The Distributed Version Control Revolution

Charles Bailey, Bloomberg LP
@hashpling

What is a version control system?

- A tool to manage manually authored artifacts created in the production of software

What does source control provide?

- Reversibility
- Concurrency
- Annotation

SCCS

- Development started in 1972
- Originally implemented in SNOBOL4
- Still under development

In the second step, the programmer “marks up” the file named “modx.a” generated by the get command with the UNIX editor, an interactive context editor styled after QED. **If necessary, this step may span several days or even weeks**, and may involve several editor sessions and compilations.

Mark J Rochkind, The Source Code Control System, IEEE Transactions of Software Engineering, December 1975

RCS

- Very similar to SCCS
- Uses reverse deltas instead of forward deltas[citation needed]
- Open-sourced much earlier than SCCS (1990 vs 2006)

Revision numbers

- Revisions on “trunk”: 1.328
- Branching: 1.94.1.23
- Can assign “symbolic names” with revisions

CVS

- Module based, groups of files are versioned together
- Locking is optional
- Programmers can work on different machines

CVS

- RCS under the hood
- Files are versioned individually
- Merging requires discipline and care

Subversion

- [In]famously: CVS done right
- Branches and tags are just copies
- Merge tracking since 1.5

Reasons like “Trouble Report 5576: change SUM header” are what one like to see. Sometimes, unfortunately, one sees instead things like “Another bug” or “Tried again.”

Mark J Rochkind, The Source Code Control System, IEEE Transactions of Software Engineering, December 1975

```
tree a30c33a8f1971ea2a8375481747d32e7d6c46555
parent 61f76a3612db199a9eb9090c2605d8fc35ffc41c
author Kirill Smelkov <kirr@mns.spb.ru> 1396820786 +0400
committer Junio C Hamano <gitster@pobox.com> 1396906846 -0700
```

tree-diff: rework diff_tree() to generate diffs for multiparent cases as well

Previously diff_tree(), which is now named ll_diff_tree_sha1(), was
generating diff_filepair(s) for two trees t1 and t2, and that was
usually used for a commit as t1=HEAD~, and t2=HEAD - i.e. to see changes
a commit introduces.

In Git, however, we have fundamentally built flexibility in that a
commit can have many parents - 1 for a plain commit, 2 for a simple merge,
but also more than 2 for merging several heads at once.

For merges there is a so called combine-diff, which shows diff, a merge
introduces by itself, omitting changes done by any parent. That works
through first finding paths, that are different to all parents, and then
showing generalized diff, with separate columns for +/- for each parent.
The code lives in combine-diff.c .

There is an impedance mismatch, however, in that a commit could
generally have any number of parents, and that while diffing trees, we
divide cases for 2-tree diffs and more-than-2-tree diffs. I mean there
is no special casing for multiple parents commits in e.g.
revision-walker .

That impedance mismatch *hurts* *performance* *badly* for generating
combined diffs - in "combine-diff: optimize combine_diff_path
sets intersection" I've already removed some slowness from it, but from
the timings provided there, it could be seen, that combined diffs still
cost more than an order of magnitude more cpu time, compared to diff for
usual commits, and that would only be an optimistic estimate, if we take
into account that for e.g. linux.git there is only one merge for several
dozens of plain commits.

That slowness comes from the fact that currently, while generating
combined diff, a lot of time is spent computing diff(commit,commit^2)
just to only then intersect that huge diff to almost small set of files
from diff(commit,commit^1).

That's because at present, to compute combine-diff, for first finding
paths, that "every parent touches", we use the following combine-diff
property/definition:

```
D(A,P1...Pn) = D(A,P1) ^ ... ^ D(A,Pn)      (w.r.t. paths)
```

where

D(A,P1...Pn) is combined diff between commit A, and parents Pi

and

D(A,Pi) is usual two-tree diff Pi..A

So if any of that D(A,Pi) is huge, tracting 1 n-parent combine-diff as n
1-parent diffs and intersecting results will be slow.

And usually, for linux.git and other topic-based workflows, that
D(A,P2) is huge, because, if merge-base of A and P2, is several dozens
of merges (from A, via first parent) below, that D(A,P2) will be diffing
sum of merges from several subsystems to 1 subsystem.

The solution is to avoid computing n 1-parent diffs, and to find
changed-to-all-parents paths via scanning A's and all Pi's trees
simultaneously, at each step comparing their entries, and based on that
comparison, populate paths result, and deduce we could *skip*
recursing into subdirectories, if at least for 1 parent, sha1 of that
dir tree is the same as in A. That would save us from doing significant
amount of needless work.

Such approach is very similar to what diff_tree() does, only there we
deal with scanning only 2 trees simultaneously, and for n+1 tree, the
logic is a bit more complex:

```
D(T,P1...Pn) calculation scheme
-------------------------------

D(T,P1...Pn) = D(T,P1) ^ ... ^ D(T,Pn)         (regarding resulting paths set)

    D(T,Pj)                     - diff between T..Pj
    D(T,P1...Pn)                - combined diff from T to parents P1,...,Pn
```

We start from all trees, which are sorted, and compare their entries in
lock-step:

```
     T     P1       Pn
     -     -        -
    |t|   |p1|     |pn|
    |-|   |--| ... |--|      imin = argmin(p1...pn)
    | |   | |     | |
    |-|   |--|     |--|
    |.|   |. |     |. |
     .     .        .
     .     .        .
```

at any time there could be 3 cases:

```
    1)  t < p[imin];
    2)  t > p[imin];
    3)  t = p[imin].
```

Schematic deduction of what every case means, and what to do, follows:

```
1)  t < p[imin]  ->  ∀j t ∉ Pj  ->  "+t" ∈ D(T,Pj)  ->  D += "+t";  t↓

2)  t > p[imin]

    2.1) ∃j: pj > p[imin]  ->  "-p[imin]" ∉ D(T,Pj)  ->  D += ø;  ∀ pi=p[imin]  pi↓

    2.2) ∀i  pi = p[imin]  ->  pi ∉ T  ->  "-pi" ∈ D(T,Pi)  ->  D += "-p[imin]";  ∀i
pi↓

3)  t = p[imin]

    3.1) ∃j: pj > p[imin]  ->  "+t" ∈ D(T,Pj)  ->  only pi=p[imin] remains to
investigate
    3.2) pi = p[imin]  ->  investigate δ(t,pi)
     |
     |
     v

    3.1+3.2) looking at δ(t,pi) ∀i: pi=p[imin] - if all != ø  ->

                        ⎧ δ(t,pi)  - if pi=p[imin]
                ->  D += ⎨
                        ⎩ "+t"     - if pi>p[imin]


    in any case t↓  ∀ pi=p[imin]  pi↓
```

~

For comparison, here is how diff_tree() works:

```
D(A,B) calculation scheme
-------------------------

    A     B
    -     -
   |a|   |b|    a < b   ->  a ∉ B   ->   D(A,B) +=  +a    a↓
   |-|   |-|    a > b   ->  b ∉ A   ->   D(A,B) +=  -b    b↓
   | |   | |    a = b   ->  investigate δ(a,b)            a↓ b↓
   |-|   |-|
   |.|   |.|
    .     .
    .     .
```

~~~~~~~~

This patch generalizes diff tree-walker to work with arbitrary number of
parents as described above - i.e. now there is a resulting tree t, and
some parents trees tp[i] i=[0..nparent). The generalization builds on
the fact that usual diff

```
D(A,B)
```

is by definition the same as combined diff

```
D(A,[B]),
```

so if we could rework the code for common case and make it be not slower
for nparent=1 case, usual diff(t1,t2) generation will not be slower, and
multiparent diff tree-walker would greatly benefit generating
combine-diff.

What we do is as follows:

1) diff tree-walker ll_diff_tree_sha1() is internally reworked to be
   a paths generator (new name diff_tree_paths()), with each generated path
   being `struct combine_diff_path` with info for path, new sha1,mode and for
   every parent which sha1,mode it was in it.

2) From that info, we can still generate usual diff queue with
   struct diff_filepairs, via "exporting" generated
   combine_diff_path, if we know we run for nparent=1 case.
   (see emit_diff() which is now named emit_diff_first_parent_only())

3) In order for diff_can_quit_early(), which checks

       DIFF_OPT_TST(opt, HAS_CHANGES))

   to work, that exporting have to be happening not in bulk, but
   incrementally, one diff path at a time.

   For such consumers, there is a new callback in diff_options
   introduced:

       ->pathchange(opt, struct combine_diff_path *)

   which, if set to !NULL, is called for every generated path.

   (see new compat ll_diff_tree_sha1() wrapper around new paths
    generator for setup)

4) The paths generation itself, is reworked from previous
   ll_diff_tree_sha1() code according to "D(A,P1...Pn) calculation
   scheme" provided above:

   On the start we allocate [nparent] arrays in place what was
   earlier just for one parent tree.

   then we just generalize loops, and comparison according to the
   algorithm.

Some notes(*):

1) alloca(), for small arrays, is used for "runs not slower for
   nparent=1 case than before" goal - if we change it to xmalloc()/free()
   the timings get ~1% worse. For alloca() we use just-introduced
   xalloca/xalloca_free compatibility wrappers, so it should not be a
   portability problem.

2) For every parent tree, we need to keep a tag, whether entry from that
   parent equals to entry from minimal parent. For performance reasons I'm
   keeping that tag in entry's mode field in unused bit - see S_IFXMIN_NEQ.
   Not doing so, we'd need to alloca another [nparent] array, which hurts
   performance.

3) For emitted paths, memory could be reused, if we know the path was
   processed via callback and will not be needed later. We use efficient
   hand-made realloc-style path_appendnew(), that saves us from ~1-1.5%
   of potential additional slowdown.

4) goto(s) are used in several places, as the code executes a little bit
   faster with lowered register pressure.

Also

- we should now check for FIND_COPIES_HARDER not only when two entries
  names are the same, and their hashes are equal, but also for a case,
  when a path was removed from some of all parents having it.

  The reason is, if we don't, that path won't be emitted at all (see
  "a > xi" case), and we'll just skip it, and FIND_COPIES_HARDER wants
  all paths - with diff or without - to be emitted, to be later analyzed
  for being copies sources.

  The new check is only necessary for nparent >1, as for nparent=1 case
  xmin_eqtotal always =1 =nparent, and a path is always added to diff as
  removal.

~~~~~~~~

Timings for

```
    # without -c, i.e. testing only nparent=1 case
    `git log --raw --no-abbrev --no-renames`
```

before and after the patch are as follows:

```
                navy.git        linux.git v3.10..v3.11

    before      0.611s          1.889s
    after       0.619s          1.907s
    slowdown    1.3%            0.9%
```

This timings show we did no harm to usual diff(tree1,tree2) generation.
From the table we can see that we actually did ~1% slowdown, but I think
I've "earned" that 1% in the previous patch ("tree-diff: reuse base
str(buf) memory on sub-tree recursion", HEAD~~) so for nparent=1 case,
net timings stays approximately the same.

The output also stayed the same.

(*) If we revert 1)-4) to more usual techniques, for nparent=1 case,
    we'll get ~2-2.5% of additional slowdown, which I've tried to avoid, as
   "do no harm for nparent=1 case" rule.

For linux.git, combined diff will run an order of magnitude faster and
appropriate timings will be provided in the next commit, as we'll be
taking advantage of the new diff tree-walker for combined-diff
generation there.

P.S. and combined diff is not some exotic/for-play-only stuff - for
example for a program I write to represent Git archives as readonly
filesystem, there is initial scan with

    `git log --reverse --raw --no-abbrev --no-renames -c`

to extract log of what was created/changed when, as a result building a
map

```
    {}  sha1    ->  in which commit (and date) a content was added
```

that `-c` means also show combined diff for merges, and without them, if
a merge is non-trivial (merges changes from two parents with both having
separate changes to a file), or an evil one, the map will not be full,
i.e. some valid sha1 would be absent from it.

That case was my initial motivation for combined diffs speedup.

Signed-off-by: Kirill Smelkov <kirr@mns.spb.ru>
Signed-off-by: Junio C Hamano <gitster@pobox.com>

```
[...]

Schematic deduction of what every case means,
and what to do, follows:

1)  t < p[imin]  ->  ∀j t ∉ Pj  ->  "+t" ∈
D(T,Pj)  ->  D += "+t";  t↓

2)  t > p[imin]

    2.1) ∃j: pj > p[imin]  ->  "-p[imin]" ∉
D(T,Pj)  ->  D += ∅;  ∀ pi=p[imin]  pi↓

[...]
```

Concurrency models

- Locking
- Merge before commit
- Commit before merge

Decentralized or Distributed?

- Decentralized: there's no single "server" that controls the history of a project

- Distributed: the operations traditionally performed by a "server" and now distributed among "clients"

What does it mean to be “distributed”?

- I can commit artifacts on one node and the history which I create is complete and equivalent to history created on any other node

Myths of version control

- Conflict resolution by merging is intractably difficult, so we'll have to settle for locking.

- Change history representation as a snapshot sequence is perfectly dual to the representation as change/add/delete/rename sequences.

The Next Generation

- BitKeeper (2000)
- Arch (2002)
- Monotone (2003)
- Git (2005)
- Mercurial (2005)

Anyone can make a commit

- No monotonic version numbers
- A commit references the commits from which it is derived
- Commits form a graph

Acknowledging reality

- Everybody branches, all the time

- Merging is expected, and expected to be cheap

dVCS Advantages

- Record history locally
- No need to give “commit” access to many people
- Develop first; decide to branch later

When all of the modules were accessed at their latest level, they totaled 740 719 lines. This number may be taken as the minimum number of lines which must be kept, assuming that only the latest version is needed. At an additional space cost of 37 percent, SCCS not only keeps the necessary versions (one for each customer, system test, and development), but also can regenerate any module at any point since it was placed under SCCS control, as well as maintain a complete history of the changes to the project's software.

Mark J Rochkind, The Source Code Control System, IEEE Transactions of Software Engineering, December 1975

Risks and costs

- More complex set of commands

- Greater set of possible workflows — risk of choosing an overcomplicated one

Questions