Static Profiling: Why should you try it?

Women in Compilers and Tools Meetup Series
30 June 2022

Angélica Moreira
WHO AM I?

- From **Belo Horizonte, MG - Brazil***

- Pursuing a PhD in Computer Science at **Federal University of Minas Gerais (UFMG)** in Brazil
  - Advised by dr. Fernando Magno Quintão Pereira (from UFMG)
  - Co-advised by dr. Guilherme Ottoni (from Meta)

- Working in compilers research area as a Graduate Student Researcher for almost 4 years (mostly **LLVM**)
  - Projects highlight:
    - Dead Code Elimination
    - Basic Block Reordering
    - Static Branch Predictor
    - MLIR-based Compiler for the MSCCL** project

* Yep, the city where Brazil was humiliated by Germany in the world cup, 7x1 :'(
** MSCCL stands for Microsoft Collective Communication Library (MSCCL) is a platform to execute custom collective communication algorithms for multiple accelerators supported by Microsoft Azure. [https://github.com/microsoft/msccl](https://github.com/microsoft/msccl)
Basic Concepts

Prediction vs. Probability vs. Frequency
Prediction vs. Probability vs. Frequency

• For instance, in this code:

```plaintext
b1: if (condition)
b2:    statement;
b3:    else statement;
```

Prediction vs. Probability vs. Frequency

• For instance, in this code:

```plaintext
b1: if (condition)
b2: statement;
b3: else statement;
```

- A branch probability:
  - "branch b1->b2 81% to be taken, while b1->b3 has 19% of being taken."

PREDICTION VS. PROBABILITY VS. FREQUENCY

• For instance, in this code:

```c
b1: if (condition)
    b2: statement;
    b3: else statement;
```

– A branch frequency:
  • "b1 executes 80 times, in 65 of which it branches to b2, and in 15 it branches to b3."

SOFTWARE-BASED BRANCH PREDICTION
Software-Based Branch Prediction

1. Entry
2. `scanf(&x)`
3. `if (x <= 100)`
   - `printf("Cat person."))`
   - `printf("Dog person."))`
Software-Based Branch Prediction

Which way are we more likely to go?

```c
entry
scanf(&x)
if (x <= 100)
    printf("Cat person.")
else
    printf("Dog person.")
```
Software-Based Branch Prediction

entry

scanf(&x)

if (x <= 100)

printf("Cat person.")

20%?

80%?

printf("Dog person.")
How it's usually done...

```c
entry

if (x <= 100)
    printf("Cat person.");
else
    printf("Dog person.");
```
Dynamic Profiling

```c
scanf(&x)
if (x <= 100)
  printf("Dog person.")
else
  printf("Cat person")
```

Inputs

- 10
- 50
- 100
- 200

```
entry

scanf(&x)
if (x <= 100)
```
Dynamic Profiling

```c
scanf(&x)
if (x <= 100)
  printf("Dog person.")
else
  printf("Cat person.")
```
What are the common techniques?

printf("Dog person.")

printf("Cat person.")
Dynamic Profiling

• Sampling-based approaches:
  – Runs the program
  – **Samples** instructions executed
  – Records sampled branch executions
Sampling-based approaches:

- Runs the program
- **Samples** instructions executed
- Records sampled branch executions

### Dynamic Profiling

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Samples</th>
<th>Command</th>
<th>Shared Object</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.07%</td>
<td>242</td>
<td>Puzzle</td>
<td>Puzzle</td>
<td>[.] 0x00000000000005cf</td>
</tr>
<tr>
<td>+ 18.51%</td>
<td>0x4005b6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ 4.96%</td>
<td>0x4005ef</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- 7.24%</td>
<td>73</td>
<td>Puzzle</td>
<td>Puzzle</td>
<td>[.] 0x00000000000005c8</td>
</tr>
<tr>
<td>+ 6.65%</td>
<td>0x4005b6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.59%</td>
<td>0x4005ef</td>
<td>(predicted:83.3%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- 7.13%</td>
<td>72</td>
<td>Puzzle</td>
<td>Puzzle</td>
<td>[.] 0x00000000000005ef</td>
</tr>
<tr>
<td>+ 5.62%</td>
<td>0x4005b6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ 1.40%</td>
<td>0x4005ef</td>
<td>(predicted:92.9%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- 6.58%</td>
<td>66</td>
<td>Puzzle</td>
<td>Puzzle</td>
<td>[.] 0x00000000000005bb</td>
</tr>
<tr>
<td>+ 5.17%</td>
<td>0x4005b6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ 1.31%</td>
<td>0x4005ef</td>
<td>(predicted:92.3%)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Profiling

- Instrumentation based approaches:
  - Instrument every branch in the program with a counter
  - Increment counter whenever branch executes
  - Much higher CPU and memory overhead
  - Intrusive
Speaking of intrusiveness...
Motivation

But why static branch prediction?
Motivation

But why static branch prediction?

Collecting profile data sometimes is really difficult …
Motivation

Collecting profile data sometimes is really difficult …

Impossible?
Collecting profile data

Impossible?

Yes, think about mobile apps.
Static Profiling

• Look only at the code
Static Profiling

• Look only at the **code** (no execution!)
Static Profiling

- Look only at the code (no execution!)
Static Profiling

• Look only at the **code** (no execution!)
• Try to infer branch likelihood
Static Branch Prediction in the Wild

- Heuristic-based
- Machine Learning-based

Suggestion of paper to read:
Heuristic-Based Static Profiling

entry

if (p == nullptr)

printf("Error!")
exit()

printf("Wash your hands")
Heuristic-Based Static Profiling

if (p == nullptr)

printf("Error!")
exit()

printf("Wash your hands")

Error path is probably more likely to not execute often? (heuristics)
Heuristic-Based Static Profiling

if (p == nullptr)
    printf("Error!\n")
    exit();

So we might guess these!

printf("Wash your hands")
LLVM’s Static Branch Prediction

• Based solely on heuristics
  – Paper from Ball & Larus\(^1\)
• Implemented in the BranchProbabilityInfo analysis pass

Heuristic-Based Static Profiling

• Very ad-hoc solution
• Relies on compiler developers themselves coming up with clever heuristics
Machine Learning-Based Static Profiling

- Collect corpus of programs, and static features that describe them
- Record their branch execution behaviour
- Train ML model based on features + branch data
- Create static profiles for unknown programs based on trained knowledge!
Some work in this area

To read click **here**

Evidence-based Static Branch Prediction using Machine Learning

Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn

Department of Computer Science

Campus Box 430

University of Colorado

Boulder, CO 80309-4430 USA

September 19, 1996

Abstract

Correctly predicting the direction that branches will take is increasingly important in today’s wide-issue computer architectures. The name program-based branch prediction is given to static branch prediction techniques that base their prediction on a program’s structure. In this paper, we investigate a new approach to program-based branch prediction that uses a body of existing programs to predict the branch behavior in a new program. We call this approach to program-based branch prediction evidence-based static prediction, or ESP. The main idea of ESP is that the behavior of a corpus of programs can be used to infer the behavior of new programs. In this paper, we use neural networks and decision trees to map static features associated with each branch to a prediction that the branch will be taken. ESP shows significant advantages over other prediction mechanisms. Specifically, it is a program-based technique, it is effective across a range of programming languages and programming styles, and it does not rely on the use of expert-defined heuristics.

In this paper, we describe the application of ESP to the problem of static branch prediction and compare our results to existing program-based branch predictors. We also investigate the applicability of ESP across computer architectures, programming languages, compilers, and run-time systems. We provide results showing how sensitive ESP is to the number and type of static features and programs included in the ESP training set, and compare the efficacy of static branch prediction for various compilers. Averaging over a body of 43 C and Fortran programs, ESP branch prediction results in a miss rate of 20%, as compared with the 23% miss rate obtained using the best existing program-based heuristics.

To read click **here**

Profile Guided Optimization without Profiles: A Machine Learning Approach

Nadav Rotem
Meta, Inc.

Chris Cummins
Meta AI

January 5, 2022

Abstract

Profile guided optimization is an effective technique for improving the optimization ability of compilers based on dynamic behavior, but collecting profile data is expensive, cumbersome, and requires regular updating to remain fresh.

We present a novel statistical approach to inferring branch probabilities that improves the performance of programs that are compiled without profile guided optimizations. We perform offline training using information that is collected from a large corpus of binaries that have

To read click **here**

VESPA: Static Profiling for Binary Optimization

ANGÉLICA APARECIDA MOREIRA, UFMG, Brazil

GUILHERME OTTONI, Facebook, Inc., USA

FERNANDO MAGNO QUINTÃO PEREIRA, UFMG, Brazil

Over the past few years, there has been a surge in the popularity of binary optimizers such as BOLT, Propeller, Janus and HALO. These tools use dynamic profiling information to make optimization decisions. Although effective, gathering runtime data presents developers with inconveniences such as unrepresentative inputs, the need to accommodate software modifications, and longer build times. In this paper, we revisit the static profiling technique proposed by Calder et al. in the late 90’s, and investigate its application to drive binary optimizations, in the context of the BOLT binary optimizer, as a replacement for dynamic profiling. A few core modifications to Calder et al.’s original proposal, consisting of new program features and a new regression model, are sufficient to enable some of the gains obtained through runtime profiling. An evaluation of BOLT powered by our static profiler on four large benchmarks (clang, GCC, MySQL and PostgreSQL) yields binaries that are 5.47% faster than the executables produced by clang -O3.
Optimize binaries using static profile inferred by a machine learning model

**Vintage ESP Amended (VESPA)**
- Extension of Calder's work: Evidence-Based Static Branch Prediction (ESP) \(^1\)

---

Calder vs VESPA

- Collect corpus of programs, and static features that describe them
- Record their branch execution behaviour
- Train ML model based on features + branch data
- Create static profiles for unknown programs based on trained knowledge!
Infrastructurereview - Static BOLT usage!

1. Run binary with perf, collect counters
2. Convert the perf report to bolt input format
3. Optimize binary

4. Extract static features from binary
5. Feed features to ML model, extract probabilities
6. Optimize binary with inferred probabilities
ML Pipeline Overview

Statically extract 56 features for each branch in the program

Data Preparation
- Cleaning
- Scaling numeric features
- Encoding categorical features

Train predictive models / Perform predictions using the model

Features
- Categorical
- Numerical
Statically extract **56** features for each branch in the program

Data Preparation

- Cleaning
- Scaling numeric features
- Encoding categorical features

**How?**

Train predictive models / **Perform predictions** using the model

ML Model

Features

- Categorical
- Numerical
Feature Miner

- Implemented the FeatureMiner pass in BOLT
- Runs after binary disassembling/CFG construction
- Analyzes the CFG to collect static features proposed by Calder et al.\[1\], as well as others devised by us for each branch.

Statically extract 56 features for each branch in the program.

Data Preparation:
- Cleaning
- Scaling numeric features
- Encoding categorical features

Train predictive models / Perform predictions using the model

ML Model

Features:
- Categorical
- Numerical

Which features?
AN EXAMPLE

isLoopHeader

entry:
  br label %for.cond

for.cond:
  %sum.0 = phi i32 [ 0, %entry ], [ %add, %for.inc ]
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i32 %i.0, 100
  br i1 %cmp, label %for.body, label %for.end

T  F

for.body:
  %add = add nsw i32 %sum.0, %i.0
  br label %for.inc

for.inc:
  %inc = add nsw i32 %i.0, 1
  br label %for.cond

for.end:
  %cmp1 = icmp sgt i32 %sum.0, 101
  br i1 %cmp1, label %if.then, label %if.end

T  F

if.then:
  %dec = add nsw i32 %sum.0, -1
  br label %if.end

if.end:
  %sum.1 = phi i32 [ %dec, %if.then ], [ %sum.0, %for.end ]
  ret i32 %sum.1
This conditional branch is the header of this loop. Thus, \texttt{isLoopHeader} = \texttt{True}
This conditional branch is not the header of any loop. Thus, isLoopHeader = False
AN EXAMPLE

isLoopHeader

entry:
    br label %for.cond

for.cond:
    %sum.0 = phi i32 [ 0, %entry ], [ %add, %for.inc ]
    %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
    %for.cond = %add %i.0 1
    br label %for.inc

for.body:
    %add = add nsw i32
    br label %for.inc

for.inc:
    %inc = add nsw i32 %i.0, 1
    br label %for.cond

if.then:
    %dec = add nsw i32 %sum.0, -1
    br label %if.end

if.end:
    %sum.1 = phi i32 [ %dec, %if.then ], [ %sum.0, %for.end ]
    ret i32 %sum.1

And 55 more others!
ML Pipeline Overview

Statically extract **56** features for each branch in the program

Data Preparation

- Cleaning
- Scaling numeric features
- Encoding categorical features

Train predictive models / **Perform predictions** using the model

ML Model

Features

- Categorical
- Numerical

What do these predictions look like?
Our models provide estimates for the probabilities of branches being taken/not taken

```
scanf(&x)
if (x > 100)

printf("Do not get vaccinated for COVID")
printf("Get vaccinated for COVID")

Δ%?
96%?
```
However, BOLT assigns \textit{frequencies} to branches, not probabilities.

```
scanf(&x)
if (x > 100)
```

400?

\texttt{printf("Do not get vaccinated for COVID")}

9600?

\texttt{printf("Get vaccinated for COVID")}
Our model outputs **probabilities**, yet we need **frequencies**. How to convert between the two?

However, BOLT assigns **frequencies** to branches, not probabilities.

```c
entry
scanf(&x)
if (x > 100)
printf("Do not get vaccinated for COVID")

printf("Get vaccinated for COVID")
```

400?

?
Getting Frequencies out of Probabilities

• Technique proposed by Wu & Larus\(^1\)
  
  - Calculate basic block and Control Flow Graph (CFG) edge frequencies intra-procedurally (within functions)
  
  - Propagate probabilities starting from the entry block, according to these equations:

\[
\begin{align*}
  bfreq(b_i) &= 1 & \text{(if entry block)} \\
  bfreq(b_i) &= \sum_{b_p \in \text{pred}(b_i)} freq(b_p \rightarrow b_i) & \text{(otherwise)} \\
  freq(b_i \rightarrow b_j) &= bfreq(b_i) \times \text{prob}(b_i \rightarrow b_j)
\end{align*}
\]

Getting Frequencies out of Probabilities

• Technique proposed by Wu & Larus\(^1\)
  • Calculate basic block and Control Flow Graph (CFG) edge frequencies intra-procedurally (within functions)
  • Propagate probabilities starting from the entry block, according to these equations:

\[
bfreq(b_i) = \sum_{b_p \in \text{pred}(b_i)} bfreq(b_p)
\]

\[
 freq(b_i \rightarrow b_j) = bfreq(b_i)
\]

This technique estimates execution frequency (not absolute counts) with static program analysis!!!!

Caveats to Static Inference

• Indirect branches cannot have their targets inferred statically
  – Adds imprecision to **intra-procedural** inference!

• Similarly, indirect procedure calls (virtual method invocations, function pointer calls, etc.) also cannot be resolved statically
  – Adds imprecision to **inter-procedural** inference!
Experiments
"BOLT using static profile data can still provide some of the gains as using real profile data."
Setup

- Trained models on a dataset with 243 programs:
  - Corpus of 2,093,873 two-way conditional branched but only 513,316 associated with branch predictions
- All binaries used in training and as baseline were compiled using Clang 12 with -O3
- 80% of the branches used for training and 20% for test and validation
RQ2: What are the performance gains of our approach when compared to the baseline compiler at its highest optimization level, and to a binary optimized with dynamic profiling information?
Experiments

Our baseline is the original binary, optimized with -O3
BOLT+Full Perf indicates a binary optimized using BOLT’s traditional usage: with a dynamic execution profile collected with `perf`
BOLT+Limited Perf indicates a binary optimized using BOLT’s traditional usage: with a stripped dynamic execution profile collected with `perf`.
BOLT+Calder indicates a binary built using BOLT with a static profile inferred using the heuristics due to Calder.
Experiments

BOLT+Wu Larus indicates a binary built using BOLT with a static profile inferred using the heuristics due to Wu et al.
BOLT+Unbiased indicates a binary built using BOLT and assuming every two-way branch has a 50/50 probability split.
Experiments

BOLT+No Profile is a binary built with BOLT fed with no profile information at all.
These two indicate binaries built with trivial profiles that assume every two-way branch takes either direction with 100% probability.
And finally, BOLT+VESPA is a binary built using BOLT with our technique for static profile inference.
Experiments

Far from the dynamic results, but good performance improvements on top of -O3

VESPA features set beats ESP feature set!

Sanity check
Experiments

MySQL + Sysbench OLTP_POINT_SELECT

BOLT+Full Perf
BOLT+Limited Perf
BOLT+VESPA
BOLT+Calder
BOLT+Wu Larus
BOLT+Unbiased
BOLT+No Profile
BOLT+Never
BOLT+Always

Throughput Improvement

-20.00%  0.00%  20.00%  40.00%

49.72%  28.81%  8.63%  6.08%  4.42%  3.76%  0.90%  -6.87%  -10.22%

Same behaviour occurs, where VESPA provides significant benefits on top of baseline and BOLT with trivial/heuristics-based profiles.
VESPA is still far away from a dynamic profiler but it does deliver considerable speedups on top of the baseline.

ESP feature set is defeated by VESPA feature set!

Sanity check
What did you learn today?

- Ways to do software branch prediction;
- Types of static branch predictors;
- How we could do static profiling;
- How we can use ML as a tool to help in the task of generating static profile;
- How we can get branch and block frequencies when you only have probabilities.