeaway: our intuition is often wrong.

Not just at a macro level,
but at a micro level.

You may be able to narrow down that this computation of x is slow,
but if you examine it carefully... what parts of it are slow?

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

– Donald Knuth

That Saying You Were Expecting

Feeling lucky?
Maybe you optimized a slow part.



To make your programs or systems fast,
you need to find out what is currently slow and improve it (duh!).

Up until now, it's mostly been about
“let's speed this up”.
We haven't taken much time to decide what we should speed up.

General idea:

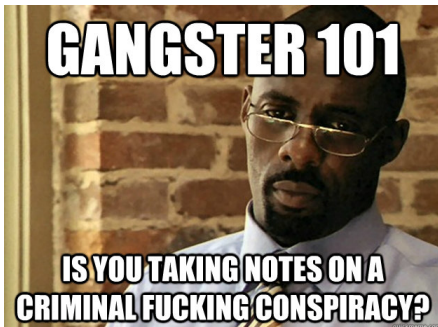
collect data on what parts of the code
are taking up most of the time.

- What functions get called?
- How long do functions take?
- What's using memory?

There is always the “informal” way: “debug” by print statements.

On entry to foo, log “entering function foo”, plus timestamp.

On exit, log “exiting foo”, also with timestamp.



Kind of works, and [JZ] has used it... But this approach is not necessarily good.

This is “invasive” profiling—change the source code of the program to add instrumentation (log statements).

Plus we have to do a lot of manual accounting.

Doesn't really scale.

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

==20734== Profiling application: target/release/nbody-cuda

==20734== Profiling result:

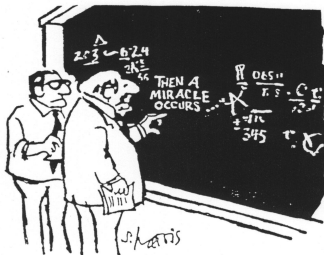
Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	100.00%	10.7599s	1	10.7599s	10.7599s	10.7599s	calculate_forces
	0.00%	234.72us	2	117.36us	100.80us	133.92us	[CUDA memcpy HtoD]
	0.00%	94.241us	1	94.241us	94.241us	94.241us	[CUDA memcpy DtoH]
API calls:	97.48%	10.7599s	1	10.7599s	10.7599s	10.7599s	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

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

Like debugging: if you get to be a wizard you can maybe do it by code inspection.



I think you should be a little more specific, here in Step 2

But speculative execution inside your head is harder for perf than debugging.

So: we want to use tools and do this in a methodical way.

So far we've been looking at small problems.

Must **profile** to see what's slow in a large program.

Two main outputs:

- flat;
- call-graph.

Two main data gathering methods:

- statistical;
- instrumentation.

Flat Profiler:

- Only computes average time in a particular function.
- Does not include other (useful) information (callees).

Call-graph Profiler:

- Computes call times.
- Reports frequency of function calls.
- Gives a call graph: who called what function?

Statistical:

Mostly, take samples of the system state, that is:

- every 100ms, check the system state.
- will cause some slowdown, but not much.

Instrumentation:

Add instructions at specified program points:

- can do this at compile time or run time (expensive);
- can instrument either manually or automatically;
- like conditional breakpoints.

When writing large software projects:

- First, write clear and concise code.
Don't do premature optimizations—
focus on correctness.
- Profile to get a baseline of your performance:
 - allows you to easily track any performance changes;
 - allows you to re-design your program before it's too late.

Focus your optimization efforts on the code that matters.

Good signs:

- Time is spent in the right part of the system.
- Most time should not be spent handling errors; in non-critical code; or in exceptional cases.
- Time is not unnecessarily spent in OS.



Kitchener driver follows GPS directions right into Lake Huron...

You can always profile your systems in development, but that might not help with complexities in production.

The constraints on profiling production systems are that the profiling must not affect the system's performance or reliability.

We'll first talk about `perf`, the profiler recommended for use with Rust. This is Linux-specific, though.

The `perf` tool is an interface to the Linux kernel's built-in sample-based profiling using CPU counters.

```
[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		

The first thing to do is to compile with debugging info, go to your `Cargo.toml` file and add:

```
[profile.release]
debug = true
```

This means that `cargo build --release` will now compile the version with debug info.

Run the program using `perf record`, which will sample the execution of the program to produce a data set.

Then there are three ways we can look at the code: `perf report`, `perf annotate`, and a flamegraph.

During development of some of the code exercises, I used the CLion built-in profiler for this purpose.

It generates a flamegraph for you too, and I'll show that for how to create the flamegraph as well.

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());  
}
```

```
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?

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.

Rust: usually no adaptive runtime system.

POGO:

- observe actual runs;
- predict the future.

So, we need multi-step compilation:

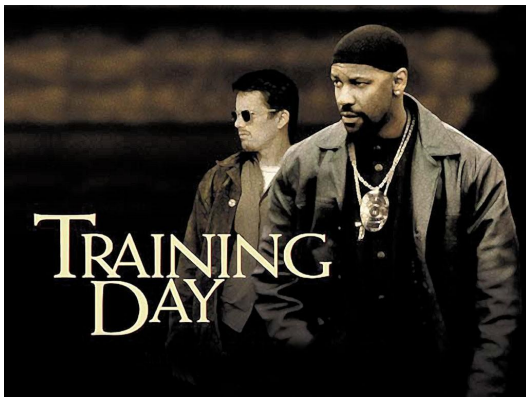
- compile with profiling;
- run to collect data;
- recompile with profiling data to optimize.

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).



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.

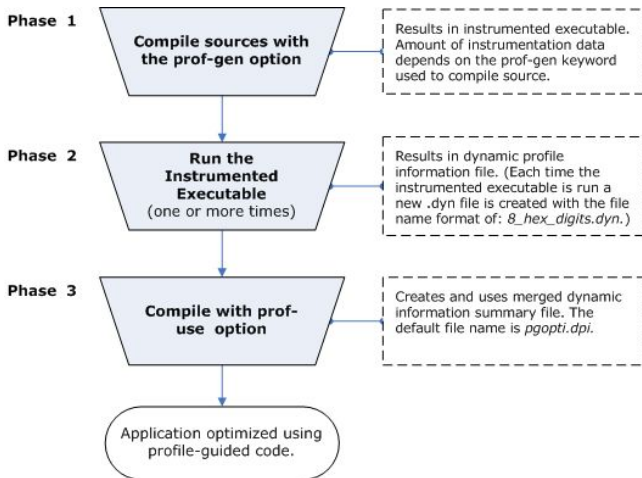
Or you might end up teaching it that almost nothing is
important... (“everything's on the exam!”)

Finally, compile the program again.

Inputs: source plus training data.

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

Summary Graphic



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.

```
# 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 '.profrac' files in /tmp/pgo-data
./main mydata1.csv
./main mydata2.csv
./main mydata3.csv

# STEP 3: Merge and post-process all the '.profrac' 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
```

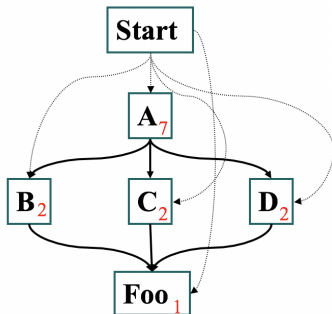
In the optimize phase, compiler uses the training data for:

- 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

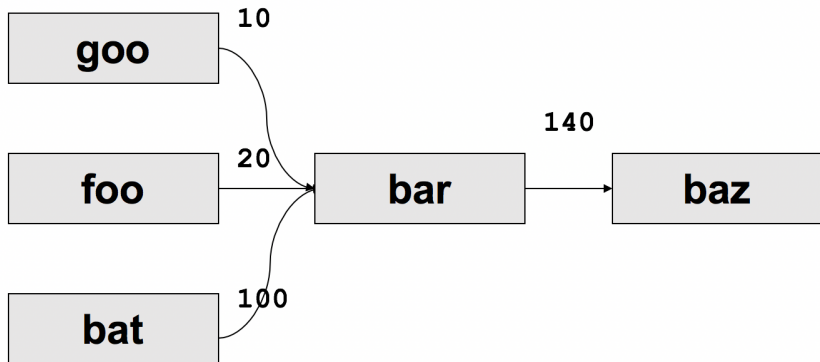
Most performance gains from inlining.

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.

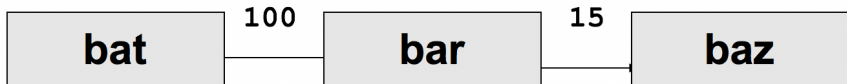


Example 2 of relationships between functions.
Numbers on edges represent the number of invocations:

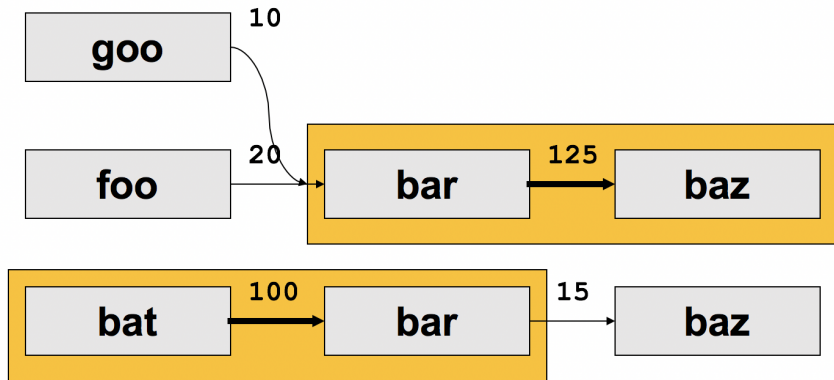


The POGO View of the World

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



The POGO View of the World



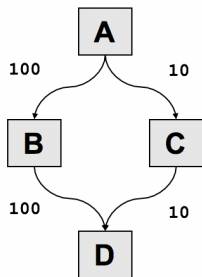
Call graph profiling data also good for packing blocks.

Put most common cases nearby.

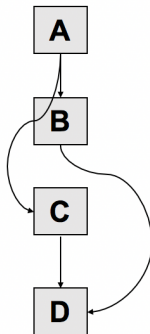
Put successors after their predecessors.

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

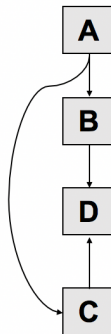
Calling a function in same page as caller = “page locality”.



Default layout



Optimized layout



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...