Lecture 28 — Profiler Guided Optimization

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 6, 2017

Profiler Guided Optimization

The compiler does a great deal of static analysis of the code you've written and makes its best guesses about what is likely to happen.

The canonical example for this is branch prediction.

There is an if-else block and the compiler will then guess about which is more likely and optimize for that version.

Profiler Guided Optimization

```
void whichBranchIsTaken(int a, int b) {
        if (a < b) {
                printf(" a is less than b. \n");
        } else {
                printf(" b is greater than or equal to a. \n");
        }
}

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


void switchCaseExpansion(int i) {
        switch(i) {
                case 1:
                        printf(" Case 1 was chosen \n");
                        break;
                case 2:
                        printf(" Case 2 was chosen \n");
                        break;
        }
}
```

Hints and Java

There exists a solution to this, and it is that we can give hints to the compiler, but that's a manual process.

The Java HotSpot virtual machine will update its predictions on the fly.

There are some initial predictions and if they turn out to be wrong, the Just In Time compiler will replace it with the other version.

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.

Three Steps

Solving this problem is the goal of POGO.

It is taking the data from some actual runs of the program and using that to inform the predictions.

This necessitates a multi-step compile: first compile the code, run it to collect data, then recompile the code using the data we collected.

Step One: Measure

Step one is to generate an executable with instrumentation.

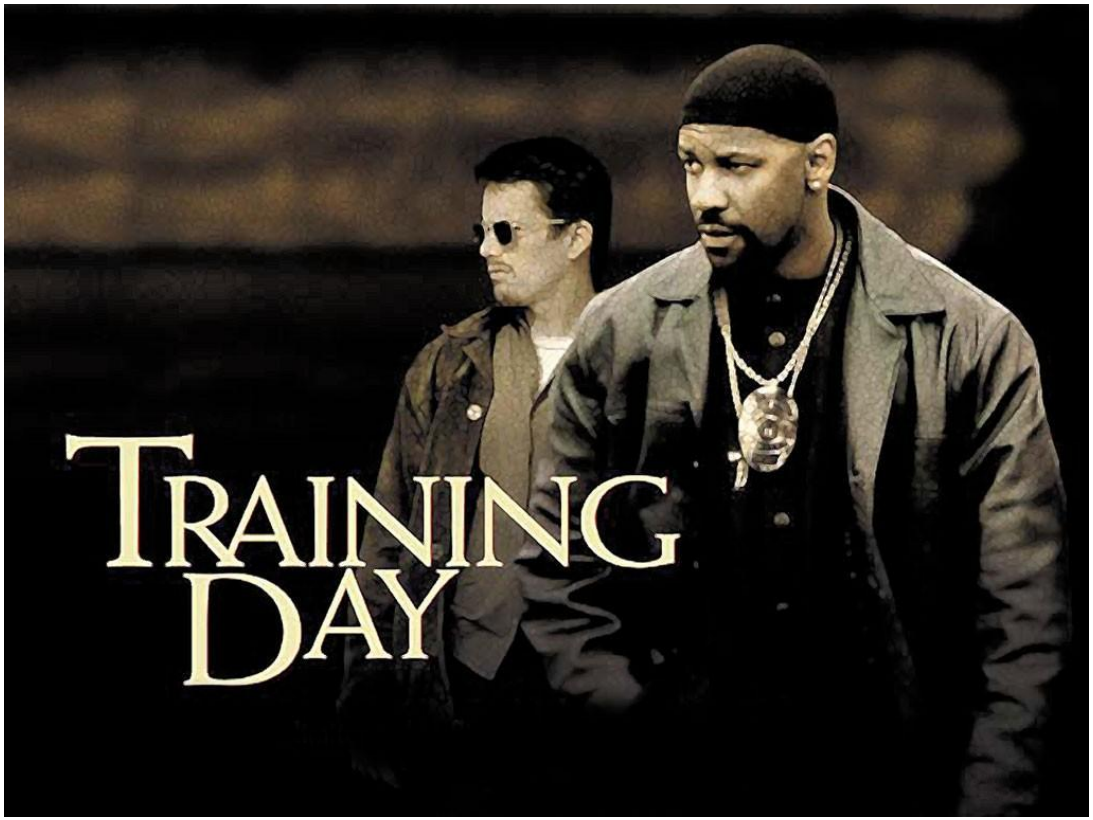
The compiler inserts a bunch of probes into the generated code that are used to record data.

Three types of probe are inserted:

- Function entry probes
- Edge probes
- Value probes

When this phase is complete, there is an instrumented executable and an empty database file where the training data goes.

Step Two: Training Day



Step Two: Training Day

Run the instrumented executable through real-world scenarios.

Ideally you will spend the training time on the performance-critical sections.

It does not have to be a single training run, of course, data can be collected from as many runs as desired.

Step Two: Training Day

You are not trying to exercise every part of the program (this is not unit testing)!

In fact, trying to use every bell and whistle of the program is counterproductive.

If the usage data does not match real world scenarios then the compiler has been given the wrong information about what is important.

Or you might end up teaching it that almost nothing is important...

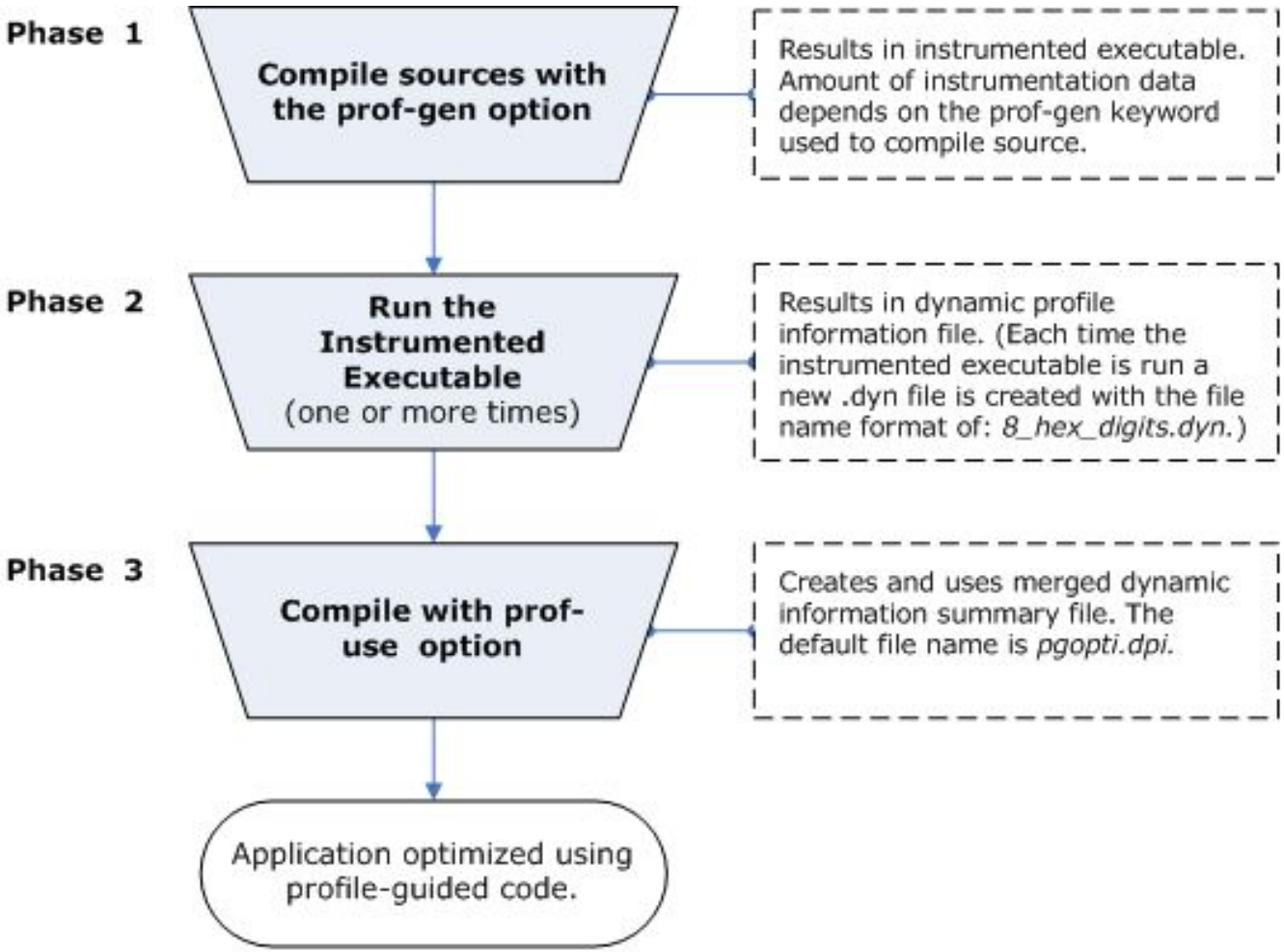
Step Three: Recompile

Step three is a recompile.

This time, in addition to the source files, the training data is fed into the compiler for a second compile.

This data is applied to produce a better output executable than could be achieved by static analysis alone.

Summary Graphic



Save Some Steps

It is not necessary to do all three steps for every build.

Old training data can be re-used until the code base has diverged significantly enough from the instrumented version.

Recommended workflow: one developer performs these steps and check the training data into source control so that other developers can make use of it .

Define “Better”

What does it mean for it to be better?

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

The algorithms will aim to minimize the size of code of areas that are “cold” .

Something less than 5% of methods should be compiled for speed.

Training Runs

It is possible that we can combine multiple training runs and we can manually give some suggestions of what scenarios are important.

Obviously the more a scenario runs in the training data, the more important it will be, as far as the POGO optimization routine is concerned.

Multiple runs can be merged 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.

Let’s 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, the training data is used to do the following optimizations:

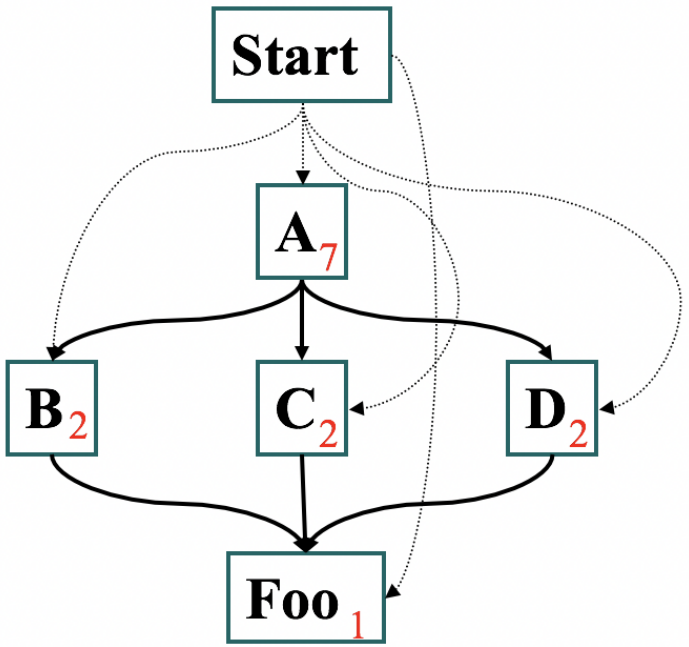
1. Full and partial inlining
2. Function layout
3. Speed and size decision
4. Basic block layout
5. Code separation
6. Virtual call speculation
7. Switch expansion
8. Data separation
9. Loop unrolling

Muh Gainz

The majority of the performance gains relate to the inlining decisions.

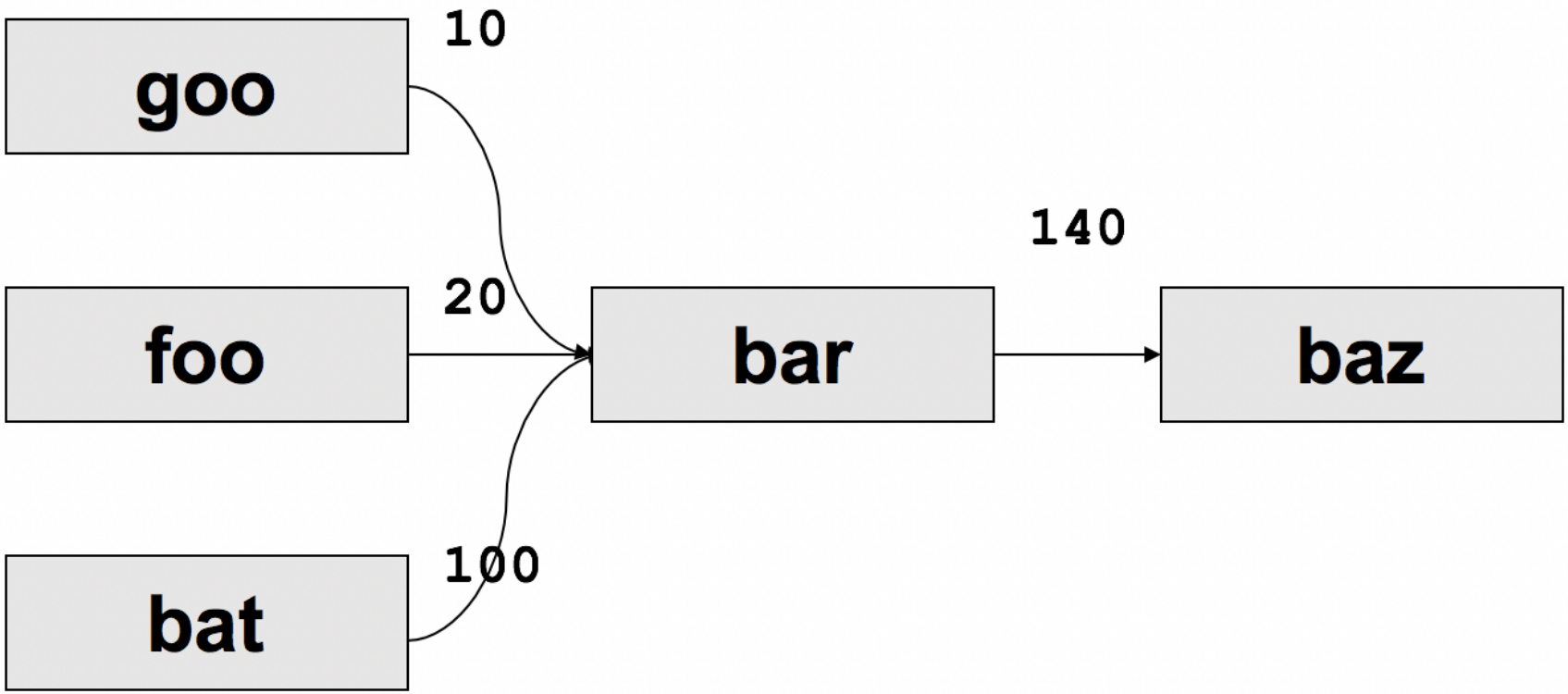
These decisions are based on the call graph path profiling.

The behaviour of function `foo` may be very different when calling it from `bar` than it is when calling it from function `baz`.



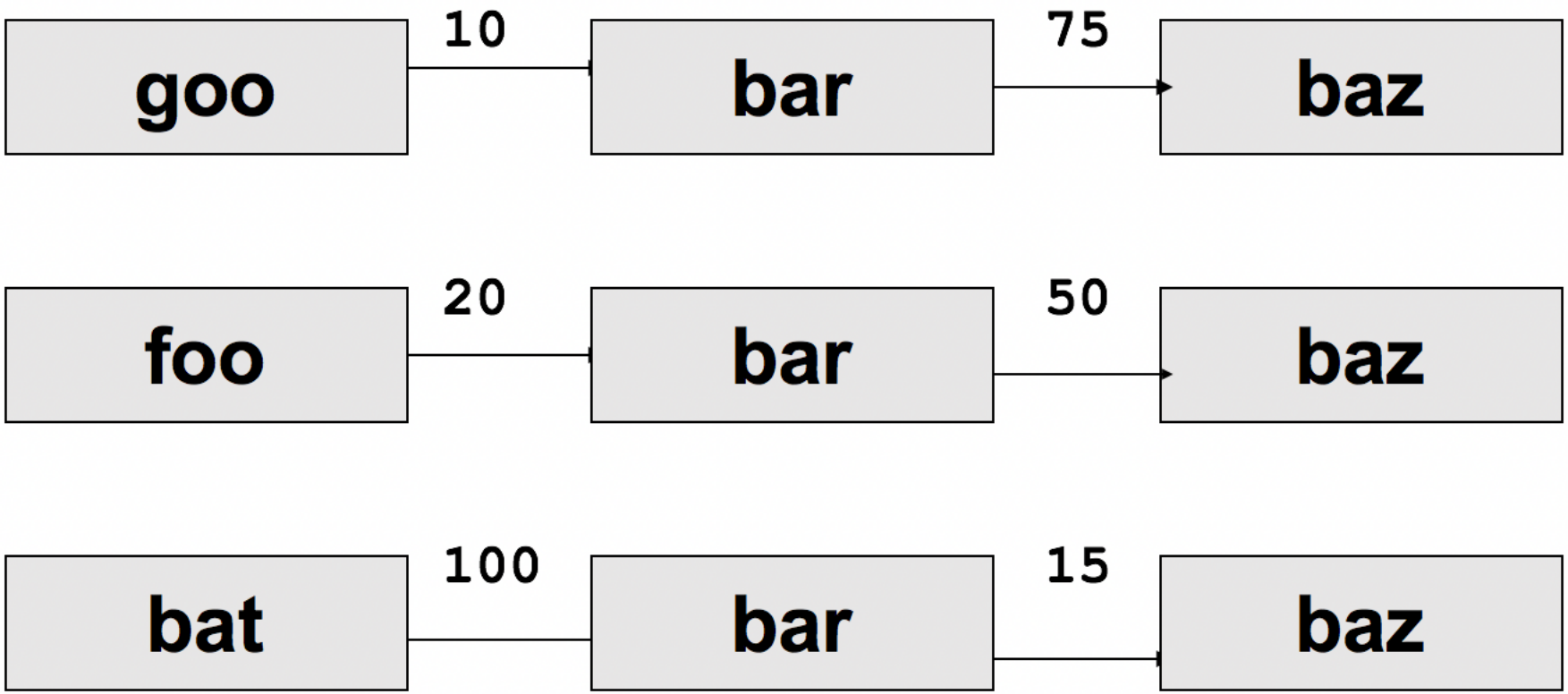
Another Call Graph

Consider another diagram showing the relationships between functions, in which the numbers on the edges represent the number of invocations:

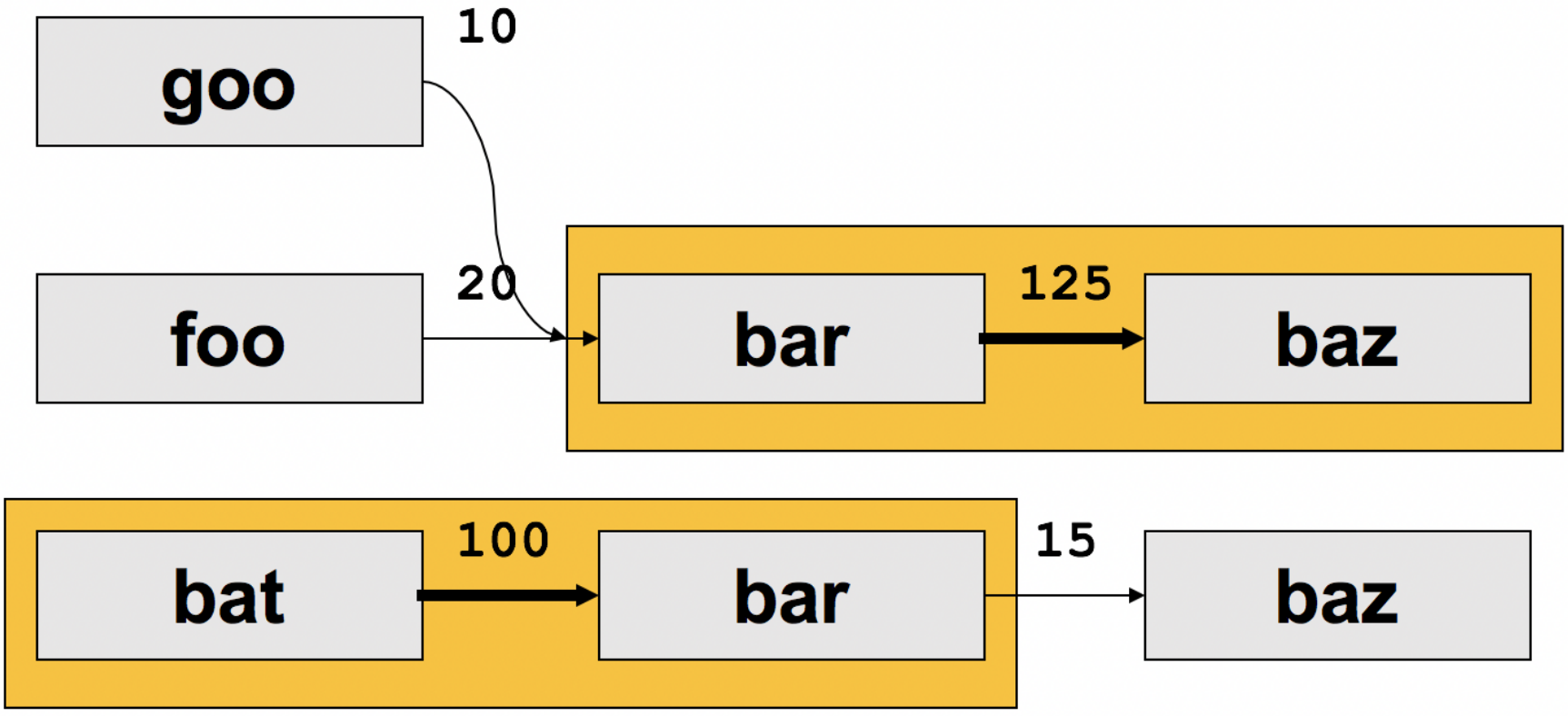


The POGO View of the World

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



The POGO View of the World



Page Locality

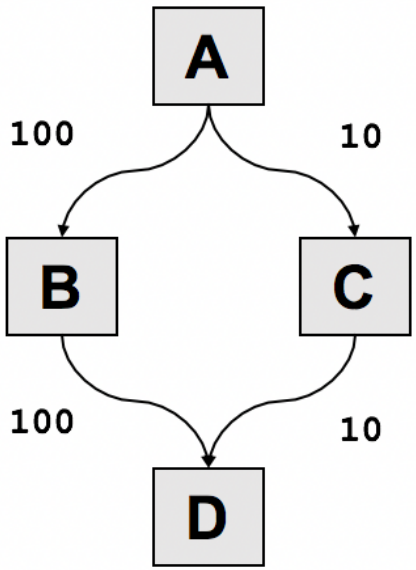
Packing the blocks is also done based on this call graph profiling.

The most common cases will be put next to each other, and, where possible, subsequent steps are put next to each other.

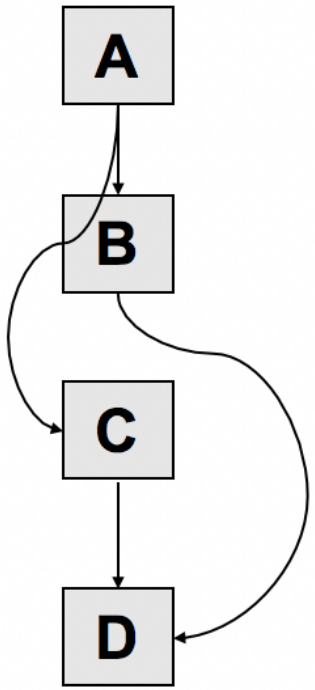
The more we can pack related code together, the fewer page faults we get by jumping to some other section, causing a cache miss...

If the function being called is in the same page as the call, it has achieved “page locality”.

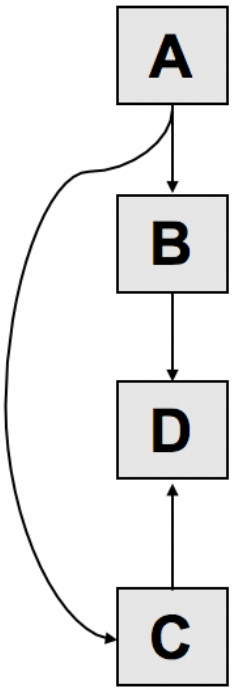
Block Layout



Default layout



Optimized layout



Dead Code?

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

I don’t think they actually mean truly dead code, the kind that is compile-time determined to be unreachable.

Instead they mean code that never gets invoked in any of the training runs.

But how well does it work?

Benchmark Results

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 the results in a synthetic benchmark translate to real-world application performance...