Source-based Code Coverage for Embedded Use Cases

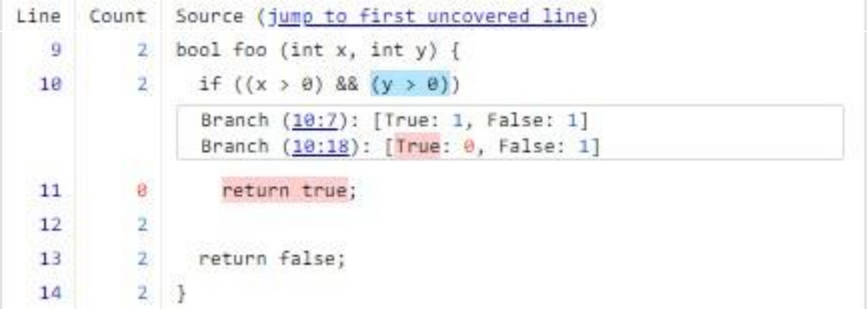
Alan Phipps, Texas Instruments

Cody Addison, Nvidia

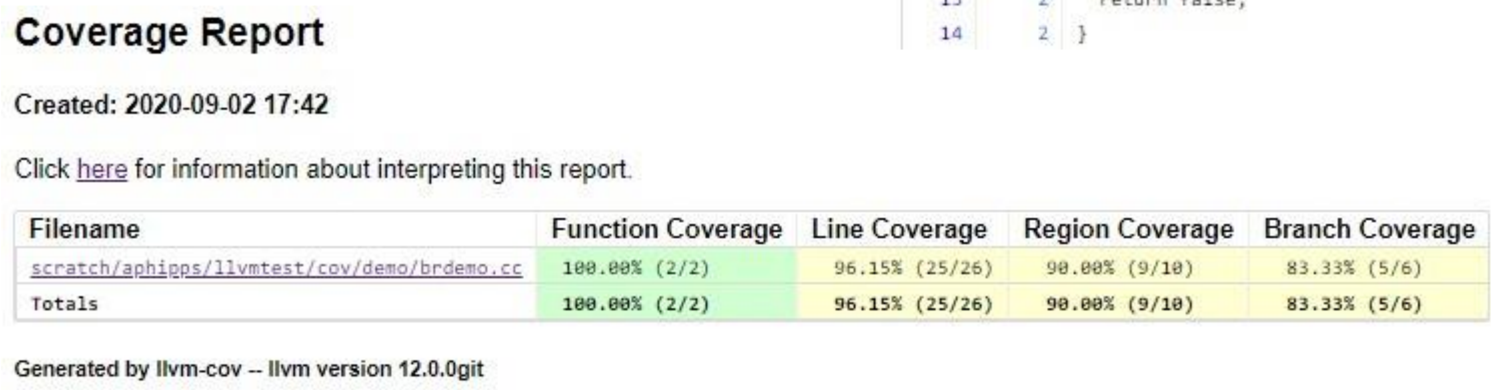
2020 LLVM Developers' Meeting, October 2020

What is Code Coverage?

- A measurement for how thoroughly code has been executed during testing
  - All sections of code have an associated test
  - Un-executed code may be at higher risk of having lurking bugs



```
Line   Count   Source (jump to first uncovered line)
   9       2   bool foo (int x, int y) {
  10       2     if ((x > 0) && (y > 0))
                   Branch (10:7): [True: 1, False: 1]
                   Branch (10:18): [True: 0, False: 1]
  11       0       return true;
  12       2
  13       2     return false;
  14       2   }
```



Coverage Report

Created: 2020-09-02 17:42

Click here for information about interpreting this report.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
scratch/aphipps/llvmtest/cov/demo/brdemo.cc	100.00% (2/2)	96.15% (25/26)	90.00% (9/10)	83.33% (5/6)
Totals	100.00% (2/2)	96.15% (25/26)	90.00% (9/10)	83.33% (5/6)

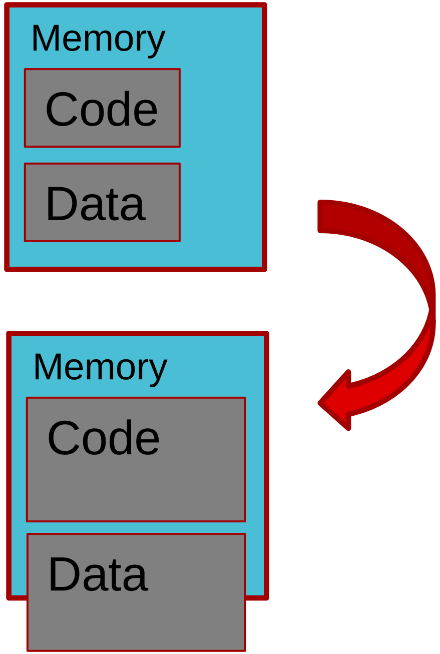
Generated by llvm-cov -- llvm version 12.0.0git

The Challenge

- Embedded devices usually have tight memory requirements

- LLVM Source-based Code Coverage has large size requirements
  - Additional instructions added to instrument your code
  - Additional runtime code to control profile data merging
    - This includes counter merging and profile data input and output
  - Additional data sections to track counters and coverage data

- There ARE things we can do to reduce the size!



Memory

Code

Data

Memory

Code

Data

1. Must all data sections be in memory?

- No!

- Code Coverage relies on several data sections:
  - `__llvm_prf_cnts` : Profile counters, incremented at runtime
  - `__llvm_covfun` : Coverage Function Records
  - `__llvm_covmap` : Coverage Mapping Data Records
  - `__llvm_prf_data` : Profile Data
  - `__llvm_prf_names` : Profile Function names

These sections may comprise **80%-90% of the data** but *do not require runtime modification*

1. Must all data sections be in memory?

- No!

- Code Coverage relies on several data sections:
  - `__llvm_prf_cnts` : Profile counters, incremented at runtime
  - `__llvm_covfun` : Coverage Function Records
  - `__llvm_covmap` : Coverage Mapping Data Records
  - `__llvm_prf_data` : Profile Data
  - `__llvm_prf_names` : Profile Function names

Move these sections *out of memory*, preserved in object file

- Modify `llvm-profdata` to accept an **object file argument**
  - Move it *off-line*: Combine its data with *downloaded* profile counters, producing an indexed profile data file

2. Can we reduce runtime support?

- We just moved most processing of raw profile data off-line

- Runtime features are included that are *unnecessary* for embedded platforms
  1. Runtime counter merging
  2. Use of environment variable to control where output goes
  3. Indexed profile writing output
  4. Buffering data for writing output
  5. Reading data input in for profile-guided optimization (PGO)

- How big is `libclang_rt.profile.a`? **100kb for Armv7m**!

compiler-rt/lib/profile/CMakeLists.txt:

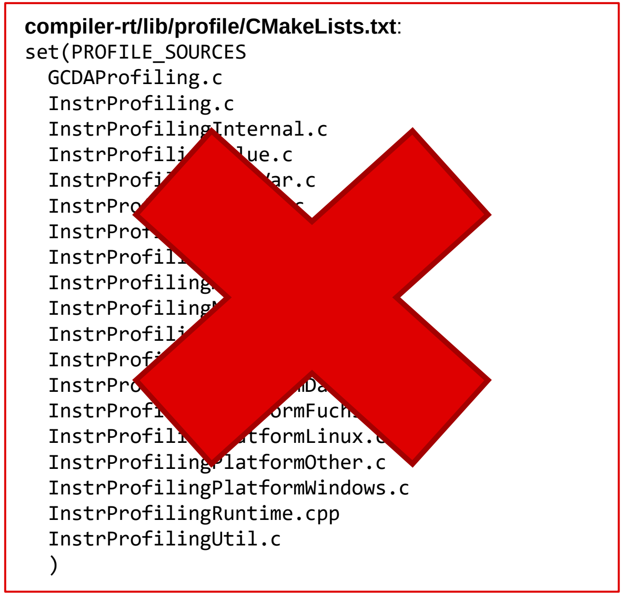
```
set(PROFILE_SOURCES
  GCDAProfiling.c
  InstrProfiling.c
  InstrProfilingInternal.c
  InstrProfilingValue.c
  InstrProfilingBiasVar.c
  InstrProfilingBuffer.c
  InstrProfilingFile.c
  InstrProfilingMerge.c
  InstrProfilingMergeFile.c
  InstrProfilingNameVar.c
  InstrProfilingVersionVar.c
  InstrProfilingWriter.c
  InstrProfilingPlatformDarwin.c
  InstrProfilingPlatformFuchsia.c
  InstrProfilingPlatformLinux.c
  InstrProfilingPlatformOther.c
  InstrProfilingPlatformWindows.c
  InstrProfilingRuntime.cpp
  InstrProfilingUtil.c
  )
```

2. Can we reduce runtime support?

- We just moved most processing of raw profile data off-line

- Runtime features are included that are *unnecessary* for embedded platforms
  1. Runtime counter merging
  2. Use of environment variable to control where output goes
  3. Indexed profile writing output
  4. Buffering data for writing output
  5. Reading data input in for profile-guided optimization (PGO)

- How big is `libclang_rt.profile.a`? **100kb for Armv7m**!
- If we only support for basic writing of counters and remove everything else → **4kb for Armv7m!**

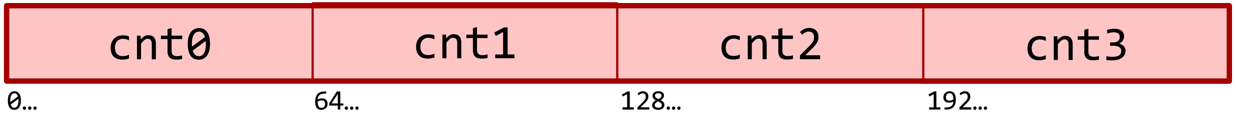


compiler-rt/lib/profile/CMakeLists.txt:

```
set(PROFILE_SOURCES
  GCDAProfiling.c
  InstrProfiling.c
  InstrProfilingInternal.c
  InstrProfili    lue.c
  InstrProfil        ar.c
  InstrPro
  InstrProf
  InstrProfil
  InstrProfiling
  InstrProfiling
  InstrProfil
  InstrProfi
  InstrPro            mDa
  InstrProfi           ormFuch
  InstrProfil     atformLinux.c
  InstrProfilingPlatformOther.c
  InstrProfilingPlatformWindows.c
  InstrProfilingRuntime.cpp
  InstrProfilingUtil.c
  )
```

3. What about counter size?

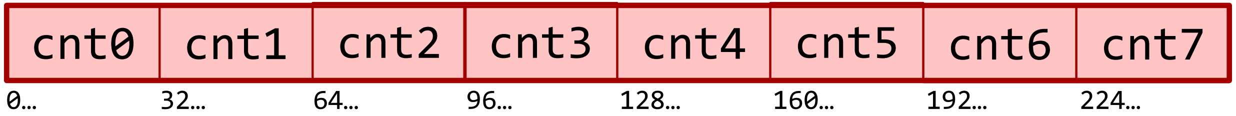
- Remember…. we made `__llvm_prf_cnts` the *only coverage data section in memory*
  - *But this is comprised of counters that are 64bits in size*



cnt0 cnt1 cnt2 cnt3

0… 64… 128… 192…

- Embedded applications can get away with smaller counter sizes
- Reduce the counter size to 32bits – ***50% reduction in size!***



cnt0 cnt1 cnt2 cnt3 cnt4 cnt5 cnt6 cnt7

0… 32… 64… 96… 128… 160… 192… 224…

- Even better: make counter size configurable to any reasonable size (16bits, 8bits)
  - Use saturating addition to prevent against overflow on small counter sizes

Thank you!