ECE459: Programming for Performance Winter 2017

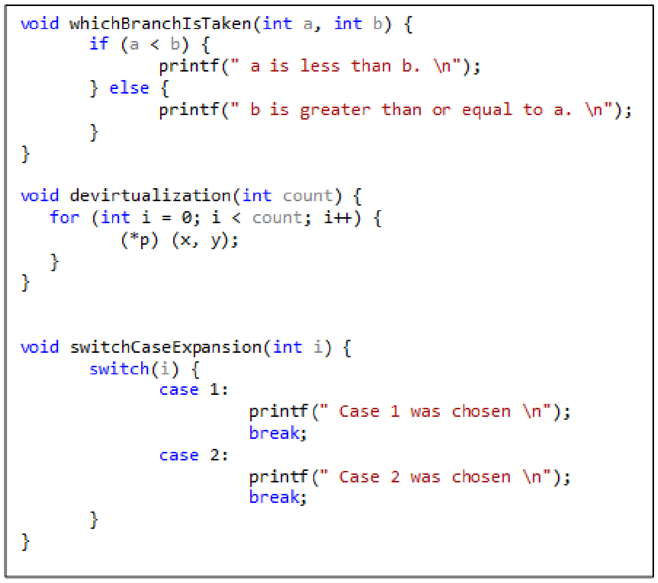
Lecture 28 — Profiler Guided Optimization (POGO)

Jeff Zarnett

Profiler Guided Optimization (POGO)

In 2015 we were fortunate enough to have a guest lecture from someone at Microsoft actually in the room to give the guest lecture on the subject of Profile Guided Optimization (or POGO). Try as I might, I was not able to convince him to fly in just for this lecture. For an alternative take on the subject, you could watch the full video at `https://www.youtube.com/watch?v=zEsdBcu4R00`. We'll use an excerpt in this lecture.

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. Consider three examples from [Ast13a]:



```
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;
        }
}
```

Just looking at this, which is more likely, `a > b` or `a <= b`? Assuming there's no other information in the system the compiler can believe that one is more likely than the other, or having no real information, use a fallback rule. This works, but what if we are wrong? Suppose the compiler decides it is likely that `a` is the larger value and it optimizes for that version. However, it is only the case 5% of the time, so most of the time the prediction is wrong. That's unpleasant. But the only way to know is to actually run the program.

There are similar questions raised for the other two examples. What is the "normal" value for some pointer `p`? If we do not know, the compiler cannot do devirtualization (replace this virtual call with a real one). Same thing with `i`: what is its typical value? If we know that, it is our prediction. Actually, in a switch-case block with many options, could we rank them in descending order of likelihood?

There exists a solution to this, and it is that we can give hints to the compiler, but that's a manual process. Automation is a good thing and this lecture is about that. These sorts of things already exist for Java! 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. That's neat! I don't know for certain but I suspect the .NET runtime will do the same for something like C#. But this is C(++) (Sparta) and we don't have that: 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.

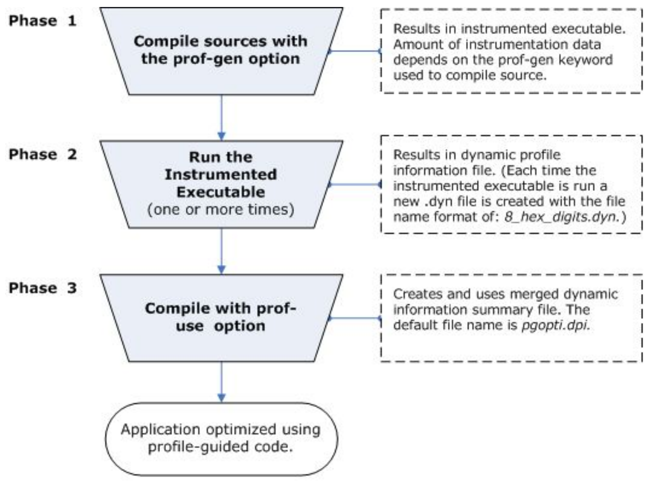
Solving this problem is the goal of POGO. It is taking the data from some actual runs of the program and using that to inform the predictions. This necessitates a multi-step compile: first compile the code, run it to collect data, then recompile the code using the data we collected. Let's expand on all three steps.

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, and value probes. A function entry probe, obviously, counts how many times a particular function is called. An edge probe is used to count the transitions (which tells us whether an if branch is taken or the else condition). Value probes are interesting; they are used to collect a histogram of values. Thus, we can have a small table that tells us the frequency of what is given in to a `switch` statement. When this phase is complete, there is an instrumented executable and an empty database file where the training data goes [Ast13a].

Step two is 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. Still, it is important to note that you are not trying to exercise every part of the program (this is not unit testing); instead it should be as close to real-world-usage as can be accomplished. In fact, trying to use ever 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...

Ste three is a recompile. This time, in addition to the source files, the training data is fed into the compiler for a second compile. and this data is applied to produce a better output executable than could be achieved by static analysis alone.

The Intel Developer Zone[1] explains the process in a handy infographic:


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.

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. According to [Ast13a], the recommended workflow is for one developer to perform these steps and check the training data into source control so that other developers can make use of it in their builds.

What does it mean for it to be better? We have already looked at an example about how to predict branches. Predicting it correctly will be faster than predicting it incorrectly, but this is not the only thing. The algorithms will aim for speed in the areas that are "hot" (performance critical and/or common scenarios). The algorithms will alternatively aim to minimize the size of code of areas that are "cold" (not heavily used). It is recommended in [Ast13a] that less than 5% of methods should be compiled for speed.

[1]Source: `https://software.intel.com/en-us/node/522721`

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). (We'll come back to the video again in a little bit to see more of it at the end.)

Behind the Scenes

In the optimize phase, the training data is used to do the following optimizations [Ast13b]:

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

For the most part we should be familiar with the techniques that are listed as being other compiler optimizations we have previously discussed. The new ones are (3) speed and size decision, which we have just covered; and items (4) and (5) which relate to how to pack the generated code in the binary.

This table, condensed from [Ast13b] summarizes the gains to be made. 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%

There are more details in the source as to how many functions are used in a typical run and how many things were inlined and so on. But we get enough of an idea from the last row of how much we are speeding up the program, plus some information about why. We can speculate about how well the results in a synthetic benchmark translate to real-world application performance, but at least from this view it does seem to be a net gain.

References

[Ast13a] Ankit Asthana. Building faster native applications, 2013. Online; accessed 8-January-2016. URL: https://blogs.msdn.microsoft.com/vcblog/2013/04/04/build-faster-and-high-performing-native-applications-using-pgo/.

[Ast13b] Ankit Asthana. Profile guided optimization, 2013. Online; accessed 8-January-2016. URL: http://nwcpp.org/talks/2013/ProfileGuidedOptimizationMarch21st.pptx.