LLVM Based Profile Guided Optimization For Mobile Devices

LLVM COMPILERS TEAM

Authors: Yakushkin Sergey
Kosov Pavel

Agenda

- Description of profile-guided optimization (PGO)
- PGO implementation details in different compilers
- PGO related works and researches
- Ways for PGO improvement

Huawei

- Big amount of mobile devices (OS Android)
- Strong competition demands develop best solution for improving user experience
- PGO is a one of possible ways for such improvement

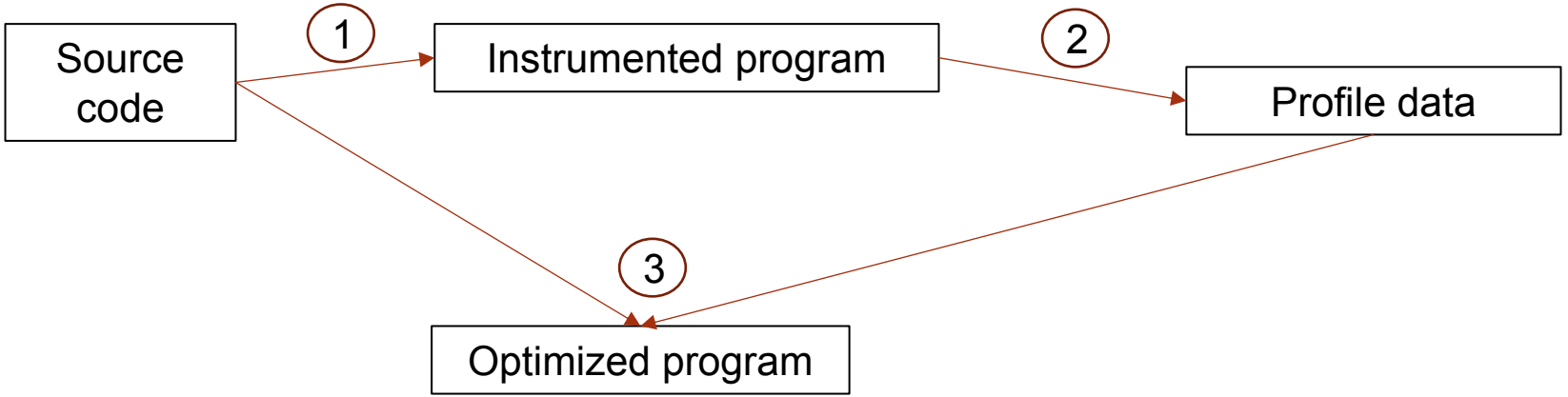
Description of PGO

Also known as Feedback Driven Optimization
It is not an optimization, it is an approach.

Compiler use data about real cases of program using

Ways to get profile data:
- Instrumentation
- Sampling

Instrumentation (in LLVM)



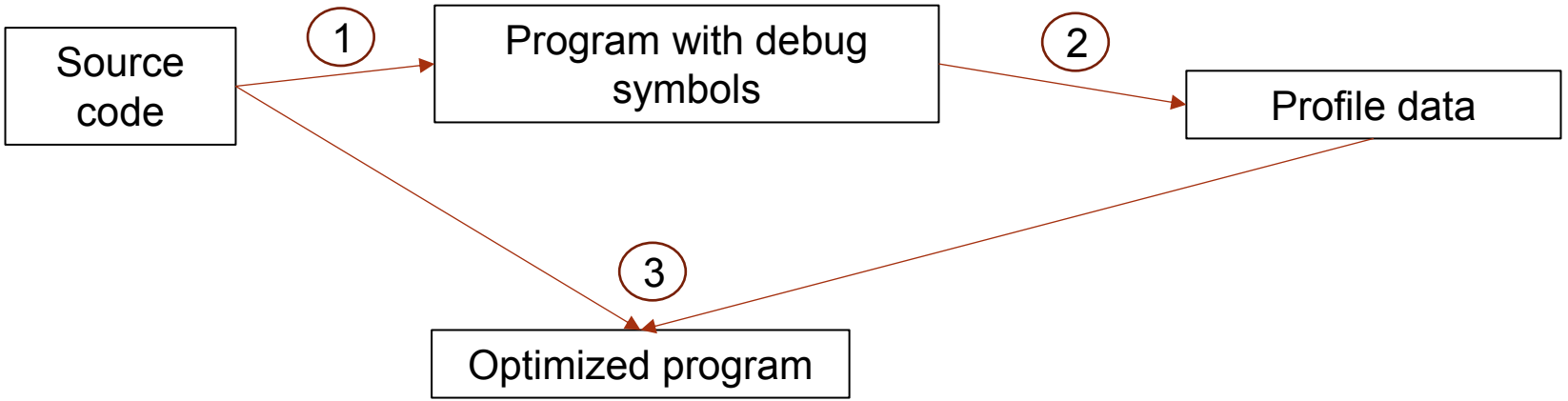
1 – Compilation with insertion of counters (fprofile-generate)
2 – Program execution. Creation of file with profile data (profdata file)
3 – Compilation with profdata file (fprofile-use)

Instrumentation (in LLVM)

Code insertion:

1) Counters:
   1.1) Add to entry point
   1.2) Calculation of minimal spanning tree (MST)
   1.3) Add counters to edges which are not in MST
2) Probes for indirect call addresses
3) Probes for functions parameter (only for memcpy / memmove / memset)

Sampling (in LLVM)



1 – Compilation with debug information
2 – Program execution with profiler (e.g perf for Linux), creation of profdata file
3 – Compilation with profdata file (fprofile-use)

Sampling

- Profiler stops the program with user-provided frequency (default 1000 Hz for perf)
- Collects info using hardware counters
- Map collected information to current instruction of profiled program

Profiler can collect a lot of different information: CPU cycles, instructions count, cache misses, context switching etc.

Instrumentation vs Sampling

Advantages of instrumentation:

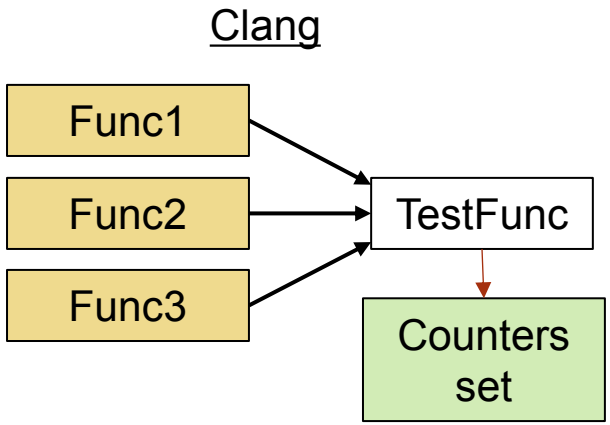
- Accuracy, determinism
- Ability to determine indirect call addresses and functions parameters values

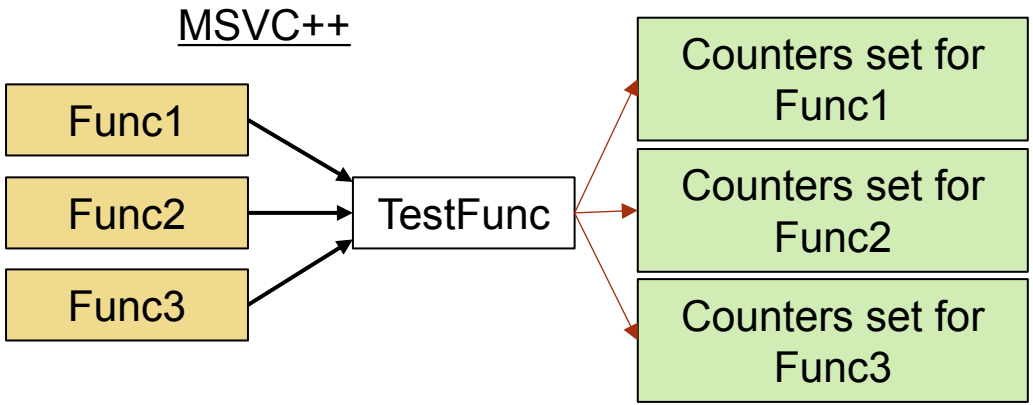
Advantages of sampling:

- Low overhead
- Ability to collect information from hardware counters

Comparison of PGO implementation with MSVC++

MSVC++ creates different counters set for each caller





Comparison of PGO implementation with IntelC++ Compiler (ICC)

ICC with instrumentation allows to get following data:

- CPU cycle inside functions and loops
- Iterations count for loops (maximum, minimum, average)

Also ICC allow to use fine-grained setup of instrumentation level

Projects and Researches

Projects:

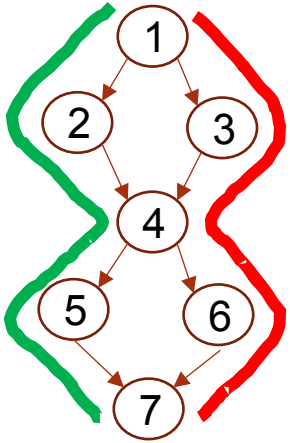
- BOLT (Facebook). Post-link optimizer application's code layout. Use Call-Chain Clustering algorithm (improved Pettis-Hansen).
- Propeller (Google). Enhanced BOLT for using in distributed systems and with lower memory foot-print

Researches:

- GOA (grant DARPA). Optimizer for power-efficiency
- CodeStitcher (grant Huawei, IBM). An inter-procedural basic block code layout optimizer

Ways for PGO improvement in LLVM

- Reduce number of counters by improvement control-dependence information analysis
- Enhance heuristics of Call-Chain Clustering for reducing number of instructions cache misses
- Add profiling of hot paths (bbvectors) in functions:



Path 1-2-4-5-7 was taken N times
Path 1-3-4-6-7 was taken M times
...

Ways for PGO improvement in LLVM

- Collect info for each caller about bbvectors that were taken

Optimizations which will benefit from bbvectors info

- Partial inlining
- Functions specialization
- Link time code layout
- Code size reducing (by applying different sets of optimization for cold and hot functions)

Ways for PGO improvement in LLVM

- Reduce number of data cache misses
- Setup level of instrumentation
- Profile CPU cycles
- Use profile data in next optimizations: vectorization, code size reducing, loop unrolling, loop peeling etc.

List of possible improvements on LLVM official site:
https://llvm.org/OpenProjects.html

Thank you

Contacts:
Yakushkin.Sergey@Huawei.com
Kosov.Pavel@Huawei.com