Convert interactive rebase to C

Abstract

Many components of Git are still in the form of shell and Perl scripts. This has certain advantages of being extensible but causes problems in production code on multiple platforms like Windows. I propose to rewrite a couple of shell and perl scripts into portable and performant C code, making them built-ins. The major advantage of doing this is improvement in efficiency and performance.
Much more scripts like git-am , git-pull, git-branch have already been rewritten in C. Much more scripts like git-rebase, git-stash, git-add --interactive are still present in shell and perl scripts. I propose to work in “git-rebase --interactive”.

‘git-rebase --interactive’ makes a list of commits which are about to be rebased and lets the user edit that list before rebasing and can also can be used for splitting commits.

Shell Scripts:

Although shell scripts are more faster can be extensible in functionality and can be more easier to write, they introduce certain disadvantages.

1. Dependencies: The scripting languages and shell scripts make more productive code but there is an overhead of dependencies. The shell scripts are lighter and simpler and call other executables to perform non-trivial tasks. Taking 'git-rebase--interactive.sh' shell script for example. sed, rm, echo, test are constantly present in it. These look common to POSIX platforms but for non-POSIX platforms there needs some extra work for porting these commands. For example, in Git for Windows, the workaround for these commands in non-POSIX platform adds some extra utilities and adds MSYS2 shell commands and needs to pack language runtime for Perl. This increases the installation size and requires more disk space. Again, adding more batteries again needs implementation in all of the dependency libraries and executables.

2. Inefficiency: Git has internal caches for configuration values, the repository index and repository objects. The porcelain commands do not have access to git's internal API and so they spawn git processes to perform git operations. For every git invocation, git would re-read the user's configuration files, repository index, repopulate the filesystem cache, etc. This leads to overhead and unnecessary I/O. Windows is known to have worse I/O performance compared to Linux. There is also slower I/O performance of HDD compared to SSD. This unnecessary I/O operations causes runtime overhead and becomes slower in poor I/O performance setups. Now, writing the porcelain into C built-ins leverages the git API and there is no need of spawning separate git processes, caching can be used to reduce unnecessary I/O processes.

3. Spawning processes is less performant: Shell scripts usually spawn a lot of processes. Shell scripts are very lighter and hence have limited functionalites. For git-rebase--interactive to work it needs to perform lots of git operations like git rev-parse git config and thus spawns git executable processes for performing these operations. Again for invoking git config and providing configuration values, it spawn new processes to handle that. Spawning is implemented by fork() and exec() by shells. Now, on systems that do not support copy-on-write semantics for fork(), there is duplication of the memory of the parent process for every fork() call which turns out to be an expensive process. Now, in Windows, Git uses MSYS2 exclusively to emulate fork() but since, Windows doesnot support forking semantics natively, the workaround provided by MSYS2 emulates fork() without copy-on-write semantics. Doing this creates another layer over Windows processes and thus slows git.

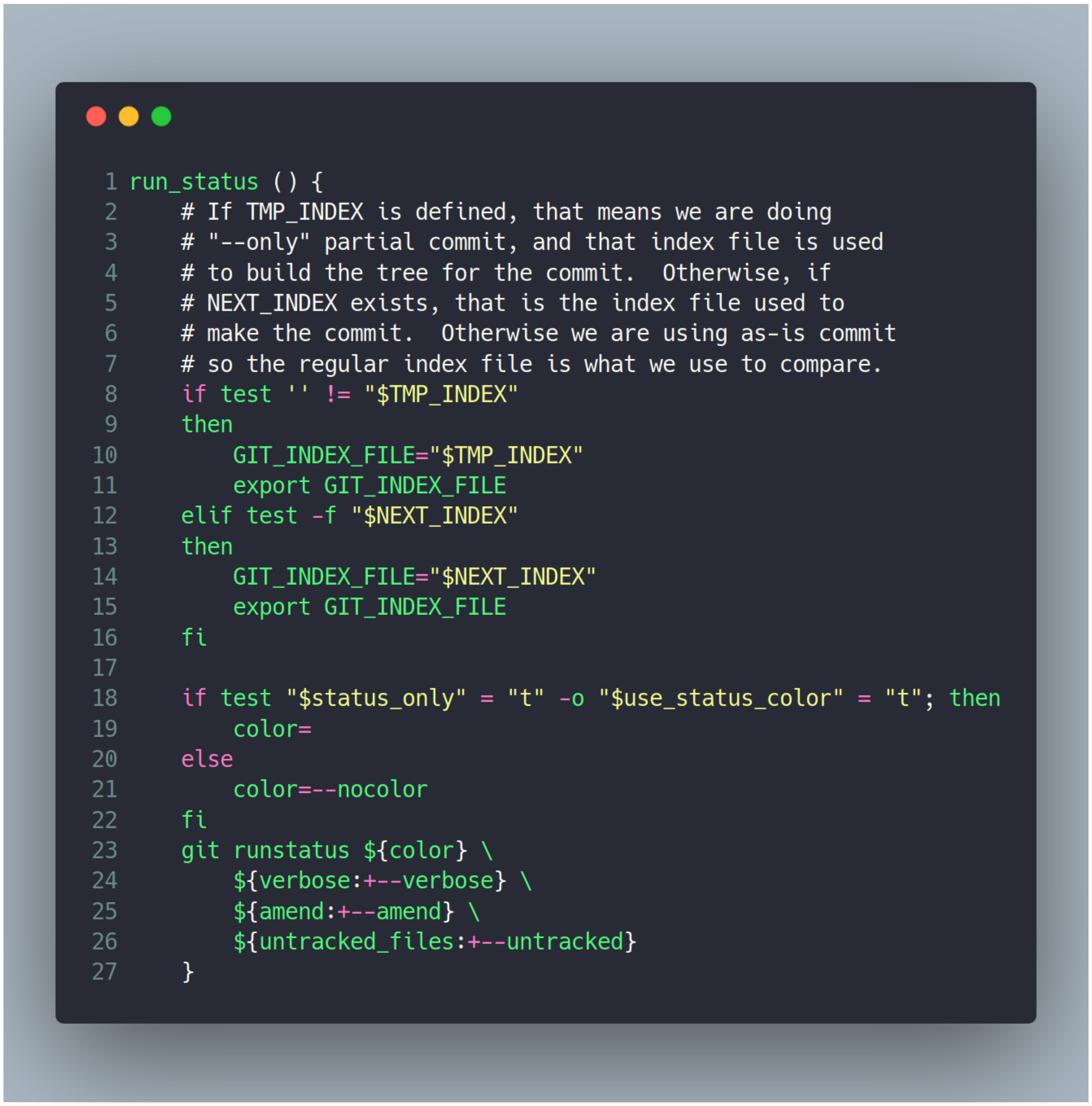
Rewriting C built-ins

These above mentioned problems need to be fixed. The only fix for these problems would be to write built-ins in C for all these shell scripts leveraging the git API. Writing in built-in reduces the dependency required by shell scripts. Since, Git is native executable in Windows, doing this can make MSYS2 POSIX emulation obsolete. Then, using git's internal API and C data types, built-in git_config_get_value() can be used to get configuration value rather than spawning another git-config process. This removes the necessary to re-read git configuration cache every-time and reduces I/O. Furthermore, git-rebase--interactive will be more faster and show consistent behaviour as instead of spawning another process and parsing command-line arguments manually, they can be hardcoded to be built-in and leverage all the required git's internal API's like parse-options.
To implement git-rebase--interactive in C, I propose to avoid spawning lots of external git processes and reduce redundant I/O by taking advantage of the internal object, index and configuration cache. I propose to use C data structures instead of needless parsing as C has richer and performant data structures. Windows will definitely benefit from these rewrites as the use of MSYS2 POSIX emulation would be obsolete and unnecessary I/O processes would be reduced and there won't be need of spawning external git processes.
But, while rewriting regressions should be totally avoided. The code should not cause regression and introduce bugs not present in previous scripts. Thought the rewrite wouldn't look more simpler like previous scripts, it should not have different behaviour or bugs compared to the script and should not be hard to maintain compared to the script.
There has been some development in git-rebase--interactive as seen on https://public-inbox.org/git/CAKk8isqj3OusAE8OJtcys0a-Yj9fgQNn=DtLe-ZGYNzcKp=-3Q@mail.gmail.com/. To maximize the productivity, the findings from the patch submitted can be used. Since, there are already much discussions regarding the rewrite.

Potential Problems

New Re-write code becomes less simpler and Bugs could get introduced:

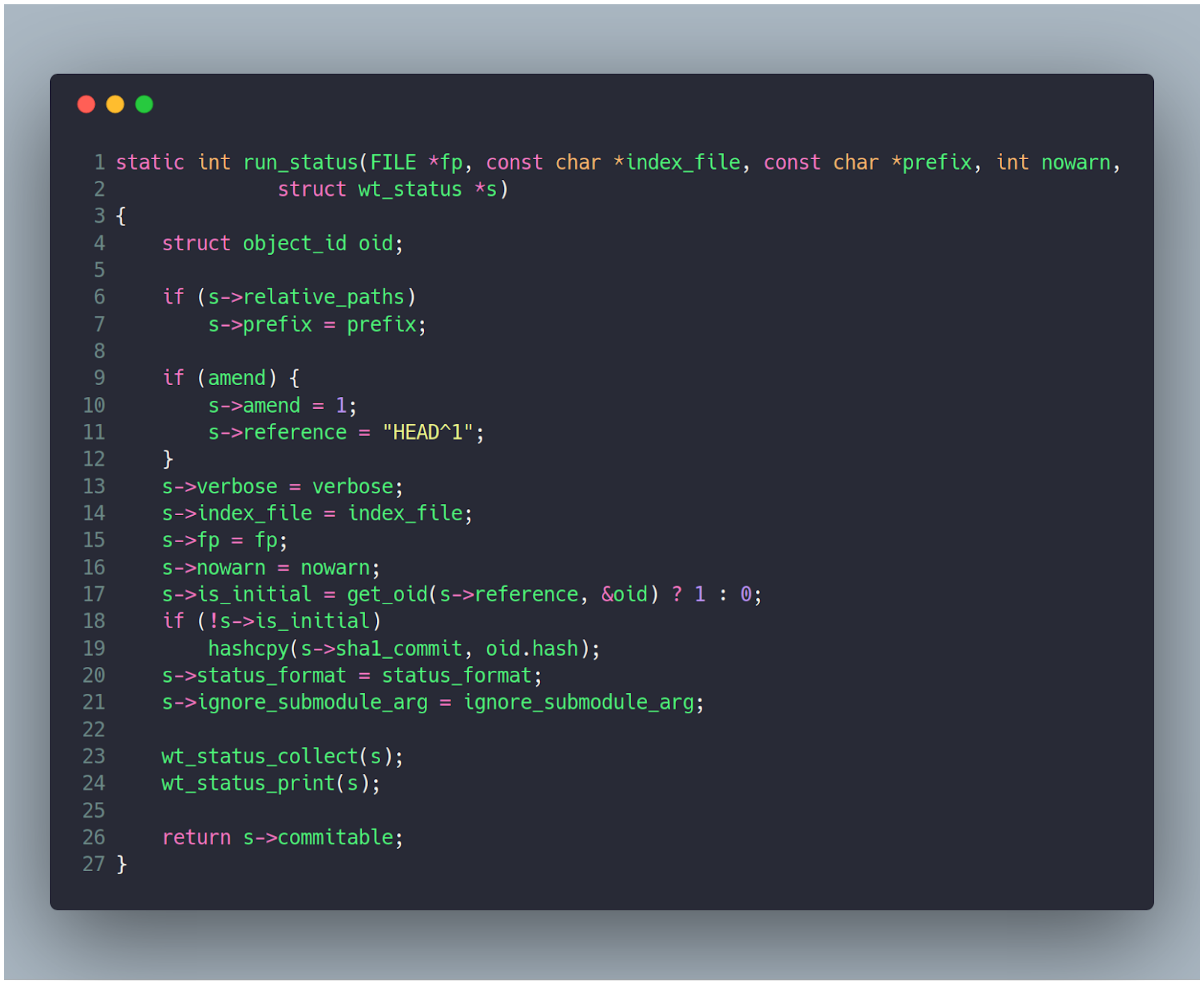
C is a more verbose language than shell script and looks less simpler than shell scripts. Doing things in less than 4 lines of shell script in C can take up more than 12 lines to make it more simpler. For example, in a previously used script `git-commit.sh` we can see,



```
run_status () {
    # If TMP_INDEX is defined, that means we are doing
    # "--only" partial commit, and that index file is used
    # to build the tree for the commit.  Otherwise, if
    # NEXT_INDEX exists, that is the index file used to
    # make the commit.  Otherwise we are using as-is commit
    # so the regular index file is what we use to compare.
    if test '' != "$TMP_INDEX"
    then
        GIT_INDEX_FILE="$TMP_INDEX"
        export GIT_INDEX_FILE
    elif test -f "$NEXT_INDEX"
    then
        GIT_INDEX_FILE="$NEXT_INDEX"
        export GIT_INDEX_FILE
    fi

    if test "$status_only" = "t" -o "$use_status_color" = "t"; then
        color=
    else
        color=--nocolor
    fi
    git runstatus ${color} \
        ${verbose:+--verbose} \
        ${amend:+--amend} \
        ${untracked_files:+--untracked}
    }
```

This script has been rewritten as builtin in C. These lines of code have been replaced in C providing more built-in performance but looks less simpler and contains high multiline functions in `commit.c`



```c
static int run_status(FILE *fp, const char *index_file, const char *prefix, int nowarn,
              struct wt_status *s)
{
    struct object_id oid;

    if (s->relative_paths)
        s->prefix = prefix;

    if (amend) {
        s->amend = 1;
        s->reference = "HEAD^1";
    }
    s->verbose = verbose;
    s->index_file = index_file;
    s->fp = fp;
    s->nowarn = nowarn;
    s->is_initial = get_oid(s->reference, &oid) ? 1 : 0;
    if (!s->is_initial)
        hashcpy(s->sha1_commit, oid.hash);
    s->status_format = status_format;
    s->ignore_submodule_arg = ignore_submodule_arg;

    wt_status_collect(s);
    wt_status_print(s);

    return s->commitable;
}
```

We can see the C version requires more lines of code and seem complex than the script. But, this C code is not depending on external libraries and dependencies hence provides good performance and can be optimized further. The C codes after re-writing can be further documented, optimized and designed to make it more simpler for maintaining it. The functions can be documented with well-defined inputs, outputs and behaviour. The internal git API can be used and the recurring codes can be modified which can shrink the codebase and make git codebase smaller and resulting in smaller installation size in different platforms.

Now, while doing this rewrite there is high possibility of introducing bugs. Though, the test-suite can catch obvious bugs, other less-obvious bugs can escape from test scenarios. To fix this problem, new tests with the co-ordination of other maintainers and contributors should be written for the cases where the test fails to catch bugs. Other code coverage tools can also be used to ensure that the test suite for the command tests all code in builtin. Though it may take more time, careful rewriting and review of code must be done seriously by doing one to one line translations of shell scripts to C. The re-written code after tests must go through rigorous review process to find and squash bugs present in it.

"Given enough eyeballs, all bugs are shallow." - Linus Torvalds

Timeline and Development Cycle

- Apr 23: Accepted student proposals announced.

- Apr 23 onwards: Researching of all the test suites. Discussion of possible test improvements in for git-rebase--interactive.

  Firstly, the test suite coverage of every command will be reviewed using gcov and kcov. The test suite might not be perfect or comprehensive but must cover all the major code paths and command-line switches of the script. The git-rebase--interactive.sh will be closely studied during this time and the tests concerning all the git-rebase--interactive components will be checked for any cases where the test might not be successful to check.

- May 1: Rewriting skeleton begins.
  The shell scripts are translated on a line-by-line basis into C code. The C code will be written in a way to maximize the use of git internal API.
  Firstly, I'll clone https://github.com/dscho/git and checkout sequencer-shears branch, Johannes has worked on a patch to depreciate `--preserve-merges`. My work will build on this patch. I will then start implementing `git-rebase.sh` to call the sequencer when --interactive option is invoked.

- May 13: The modified `git-rebase.sh` file which is made to call the sequencer is then sent for review. This will be the initial patch and would go through a lot of review process. The script will then be renamed to `git-rebase--preserve.sh`

- May 13 - June 10: Make second versions with more improvements and more batteries ready for next review cycle. After sending many versions of the patch, the working will now be of converting `git-rebase--preserve.sh` to C language incrementally.

- June 10 - Jul 20: Now, after these phases, while `git-rebase --preserve.c` file which was converted in the previous cycle is in review process, I will start converting functions of `git-rebase--interactive.sh` to C and migrate them to `git-rebase--helper.c`. I found much of the previous conversion processes, the codes from shell scripts were migrated to helper. Since, `git-rebase--interactive.sh` is a big script file spanning over 922 sloc, I will start rewriting the smaller functions like do_pick(), do_rest() which are kind of wrapper

functions and then start working on the longer and extra brain-cycle needing functions like pick_one() which covers a large portion of the script and will probably double up in C code. I will try to use data structures and avoid repetition of codes to minimize the codebase of the C conversion.
The patches around this cycle will be sent in patch series style. Polishing of the patch will happen with every iterative patch versions.

- Jul 20 - Aug 5: During this cycle, I'll be active in discussion with the community regarding the codebase and I'll start polishing the codebase and again check for code-repetition. During this time, I'll with the close help from my mentors start benchmarking, optimizing the code and improving the code coverage. This will be a hard cycle as optimizing code is difficult. For benchmarking, there are some scripts present in `t/perf` which will be used.

- Aug 14: Submit final patches.
  By this time, the git-rebase--interactive.sh will have been completely written in C and now git can leverage the performance provided by the new builtin code. If there is time for more, then I'll convert `git-rebase--am.sh` to C code.

- Aug 22: Results announced.

- Apr 24 - Aug 14: "What I'm working on" is written and posted in my blog regarding GSoC with Git.
  A blog series will be written in a weekly basis of my current findings and will write about "What I'm working on" to further provide information about my development of summer project.

About me

I'm Pratik Karki and am studying bachelors in Computer Engineering in Advanced College of Engineering and Management (Affiliated to Institute of Engineering, Tribhuvan University), Kupondole, Lalitpur, Nepal. I've been writing C, C++, Ruby, Perl, Clojure, Lisp, Java, JS, Erlang, Rust for 3 years, and contributed to some projects which can be seen in Github. I've been doing independent contract works for small upcoming startup in Nepal. I have been planning to contribute in Git for a long time. Thanks, to GSoC I've submitted small patch as a microproject: test: avoid pipes in git related commands for test suite. I am looking forward to submitting more patches to Git on a long time basis.