Analysis of Test Coverage Data on a Large-Scale Industrial System

by Erik Sven Vasconcelos Jansson
<erija578@student.liu.se>

Examinator: Lena Buffoni - Advisor: Bernhard Thiele
Department of Computer and Information Science
at Linköping University (LiTH), Sweden

September 15, 2016
Importance of *software testing*, within any *mission critical software* products.

See for e.g. NASA's *Climate Orbiter*...

The *code/test coverage metric* can be used in determining e.g. *testing holes*.

```c
1 int factorial(int n) {
2     if (n == 0) return 1;
3     else if (n < 0) return -1;
4     return factorial(n - 1)*n;
5 }
6
7 int main(int, char**) {
8     assert(factorial(0) == 1);
9     assert(factorial(4) == 24);
10    return 0;
11 }
```

---

1 Climate Orbiter: http://mars.nasa.gov/msp98/
Project at *Ericsson (R&D) Linköping*, the given *large-scale industrial system*.

*Production & unit tests*: 960 KSLOC, the *target function tests*: 564 KSLOC.

*Tasks*: integrate a *coverage gathering*, and *analysis system* for *function tests*; determine “*feasibility*” of the addition.

*Goal*: develop framework, upon which *test case selection/prioritization* could be derived in future work. PL3 ≈ 18h.
Motivation

- **Software testing** is **complicated** and **expensive**, research \([B^{+}81]\) has shown 50% project effort.
- **Coverage data** provides **insight** into the **nature** of **tests**; giving useful information about them.
- Research in **large-scale test coverage is scarce**, according to several papers such as \([ABR^{+}11]\)
- **Gathering & analysis** is problematic \([ABR^{+}11]\)
- Articles don’t present **development experience**.
Problems

- **Coverage gathering is problematic on large-scale systems:**
  - Existing coverage tools *don’t integrate well* on all setups.
  - *Performance* and *resource usage* might be *neg. affected*.
  - *Research question: how feasible is such an extension?*

- **Analyzing raw test coverage data manually isn’t feasible:**
  - *Huge amounts* of data produced for *large-scale software*.
  - *Difficult* to extract any *meaningful test case properties*.
  - *Research question: what does the analysis tools find?*
Challenges

- Performance sensitive system.
- Executes a series of *daemons*.
- Coverage for a *remote target*.
- *Huge amount of profile data*.
- *Isolate* changes in *behaviour*.

```c
1 static int bb[4] = {};  
2 static int factorial(int n) {  
3     ++bb[0];  
4     if (n == 0) {  
5         ++bb[1];  
6         return 1;  
7     } else if (n < 0) {  
8         ++bb[2];  
9         return -1;  
10    }  
11    ++bb[3];  
12    return factorial(n - 1) * n;  
13 }  
14  
15 int main(int, char**) {  
16    assert(factorial(0) == 1);  
17    assert(factorial(4) == 24);  
18    printf("bb 0: %i", bb[0]);  
19    printf("bb 1: %i", bb[1]);  
20    printf("bb 2: %i", bb[2]);  
21    printf("bb 3: %i", bb[3]);  
22    return 0;  
23 }  
```

2 emacs?: videohive.net/item/burning-notebook/
Instrument production code with coverage capabilities.

Measure the performance effects of these, statistically.

Add flushing signal, to deal with the daemon software.

Extend testing system to fetch coverage on test’s end.

Build/find tool to analyze, and locate test similarities.
Production code is built with GCC → GCov is built-in, & proven to have minimal performance impact, \( \approx 3\% \).

- e.g.: `-fprofile-arcs -ftest-coverage -O0`.

- However, remote target → coverage is dumped wrong, solved with `GCOV_PREFIX` & `GCOV_PREFIX_STRIP`.

- Enabled in the *build system* with: `--coverage` flag. (←)
Several instrumented daemons on the target device.

Coverage information is dumped when terminated, however, device reboots if this is done... Not good.

By calling `__gcov_flush()`, processes can dump, without being terminated explicitly (no rebooting).

Synchronize flush: `kill -sUSR1 $(pidof *)`, handler enabled on all daemons by: `--coverage`. (←)
Solution
Automatic Fetching

- Extends the Maven/Jcat testing flow.
- Automatically flushes coverage down, splitting the individual test case data.
- Switched with `fetch.coverage = (testcase | testsuite)`.

![Diagram]

- onStart(): Clear Coverage
  - `fetch.coverage!="false`
- onExecute(): Test Suite
- onFinish(): Fetch Coverage
  - `fetch.coverage=testsuite`
  - `fetch.coverage=testcase`

1. fetch
2. upload
3. download

TS
DUT
1. fetch
2. upload
3. download

10 / 25
How to measure test case similarities? see Jaccard index, Hamming distance.

Initial solution modified gcov-tool.

Current implementation scovat.py, gives set operations on coverage data, required by Hamming & Jaccard. (←)

\[ J(A, B) = \frac{|A \cap B|}{|A \cup B|}, \quad d_J = 1 - J(A, B) \]
Solution
Usage Workflow

1. Build software with `bb --coverage <recipe>`, enabling coverage instrumentations and daemon signal handlers setup.
2. Upload software packages with `stp_setup <deviceid>`.
3. Enable the `fetch.coverage=testcase` testing property, enabling the Maven/JCat testing framework fetch coverages.
4. Execute the desired test case/suite, e.g. `PL2` on the devices.
5. Test coverage data is continuously fetched to the developer.
6. `scovat.py -gb $BUILDLOC -o cache` generates the intermediate coverage format, used later for modifying data.
7. `scovat.py -ao analysis cache/<A> cache/<B>` gives *Jaccard coefficient*, and *Hamming distance* for (A, B).
- Software *instrumentation* haven’t made any tests unstable.
- Fetching coverage from target → developer adds 20s time, largely caused by bugs in the *Trilead SSH* implementation.
- Consumes a *fixed disk space* of 8.3 MiB for each test case, which is transferred & removed continuously when fetched.
- The *suite execution time*, *processor*, and *memory usage* are measured statistically after a *t-distribution*, $\alpha = 5\%$, $n = 4$. 
<table>
<thead>
<tr>
<th>Sampled Dataset</th>
<th>Lower Time (h)</th>
<th>Upper Time (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instrumented</td>
<td>2.656</td>
<td>2.911</td>
</tr>
<tr>
<td>Non-Instrumented</td>
<td>2.075</td>
<td>2.260</td>
</tr>
</tbody>
</table>
## Results

### Memory Usage

<table>
<thead>
<tr>
<th>Sampled Dataset</th>
<th>Lower Usage (%)</th>
<th>Upper Usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instrumented</td>
<td>16.7663</td>
<td>18.8446</td>
</tr>
<tr>
<td>Non-Instrumented</td>
<td>17.8428</td>
<td>18.0185</td>
</tr>
</tbody>
</table>
Results
Processor Usage
## Results

### Processor Usage

<table>
<thead>
<tr>
<th>Sampled Dataset</th>
<th>Lower Usage (%)</th>
<th>Upper Usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instrumented</td>
<td>3.43483</td>
<td>4.12369</td>
</tr>
<tr>
<td>Non-Instrumented</td>
<td>4.28527</td>
<td>5.40044</td>
</tr>
</tbody>
</table>
Retrieved from the primary development test suite.

Upon producing *lcov report* with fetched coverage:

- *Statement coverage:* 59.3% out of 96,158 (lines).
- *Function coverage:* 70.7%, of 23,870 (functions).
- *Branch coverage:* 24.6%, (retrieved from: *Gcov*).
Demonstration of *test similarity analysis*, with 3 tests:

- **A**: IPForwarding#testCliRejectsInvalidAddressOnDstMo,
- **B**: IPForwarding#testCliRejectsInvalidAddressOnNexthopMo,
- **C**: PL1#testPL1TestSuite, *all three from Ericsson’s tests*.

Both **A** and **B** should exercise very similar code locations.

While **C** exercises more varied locations, different from **A**.

*Similarity* leads to *potential for test redundancy* [CNM07].

*Note!: exercising similar locations ⇨ exactly same tests!*
<table>
<thead>
<tr>
<th>Criterion</th>
<th>$d_H(A, B)$</th>
<th>$A \cap B$</th>
<th>$A \cup B$</th>
<th>$J(A, B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>0</td>
<td>400</td>
<td>400</td>
<td>1.00000</td>
</tr>
<tr>
<td>Function</td>
<td>0</td>
<td>90</td>
<td>90</td>
<td>1.00000</td>
</tr>
<tr>
<td>Branch</td>
<td>0</td>
<td>132</td>
<td>132</td>
<td>1.00000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Criterion</th>
<th>$d_H(A, C)$</th>
<th>$A \cap C$</th>
<th>$A \cup C$</th>
<th>$J(A, C)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>21691</td>
<td>398</td>
<td>22089</td>
<td>0.01801</td>
</tr>
<tr>
<td>Function</td>
<td>5369</td>
<td>90</td>
<td>5459</td>
<td>0.01648</td>
</tr>
<tr>
<td>Branch</td>
<td>21960</td>
<td>123</td>
<td>22083</td>
<td>0.00557</td>
</tr>
</tbody>
</table>
Conclusions

- **Feasibility**: deemed possible, since tests aren’t unstable; however, it increases softw. execution time significantly.

- **Measurements**: project has similar coverage to Google’s average project (C) coverage (statement) measurement.

- **Interpretation**: analysis tool/method, show locations of: test case similarity, and pot. test redundancy, [CNM07].

- **Limitation**: still requires engineers to verify redundancy.

- **Future Work**: clustering, test selection & prioritization.
In a nutshell...

- **Software testing** is hard; test coverage give us valuable metadata about tests.
- Gathering & analyzing code coverages on large-scale systems proves difficult.
- Integration of the system was feasible, but prog. execution time was affected.
- Metrics of a real, large-scale test suite were given since such data was scarce.
- Finally, analysis tool shows similarities between test cases, which sifts results.

---

3 Thesis Defense: https://www.xkcd.com/1403/
Bibliography

