Lightweight Instrumentation using Debug Information

Ellis Hoag & Kyungwoo Lee

Agenda

1. Background
2. Instrumentation overview
3. Lightweight Instrumentation
4. Extensions
    a. Function Coverage
    b. Block Coverage

Binary Size

- Includes:
  - `.text` section
  - `.data` section
  - No debug info
- Mobile apps use `-Os`/`-Oz`
  - Less network pressure
  - Less storage pressure
  - Better performance

Efficient Profile-Guided Size Optimization for Native Mobile Applications

Kyungwoo Lee
Meta
Menlo Park, CA, USA
kyulee@fb.com

Ellis Hoag
Meta
Menlo Park, CA, USA
ellishoag@fb.com

Nikolai Tillmann
Meta
Menlo Park, CA, USA
nikolait@fb.com

Abstract

Positive user experience of mobile apps demands they not only launch fast and run fluidly, but are also small in order to reduce network bandwidth from regular updates. Conventional optimizations often trade off size regressions for performance wins, making them impractical in the mobile space. Indeed, *profile-guided optimization* (PGO) is successful in server workloads, but is not effective at reducing size and page faults for mobile apps. Also, profiles must be collected from instrumenting builds that are up to 2*X* larger, so they cannot run normally on real mobile devices.

In this paper, we first introduce *Machine IR Profile* (MIP), a lightweight instrumentation that runs at the machine IR level. Unlike the existing LLVM IR instrumentation coun-

CCS Concepts: • **Software and its engineering** → **Compilers**; *Runtime environments*; • **Computer systems organization** → Embedded software; • **General and reference** → Performance.

Keywords: profile-guided optimizations, size optimizations, machine outlining, mobile applications, iOS

ACM Reference Format:

Kyungwoo Lee, Ellis Hoag, and Nikolai Tillmann. 2022. Efficient Profile-Guided Size Optimization for Native Mobile Applications. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22), April 02–03, 2022, Seoul, South Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3497776.3517764

IRPGO Overview

```
clang -fprofile-generate main.cpp
```

- Injects probes
  - Edge counts
  - Indirect function calls (value profiling)

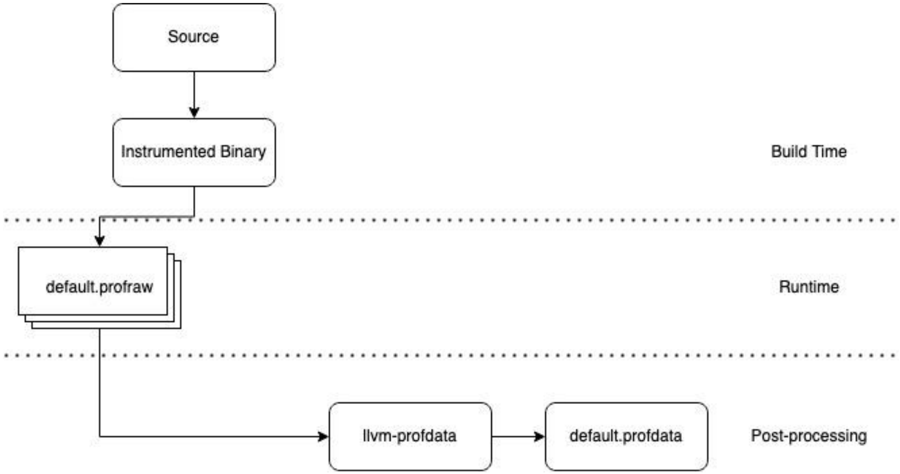
IRPGO Overview

Runtime

- Dump “raw profiles”

IRPGO Overview

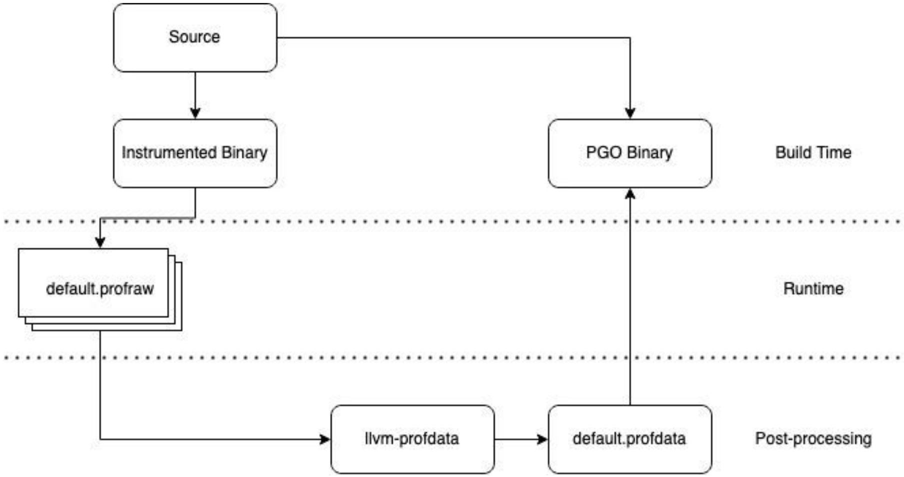
Post-processing




```
llvm-profdata merge default_*.profraw -o default.profdata
```

IRPGO Overview

Optimization



IRPGO Instrumented Binary

Test with the `clang` binary

```
$ cmake -GNinja -DLLVM_ENABLE_PROJECTS="clang"         \
  -DCMAKE_C_COMPILER="..." -DCMAKE_CXX_COMPILER="..." \
  -DCMAKE_C_FLAGS="..." -DCMAKE_CXX_FLAGS="..."        \
  -DCMAKE_BUILD_TYPE=RelWithDebInfo ../llvm/
$ ninja clang
```

IRPGO Instrumented Binary

47% size overhead!

Section	Base	IRPGO
`.text`	65.8 Mi	93.0 Mi
`__llvm_prf_cnts`		14.5 Mi
`__llvm_prf_names`		7.68 Mi
`__llvm_prf_data`		4.50 Mi
Total Binary Size	119 Mi	175 Mi
Overhead		**47%**

```
CMAKE_CXX_FLAGS="-fprofile-generate -mllvm -disable-vp"
```

IRPGO `__llvm_prf_cnts`

- 26% of overhead
- 64 bit counters

```
@__profc__Z3foov = private global [5 x i64] zeroinitializer, section "__llvm_prf_cnts", comdat, align 8
```

IRPGO `__llvm_prf_names`

- 14% of overhead
- Function Names
  - Compressed or uncompressed
- Unused at runtime!

```
@__llvm_prf_nm = private constant [17 x i8] c"...", section "__llvm_prf_names", align 1
```

IRPGO `__llvm_prf_data`

Correlates raw profile data to their functions

- 8% of overhead
- (mostly) Unused at runtime!

```
// Simplified from InstrProfData.inc
struct InstrProfData {
 int NameRef;
 int FuncHash;
 int *RelativeCounterPtr;
 int *FunctionPointer;
 int *Values;
 int NumCounters;
 int NumValueSites[2];
};
```

IRPGO __llvm_prf_data

Values pointer changes at runtime

Unsupported fields

```
// Simplified from InstrProfData.inc
struct InstrProfData {
 int NameRef;
 int FuncHash;
 int *RelativeCounterPtr;
 int *FunctionPointer;
 int *Values;
 int NumCounters;
 int NumValueSites[2];
};
```

IRPGO __llvm_prf_data

Derives function name using __llvm_prf_names

```
// Simplified from InstrProfData.inc
struct InstrProfData {
 int NameRef;
 int FuncHash;
 int *RelativeCounterPtr;
 int *FunctionPointer;
 int *Values;
 int NumCounters;
 int NumValueSites[2];
};
```

IRPGO __llvm_prf_data

Function CFG hash

```
// Simplified from InstrProfData.inc
struct InstrProfData {
 int NameRef;
 int FuncHash;
 int *RelativeCounterPtr;
 int *FunctionPointer;
 int *Values;
 int NumCounters;
 int NumValueSites[2];
};
```

IRPGO `__llvm_prf_data`

Counter pointer

Function pointer

```
// Simplified from InstrProfData.inc
struct InstrProfData {
 int NameRef;
 int FuncHash;
 int *RelativeCounterPtr;
 int *FunctionPointer;
 int *Values;
 int NumCounters;
 int NumValueSites[2];
};
```

IRPGO `__llvm_prf_data`

Number of counters

```
// Simplified from InstrProfData.inc
struct InstrProfData {
 int NameRef;
 int FuncHash;
 int *RelativeCounterPtr;
 int *FunctionPointer;
 int *Values;
 int NumCounters;
 int NumValueSites[2];
};
```

IRPGO `__llvm_prf_data`

- Marked as used
  - `llvm.compiler.used`
- Function pointer reference
  - Prevents dead stripping

```
// Simplified from InstrProfData.inc
struct InstrProfData {
 int NameRef;
 int FuncHash;
 int *RelativeCounterPtr;
 int *FunctionPointer;
 int *Values;
 int NumCounters;
 int NumValueSites[2];
};
```

Solutions?

Goal: Collect a **profile** using a **small binary**

- AutoFDO (sampling-based PGO)
  - ✅ Zero binary size overhead
  - ❌ Not precise
  - ❌ Hardware counters may be unavailable

Solutions?

Goal: Collect a **profile** using a **small binary**

- AutoFDO (sampling-based PGO)
  - ✅ Zero binary size overhead
  - ❌ Not precise
  - ❌ Hardware counters may be unavailable
- Extract `__llvm_prf_data` section to file
  - ✅ Smaller binary size overhead
  - ❌ Difficult to get right
    - Relative relocations
    - Comdat sections
  - Efficient Profile-Guided Size Optimization for Native Mobile Applications [1]

Debug Info Correlation

Use debug info of counters to populate `InstrProfData`

- Extract from debug info
  a. Counter Address
  b. Function Pointer

```
// Simplified from InstrProfData.inc
struct InstrProfData {
 int NameRef;
 int FuncHash;
 int *RelativeCounterPtr;
 int *FunctionPointer;
 int *Values;
 int NumCounters;
 int NumValueSites[2];
};
```

Debug Info Correlation

Additional constant metadata

1. Function name
2. Function hash
3. Number of counters

```
// Simplified from InstrProfData.inc
struct InstrProfData {
  int NameRef;
  int FuncHash;
  int *RelativeCounterPtr;
  int *FunctionPointer;
  int *Values;
  int NumCounters;
  int NumValueSites[2];
};
```

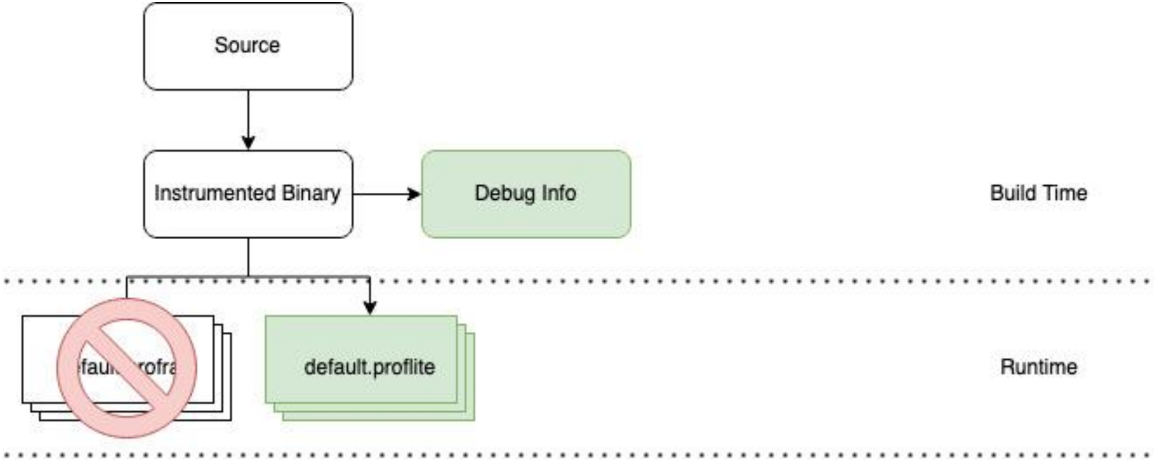
```
!1 = distinct !DIGlobalVariable(name: "__profc__Z3foov", ..., annotations: !11)
!11 = !{!12, !13, !14}
!12 = !{!"Function Name", !"_Z3foov"}
!13 = !{!"CFG Hash", i64 742261418966908927}
!14 = !{!"Num Counters", i32 1}
```

Debug Info Correlation

47% ➡ 36% size overhead

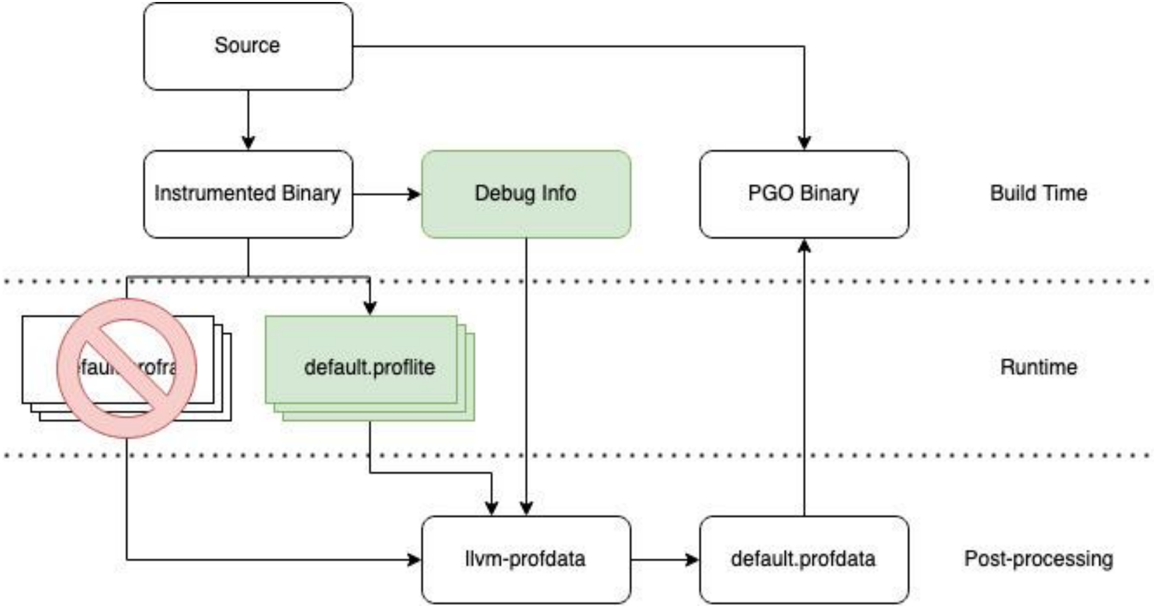
Section	Base	IRPGO	Lightweight
`.text`	65.8 Mi	93.0 Mi	90.3 Mi
`__llvm_prf_cnts`		14.5 Mi	14.5 Mi
`__llvm_prf_names`		7.68 Mi	
`__llvm_prf_data`		4.50 Mi	
Total Binary Size	119 Mi	175 Mi	162 Mi
Overhead		**47%**	**36%**

Debug Info Correlation




```
clang -g -fprofile-generate -mllvm -debug-info-correlate main.cpp
```

Debug Info Correlation




```
llvm-profdata merge --debug-info <dbg> default_*.proflite -o default.profdata
```

Function Entry Coverage

Goal: Identify **called** functions

- Test coverage
- Production coverage
    - Dead code detection

Function Entry Coverage

- Inject instructions at function entry
- Single byte global
  - Initialized to `0xff`
- 9 bytes on AArch64
  - 1 byte global
  - 8 bytes for two AArch64 instructions

```
adrp    x8, .coverage_byte
strb    wzr, [x8, .coverage_byte]

.coverage_byte:
        .byte   255
```

Lightweight Function Entry Coverage

47% ➡ 5% size overhead

Section	Base	IRPGO	Lightweight	Function Coverage
`.text`	65.8 Mi	93.0 Mi	90.3 Mi	69.5 Mi
`__llvm_prf_cnts`		14.5 Mi	14.5 Mi	163 Ki
`__llvm_prf_names`		7.68 Mi		
`__llvm_prf_data`		4.50 Mi		
Total Binary Size	119 Mi	175 Mi	162 Mi	125 Mi
Overhead		**47%**	**36%**	**5%**

Basic Block Coverage

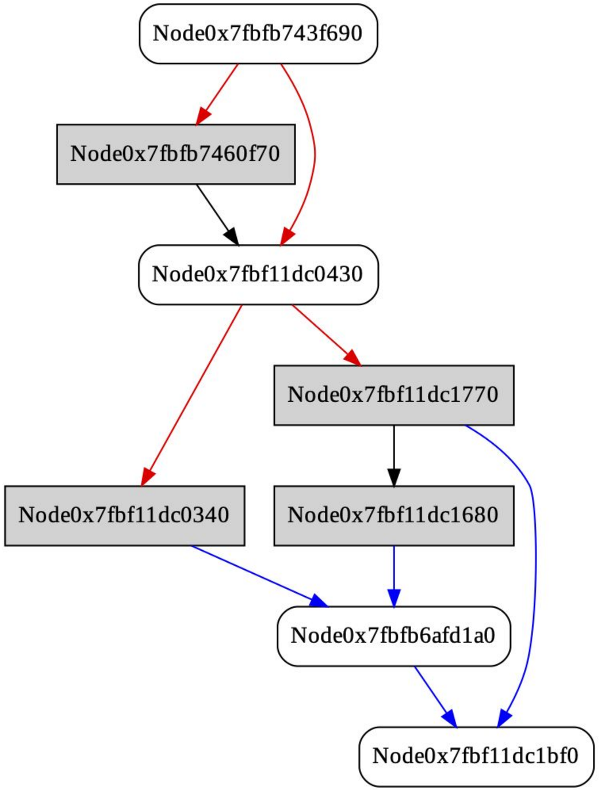
Goal: Identify **cold** blocks to **outline**

- Single byte global
- Less precise than edge counts
- Outlining opportunities

Basic Block Coverage

Goal: Identify **cold** blocks to **outline**

- Instrument every block?
  - Edge counts use Knuth's alg [2]
- ~60% blocks instrumented
- In review
  - https://reviews.llvm.org/D124490



Basic Block Coverage

47% ➡ 17% size overhead

Section	Base	IRPGO	Lightweight	Function Coverage	Block Coverage
`.text`	65.8 Mi	93.0 Mi	90.3 Mi	69.5 Mi	82.1 Mi
`__llvm_prf_cnts`		14.5 Mi	14.5 Mi	163 Ki	1.38 Mi
`__llvm_prf_names`		7.68 Mi			
`__llvm_prf_data`		4.50 Mi			
Total Binary Size	119 Mi	175 Mi	162 Mi	125 Mi	139 Mi
Overhead		**47%**	**36%**	**5%**	**17%**

Special Thanks

- Wenlei He
- Nikolai Tillmann
- Julian Mestre
- Sergey Pupyrev
- Greg Clayton

Sources and Links

[1] Kyungwoo Lee, Ellis Hoag, and Nikolai Tillmann. 2022. Efficient Profile-Guided Size Optimization for Native Mobile Applications. In Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction. https://doi.org/10.1145/3497776.3517764

[2] Donald E. Knuth, Francis R. Stevenson. Optimal measurement of points for program frequency counts. BIT Numerical Mathematics 1973, Volume 13, Issue 3, pp 313-322

- Lightweight Instrumentation
  - https://discourse.llvm.org/t/instrprofiling-lightweight-instrumentation/59113
  - https://reviews.llvm.org/D115693
  - https://reviews.llvm.org/D115915
- Function Entry Coverage
  - https://reviews.llvm.org/D116180
- Minimal Block Coverage
  - https://reviews.llvm.org/D124490