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 2X 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 count-

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:
IRPGO Overview

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

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Base</th>
<th>IRPGO</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>65.8 Mi</td>
<td>93.0 Mi</td>
</tr>
<tr>
<td>__llvm_prf_cnts</td>
<td></td>
<td>14.5 Mi</td>
</tr>
<tr>
<td>__llvm_prf_names</td>
<td></td>
<td>7.68 Mi</td>
</tr>
<tr>
<td>__llvm_prf_data</td>
<td></td>
<td>4.50 Mi</td>
</tr>
<tr>
<td>Total Binary Size</td>
<td>119 Mi</td>
<td>175 Mi</td>
</tr>
<tr>
<td>Overhead</td>
<td></td>
<td>47%</td>
</tr>
</tbody>
</table>

**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!

```c
@__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!

```c
// 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

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

Derives function name using __llvm_prf_names
IRPGO__llvm_prf_data

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

```cpp
// 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 \textbf{profile} using a \textbf{small binary}

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

- Extract \texttt{\_\_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

```c
// 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

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

```c
!1 = distinct !DIGlobalVariable(name: "__profc__Z3foov", ..., annotations: !11)
!11 = !{!12, !13, !14}
!12 = !{"Function Name", "_Z3foov"}
!13 = !{"CFG Hash", 164 742261418966908927}
!14 = !{"Num Counters", i32 1}
```
## Debug Info Correlation

47% ➞ 36% size overhead

<table>
<thead>
<tr>
<th>Section</th>
<th>Base</th>
<th>IRPGO</th>
<th>Lightweight</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>65.8 Mi</td>
<td>93.0 Mi</td>
<td>90.3 Mi</td>
</tr>
<tr>
<td>__llvm_prf_cnts</td>
<td>14.5 Mi</td>
<td>14.5 Mi</td>
<td></td>
</tr>
<tr>
<td>__llvm_prf_names</td>
<td>7.68 Mi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__llvm_prf_data</td>
<td>4.50 Mi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Binary Size</td>
<td>119 Mi</td>
<td>175 Mi</td>
<td>162 Mi</td>
</tr>
<tr>
<td>Overhead</td>
<td></td>
<td>47%</td>
<td>36%</td>
</tr>
</tbody>
</table>
Debug Info Correlation

```
clang -g -fprofile-generate -mllvm -debug-info-correlate main.cpp
```
llvm-profdata merge --debug-info <dbg> default_*.*.profdata -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

<table>
<thead>
<tr>
<th>Section</th>
<th>Base</th>
<th>IRPGO</th>
<th>Lightweight</th>
<th>Function Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>65.8 Mi</td>
<td>93.0 Mi</td>
<td>90.3 Mi</td>
<td>69.5 Mi</td>
</tr>
<tr>
<td>__llvm_prf_cnts</td>
<td>14.5 Mi</td>
<td>14.5 Mi</td>
<td>163 Ki</td>
<td></td>
</tr>
<tr>
<td>__llvm_prf_names</td>
<td>7.68 Mi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>__llvm_prf_data</td>
<td>4.50 Mi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Binary Size</td>
<td>119 Mi</td>
<td>175 Mi</td>
<td>162 Mi</td>
<td>125 Mi</td>
</tr>
<tr>
<td>Overhead</td>
<td></td>
<td>47%</td>
<td>36%</td>
<td>5%</td>
</tr>
</tbody>
</table>
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](https://reviews.llvm.org/D124490)
## Basic Block Coverage

47%  ➪  17% size overhead

<table>
<thead>
<tr>
<th>Section</th>
<th>Base</th>
<th>IRPGO</th>
<th>Lightweight</th>
<th>Function Coverage</th>
<th>Block Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>65.8 Mi</td>
<td>93.0 Mi</td>
<td>90.3 Mi</td>
<td>69.5 Mi</td>
<td>82.1 Mi</td>
</tr>
<tr>
<td>__llvm_prf_cnts</td>
<td>14.5 Mi</td>
<td>14.5 Mi</td>
<td>163 Ki</td>
<td></td>
<td>1.38 Mi</td>
</tr>
<tr>
<td>__llvm_prf_names</td>
<td>7.68 Mi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>__llvm_prf_data</td>
<td>4.50 Mi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Binary Size</td>
<td>119 Mi</td>
<td>175 Mi</td>
<td>162 Mi</td>
<td>125 Mi</td>
<td>139 Mi</td>
</tr>
<tr>
<td>Overhead</td>
<td>47%</td>
<td>36%</td>
<td>5%</td>
<td>17%</td>
<td></td>
</tr>
</tbody>
</table>
Special Thanks

- Wenlei He
- Nikolai Tillmann
- Julian Mestre
- Sergey Pupyrev
- Greg Clayton
Sources and Links


- **Lightweight Instrumentation**
  - [https://discourse.llvm.org/t/instrprofiling-lightweight-instrumentation/59113](https://discourse.llvm.org/t/instrprofiling-lightweight-instrumentation/59113)
  - [https://reviews.llvm.org/D115693](https://reviews.llvm.org/D115693)
  - [https://reviews.llvm.org/D115915](https://reviews.llvm.org/D115915)

- **Function Entry Coverage**
  - [https://reviews.llvm.org/D116180](https://reviews.llvm.org/D116180)

- **Minimal Block Coverage**
  - [https://reviews.llvm.org/D124490](https://reviews.llvm.org/D124490)