Trends in Very Large Scale Continuous Integration

By Dr. Tim A. D. Henderson
Build, Test, Release
Google LLC
tadh@google.com
hackthology.com
linkedin.com/in/tadh/
Agenda

- What is Continuous Integration
- The Goals of Continuous Integration
- What is "Very Large Scale"
- Challenges Posed by Large Scale CI
- Solutions Space
- Trends
What is Continuous Integration?
A system and practice of automatically merging changes into a source of truth for your organization's source code and related artifacts.
Merging Changes

Change A

@@ -133,18 +133,18 @@ func (g *gen) fnStmt

```go
for idx, param := range fn.params {
-   r := g.newRegister()
+   r := g.newRegister(param.ptype)
   g.symbols.Put(param.name, r)
   entry.add(&Instruction{
+      Op:     ops.PRM,
+      A:      &Constant{value: idx, ctype: param.ptype},
+      Result: r})
}
```
Merging Changes

Change B

```go
@@ -160,18 +160,22 @@ func (g *gen) createClosure

    if inner, is := operand.(*Closure); is {
        operand, blk = g.createClosure(blk, inner)
        rewrite[inner.fn.String()] = &ClosureRegister{
-            id: len(registers),
+            id: len(registers),
+            rtype: operand.Type(),
        }
    }
```
Existing versions in your source of truth (main branch)
The new changes are merged using a text based merge

```plaintext
for idx, param := range fn.params {
    r := g.newRegister()
+   r := g.newRegister(param.ptype)
    g.symbols.Put(param.name, r)
    entry.add(&Instruction{
      Op: ops.PRM,
      A: &Constant{value: idx, ctype: param.ptype},
      Result: r})
}
```
These changes are linearized into a specific order
Just merging isn't enough, we need to run tests.

<table>
<thead>
<tr>
<th>Builds &amp; Tests</th>
<th>3</th>
<th>F</th>
<th>5</th>
<th>W</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₀</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>F</td>
<td>F</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>T₁</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>T₂</td>
<td>P</td>
<td>P</td>
<td>F</td>
<td>F</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>T₃</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
</tbody>
</table>
Tests should be run both before the merge and after.
What is Continuous Integration?

A system and practice of automatically merging changes into a source of truth for your organization's source code and related artifacts.

1. Automatically integrates new changes into the main branch in your source code management system.

2. Runs builds and tests to ensure the code still compiles and the tests pass.

3. May include additional functionality.
Goals of Continuous Integration
Goals of Continuous Integration

1. Happy Developers!
   a. Main branch is not constantly broken
   b. Provides quick "dev loop" feedback
   c. Provides tools for managing debugging and fixing breakages
   d. Handle flaky tests

2. Trustworthy Releases
   a. Runs all tests which could affect the result for a release
   b. Provides results on-time at team defined frequency (ex. 1 per hour, 4 per day, 2 per week)
   c. Ensures releases can be made reliably
   d. Handle flaky tests
What is "Very Large Scale"?
Prerequisite for Very Large Scale:

All code being integrated is integrated into the same branch in the same repository. Code which integrates into different branches or different repositories can be efficiently sharded into separate CI instances.
Very Large Scale

1. Supports commit submission rates **exceeding** the minimum time it takes to run a single build or test on the code base.

2. Multiple **resource management** techniques have been applied to manage resource demand.

3. Scale continues to grow and it remains an **ongoing** organizational priority to manage.
Scaling Factors

- Size of code base (# of lines)
- # of tests
- # of test configurations
- # of test environments (server, web, iOS, android, etc…)
- Frequency of commits
- Frequency of releases
- # of developers (users)
- # of distinct "projects" or "release artifacts"
- # of flaky (non-deterministic) tests
- # of flaky machines
- Complexity of test environment: hermetic unit test \(\Rightarrow\) multi-machine and platform end-to-end system tests.
Solutions Space for Scaling Scenarios
Solutions Space for Scaling Scenarios

1. Limit the number of commits which get tested.

2. Limit the number of tests which get run.
Research and Industrial Trends
Understand which tests need to be run.

Use static analysis coarse (build dependencies) ⇒ fine (program dependence graph) grained to determine which tests are affected by a change.


Understand which tests need to be run.

Only buzz_client_tests are run and only Buzz project needs to be updated.

Not every test needs to be run every change

Changes

Builds & Tests

T₀  T₁  T₂  T₃

3  F  5  W  C  A  B

P  P  P  F  F  F  P

F  F  F  F  P  P  P

P  P  P  F  F  P  P

P  P  P  P  P  P  P

means test was affected by the change
Huge savings to be had by skipping unaffected tests

- **Builds & Tests**
  - $T_0$
  - $T_1$
  - $T_2$
  - $T_3$

- **Changes**
  - 3
  - F
  - 5
  - W
  - C
  - A
  - B

- **Test Status**
  - Green square: Pass
  - Red square: Fail
  - Yellow star: Test affected by change

- Explanation:
  - $T_0$: Pass
  - $T_1$: Pass
  - $T_2$: Fail
  - $T_3$: Pass

- Note: Yellow stars indicate tests affected by changes.
Throttling test executions to prevent delays

In order to manage the finite resource of build and test execution machines, do not run tests at every commit. Wait until the execution system will have resources and then schedule (enqueue) all tests which need to be run. Execution system should prioritize latency sensitive builds and tests.

John Micco and Developer Infrastructure. "Continuous integration at google scale." Eclipse Con 2016. [slides] [Google]

Throttling test executions to prevent delays

<table>
<thead>
<tr>
<th>Builds &amp; Tests</th>
<th>3</th>
<th>F</th>
<th>5</th>
<th>W</th>
<th>C</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₀</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₁</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₃</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Changes
Throttling test execution: requires culprit finding

<table>
<thead>
<tr>
<th>Builds &amp; Tests</th>
<th>3</th>
<th>F</th>
<th>5</th>
<th>W</th>
<th>C</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_1$</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>P</td>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_2$</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>$T_3$</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Changes
In practice: parallel binary search is often used

Culprit CL is W for $T_0$

Fixing CL is W for $T_1$
But, watch out for flaky tests!

Builds & Tests

<table>
<thead>
<tr>
<th>T₀</th>
<th>P</th>
<th>F</th>
<th>P</th>
<th>F</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>T₂</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>T₃</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>P</td>
</tr>
</tbody>
</table>

Wait! This test wasn't broken it was just flaky!!
Combining dependency based selection and throttling

Builds & Tests

<table>
<thead>
<tr>
<th>T₀</th>
<th>P</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>T₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Changes

1. 3
2. F
3. 5
4. W
5. C
6. A
7. B

Test was affected by the change
Dependency information speeds up culprit finding

Builds & Tests

T₀
P
P

T₁
F
P

T₂

T₃
P

3 F 5 W C A B

Changes

test was affected by the change
Predicting which tests are most likely to fail.

Academic work includes a broad categories of techniques from precise static analysis to coarse grained heuristics + machine learning. Industrial implementations tend to use heuristics as analysis based approaches is challenging. **Must account for flakiness.**


Combining dependency based selection and throttling

Builds & Tests

<table>
<thead>
<tr>
<th>T₀</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>F</td>
<td></td>
<td>P</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Changes

3 → F → 5 → W → C → A → B

test was affected by the change
Adding in Predictive Selection

Builds & Tests

- $T_0$
- $T_1$
- $T_2$
- $T_3$

Changes

- 3
- F
- 5
- W
- C
- A
- B

- test was affected by the change
- test was skipped as it was predicted to pass

Changes from $T_0$ to $T_3$:

- $T_0$: S
- $T_1$: F
- $T_2$: P
- $T_3$: S
Predicting which tests are most likely to fail.

Manage the flaky tests

Flaky tests are tests with non-deterministic outcomes. These must be managed at every stage in a CI system. They should be automatically identified and triaged, potentially excluded from release gating, and surfaced to developers with tooling support to identify the root causes of the nondeterminicity.


Detecting flakes with re-runs of failures and known flaky tests

Builds & Tests

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T₀</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₁</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₃</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Changes

- test was affected by the change
- test was skipped as it was predicted to pass
- test was run multiple times to deflake
- test is flaky (aka. nondeterministic)
Manage the flaky tests

Flaky tests are a significant source of transitions at Google: in our data over 80% of observed transitions were caused by confirmed flaky results.

Fig. 1. Performance of the transition count algorithm (Section VI-B), on data with and without flaky executions.

Fig. 2. The sources of transitions in raw test data.

Managing demand through economics

It is important to ensure the end user is aware of the costs of their usage of CI. In large integrated organizations using a monorepo this can be challenging. One approach is to force each product to internally pay for their expected usage and then throttle them based on how many resources they actually bought. This will encourage developers to optimize their tests.

Predefined limits throttle the execution of certain tests

Builds & Tests

<table>
<thead>
<tr>
<th></th>
<th>T₀</th>
<th></th>
<th>T₁</th>
<th></th>
<th>T₂</th>
<th></th>
<th>T₃</th>
</tr>
</thead>
</table>

Changes

- 3  | F  | 5  | W  | C  | A  | B  |

- test was affected by the change
- test was skipped as it was predicted to pass
- test was run multiple times to deflake
- test is flaky (aka. nondeterministic)
- test was throttled due to predefined limits.
Understanding what CI model works best

There are many different CI models. Some organizations enforce correctness by serializing the merge and test operations to prevent bad commits from being integrated. Others allow a small percentage of “collisions” and then apply culprit finding and automatic rollback. There are many open questions, here are three:

1. At what stages (pre-merge, post-merge, release) do the various options for test selection work most effectively?

2. What merge-gating techniques are most cost effective?

3. Are there ways to estimate the economic cost of test failure and prioritize tests with higher costs — not just tests which are predicted to be failure prone?
Complexity

As CI systems grow in features and smart capabilities the complexity of the system needs to be actively managed. Some techniques the system used a few years ago may be moderately effective but be too costly in either resources or CI developer time to scale with the growing demands of your organization.

Trade-offs between resources spent on testing and CI infrastructure have to be made against implementation, scaling, and maintenance costs of better and smarter algorithms.
Questions?

mail me: tadh@google.com
visit me: hackthology.com
Citations


John Micco and Developer Infrastructure. "**Continuous integration at google scale.**" Eclipse Con 2016. [slides] [Google]

A. Memon et al., “**Taming Google-scale continuous testing,**” International Conference on Software Engineering: Software Engineering in Practice Track ICSE-SEIP, 2017. https://dx.doi.org/10.1109/ICSE-SEIP.2017.16.