Branch Coverage: Squeezing more out of LLVM Source-based Code Coverage

Alan Phipps, Texas Instruments

2020 LLVM Developers’ Meeting
What is Source-based Code Coverage?

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

• Supported Coverage criteria (in increasing level of granularity)
  – Function
    • Percentage of code functions executed at least once
  – Line
    • Percentage of code lines executed at least once
  – Region
    • Percentage of code statements executed at least once
Basic Phases (High Level)

- Counter Allocation and Counter-to-Source Region Mapping (clang)
- Counter Instrumentation in LLVM IR (clang)
- Test Execution
- Data Visualization (llvm-profdata & llvm-cov)
Counter Region Mapping and Instrumentation

• Counters are inserted into basic blocks of generated code mapped to source

```java
line 9: bool foo(int x, int y) {
    Counter1++
line 10:    if ((x > 0) && (y > 0))
        ^Counter2++
line 11:        return true;
line 12:    Counter3++
line 13:    return false;
line 14: }
```

• `Counter1` instrumented to track
  • Region (9:24 → 10:23)
  • Function (line 9 – foo())
  • Line (line 10)
  • Statement: if-stmt

• `Counter2` instrumented to track
  • Region (10:18 → 10:25)
  • Statement (y > 0)

• `Counter3` instrumented to track
  • Region (11:0 → 11:12)
  • Line coverage (line 11)

• `(Counter1 – Counter3)` tracks
  • Region (12:0 → 14:0)
  • Line coverage (line 13)
LLVM Coverage Visualization

• LLVM Coverage Utility (llvm-cov)

• Text (llvm-cov)

```
bool foo (int x, int y) {
  if ((x > 0) && (y > 0))
    return true;
  return false;
}
```

Coverage Report

Created: 2020-09-09 15:28

Click here for information about interpreting this report.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Function Coverage</th>
<th>Line Coverage</th>
<th>Region Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>scratch/aphiops/llvmtest/cov/demo/brdemo.cc</td>
<td>100.00% (2/2)</td>
<td>96.15% (25/26)</td>
<td>90.00% (9/10)</td>
</tr>
<tr>
<td>Totals</td>
<td>100.00% (2/2)</td>
<td>96.15% (25/26)</td>
<td>90.00% (9/10)</td>
</tr>
</tbody>
</table>

Generated by llvm-cov -- llvm version 12.0.0git
Why is branch Coverage Important?

<table>
<thead>
<tr>
<th>Line</th>
<th>Cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
</tr>
</tbody>
</table>

- There are two conditions on line 10 that form a decision: \((x > 0), (y > 0)\)
- Line 11 shows that “return true” was executed once
  - What was the execution path through the control flow that facilitated this?
  - What was the execution path through the control flow around this?
  - If we don’t know, we can’t be sure we are executing all paths!
- Branch Coverage tells us this!
  - How many times is each condition taken (True) or not taken (False)?
**LLVM Coverage Visualization + Branch Coverage**

- **LLVM Coverage Utility (llvm-cov)**

  ```
  bool foo (int x, int y) {
    if ((x > 0) && (y > 0)) {
      return true;
    }
    return false;
  }
  ```

- **Text (llvm-cov)**

  ```
  bool foo (int x, int y) {
    if ((x > 0) && (y > 0)) {
      return true;
    }
    return false;
  }
  ```

**Coverage Report**

Created: 2020-09-02 17:42

Click here for information about interpreting this report.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Function Coverage</th>
<th>Line Coverage</th>
<th>Region Coverage</th>
<th>Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>scratch/aphipps/llvmtest/cov/demo/brdemo.cc</td>
<td>100.00% (2/2)</td>
<td>96.15% (25/26)</td>
<td>90.00% (9/10)</td>
<td>83.33% (5/6)</td>
</tr>
<tr>
<td>Totals</td>
<td>100.00% (2/2)</td>
<td>96.15% (25/26)</td>
<td>90.00% (9/10)</td>
<td>83.33% (5/6)</td>
</tr>
</tbody>
</table>
Goal: Ensure 100% Branch Coverage

- C short-circuit semantics on logical operators
  - Testing all individual conditions also tests corresponding decisions

```c
bool foo(int x, int y) {
    if ((x > 0) && (y > 0))
        return true;
    return false;
}

foo(1, 0): (x > 0) = true
(x > 0) && (y > 0) = false
(y > 0) = false

foo(0, 1): (x > 0) = false
(y > 0) = ... not executed!
(x > 0) && (y > 0) = false

foo(1, 1): (x > 0) = true
(y > 0) = true
(x > 0) && (y > 0) = true
```
How is Branch Coverage implemented?
Clang Source Region Creation

- Counter-to-Source Region Mapping (clang)
- Counter Instrumentation in LLVM IR (clang)
- Test Execution
- Data Visualization

Regions created based on AST walk
CounterMappingRegion

struct CounterMappingRegion {
    enum RegionKind {
        /// A CodeRegion associates some code with a counter.
        CodeRegion,

        /// An ExpansionRegion represents a file expansion region that associates
        /// a source range with the expansion of a virtual source file, such as
        /// for a macro instantiation or #include file.
        ExpansionRegion,

        /// A SkippedRegion represents a source range with code that was skipped
        /// by a preprocessor or similar means.
        SkippedRegion,

        /// A GapRegion is like a CodeRegion, but its count is only set as the
        /// line execution count when its the only region in the line.
        GapRegion,

        /// A BranchRegion represents leaf-level boolean expressions and is
        /// associated with two counters, each representing the number of times the
        /// expression evaluates to true or false.
        BranchRegion
    }

    /// Primary Counter that is also used for Branch Regions (TrueCount).
    Counter TrueCount;

    /// Secondary Counter used for Branch Regions (FalseCount).
    Counter FalseCount;

    unsigned FileID, ExpandedFileID;
    unsigned LineStart, ColumnStart, LineEnd, ColumnEnd;

};

CounterMappingRegion
associates a source range with a counter. It uses RegionKind to
identify how to interpret its data.

1.) Extend RegionKind to include a new BranchRegion kind to
represent branch-generating conditions

2.) Use existing Counter to represent “True” BranchRegion
counts

3.) Add a second Counter to represent “False” BranchRegion
counts
Counter Region Mapping (clang)

- Instrumentation profile **Counters** are already created for statement regions
  - We can *trivially* reuse them to calculate Branch condition counts!
  - A **Counter** can also refer to an arithmetic expression between two counters

```c
Counter1++
if ( C ) {
    Counter2++
    ...
}
```

- **Counter1** maps to “Parent” region
- **Counter2** maps to If-Stmt “Then” region
- For **BranchRegion(C)**
  - C.TrueCounter = Counter2
  - C.FalseCounter = Counter1 – Counter2

This is true for all control-flow statements: if, for, while, switch, ternary ?:
Clang Counter Instrumentation

Counter-to-Source Region Mapping (clang)

Counter Instrumentation in LLVM IR (clang)

Test Execution

ASTs lowered to LLVM IR
Since we reuse counters, no special instrumentation needed! … except …

Data Visualization
Counter Instrumentation for Logical Operators

I have to instrument a new counter (**Counter3**) to track C2’s counts

```c
bool X = C1 || C2 || Counter3++
```

- **Counter1** maps to “Parent” region
- **Counter2** maps to “C2”, the right-hand-side, representing C2 execution count
- C short-circuit semantics on logical operators
  - **Counter2** increments *only* when C1 is false
- For **BranchRegion(C1)**
  - C1.FalseCounter = Counter2
  - C1.TrueCounter = Counter1 – Counter2
- For **BranchRegion(C2)**
  - C2.FalseCounter = **Counter3**
  - C2.TrueCounter = **Counter2** – **Counter3**
Data Visualization

Counter-to-Source Region Mapping (clang)

Counter Instrumentation in LLVM IR (clang)

Test Execution

Data Visualization (llvm-cov)

Data Decoded and Statistics Calculated
Visualization (llvm-cov)

- Decode mapping regions and filter based on Function and Macro Expansion

```
line 9: bool foo(int x, int y) {
line 10:  if ((x > 0) && (y > 0))
line 11:    return true;
line 12:  return false;
line 13:  return false;
line 14: }
```

Function (foo)
- CodeRegion1 (9:24-10:23)
- CodeRegion2 (11:0-11:12)
- CodeRegion3 (12:0-14:0)

BranchRegions:
- BranchRegion1 (10:5-10:11)
- BranchRegion2 (10:16-10:22)

```
line 18: #define MAX(x,y) ((x) > (y) ? (x) : (y))
```

Expansion (MAX)
- CodeRegion1 (18:18-18:40)

BranchRegions:
- BranchRegion1 (18:19-18:24)

```
line 19: bool bar(int x, int y) {
line 20:  return MAX(x,y);
line 24: }
```

Function (bar)
- CodeRegion1 (19:24-24:0)
- ExpansionRegion1 (20:10-20:13)
Visualization (llvm-cov) SubViews

- Extend notion of region **SubView** to include branches
  - **SubViews** are demarcated nested views in the source-code
  - Branches on the same line are grouped into the same **SubView**
  - **SubViews** are also used to demarcate macro expansions
    - Macro expansions can be recursive
    - Macro expansions can contain conditions

- Extend summary reports to include Branch Coverage
  - Add **BranchCoverageInfo** class

```c
if (BRANCH_MACRO(arg1, arg1))
    #define BRANCH_MACRO(x, y) (x == y)
    Branch (9:28): [True: 2, False: 0]

printf("This executes on a macro expansion\n");
```

**BranchCoverageInfo**
- Total # of Branches (2 per region)
- # Branches executed *at least once*
Branch Coverage Future Optimizations

• Better counter reuse for logical operators
  – Nested conditions: `bool myval = (C1 && C2 && (C3 || C4));`

• Enable HTML ToolTip “hover” capability on source conditions
  – Hovering will reveal actual True/False Branch Counts
  – Similar to how region coverage counts show up today

• Better identification of special branch regions
  – Identify an *implicit* default Case in a `switch` statement
  – Identify the sense of constant-folded conditions: *always* True or *never* True
What’s Next: MC/DC

- **Ultimate Goal**: Modified Condition/Decision Coverage (MC/DC)
  - Percentage of all condition outcomes that *independently* affect a decision outcome
  - *Built on top of branch-coverage*

- Usually involves emitting a truth table to confirm all possibilities
Observations on GCC Branch Coverage

- GCC HTML (LCOV)
  - True/False Branch Data shown
    - “+” → Executed at least once
    - “-” → Not Executed (i.e. “0”)
    - Hover to see counts

- GCC Text (GCOV)
  - Difficult to tie branches to source
    - Which branch goes with which condition?
    - Which branch represents taken vs not taken?
  - In other contexts...
    - May see additional branches that aren’t visible in source code
    - Some branches may be removed
      - GCC advises against using optimization with code coverage

```c
function Z3fooii called 2 returned 100% blocks executed 80%
  2:    9:bool foo (int x, int y) {
  2:   10:  if ((x > 0) && (y > 0))
  2:   11:    return true;
  2:   12:  return false;
  2:   13: return false;
  2:   14:}
```

```c
branch 0 taken 1 (fallthrough)
branch 1 taken 1
branch 2 taken 0 (fallthrough)
branch 3 taken 1
```

```c
#####:   11:    return true;
```

```c
2:   12:
```

```c
2:   13:    return true;
```

```c
2:   14:}
```
function _Z3fooii called 2 returned 100% blocks executed 80%

branch 0 taken 1 (fallthrough)
branch 1 taken 1
branch 2 taken 0 (fallthrough)
branch 3 taken 1

#####: 11: return true;
-: 12:
2: 13: return false;
-: 14:}

branch (10:7): [True: 1, False: 1]
branch (10:18): [True: 0, False: 1]
Current State of LLVM Branch Coverage

• Implementation is complete -- in the process of upstreaming the work!
  – Phabricator Review https://reviews.llvm.org/D84467

• Should be included with stock LLVM Source-based Code Coverage

• A lot of ways to improve branch coverage! Want to be involved?
  – Contact me! a-phipps@ti.com
Thank you!

- Acknowledgements
  - Vedant Kumar, Apple
  - Cody Addison, Nvidia
  - Alan Davis, Texas Instruments