Agenda

• Introduction
• Rationale
• Technical Overview
• Tooling Access and Usage
• Future Direction
• Q&A
About Us

**SW Quality organization**
within Arm’s
Open Source Software Group
Basil Eljuse - Principal SW Engr – Tech Lead
Saul Romero - Staff SW Engr – Tooling Specialist

**Focus is on**
Quality improvement initiatives
Common hard tooling problems
Automation improvements
Mostly internal faced

**Public contributions**
big.LITTLE sched-tests (precursor to LISA tool)
scmi-tests (part of ACS)
qa-tools (most recent contribution)
Rationale
Why we went down this path?

Motivation

- Emphasis on ‘demonstrable quality’ more than ever
- Lack of measures => ‘flying blind’
- Code coverage is one useful measure
- Code coverage – feedback with potential for actionable outcomes
  - indicator of test coverage
    - Is my test-set good enough?
    - Can I direct my test effort better?
- residual risk to quality
  - What am I not covering with my current tests?

Problem Statement

- Firmware projects - Traditional coverage tooling with code instrumentation not an option
  - Memory constraint platforms
    - code size limitations
  - Higher degree of platform code dependency
    - emulation expensive and less desirable
  - No COTS tooling available

- Need: Perform code coverage measurement without doing code instrumentation.
Trace-based Coverage Tooling Design

https://gitlab.arm.com/tooling/qa-tools/-/blob/master/coverage-tool/docs/design_overview.md

- **Capture Phase**
  - Fastmodel - MTI based custom plugin captures trace with instructions executed

- **Analysis Phase**
  - Dwarf signature (-g compiler flag) – C source mapping
  - Object dump data – Instruction level mapping

- **Visualisation Phase**
  - Lcov reports

**Overview**

- **Source Files**
  - Elf files: Built with -g option
  - Trace log files: From CI or local runs
  - Config files: Configuration parameters

- **Tools**
  - Objdump
  - Readelf

- **Post-Processor 1**
  - Intermediate layer
  - Json data: Self contained intermediate report to be consumed by reporting clients or further post processing

- **Post-Processor 2**
  - Translates from intermediate layer to .info files to be consumed by LCOV

- **LCOV genhtml**
  - Self contained intermediate report to be consumed by reporting clients or further post processing
Current Tooling Capability

What is supported today?

- Statement coverage
- Function coverage
- Branch Coverage
- Merging of related coverage reports
- Baseline viewing of coverage info

**Lcov Report View**

<table>
<thead>
<tr>
<th>Directory</th>
<th>Line Coverage</th>
<th>Functions</th>
<th>Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>85.3%</td>
<td>89.3%</td>
<td>77.2%</td>
</tr>
<tr>
<td>llc</td>
<td>85.0%</td>
<td>89.0%</td>
<td>77.0%</td>
</tr>
<tr>
<td>zircon</td>
<td>85.1%</td>
<td>89.2%</td>
<td>77.4%</td>
</tr>
<tr>
<td>nanos</td>
<td>85.2%</td>
<td>89.3%</td>
<td>77.5%</td>
</tr>
<tr>
<td>compiler</td>
<td>85.3%</td>
<td>89.4%</td>
<td>77.6%</td>
</tr>
<tr>
<td>core</td>
<td>85.4%</td>
<td>89.5%</td>
<td>77.7%</td>
</tr>
<tr>
<td>driver</td>
<td>85.5%</td>
<td>89.6%</td>
<td>77.8%</td>
</tr>
<tr>
<td>dev</td>
<td>85.6%</td>
<td>89.7%</td>
<td>77.9%</td>
</tr>
<tr>
<td>test</td>
<td>85.7%</td>
<td>89.8%</td>
<td>78.0%</td>
</tr>
<tr>
<td>main</td>
<td>85.8%</td>
<td>89.9%</td>
<td>78.1%</td>
</tr>
<tr>
<td>perf</td>
<td>85.9%</td>
<td>90.0%</td>
<td>78.2%</td>
</tr>
<tr>
<td>all</td>
<td>86.0%</td>
<td>90.1%</td>
<td>78.3%</td>
</tr>
</tbody>
</table>

LCOV - code coverage report

```c
1: assert (L & SCTR_M_RBT);         // assert (L & SCTR_M_RBT);
2: if (L != 0x0) {                 // assert (L & SCTR_M_RBT);
    else assert (L & SCTR_M_RBT);  // assert (L & SCTR_M_RBT);
    #ifdef ENABLE_ASSERTIONS
    #endif
    /* Perform remaining generic architectural setup from G3 */
    /* Perform platform setup in M1 */
    /* Get the image id of next image to load and run. */
    #endif
    #ifdef TRUSTED_BOOT_ROOT
    /* We currently interpret any image id other than */
    /* M1, IMAGE_0 as the start of firmware upgrade. */
    /* Console flushed. */
```
Capture Phase - Details

Model Trace Interface Plugin

- Instantiate the MTI plugin instance
- Register plugin instance with Simulation
- Discover a trace source “INSTR”
- Register callback handler to record trace “field” capture in memory
- At termination dump the trace info from memory to file

Useful Reference - Model Trace Interface Reference Manual v1.1

https://gitlab.arm.com/tooling/qa-tools/-/blob/master/coverage-tool/docs/plugin_user_guide.md

PREPARE:: Building the model plugin
-------------------------
make -C model-plugin PVLIB_HOME=/path/to/modellib
For TF-A CI:
PVLIB_HOME=$warehouse/SysGen/PVModelLib/$model_version/$model_build/external
Toolchain: aarch64-linux-gnu (we reused the same used by their CI)
Objects created: CoverageTrace.so, CoverageTrace.o, PluginUtils.o

EXECUTE:: Capturing a trace
-----------------
You need to add two options to your model command-line:
--plugin /path/to/CoverageTrace.so
[-C TRACE.CoverageTrace.trace-file-prefix="/path/to/TRACE-PREFIX"]

Example from TF-A CI:
/arm/warehouse/SysGen/Models/11.6/45/models/Linux64_GCC-4.9/FVP_Base_RevC-2xAEMv8A \
--data cluster0.cpu0=el3_payload.bin@0x80000000  \
--data cluster0.cpu0=ns_bl1u.bin@0x0beb8000  \
--plugin=/work/workspace/workspace/tf-worker/test-definitions/scripts/tools/code_coverage/fastmodel_baremetal/bmcov/model-plugin/CoverageTrace.so \
-C bp.flashloader0.fname=fip.bin \n-C bp.secureflashloader.fname=bl1.bin \n-C bp.ve_sysregs.exit_on_shutdown=1 \n-C pctl.startup=0.0.0.0 Q 1000 "$@"

OUTPUT:: Coverage Trace sample output:
-------------------------------
00001ce8 16 4
00001cec 16 4
00001cf0 16 4
00001cf4 16 4
00001cf8 16 4
Analysis Phase - Details

https://gitlab.arm.com/tooling/qa-tools/-/blob/master/coverage-tool/docs/reporting_user_guide.md

Elm files

Objdump

MTI Plugin

PostProcessing stage#1

Intermediate report in Json format:

```
"sources": {
  "bl/v/aarch64/bl1_arch_setup.c": {
    "functions": {
      "bl1_arch_setup": true
    },
    "lines": {
      "0": {
        "covered": true,
        "elf_index": {
          "0": {
            "5856": {
              "b2760000": 1
            }
          }
        }
      }
    }
  }
}
```

metadata and c-source files.

Listing functions and number lines (in the C file) with coverage.

Includes associated asm lines for the given c-source line.

Source tree

Elf files

MTI Plugin

Objdump

Analysis Phase - Details

https://gitlab.arm.com/tooling/qa-tools/-/blob/master/coverage-tool/docs/reporting_user_guide.md
Analysis Phase – Details

Intermediate report in Json format

PostProcessing stage#2 (generate_info_file.py)

Lcov info file

To genhtml reports

Intermediate reports in Json format

PostProcessing stage#2

Intermediate reports in Json format

PostProcessing stage#2

PostProcessing stage#2

PostProcessing stage#2

(merge.py)

Lcov info files

Merged Lcov info file

Merge info files
The LCOV open source project (http://ltp.sourceforge.net/coverage/lcov.php) for visualisation.

- Starting from the JSON file a .info (LCOV) file(s) is generated
- The HTML code is produced starting from the .info file and the original C source code.
  - includes information about line, function and branch coverage
  - allows to browse through the source files and check their coverage.
Gotchas and Learnings

Is there any catch?

- Optimisation levels (especially -Os) influence coverage stats
  - Only source lines with dwarf signature can yield coverage info
  - Optimisation can lead to functions be inlined or code removed from binary

- File encoding issues affects post processing

- Lexical analyser to help with source code parsing did not help
  - Finally used simple python text parsing logic

- Toolchain bugs affect coverage generation
Tooling Access and Usage

Where to get this tool from?

• Open sourced the MTI plugin implementation and the associated post processing scripts
  • https://gitlab.arm.com/tooling/qa-tools
• Any feedback or contributions very much welcomed.
  • See https://gitlab.arm.com/tooling/qa-tools/-/tree/master/coverage-tool#contributing
• Internally used for both TF-A, TF-M and SCP projects
  • TF-M project uses an early proof-of-concept workflow which uses LAVA setup

How can it help you?

• Tell you where to redirect your testing effort
• Address potential quality risks due to uncovered code-paths
• Data from the tool can be used to visualize ongoing coverage trend as your project evolves
• Can provide you with profiling data on executed instructions – potentially identify bottlenecks or need for better code reuse
Future Direction
What more?

• Extend the trace-based coverage measurement methodology to Silicon platforms
  • Early prototype done with Juno platform
  • Feasible; but some automation challenges persist

• MC/DC coverage
  • We can dump register values in addition to instructions executed
  • Early prototype done to show the MTI extension; but more work needed

• Alternative to a custom plugin (MTI)
  • Few possibilities with some standard fastmodel trace extensions; Early exploration!
Q&A
Thank You
Danke
Merci
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
ধন্যবাদ
תודה