Enjoy Fighting Regressions with git bisect

Christian Couder, Murex
chriscool@tuxfamily.org

LinuxCon Europe
October 23, 2013

About Git

A Distributed Version Control System (DVCS):

- created by Linus Torvalds
- maintained by Junio Hamano
- since 2005
- used by Linux Kernel, Android, Qt, Chrome OS, Perl, Ruby on Rails, ...

Usage statistics from user surveys

% of people using git bisect:

2011 (11498 people): 30%

2010 (8841 people): 28%

2009 (3868 people): 49%, 6% often
using "bisect run": 25%, 3% often

2008 (3236 people): 26%, 2% often
using "bisect run": 10%, 0.4% often

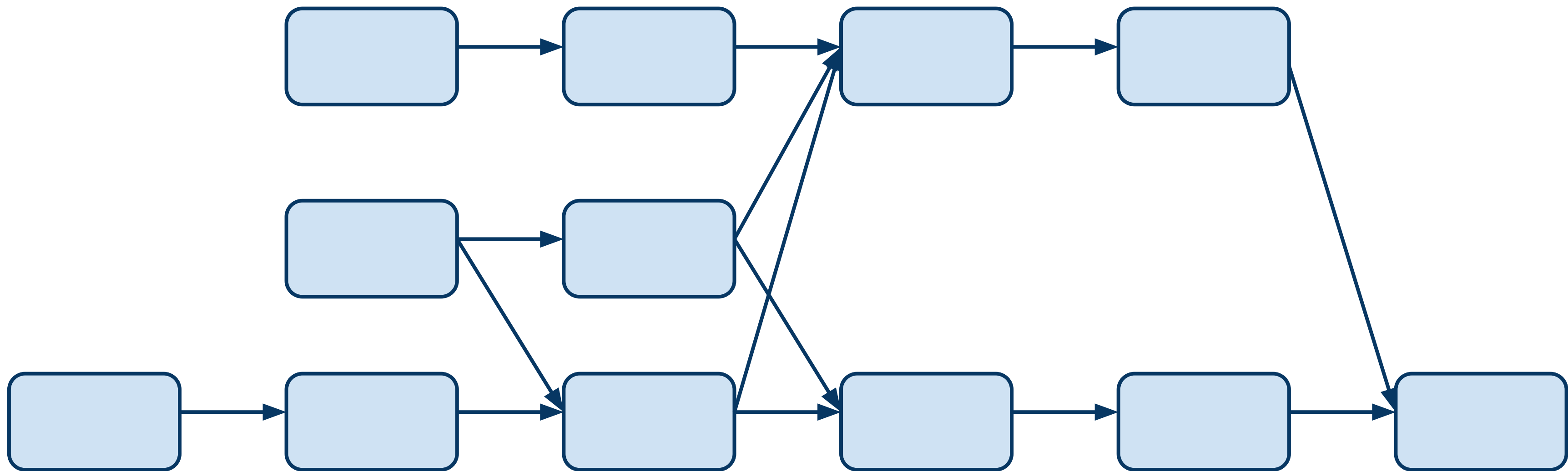
Distributed

- users have a full repository locally on their machine
- repository data is tightly compressed
- many operations don't need network access
- many very fast operations
- especially very fast checkout

Version Control Basics

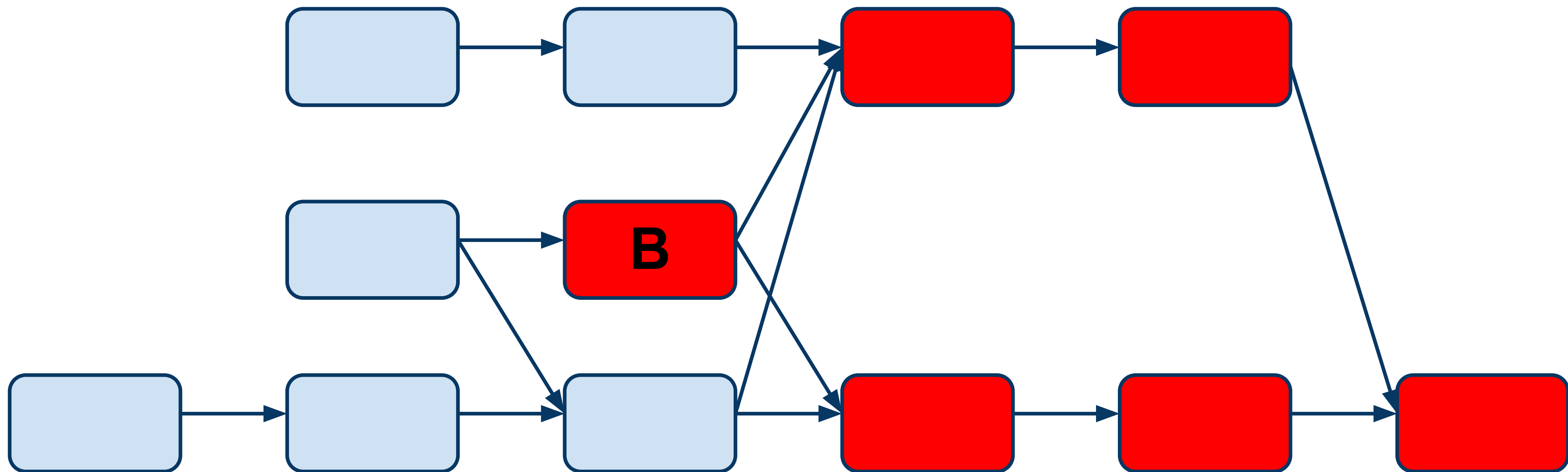
- commits are states of the managed data
- managed data is software source code
- so each commit corresponds to a software behavior
- commits are related like parents and children

Commits in Git form a DAG (Directed Acyclic Graph)



- DAG direction is from left to right
- older commits point to newer commits

First Bad Commit



- B introduces a bad behavior called "bug" or "regression"
- B is called a "first bad commit"
- red commits are called "bad"
- blue commits are called "good"

"git bisect"

Idea:

- help find a first bad commit
- use a binary search algorithm for efficiency if possible

Benefits:

- manually verifying the source code changes from only one commit is relatively easy
- the commit gives extra information: commit message, author, ...

Linux kernel example

Regression is an important problem because:

- big code base growing fast
- many different developers
- developed and maintained for many years
- many users depending on it

Development process:

- 2 weeks "merge window"
- 7 or 8 "rc" releases to fix bugs, especially regressions, around 1 week apart
- release 3.X
- stable releases 3.X.Y and distribution maintenance

Ingo Molnar about his "git bisect" use

I most actively use it during the merge window (when a lot of trees get merged upstream and when the influx of bugs is the highest) - and yes, there have been cases that i used it multiple times a day. My average is roughly once a day.

=> regressions are fought all the time

Indeed it is well known that is is more efficient (and less costly) to fix bugs as soon as possible.

=> good tools are needed by all the developers

Tools to fight regressions

- the same tools as for regular bugs (debuggers, tracing tools, ...)
- test suites
- tools similar as git bisect

Test suites

Very useful

- to prevent regressions,
- to ensure an amount of functionality and testability.

But not enough

- to find the first bad commit when a regression is discovered,
- to fight the combinatorial explosion of the number of tests to be performed.

Combinatorial explosion

Big software often

- has many different "configurations"
- may fail under one configuration but not another

N configurations, T tests and C new commits means that a release needs:

C * N * T tests performed

where N and T, at least, are growing with the size of the software

Supplementing test suites

Test suites need:

- a tool to efficiently find a first bad commit,
- a strategy to fight combinatorial explosion.

Starting a bisection and bounding it

2 ways to do it:

$ git bisect start
$ git bisect bad [<BAD>]
$ git bisect good [<GOOD>...]

or

$ git bisect start <BAD> <GOOD> [<GOOD>...]

where <BAD> and <GOOD> can be resolved to commits

Starting example

(toy example with the linux kernel)

$ git bisect start v2.6.27 v2.6.25
Bisecting: 10928 revisions left to test after this (roughly 14 steps)
[2ec65f8b89ea003c27ff7723525a2ee335a2b393] x86: clean up using max_low_pfn on 32-bit
$

=> the commit you should test has been checked out

Driving a bisection manually

1. test the current commit
2. tell "git bisect" whether it is good or bad, for example:

$ git bisect bad
Bisecting: 5480 revisions left to test after this (roughly 13 steps)
[66c0b394f08fd89236515c1c84485ea712a157be] KVM: kill file->f_count abuse in kvm

repeat step 1. and 2. until the first bad commit is found...

First bad commit found

```
$ git bisect bad
2ddcca36c8bcfa251724fe342c8327451988be0d is the first bad commit
commit 2ddcca36c8bcfa251724fe342c8327451988be0d
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date: Sat May 3 11:59:44 2008 -0700

Linux 2.6.26-rc1

:100644 100644 5cf8258195331a4dbdddff08b8d68642638eea57
4492984efc09ab72ff6219a7bc21fb6a957c4cd5 M Makefile
```

End of bisection

When the first bad commit is found:

- you can check it out and tinker with it, or
- you can use "git bisect reset", like that:

$ git bisect reset
Checking out files: 100% (21549/21549), done.
Previous HEAD position was 2ddcca3... Linux 2.6.26-rc1
Switched to branch 'master'

to go back to the branch you were in before you started bisecting

Driving a bisection automatically

At each bisection step a script or command will be launched to tell if the current commit is good or bad.

Syntax:

$ git bisect run COMMAND [ARG...]

Example to bisect a broken build:

$ git bisect run make

Automatic bisect example part 1

$ git bisect start v2.6.27 v2.6.25
Bisecting: 10928 revisions left to test after this (roughly 14 steps)
[2ec65f8b89ea003c27ff7723525a2ee335a2b393] x86: clean up using max_low_pfn on 32-bit
$
$ git bisect run grep '^SUBLEVEL = 25' Makefile
running grep ^SUBLEVEL = 25 Makefile
Bisecting: 5480 revisions left to test after this (roughly 13 steps)
[66c0b394f08fd89236515c1c84485ea712a157be] KVM: kill file->f_count abuse in kvm
running grep ^SUBLEVEL = 25 Makefile

Automatic bisect example part 2

SUBLEVEL = 25
Bisecting: 2740 revisions left to test after this (roughly 12 steps)
[671294719628f1671faefd4882764886f8ad08cb] V4L/DVB(7879): Adding cx18 Support for mxl5005s

...

...

running grep ^SUBLEVEL = 25 Makefile
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[2ddcca36c8bcfa251724fe342c8327451988be0d] Linux 2.6.26-rc1
running grep ^SUBLEVEL = 25 Makefile

Automatic bisect example part 3

2ddcca36c8bcfa251724fe342c8327451988be0d is the first bad commit
commit 2ddcca36c8bcfa251724fe342c8327451988be0d
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date: Sat May 3 11:59:44 2008 -0700

Linux 2.6.26-rc1

:100644 100644 5cf8258195331a4dbdddff08b8d68642638eea57 4492984efc09ab72ff6219a7bc21fb6a957c4cd5 M Makefile
bisect run success

Run script exit codes

0 => good
1-124 and 126-127 => bad
125 => skip
128-255 => stop

"skip": mark commit as "untestable",
"git bisect" will choose another commit to be tested

"stop": bisection is stopped immediately,
useful to abort bisection in abnormal situations

Untestable commits

Manual bisection choice:

- "git bisect visualize/view": gitk or "git log" to help you find a better commit to test
- "git bisect skip"

Possible situation with skipped commits

Possible end of bisection

There are only 'skip'ped commits left to test.
The first bad commit could be any of:
15722f2fa328eaba97022898a305ffc8172db6b1
78e86cf3e850bd755bb71831f42e200626fbd1e0
e15b73ad3db9b48d7d1ade32f8cd23a751fe0ace
070eab2303024706f2924822bfec8b9847e4ac1b
We cannot bisect more!

Saving a log and replaying it

- Saving:

$ git bisect log > bisect_log.txt

- Replaying:

$ git bisect replay bisect_log.txt

Bisection algorithm

It gives the commit that will be tested.

So the goal is to find the best bisection commit.

The algorithm used

- is "truly stupid" (Linus Torvalds)
- but works well

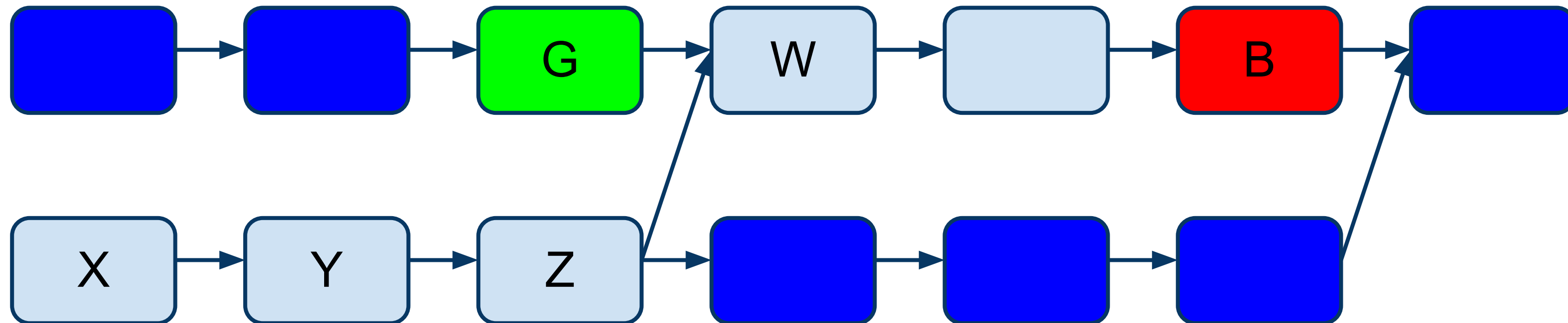
Bisection algorithm, remarks

- the algorithm is not symmetric
- it uses only one current bad commit and many good commits

2 rules to bound the graph:

- only ancestors of the bad commit are kept
- ancestors of the good commits are removed

Bisection algorithm, pitfalls



Commits X, Y and Z are not removed from the graph we are bisecting on.

So you may have to test kernels with version 2.6.25 even if you are bisecting between v2.6.26 and v2.6.27!

Or you may be on a branch with only the Btrfs driver code!

Skip algorithm

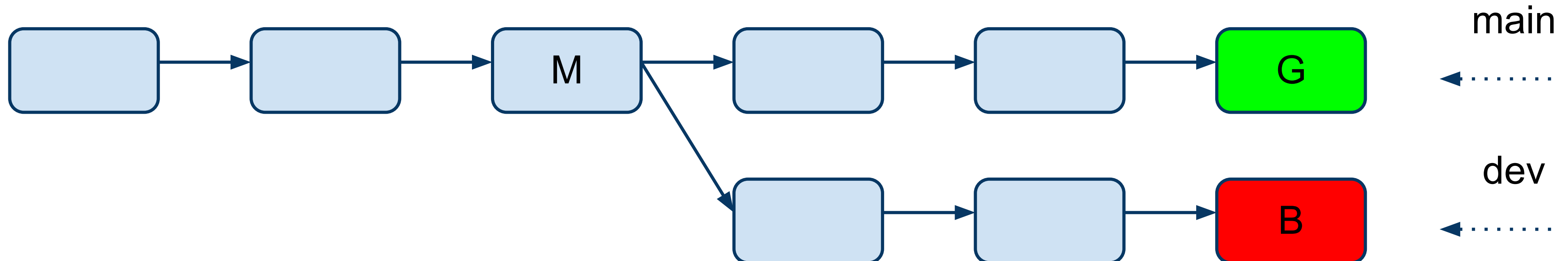
To choose another commit to test, we:

- use a pseudo random number generator
- but favor commits near the best bisection commit
- thus avoid being stuck in areas where there are many untestable commits

Checking merge bases

It is not a requirement that good commits be ancestors of the bad commit.

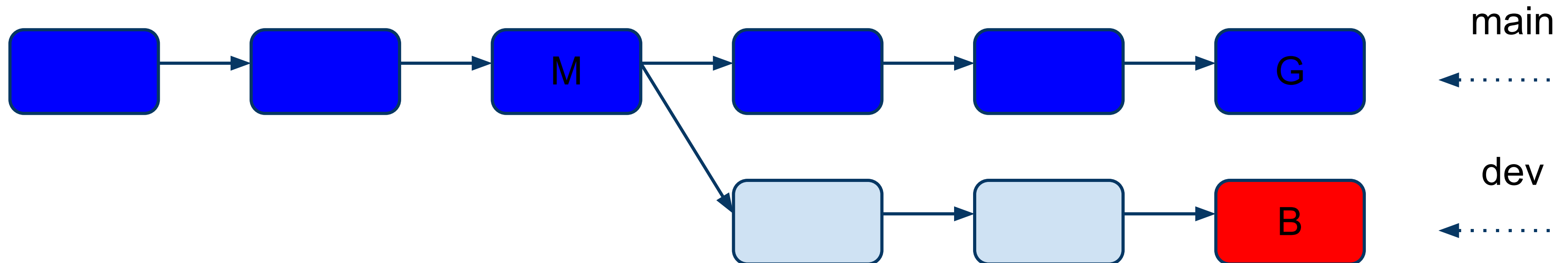
For example:



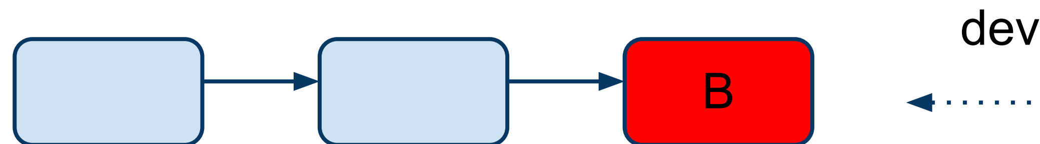
M is a "merge base" for branches "main" and "dev"

Checking merge bases

If we apply the bisection algorithm, we must remove all ancestors of the good commits.

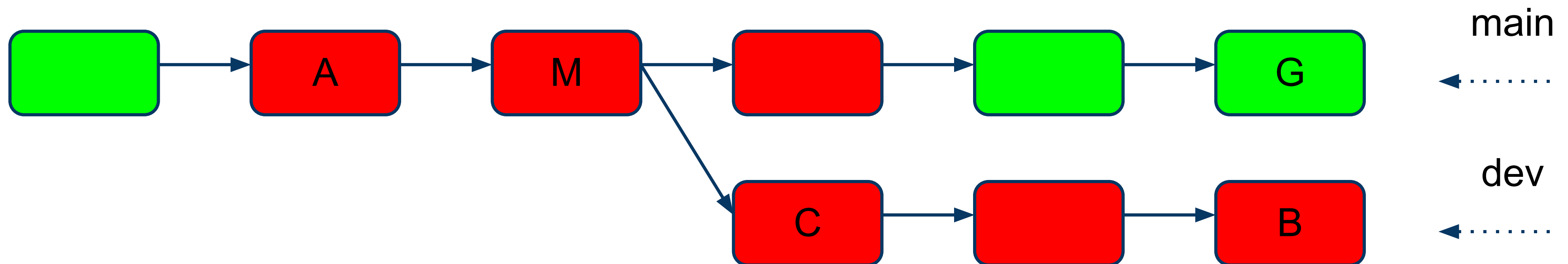


So we get only:



Checking merge bases

But what if the bug was fixed in the "main" branch?



We would find C as the first bad commit instead of A, so we would be wrong!

Checking merge bases

Solution:

- compute merge bases if a good commit is not ancestor of the bad commit
- ask the user to test merge bases first
- if a merge base is not good, stop!

For example:

The merge base BBBBBB is bad.
This means the bug has been fixed between BBBBBB and [GGGGGG,...].

"git bisect run" tips

- Bisect broken builds:

$ git bisect run make

- Bisect broken test suite:

$ git bisect run make test

- Bisect run with "sh -c":

$ git bisect run sh -c "make || exit 125; ./my_app | grep 'good output'"

Bisecting performance regressions

```
#!/bin/sh

make my_app || exit 255 # Stop if build fails
./my_app >log 2>&1 &    # Launch app in background
pid=$!                  # Grab its process ID
sleep $NORMAL_TIME      # Wait for sufficiently long

if kill -0 $pid         # See if app is still there
then        # Still running, that is bad.
   kill $pid; sleep 1; kill $pid; exit 1
else        # Already finished, we are happy.
   exit 0
fi
```

Using complex scripts

It can be worth it. For example Ingo Molnar wrote:

i have a fully automated bootup-hang bisection script. It is based on "git-bisect run". I run the script, it builds and boots kernels fully automatically, and when the bootup fails (the script notices that via the serial log, which it continuously watches - or via a timeout, if the system does not come up within 10 minutes it's a "bad" kernel), the script raises my attention via a beep and i power cycle the test box. (yeah, i should make use of a managed power outlet to 100% automate it)

General best practices

These practices are useful without "git bisect", but they are even more useful when using "git bisect":

- no commits that break things, even if other commits later fix the breakage,
- only one small logical change in each commit,
- small commits for easy review,
- good commits messages.

No bug prone merge

Merges by themselves can introduce regressions when there is no conflict to resolve.

Example:

- semantic of a function change in one branch,
- a call to the function is added in another branch.

This is made much worse when there are many changes, either needed to fix conflicts or for any other reason, in a merge. When changes are not related to the branches, they are called "evil merges" and should be avoided.

No bug prone merge 2

So what can be done:

- "git rebase" can linearize history, instead of merging, or to bisect a merge
- use shorter branches, or many topic branches
- use integration branches, like "linux-next", to prepare merges and to test

Fighting Combinatorial Explosion

C commits, T test cases, N configurations

We decide to test

- each commit only on the most common n configurations,
- and each of the N configurations only a few c times,
- and then to bisect all bugs found for the N configurations.

n * C * T : tests for the most common n configurations
c * N * T : tests for all the configurations but only c commits
b * C * log2(C) : bisection tests for the bugs found

where b is the number of bug per commits.

Test suites and git bisect

test suites:

- make it easy to write new test case to bisect
- maintain testability of the commit history

=> make bisecting easier and more efficient

git bisect:

- make it worthwhile to develop new test cases
- help overcome combinatorial explosion

=> make test suites easier to grow and more efficient

Virtuous cycle

you have a test suite:
=> why not improve its usefulness and efficiency by using git bisect?

you bisect:
=> why not save your test cases in a test suite?

Adapting your work-flow

Test suites and "git bisect" are very powerful and efficient when used together.

For example, work-flow used by Andreas Ericsson:

- write, in the test suite, a test to catch a regression
- use "git bisect run" to find the first bad commit
- fix the bug
- commit both the fix and the test script (and if needed more tests)

Adapting your work-flow 2

Results reported by Andreas from using Git and adopting this work-flow after one year:

- report-to-fix cycle went from 142.6 hours (wall-clock time) to 16.2 hours [-88%],
- each new release results in ~40% fewer bugs ("*almost certainly due to how we now feel about writing tests*").

Adapting your work-flow 3

What changed?

- tests are immediately useful
- it's often easy and straightforward
- it gives developers a break
- it's much faster
- the work done to find the bug is capitalized

This means:

- joy, happiness
- productivity

Adapting your work-flow 4

Like the work-flow used by Andreas, a good work-flow should be designed around:

- using general best practices (small logical commits, good commit messages, topic branches, no evil merge, ...),
- taking advantage of the virtuous cycle between a test suite and "git bisect".

Involving non developers

No need to be a developer to use "git bisect".

During heated discussions on linux-kernel mailing list around April 2008, David Miller wrote:

What people don't get is that this is a situation where the "end node principle" applies. When you have limited resources (here: developers) you don't push the bulk of the burden upon them. Instead you push things out to the resource you have a lot of, the end nodes (here: users), so that the situation actually scales.

Involving non developers 2

Reasons:

- It can be "cheaper" if QA people or end users can do it.
- People reporting a bug have access to the environment where the bug happens, and "git bisect" automatically extract relevant information from this environment.
- For open source project, this is a good way to get new contributors.

Plugging in other tools

Test suites and "git bisect" can be combined with more tools.

For example after "git bisect", it's possible to automatically:

- send an email to the people involved in the first bad commit, and/or
- create an entry in the bug tracking system.

gitbuilder by Avery Pennarun

Automatically build, test and bisect (if needed) all the branches in a repository

- web interface
- based on git bisect (with a few differences)

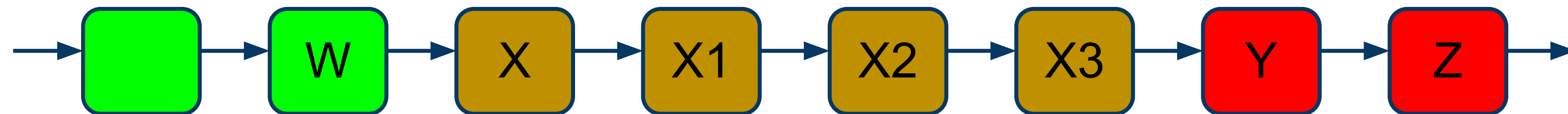
As a buffer between development and mainline it can help:

- get 0 unit test failures in mainline
- decide which branches are ready to be merged
- know what is going on in development branches

"git replace"

Sometimes the first bad commit will be in an untestable area of the graph.

For example:

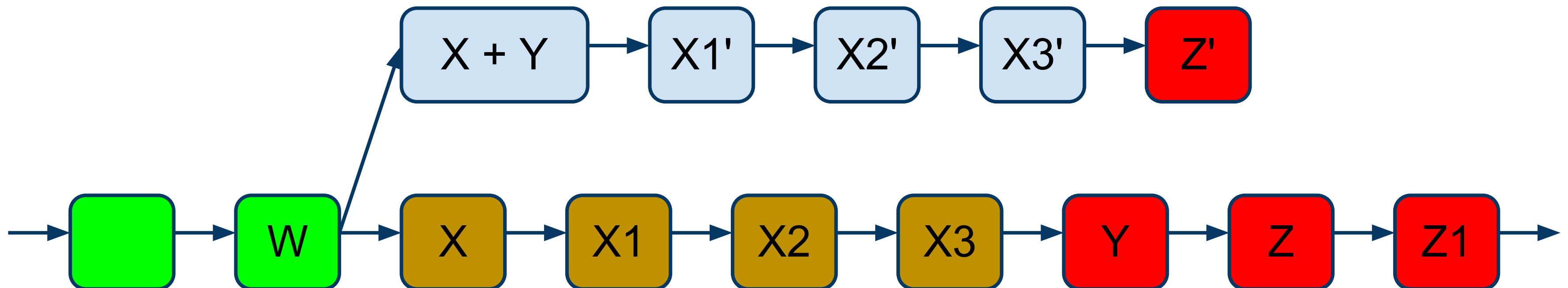


Commit X introduced a breakage, later fixed by commit Y.

"git replace" 2

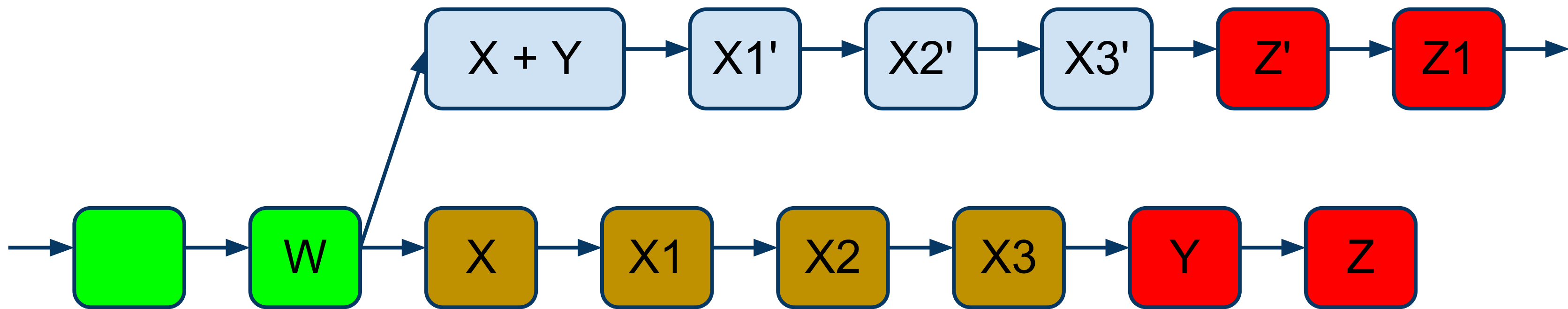
Possible solutions to bisect anyway:

- apply a patch before testing and remove it afterwards (can be done using "git cherry-pick"), or
- create a fixed up branch (can be done with "git rebase -i"), for example:



"git replace" 3

The idea is that we will replace Z with Z' so that we bisect from the beginning using the fixed up branch.



$ git replace Z Z'

Sporadic bugs

Some bugs can depend on the compiler output and small changes unrelated to the bug can make it appear or disappear.

So "git bisect" is currently very unreliable to fight sporadic bugs.

The idea is to optionally add redundant tests when bisecting, for example 1 test out of 3 could be redundant. And if a redundant test fails, we hopefully will abort early.

There is an independent project called BBChop doing something like that based on Bayesian Search Theory.

Some Conclusions

- "git bisect" nicely complements test suites and general best practices,
- it may be worth it to adopt a special work-flow,
- "git bisect" already works very well, is used a lot and is very useful.

Another Conclusion

Ingo Molnar when asked how much time it saves him:

a _lot_.

About ten years ago did i do my first 'bisection' of a Linux patch queue. That was prior the Git (and even prior the BitKeeper) days. I literally [spent days] sorting out patches, creating what in essence were standalone commits that i guessed to be related to that bug.

Another Conclusion 2

Ingo Molnar (continued):

It was a tool of absolute last resort. I'd rather spend days looking at printk output than do a manual 'patch bisection'.

With Git bisect it's a breeze: in the best case i can get a ~15 step kernel bisection done in 20-30 minutes, in an automated way. Even with manual help or when bisecting multiple, overlapping bugs, it's rarely more than an hour.

Another Conclusion 3

Ingo Molnar (continued):

In fact it's invaluable because there are bugs i would never even _try_ to debug if it wasn't for git bisect. In the past there were bug patterns that were immediately hopeless for me to debug - at best i could send the crash/bug signature to lkml and hope that someone else can think of something.

Another Conclusion 4

Ingo Molnar (continued):

And even if a bisection fails today it tells us something valuable about the bug: that it's non-deterministic - timing or kernel image layout dependent.

So git bisect is unconditional goodness - and feel free to quote that.

Many thanks to:

- Junio Hamano (comments, help, discussions, reviews, improvements),
- Ingo Molnar (comments, suggestions, evangelizing),
- Linus Torvalds (inventing, developing, evangelizing),
- many other great people in the Git and Linux communities, especially: Andreas Ericsson, Johannes Schindelin, H. Peter Anvin, Daniel Barkalow, Bill Lear, John Hawley, ...
- LinuxCon organizers and attendants,
- Murex the company I am working for.

Questions ?