Lecture 28 — Profiler Guided Optimization

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

Department of Electrical and Computer Engineering
University of Waterloo

March 13, 2018

Profiler Guided Optimization

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

Example: branch prediction.

A Branch To Predict

```
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);
    }
}
```

A Switch to Predict

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

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

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

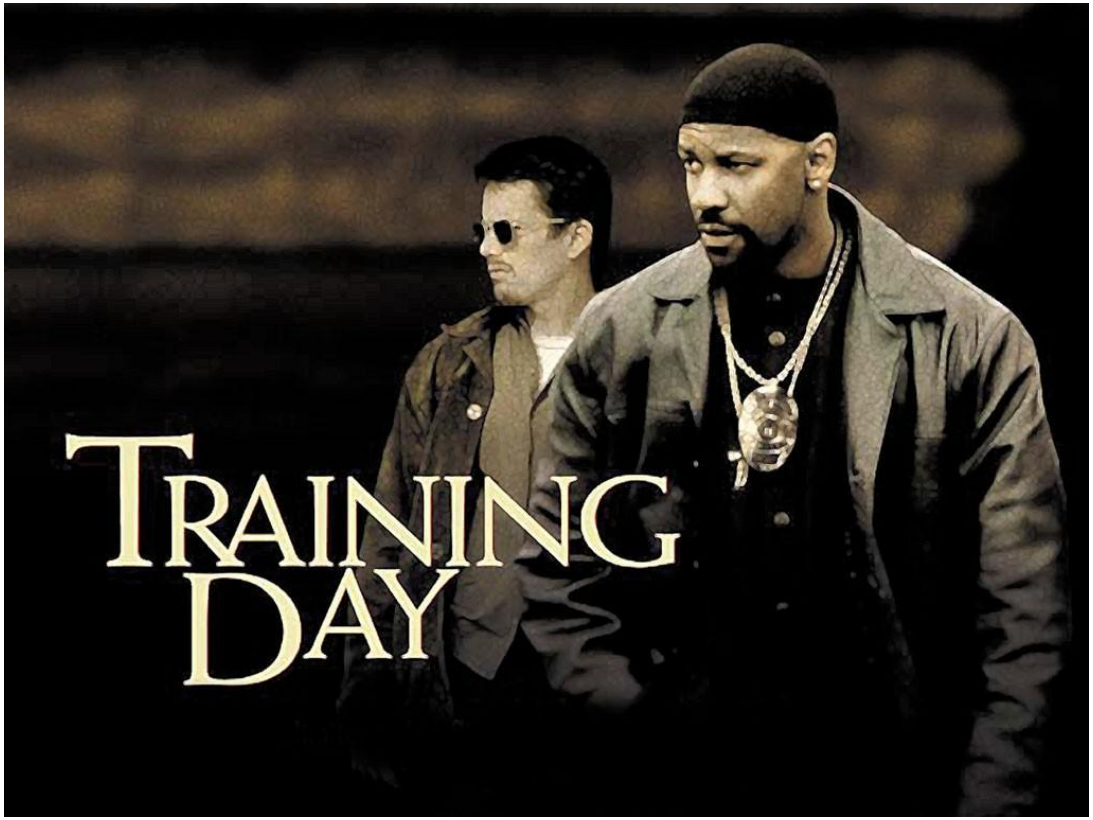
First, generate an executable with instrumentation.

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.

Step Two: Training Day

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!”)

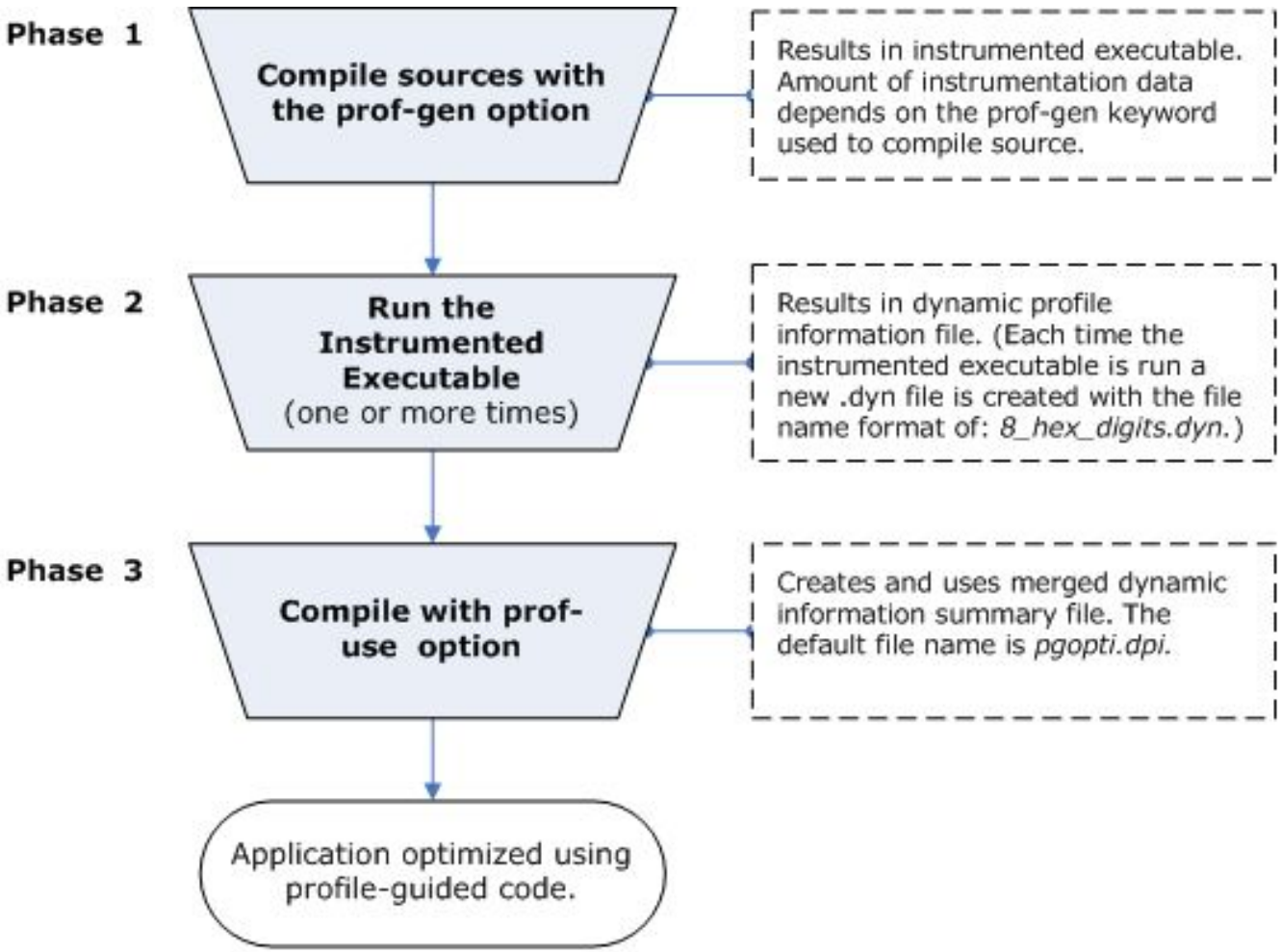
Step Three: Recompile

Finally, compile the program again.

Inputs: source plus training data.

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

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.

Show, Don't Tell

With the theory behind us, perhaps a demonstration about how it works is in order.

Let us consider an example using the N-Body problem.

You could look at the video
`https://www.youtube.com/watch?v=zEsdBcu4R00&t=21m45s`
(from 21:45 to 34:23).

Behind the Scenes

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

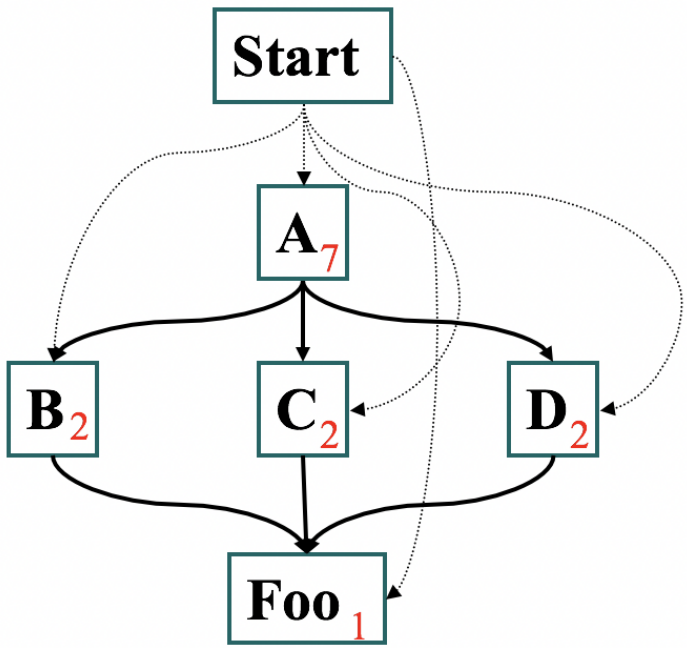
1. Full and partial inlining
2. Function layout
3. Speed and size decision
4. Basic block layout
5. Code separation
6. Virtual call speculation
7. Switch expansion
8. Data separation
9. Loop unrolling

Muh Gainz

Most performance gains from inlining.

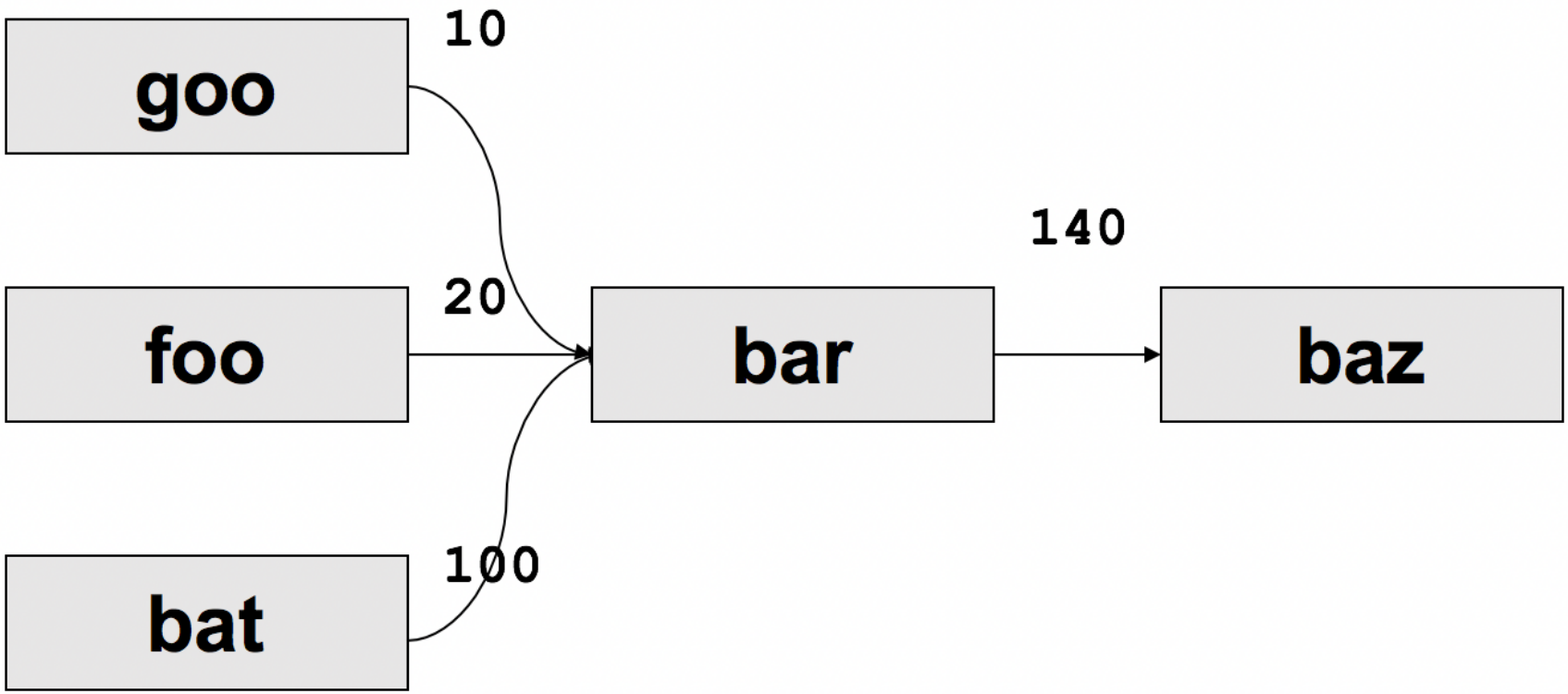
Decisions based on the call graph path profiling.

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



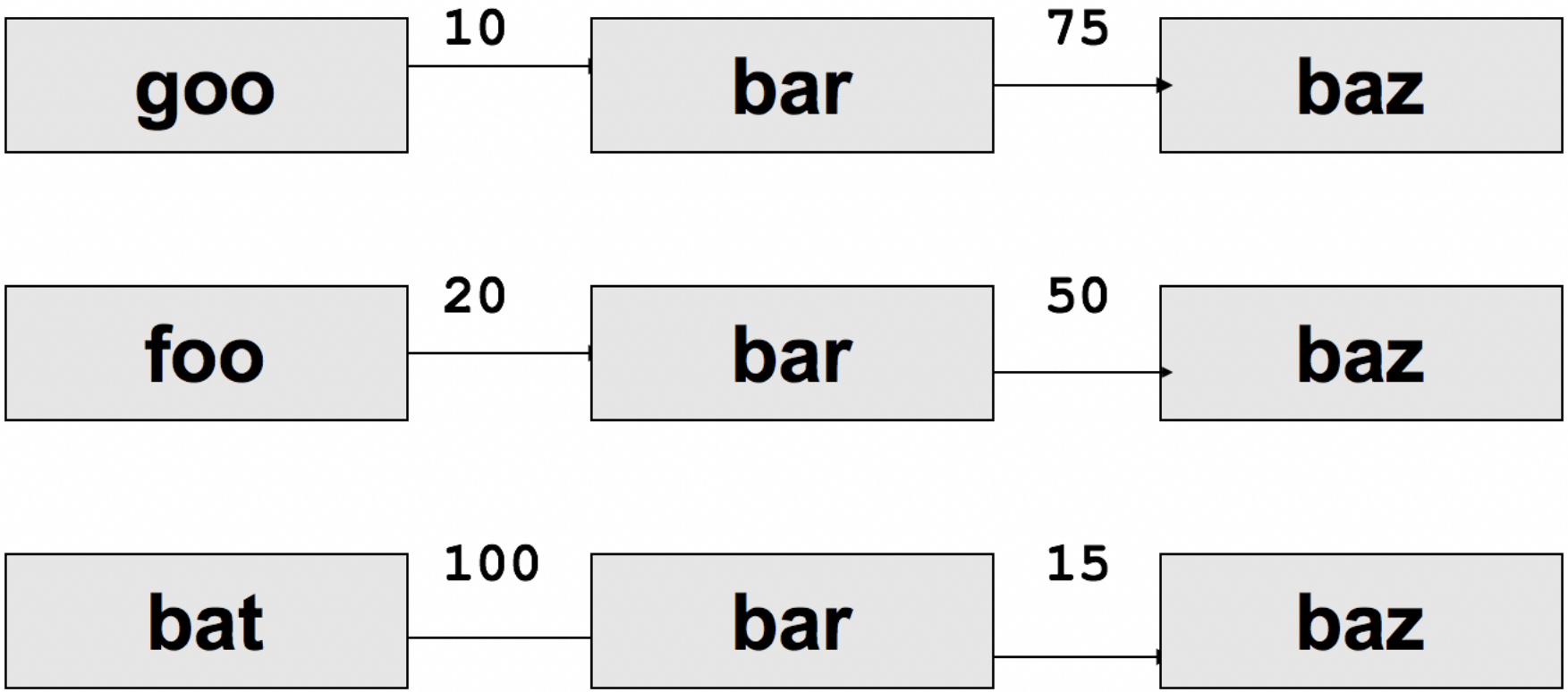
Another Call Graph

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

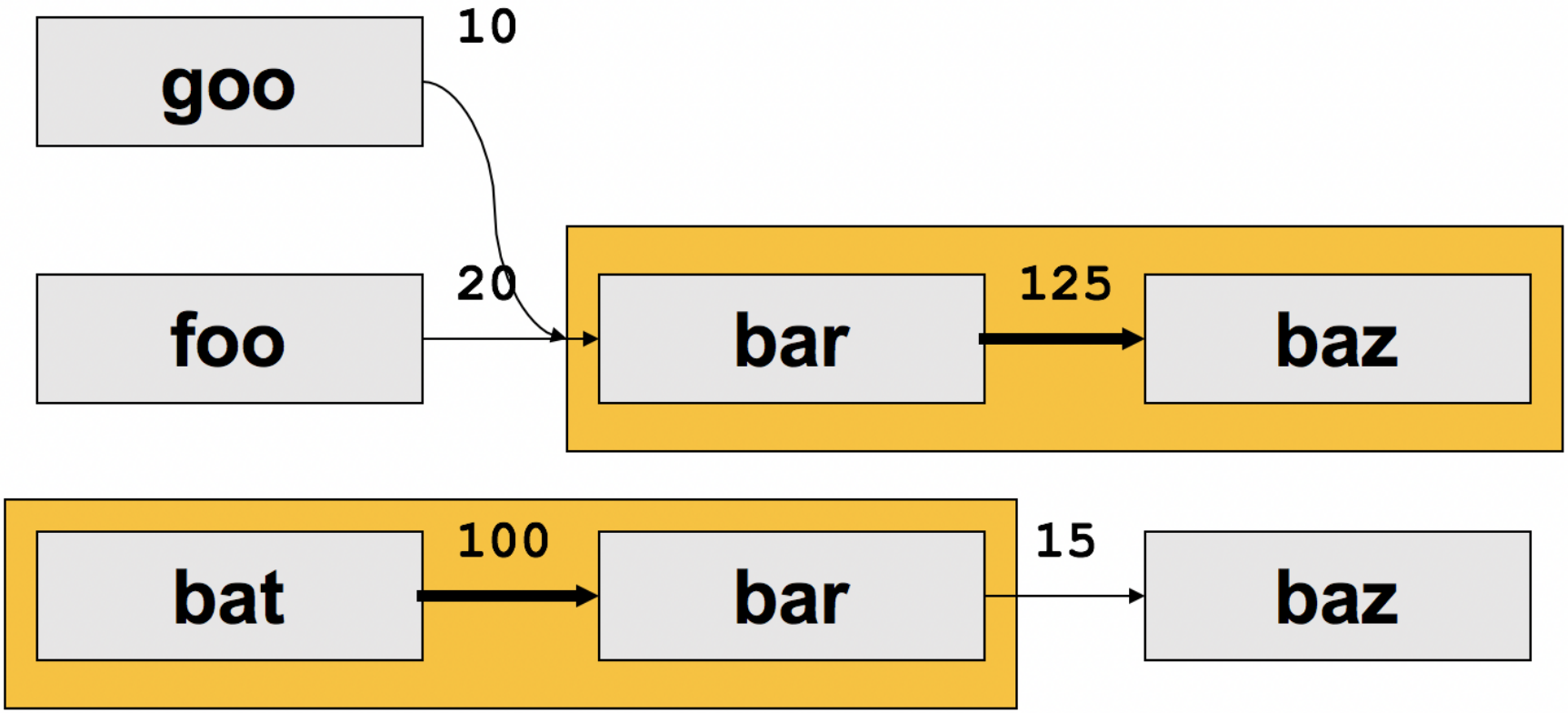


The POGO View of the World

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



The POGO View of the World



Page Locality

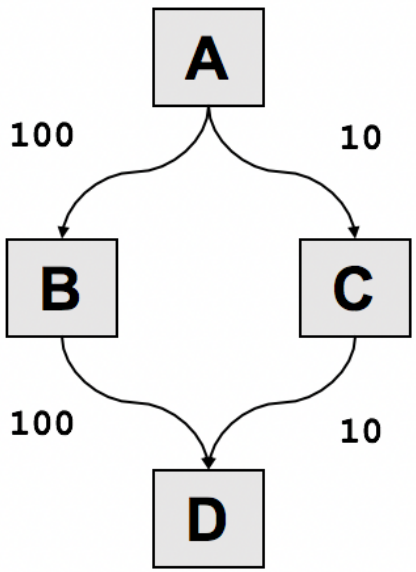
Call graph profiling data also good for packing blocks.

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

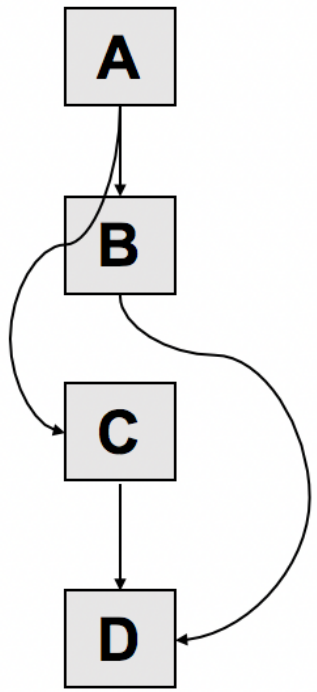
Packing related code = fewer page faults (cache misses).

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

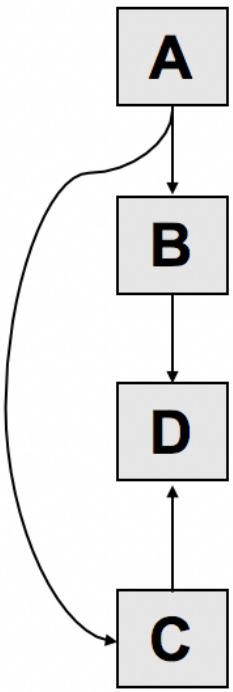
Block Layout



Default layout



Optimized 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.

Benchmark Results

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…