University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

Improving Parallelism in git-grep

Matheus Tavares Bernardino

Final Essay [v2.0]

mac 499 - Capstone Project

Program: Computer Science
Advisor: Prof. Dr. Alfredo Goldman

São Paulo
January 19, 2020

Abstract

Matheus Tavares Bernardino. **Improving Parallelism in git-grep**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2019.

Version control systems have become standard use in medium to large software development. And, among them, Git has become the most popular (Stack Exchange, Inc., 2018). Being used to manage a large variety of projects, with different magnitudes (in both content and history sizes), it must be build to scale. With this in mind, Git's grep command was made parallel using a producer-consumer mechanism. However, when operating in Git's internal object store (e.g. for a search in older revisions), the multithreaded version became slower than the sequential code. For this reason, threads were disabled in the object store case. The present work aims to contribute to the Git project improving the parallelization of the grep command and re-enabling threads for all cases. Analyzes were made on git-grep to locate its hotspots, i.e. where it was consumming the most time, and investigate how that could be mitigated. Between other findings, these studies showed that the object decompression routines accounted for up to one third of git-grep's total execution time. These routines, despite being thread-safe, had to be serialized in the first threaded implementation, because of the surrouding thread-unsafe object reading machinery. Through git-grep and object reading refactoring parallel access to the decompression code was allowed, and an speedup of more than 3x was achieved (running git-grep with 8 threads in 4 cores with hyper-threading). This successfully allowed the threads reactivation with good performance. Additionally, some previously possible race conditions in git-grep's code and a bug with submodules were fixed.

Keywords: parallelism. parallel computing. version control systems. Git. git-grep.

Glossary

Commit represents a snapshot of a given project being tracked by Git. In this document, commits are referenced in the following format: `abbreviated hash (title, yyyy-mm-dd)`.

Patch a file that specifies a collection of related changes to some specific code file(s). After applied in a repository tracked by Git, a patch can generate a commit. In the Git project, contributions are sent by email in patch format.

Patch Set or Series a group of patches that usually share the same thematic and/or goal. There might also be dependencies between a patch and its "parent".

Patch Version the patch's current iteration in the revision process. It's very common for a patch not to be merged in its first version. So after getting reviewed, the author can send a new version with the required changes.

Zlib Inflation the operation from the zlib API to retrieve data previously compressed with the DEFLATE algorithm (Deutsch, 1996). Also known as Zlib Decompression.

Definitions extracted from Tavares, 2019a, with some modifications.

Contents

Chapter 1

Introduction

1.1 Context and Historical Background

1.1.1 Version Control Systems

A version control system (VCS), as defined in the Pro Git book (Chacon and Straub, 2014), is *"a system that records changes to a file or set of files over time so that you can recall specific versions later"*. We usually talk of VCSs in terms of source code versioning, but most of them are able to handle any file format. Given a set of files to track, these systems allow the user to perform a number of operations such as: create a new version of the set with a given name and description, see the differences between versions, restore a past version, find in which version a line was changed (and, with the version description, get to know why it was changed), etc.

Using a VCS wisely can also be of great benefit for future contributors of a project. More than just the present state of the project, VCSs provide a lot of metadata on how the project has evolved over time. This is specially interesting when developers take care to create small incremental versions with informative descriptions. These can later be used to understand or recap why certain decisions were made (or more practically, why some lines of code exist). And these are just some of the many advantages of using a VCS.

1.1.2 A Summary of Git's History

Among the version control tools available today, Git, an open source distributed[1] VCS, has become the most popular for source code management. Last year's Stack Overflow Developer Survey showed that almost 90% of the interviewed developers were versioning their code through Git (see Stack Exchange, Inc., 2018). And since 2012 it has had more Google search interest then most of the other major VCSs, such as Subversion, Mercurial, Perforce and CVS (see Google Trends, 2019).

Despite its huge popularity, Git has not been around for so long. The first version was released in 2005, born from a need of the Linux kernel community. In that year, the proprietary VCS that the community had been using for three years then, BitKeeper, got its free license revoked. Linus Torvalds, the creator of Linux, and the community started to look for alternatives. But none of them seemed to fully satisfy the project needs at that moment. Some of them were too slow, some were centralized and, thus, wouldn't fit with the Linux workflow, and so on. In the meantime, Torvalds decided to work on a set of scripts that would serve him as *"a fallback [...] if nothing out there works right now"* (Torvalds, 2005). He didn't even proposed it as a VCS back then, but as "a distribution and archival mechanism". He shared the idea with the community and some members also engaged in the early development. Latter that year, the official 1.0.0 version of that set of scripts was released, already named as "Git". In the following years many other projects adopted Git as their VCS and its usage was even more spread out by online hosting services such as GitHub and GitLab.

To this date, over 1700 people have contributed to the project with more then 40000 accepted patches. Junio Hamano has been the maintainer since 2005.

1.2 Motivation and Goal

1.2.1 The `git-grep` Command

Git is composed of many commands, such as `add`, `commit`, `push`, `grep` and `rebase`. The `grep`[2] command is the one that is going to be targeted by this project. It is used to find lines in the project files that match a user-given pattern. Different from `GNU grep`, `git-grep` is Git-aware and, thus, can use Git features in its search. For example, it's possible to use

[1] In centralized VCSs, the clients query a centralized service for snapshots of the repository. The server is the one that has all the versioning information (both the tracked files and the version history). In Distributed VCSs, however, each client hold a full copy of the repository. There are pros and cons of each, but a commonly mention advantage of the second is that there is no single point of failure.

[2] https://git-scm.com/docs/git-grep

Git's text conversion drivers for binary files, search a past revision without needing to visit it, limit the search to the files being tracked, and etc.

There're two "operation modes" for `git-grep`: *working tree* and *object store*. The first corresponds to a search in the checked out files, i.e., the actual on-disk files in the current version. The latter represents a search in Git's internal data store, which allows searching in older versions of the tracked files.

1.2.2 Performance of `git-grep`

Git is currently used to manage a large number of projects of varying magnitudes (in both code and history sizes). As an example, `Gentoo`'s[3] repository is about 1.3GB while `chromium`'s[4] have 19GB of data. Since a VCS is usually a constant part of the development process, it is desired to be fast, even with larger projects. For this reason, the Git community focuses heavily on performance and scalability.

With that in mind, `git-grep` was found to become substantially slow in larger repositories, especially for object store grepping. To solve this issue and take advantage of the increasingly common multicore processors, the command was made parallel in commit `5b594f4 ("Threaded grep", 2010-01-25)`. The threaded implementation is based on the well known producer-consumer mechanism: the main thread is responsible for iterating through the tracked files (or objects, when searching in Git's internal storage) and adding them to a list. Meanwhile, the worker threads take these individual tasks from the list, read the respective file or object and peform the regular expression search. The list is subdvided in two sections: "tasks yet to be picked" and "tasks being performed or already completed". The completed ones are differentiated from the in-progress ones by a "done" flag. Every time a worker thread finishes a task it tries to update the boundary between these two sections, printing out the results of the finished tasks and removing them from the list.

The parallelization worked very well for the working tree case. However, object store grepping turned out to become even slower when multithreaded. One of the reasons for that could be that object reading (which is a rather time consuming operation) wasn't thread-safe and thus, needed to be serialized. So, because of the observed slowdown, threads were disabled for this mode in `53b8d93 ("grep: disable threading in non-worktree case", 2011-12-12)`. It must be noted, though, that this patch is from 2011 and over 200 patches have been applied to the affected code since then. So it could be the case that multithreading nowadays could bring some speedup for the object store grepping. To test

[3] https://gitweb.gentoo.org/repo/gentoo.git/
[4] https://chromium.googlesource.com/chromium/

that, we downloaded the Git repository[5] at commit `4c86140, ("Third batch", 2019-09-18)`, and applied the following small change (Patch 1.1) to re-enable threads in the object store case. Note that since the threaded code for this case hadn't been removed but only disabled, we only had to erase the condition that did it. Of course, the locks that used to protect the object store grep could have become outdated, but the code seemed to behave as expected with the said patch.

Program 1.1 Patch to re-enable threads in the object store `git-grep`

```
1   diff --git a/builtin/grep.c b/builtin/grep.c
2   index 2699001fbd..fa5135a5ea 100644
3   --- a/builtin/grep.c
4   +++ b/builtin/grep.c
5   @@ -1062,7 +1062,7 @@ int cmd_grep(int argc, const char **argv, const char *prefix)
6           pathspec.recursive = 1;
7           pathspec.recurse_submodules = !!recurse_submodules;
8
9   -       if (list.nr || cached || show_in_pager) {
10  +       if (show_in_pager) {
11                  if (num_threads > 1)
12                          warning(_("invalid option combination, ignoring --threads"));
13                  num_threads = 1;
```

The chromium's repository was used as the test data (because it's a relatively large repository in both data and commit history). Using the test metodology described at Section 3.2, the patched `git-grep` was executed with two different regular expressions in both working tree and object store, varying the number of threads. The mean elapsed times can be seen at Figures 1.1 and 1.2, in 95% confidence intervals. As we can observe, multithreading performance was still worse than sequential for the object store grep in many cases.

[5] https://git.kernel.org/pub/scm/git/git.git/

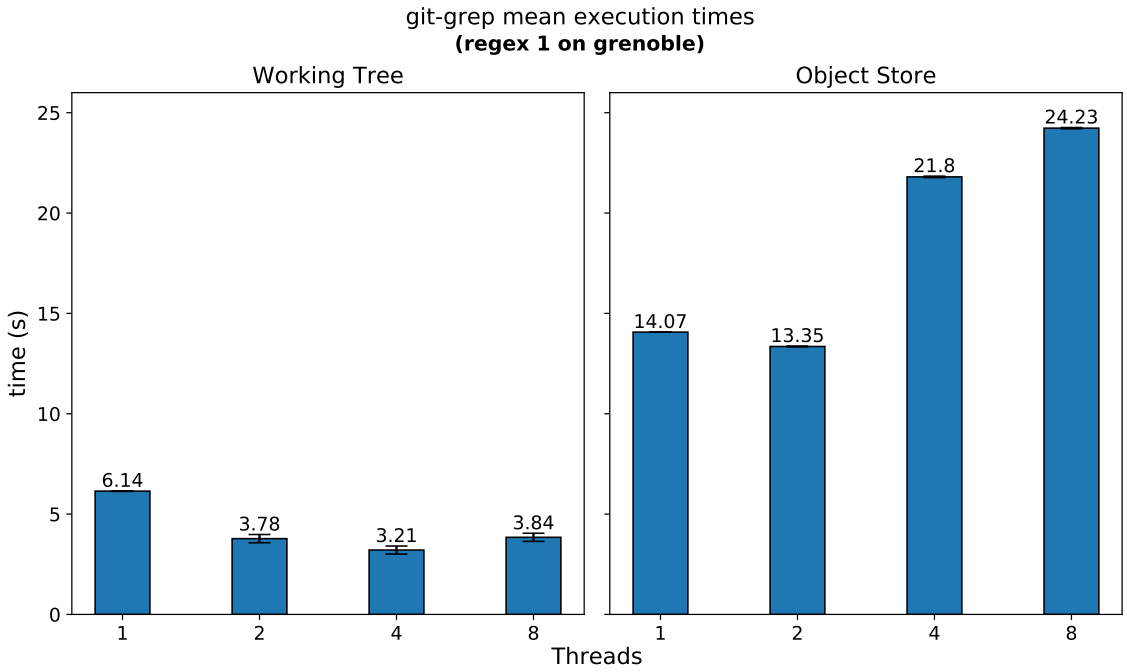

Figure 1.1: *Mean elapsed time of 30* `git grep` *executions in chromium's repository with regex 1 (*`"abcd[02]"`*), for different parameters. Executed on machine grenoble (see Appendix A).*

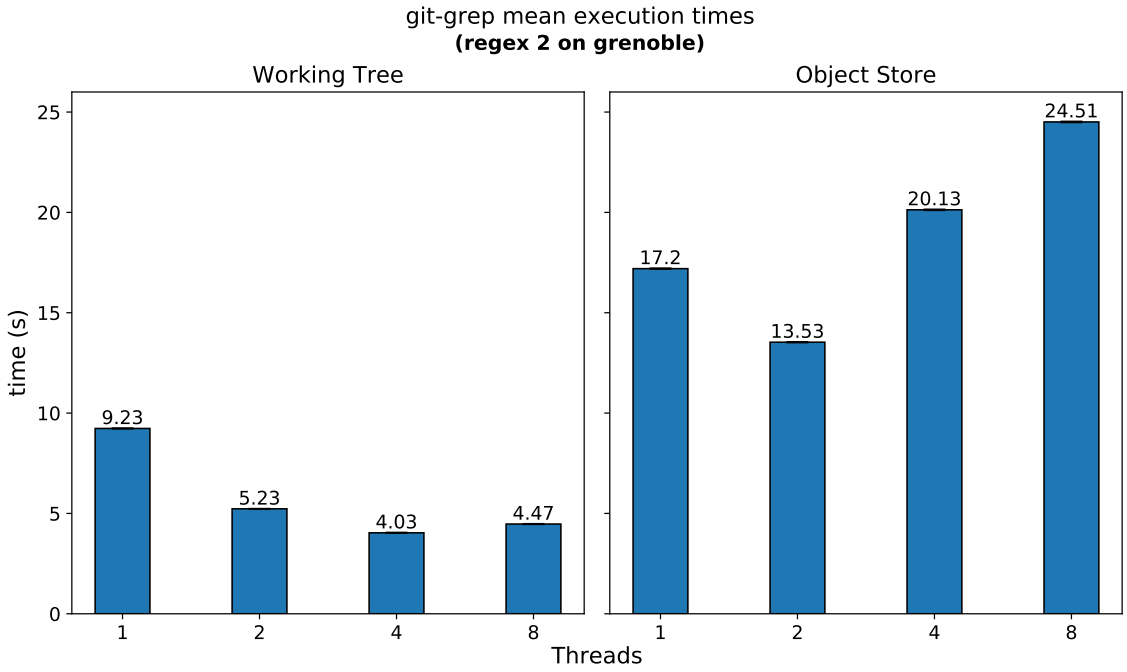

Figure 1.2: *Mean elapsed time of 30* `git grep` *executions in chromium's repository with regex 2 (*`"(static|extern) (int|double) \*"`*), for different parameters. Executed on machine grenoble (see Appendix A).*

1.2.3 Objective

The main goal of this project is to **improve `git-grep`'s parallelism, especially in the object store case**, allowing to re-enable threads with a good speedup. In this process, it is also desirable to **implement an optimized and secure thread access to the object reading functions**. These are currently not thread-safe and thus, the multithreaded code at `git-grep` must use a `mutex` when calling them. However, the more serialized sessions, the less performance gain multithreading can achieve. So, in order to improve parallelism, we aim to protect the object reading operations from the inside, taking care to avoid serializing some time consuming sections that could be safelly performed in parallel. This way, `git-grep` will no longer need its external `mutex` for object reading calls and, additionally, will have its threads performing more work in parallel, pulling a greater performance.

Besides its usefulness for `git-grep`, this thread-friendly API can be used, in the future, to introduce or improve parallelism in other sections of the codebase as well. This is interesting because many commands depend on the object reading API. Especially some that were already reported as slow under some circumstances, such as `git-blame` (Chen, 2018), and thus, could be targeted for a future parallel implementation.

1.3 Document Structure

This document is organized in the following structure: Chapter 2 describes the preparatory period taken to understand Git's codebase and discuss possible strategies with the community. Chapter 3 presents the workflow metodology adopted as well as how the performance tests were executed. Chapter 4 explains the necessary theoretical concepts regarding Git's objects and the object reading operation. Chapter 5 shows the profillings made to locate the bottlenecks in `git-grep`'s code. Chapter 6 describes the process of improving `git-grep`'s parallelism by allowing parallel decompression, and all the corner cases that had to be handled. Chapter 7 presents the benchmarking results for the code with our improvements and the final conclusions. Finally, Chapter 8 disscuss the subjective part, with the author's personal assessment and thought on this project.

As mentioned in the Glossary, all commits in this document will be referenced in the following format: `abbreviated hash (title, yyyy-mm-dd)`. This is also the format used in the Git project iteself. The full commit information can then be inspected running `git show <abbreviated hash>` in the respective repository.

Chapter 2

Preparatory Period

2.1 Getting to Know the Community

Before getting into some of Git's inner workings and the development process for this project, it's important to comment about the preparatory period. Prior to this year, I had no previous experience in contributing to Git. So before even defining the project's scope, it was necessary to take some time to get to know the community and learn the basics. I.e. how the Git community interacts, the contribution workflow, how the codebase is structured and how to do the basic actions in it such as compilling, running tests, etc.

Some great resources made this learning process easier. First, the documentation included in Git's repository (Git Project, 2019) and the very informative Git-Pro book (Chacon and Straub, 2014). Complementarily, there were many great technical blog posts on the web like Spajic, 2018, which explains how Git keeps track of the differences between working tree, staging and cache. And last but not least, the repository's own commit history and the mailing list archive are two excellent sources of information. Inspecting the commit history with a combination of `git-log`, `git-blame` and `git-show`, it was possible to revisit some of the decisions made during the development process of Git and understand why some line of codes are the way they are nowadays. And through the mailing list archive, we could also see the exchanged conversations before these decisions, as well as why some proposed ideas were not viable at that time and thus, had to be dropped.

2.2 Google Summer of Code

The idea for this project was also submitted as a Google Summer of Code (GSoC) proposal (Tavares, 2019b). GSoC[1] is a program by Google which finances students for three months to work on an open source software. One of the great oportunities GSoC brings is having the assistance of mentors from the community to help the students during the development of the project.

To write the proposal for GSoC, some additional code studying was required. In this process, the interaction with the community was vital. I didn't have the knowhow of the code to suggest an improvement plan by myself. But the more experienced developers promptly replied to my questions and helped me defining what could be improved in Git in terms of parallelism (as in Tavares, Couder, *et al.*, 2019). In the quoted mail thread, I also want to highlight Duy Nguyen's help in the initial `git-grep` profillings and implementing a proof of concept parallel decompression (Nguyen, 2019) to evaluate the potential speedup we could achieve with this project.

The project got selected[2] for GSoC 2019. To allow my mentors and the community to track my progress, I kept a weekly-updated blog (Tavares, 2019c). Note that the main goal of the GSoC proposal wasn't to improve `git-grep` threading, but to make pack access functions thread-safe (see Section 4.2), so that they could be later directly used in a threaded environment with good performance. Therefore, the work `git-grep` would be a follow-up task. However, as described in the mentioned blog, we reconsidered the initial plan during the early development phase and decided to target `git-grep` from the start. This way, we could more easily and quickly test how the pack access changes affected performance, with a code that already made use of it. Also, we decided to seek the said improvements for general object reading instead of just pack reading. This is because the strategies adopted could easily be employed in the general case, benefiting more code paths.

2.3 First Contribution

The application for a GSoC project in Git involves the previous development of a micro-project[3]. Between other objectives, these small contributions are intended to help the applicant get used to the contribution workflow. In fact, this was also my first contribution to Git, helping me to get used with the code before the actual GSoC period began.

[1] More info about GSoC is available at https://summerofcode.withgoogle.com/.
[2] https://summerofcode.withgoogle.com/projects/#6477677521797120
[3] https://git.github.io/SoC-2019-Microprojects/

The micro-project I selected was "*Use dir-iterator to avoid explicit recursive directory traversal*"[4]. The idea was to find a place in the codebase that still used the `opendir()`/`readdir()`/`closedir()` API and make it use the Git-internal `dir-iterator` API. One of the advantages of the latter is that it is not recursive, and thus doesn't have the risk of stack overflowing.

Getting to work on this micro-project was an excelent oportunity to exercise the patch sending workflow. A patch is a text file containing a set of correlated modifications to be applied over the source code of the project. It also contains a description of the changes and a reason for making them. After getting reviewed and accepted (possibly after some rerolls), they can be picked up by the project's maintainer and applied in the project's repository as a Git commit. In Git, the patch sending and reviewing happens through the mailing list. I had already worked with a mail-based workflow before, contributing to the Linux kernel. But every project has its own particularities, so this micro-project allowed me to learn more about Git's worflow.

The micro-project turned out to be more complex than we previously thought. The choosed place to make use of the `dir-iterator` API was a function in `builtin/clone.c`, used to handle the files under `.git/objects` in a local clone. In the early development, we found out some inconsistencies regarding its handling of symlinks. Basically, it was calling a lib function which had an OS-specific behavior upon certain conditions. The result was that cloning a repository with symlinks at `.git/objects` would produce different results in GNU/Linux vs. OSX or NetBSD. So, before converting it to use the `dir-iterator` API we worked to make the behavior consistent across different OSs. It was also necessary to adjust the `dir-iterator` API with options to follow symlinks and abort on errors to make it compatible with the place we were going to use it. So in the end, this series of patches turned out to have 10 patches and iterate until version 8. It contained not only the `dir-iterator` conversion at clone but also improvements and test adding to this API. Some patches were produced in collaboration with other contributors and some older patches sent to the mailing list but not merged back then, were updated and incorpored. The final list of patches sent (and already merged into `master`, being part of Git v2.23.0) are:

- **[1/10]**: clone: test for our behavior on odd objects/* content
  https://public-inbox.org/git/a2016d9d3b8e54ff9b9e6dfbd3ab4ce4a1bf7e4d.1562801254.git.matheus.bernardino@usp.br/
- **[2/10]**: clone: better handle symlinked files at .git/objects/
  https://public-inbox.org/git/47a4f9b31c03499bc1317b9a0fccb11c2f5b4d34.1562801254.git.matheus.bernardino@usp.br/

[4] https://git.github.io/SoC-2019-Microprojects/#use-dir-iterator-to-avoid-explicit-recursive-directory-traversal

- **[3/10]**: dir-iterator: add tests for dir-iterator API
  https://public-inbox.org/git/bbce6a601b9dfe018fb482298ab9e4e79968cd05.1562801254.git.matheus.bernardino@usp.br/
- **[4/10]**: dir-iterator: use warning_errno when possible
  https://public-inbox.org/git/0cc5f1f0b4ea7de4e0508316e861ace50f39de1f.1562801255.git.matheus.bernardino@usp.br/
- **[5/10]**: dir-iterator: refactor state machine model
  https://public-inbox.org/git/f871b5d3f4c916599265d34bbb0f7aeb021392c8.1562801255.git.matheus.bernardino@usp.br/
- **[6/10]**: dir-iterator: add flags parameter to dir_iterator_begin
  https://public-inbox.org/git/fe838d7eb4a3f9affca32478397abf8aca9b0230.1562801255.git.matheus.bernardino@usp.br/
- **[7/10]**: clone: copy hidden paths at local clone
  https://public-inbox.org/git/3da6408e045de6e39166227a60472bd1952664ad.1562801255.git.matheus.bernardino@usp.br/
- **[8/10]**: clone: extract function from copy_or_link_directory
  https://public-inbox.org/git/af7430eb2c29ce35691e15d68e1c59d48d6e9144.1562801255.git.matheus.bernardino@usp.br/
- **[9/10]**: clone: use dir-iterator to avoid explicit dir traversal
  https://public-inbox.org/git/e8308c74085689876e25cc88e5628cfd68fc1606.1562801255.git.matheus.bernardino@usp.br/
- **[10/10]**: clone: replace strcmp by fspathcmp
  https://public-inbox.org/git/782ca07eed2c9bac4378e5128ff996b25ed86a43.1562801255.git.matheus.bernardino@usp.br/

The cover letter for this series can be seen at: https://public-inbox.org/git/cover.1562801254.git.matheus.bernardino@usp.br/.

Chapter 3

Metodology

3.1 Workflow

The workflow adopted in this project roughly followed these steps:

1. Get to know the Git community and the contributing process, trying to make the first contributions.
2. Study the codebase and related necessary theory such as the concept of objects in Git, their storage format, etc.
3. Talk with the community on possibilities of contributions to improve parallelism in Git. Also try developing some intuition on the possible bottlenecks.
4. Profile the `git-grep` code and try to locate the current bottlenecks.
5. Try to improve `git-grep`'s parallelism with the smallest and simplest change set possible.
6. Validate the results from the previous item, checking the code corectness and repeat from item 3.

To validate the code correctness, we used the test suite already included in Git's repository. Sometimes, it was necessary to include new tests as well. The tests were executed using Travis-CI[1].

It's important to highlight the concern with producing small incremental changes at a time. Since we are making patches to a huge project being used worldwide, we must make

[1] https://travis-ci.org/

sure, at every step, that the code keeps working as expected. So going for a too big step can be risky, even though it might seem viable.

The Git repository was downloaded from https://git.kernel.org/pub/scm/git/git.git/. The patches from this project are based on the master branch. The exactly commit will depend of the patch set version, as we rebased the series before sending each version. But for the last version (v2), it was commit `4c86140 ("Third batch", 2019-09-18)`.

For all the added code related to threading, we used the `POSIX Threads` API. Note, however that Git also runs on systems where this API is not natively available. For these cases, though, the codebase already implements a conversion header. This way programmers can freely use a single wrapper API, which will then use the underlying threads implementation during compile time.

3.2 Performance Tests

To execute the performance tests both before and after the attempt for parallel improvement, chromium's codebase was chosen as the testing data. This choice came mainly because of two reasons:

- It's a relatively large repository in both history size and content size.
- Chromium's developers already reported some difficulties with a couple of Git commands that were slow in their daily usage (see Zager, 2014). We also wrote to the Chromium community asking if they still had those difficulties and they cofirmed (see Tavares, Chen, *et al.*, 2019).

Chromium contains different file formats, so two regular expressions were crafted to use as test cases:

- **Regex "1"**: `"abcd[02]"`; and
- **Regex "2"**: `"(static|extern) (int|double) \*"`.

The first is artificial, looking for a generic pattern in no particular file format. The second is a more realistic use case, strieving to find occurrences of specific pointer declarations in C/C++ code. In terms of complexity, the second takes more time to get executed. We purposely used one more complex regex to try capturing the multithreaded performance when regex matching took longer to execute.

The chromium repository was downloaded from https://chromium.googlesource.com/chromium/src/ at commit `03ae96f ("Add filters testing at DSF=2", 04-06-2019)`. A `git gc` execution was then invoked to make sure the repository's objects were packed (as the

initial goal was to improve the parallel access to packfiles). But note that the improvements made during this project should also speed up the reading of loose objects.

Unless otherwise mentioned, the machine used for the tests was `grenoble` and the command ran was `git -C <path to chromium> --no-pager grep --color=never --threads=<number of threads> <regex>`. For the test cases on the object store, an additional `--cached` option was used, and for the regex 2, `--extended-regexp` was also given. For each test case, the above command was repeated 30 times to calculate the confidence interval with 95% of confidence. The interval was calculated with the `Student's t-distribution` using a custom python script named `cint`[2] and ploted using `pyplot`[3]. Before the batches of 30 runs, two warmup executions were also dispatched to populate the machine caches.

Some tests were also performed on the machine `mango`, to check the performance with SSD. It's important to run the tests on different hardware because our problem is very closely related to I/O. Therefore, the proposed changes could bring a speedup for one storage device but be ineffective on others, or even bring a slowdown. More information on the architecture of the machines used can be seen in Appendix A.

Finally, since power management can interfere with the performance of the running applications, it's important to mention that, for `mango`, which is a laptop, all performance tests were executed while connected to the power source. `grenoble` is a desktop.

[2] https://github.com/matheustavares/cint/
[3] https://matplotlib.org/api/pyplot_api.html

Chapter 4

Theoretical Background

To understand why `git-grep` on object store was so much slower (as seen in Subsection 1.2.2), even before profilling the code, we decided to take some time to understand how the object store works. As we will later discuss, understanding the problem's domain first turned out to be more helpful for improving performance than going straight scanning the codebase for local code structures that could be optimized. So as a first step, we studied the concept behind Git objects as well as their storage format on disk. We also examined the codebase to cromprehend how they are read, since this seemed to be a considerably slow operation during `git-grep` execution. In this chapter, we will present the information gathered.

4.1 Git's Objects

When running `git-add` followed by `git-commit`, the contents of the commited files will be internally stored by Git in its object store (in the `.git/objects` directory). In fact, not only the files' contents, but also the directory tree structure and the commit itself are represented and stored as objects.

Conceptually, Git's data store can be visualized as an in-disk key-value table. Each object is stored in a compressed binary format and it's referenced by its hash. For compression, Git uses `zlib`, which implements the lossless DEFLATE algorithm (see Deutsch, 1996 for DEFLATE's specification and Roelofs *et al.*, 2006 for `zlib`'s technical details). For hashing, Git uses the SHA-1 algorithm[1] (Eastlake and Jones, 2001), which is implemented in many libraries. The actual library used by the git binary is choosen at compile-time and may be,

[1] At the time this thesis is being written, there's an ongoing plan to migrate Git's hashing from SHA-1 to SHA-256. More information about it is available at https://github.com/git/git/blob/master/Documentation/technical/hash-function-transition.txt

for example: OpenSSL, Apple's Common Crypto library or internal Git implementations. The hash is taken before compression and not only the object content is hashed, but also metadata present in the object's header.

It's important to notice that since objects are indexed by their hashes, each object is, by consequence, immutable (because changing the content would lead to a new hash and thus, a new object). So when amending a commit, for example, Git is not really modifing the said commit but creating a new one and making the current branch head point to it. The outdated version is kept for a while, but being a dangling commit, it will be eventually removed by `auto gc` (`gc` stands for `garbage collect`). This is a maintenance command ocasionally ran by Git to perform cleanup and repository optimization tasks (see section 10.7 of Chacon and Straub, 2014 for more info). One of these tasks is specifically the removal of objects that are not reachable by any commit (but this is only done after a specific time has passed since the object creation).

There are currently six *types* of objects implemented in Git. Conceptually, though, we can think of them in terms of only four: blob, tree, commit and tag. The other two, ofs-delta and ref-delta, are used to represent instances of these previous four types in a more memory efficient manner. They do that by allowing to store only what is different between two objects when they have very similar content. But we will talk more about these two delta objects in the following section. For now, let's focus on the first four. The following definitions are based on Shaffer *et al.*, 2019 and the Sections 2.6 and 10.2 of the Git Pro book (Chacon and Straub, 2014):

- **Blob**: stores a generic stream of bytes. This is the most "flexible" object type, having only a very small header and no fixed structure for its contents. The most common use for a `blob` is to store contents of files tracked by Git. But just the contents, no filesystem information such as the filename or permissions, which are stored by the `tree` object. A new `blob` is created, for example, in the invocation of `git-add`. Another use for `blobs` is to store Git notes[2].
- **Tree**: a table, mapping names to objects. Its most common use is to represent a directory tree, listing files and subdirectories. Each entry contains the filename, permissions and a reference to the object that represents it internally (a blob for regular files and another tree for subdirectories).
- **Commit**: represents one revision in the project. It holds information about the author, commiter, date, associated message, and etc. This object also contains references to: a tree object, which is a snapshot of the repository's root directory at

[2]See https://git-scm.com/docs/git-notes

the given commit; and a possible list of other commit objects, which represents its parents.

- **Annotated Tag**: Note that Git has two kinds of tags: `lightweight` and `annotated`. Both work as a label to another object (which is generally a commit). The first one is implemented as a simple file at `.git/refs/tags/`, named by the tag name, and containing the hash of the referenced object. The second one, though, is a full Git object, being stored at `.git/objects/` and referenced by its hash. It not only contains the tag name and the hash of the referenced object but also the tagger name, email and date. `Annotated` tags also permits the addition of a message and GPG[3] signing.

Figure 4.1 illustrates the structures of `blobs`, `trees` and `commits`, as well as the relationships between these objects.

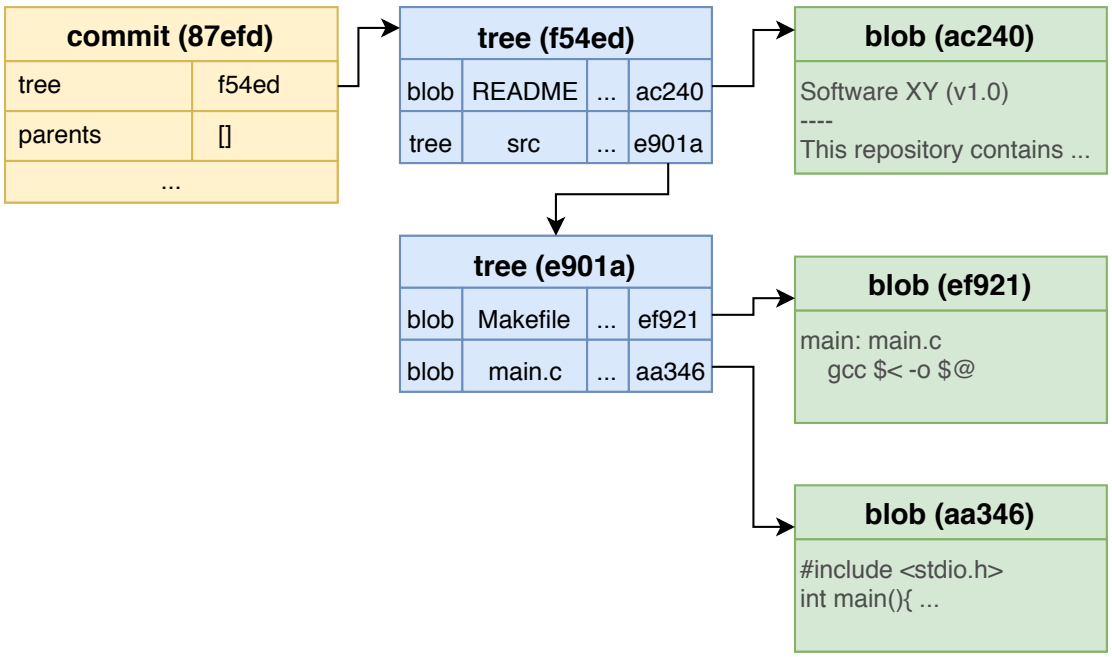

Figure 4.1: *Example diagram of Git objects in a repository.*
(Based on the GPLv2 image at https://github.com/schacon/gitbook/blob/master/assets/images/figure/objects-example.png.)

Knowing the above definitions, it's already possible to mention one of Git's optimizations when storing files: from one commit to another, it's not really necessary to recreate objects that didn't change (because both their contents and, by consequence, their hashes will be the same). So instead, Git will just reuse the blobs and trees already created, under this circumstance. An example diagram of reused objects can be seen in Figure 4.2.

[3] https://www.gnupg.org/

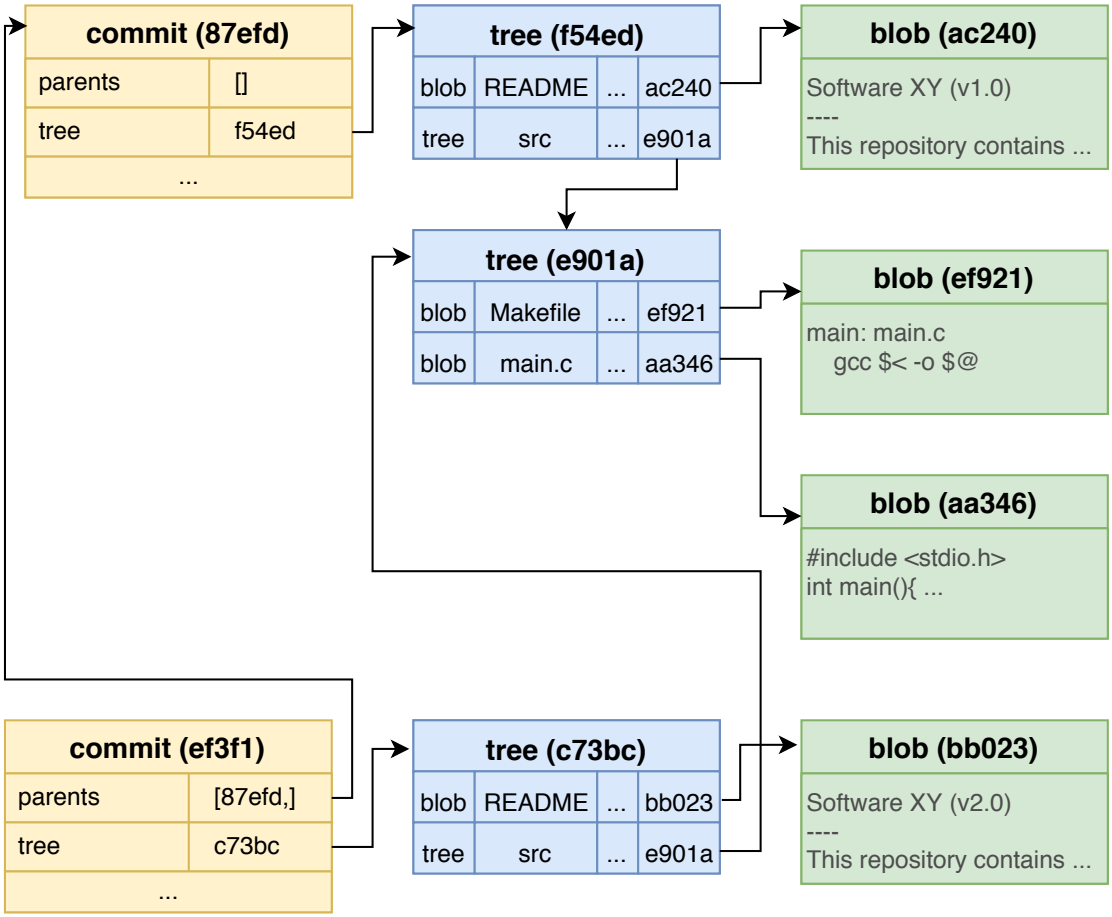

Figure 4.2: *Example diagram of Git objects being reused.*
The above figure represents a repository in which the only change between commits `87efd` *and* `ef3f1` *is the replacement of "v1.0" to "v2.0" in the* `./README` *file. Note how Git will reuse already created objects when they don't change.*

The Pro Git book by Scott Chacon and Ben Straub (Chacon and Straub, 2014) contains a dedicated chapter on Git's internals (Chap. 10) where it is possible to find more information on Git's objects and how/when they are created. Scott also hosted a very complete introductory talk on Git (Chacon, 2008), which addresses the basics on objects types.

4.2 Storing Formats

When creating new objects, Git will usually store them in the "loose" format, i.e. creating a new file at `.git/objects` containing the object's content (compressed binary) and named by its hash. However, this is not the most efficient format. For example, when changing a single line of an already commited file, Git will create a whole new blob even

though most of the content is the same.

A more economic format is the `packfile`. These files, which have the `.pack` extension, are found at `.git/objects/pack` alongside homonymous `.idx` files. The latter are used for indexing, which is crucial since `packfiles` can hold many objects in a single filesystem entry. The object entries are also stored in binary and zlib compressed format but they benefit from an extra level of compression: deltification. I.e., two or more versions of very similar objects are not stored redundanctly but, instead, a single base version is stored together with instructions on how to reconstruct the others. These instructions are stored in what's called a delta object, which is only valid in `packfiles`. As the packing format documentation says:

> "The delta data is a sequence of instructions to reconstruct an object from the base object. If the base object is deltified, it must be converted to canonical form first." - Git Project, 2018

The above quote implicitly states that is possible to have a delta's base object being another delta. This forms what is called delta chains. To be valid, every delta chain must of course end in a non-delta base. Otherwise, none of the objects in the chain would be reconstructable.

Delta objects come in two subtypes: `ref-delta` and `off-delta`. Their only difference, as described in Git Project, 2018, resides on how they reference their base objects. The first one holds the base's 20-bytes hash. The second contains the offset of the base object in relation to the delta and thus, can only be used when the base and the delta are both in the same `packfile` (a single repository can have multiple `packfiles`).

It's important to mention as well that the packing techniques (as described in Git Project, 2014) try to generate deltas from larger objects to smaller ones. And, since code files tend to grow over time, this approach will result in less deltas at the most recent revisions (the most common deltas will be in fact 'backward-deltas'). This is good for performance as users usually want to handle the most recent revisions more frequently.

Another performance feature described in Git Project, 2014 (but this time related to memory) is that Git may create deltas against files with same basename but residing in different directories. As it says "The rationale is essentially that files, like Makefiles, often have very similar content no matter what directory they live in." Finally, it is also mentioned that using the mentioned techniques and other heuristics, `packfiles` manage to achieve good IO patterns, in general.

So, being so optimized, is inevitable to question: why not creating the objects in the

packed format, already, instead of the loose one? The answer is: insertion in packfiles is not fast. As the document cited in the above paragraph states, packing involves object listing, sorting and walking with a sliding window (for deltification). Performing this operations for every `git add` or `git commit` wouldn't be viable. So, instead, objects are usually created in loose format (which is very fast since this is only a single hashed and compressed file at `.git/objects`) and them, ocasionally, they are gathered into a packfile. This is done automatically by auto-gc, upon certain conditions, trying to make the repository as optimized as possible.

4.3 Object Reading

Retrieving Git objects, which are compressed, possibly stored in different formats and deltified, is a process that requires going to a couple of phases. Not to mention the precautions taken to catch any possible failure that might occurs througout this process. Therefore, the object reading machinery is quite complex, and subdivided in many functions. But the main entry points to this machinery, are:

- `repo_read_object_file()`
- `read_object_file()`
- `read_object_file_extended()`
- `read_object_with_reference()`
- `read_object()`
- `oid_object_info()`
- `oid_object_info_extended()`

These are the highest level functions in the object reading code. An user of this machinery will usually call one of them which, in turn, delegates the underlying tasks to the respective lower level routines. One might ask why we need so many different entry points instead of a single "`read_object()`". Actually, this "`read_object()`" is in fact what `oid_object_info_extended()` implements. All the other functions above call this one to really perform the work. They exist in order to simplify the code of the object reading API users, calling the said "back-end" function with default paramenters, implementing addional checkings, etc. So we will focus on explaining how `oid_object_info_extended()` works. It receives four parameters:

- A `struct repository *`, which represents the repository where the caller wants to fetch the object from.

- A `struct object_id *`, the hash of the said object.
- A `unsigned flags` to store the combination of desired options (for example, skipping the cache, ignoring loose objects, etc.).
- The `struct object_info *` out-parameter which will be filled with the information of the desired object.

The return value is an integer code to inform failure or success. The steps done by this function are roughly the following:

1. Lookup object replacement map[4].
2. Lookup the `cached_objects` list. This is an in-memory cache for objects that are not necessarily written in disk (can be used for "fake" or temporary objects, for example).
3. Try to find a packfile in the given repository which contains the given hash.
4. In case of failure to find packfile, assume the hash is referencing a loose object and try to read it. In case of success, return the read information.
5. In case of failure, try to find the packfile again. The object could have been just packed by another Git process while this process was executing the above step.
6. If it fails, it might be the case that the given repository is a partial clone[5] and the required object is missing locally. Partial clones allow users that are not interested in the whole repository of a project to download only specific portions of it during the clone operation. Later, if a missing object is required, Git will try to fetch it on demand. So, in this case, try fetching the object and repeat from step 3. If it fails, return with error.
7. At this point, it's known that the object is packed and also in which packfile it's stored. So try retrieving the information from it and return. Or return with error, if it fails.

The code that performs steps 4 and 7 (i.e. the actual data retrieval from a loose or packed object, respectively), is done by other routines called by `oid_object_info_extended()`. They are, respectively, `loose_object_info()` and `packed_object_info()`. We won't show a pseudocode for these functions, but it's important to highlight that both need to read from disk and perform decompression, as both loose and packed objects are stored in zlib compressed format. For packed objects, there's even the additional work of reconstructing the deltas (after decompressing them as well).

[4] See https://git-scm.com/docs/git-replace
[5] https://git-scm.com/docs/partial-clone

The process of delta reconstruction is optimized with a LRU cache (i.e., a cache that removes the Least Recently Used entry when it is full and a new entry needs to be added). The motivation for this structure is that two or more different deltas may share the same base. So instead of redundanctly reading (and decompressing/reconstructing) the same base twice, the most recently used bases are kept in-memory to be reused. (Note that, for a base that is also a delta, the data stored represents the final base in this delta chain, with the deltas applied.) The key in this cache is the base object location, in the format of a pair (`packfile P, offset O in that packfile`).

The function responsible for uncompressing and delta-reconstructing objects from a `packfile` is `unpack_entry()`, which is called by the previously mentioned `packed_object_info()`. Part of `unpack_entry()`'s inner workings will be very important in the Chapter 6, so its general operation is described at the Pseudocode 4.1. It might seem odd that it removes the cache entry at phase I, instead of just picking it. But note that, the base will be re-added at phase III (updating its position in the LRU list).

Program 4.1 Pseudo-code of function `unpack-entry()`.

```
1   FUNCTION unpack_entry(P, O):
2       S <- initialize empty stack of delta locations
3
4     ▷ PHASE I: drill down to the innermost base object
5       do:
6           if (P, O) is in delta base cache:
7               DATA <- pop cached data from object at (P, O)
8               break
9           else:
10              H <- read and decompress header of object at (P, O)
11              T <- read type of object from header H
12              if T is a delta:
13                  push delta location (P, O) to stack S
14                  P, O <- get base location of delta at (P, O)
15      while T is a delta, repeat;
16
17    ▷ PHASE II: uncompress the base (if didn't retrieve it from cache)
18      If dont have DATA:
19          DATA <- read and uncompress object at (P, O)
20
21    ▷ PHASE III: apply deltas in order
22      while S is not empty:
23          add DATA to delta base cache with (P, O) as key
24          (P, O) <- pop next delta location from stack S
25          DELTA_DATA <- read and uncompress object at (P, O)
26          DATA <- apply DELTA_DATA over DATA
27      return DATA
```

With the above details on how object reading works, it is intuitivelly understandable why `git-grep` on object store takes so much longer than on the working tree. While the latter just needs to read an uncompressed file to have the data in memory, the former has to perform a lot of operations besides I/O, such as decompression, delta-reconstruction, handling of multiple packfiles and etc.

Chapter 5

Profiling

For a good implementation (or enhancement) of parallel code, it is crucial to understand where the bottleneck is in order to properly direct the parallelization effort. In the previous chapter, we presented the intutive assumption that object reading is one of the most time consuming sections of `git-grep` in the object store. Its a convincing hypothesis, especially after looking into how complex object reading is. But before we continue, it is important to show how this hypothesis was validated and, furthermost, how we got to know which subsection of object reading could be made parallel with good performance.

As seen in the previous chapter, object reading involves many smaller tasks. These can be subdivided into two categories: CPU-bound tasks and I/O-bound tasks. On the former falls CPU intensive operations like zlib decompression and delta reconstruction. On the latter falls the actual file reading. If the performance bottleneck was due to I/O-bound tasks, then we would have been in a difficult spot as parallel I/O is not always very effective. However, if it were due to CPU Bound tasks, as it turned out to be, the chances for threading improvement become much greater.

To evaluate where the said bottlenecks were, we used two profilers: `gprof`[1] and `perf`[2].

To start, Git was compiled and linked with `-pg -O0`, to disable compiler optimizations and make the generated binary write out the profile information suitable for `gprof`. One might point out that it's also important to profile the optimized code as the bottleneck sections might differ. In our case, though, they remained the same across both profilings. So we will stick with the unoptimized one since it is more faithful to the actual source files and, therefore, easier to understand.

[1] https://sourceware.org/binutils/docs-2.33.1/gprof/index.html
[2] https://perf.wiki.kernel.org/index.php/Main_Page

Running the following command in `chromium`'s repository: `git --no-pager grep --color=never --cached --threads=1 -E '(static|extern) (int|double) \*'`; and using `gprof2dot`[3] for the vizualization we geneated the diagram in Figure 5.1. Since the producer-consumer mechanism is only used when multithreaded, and that could affect the profile, the experiment was also repeated with 8 threads (for that, the Patch 1.1 was applied). The multithreaded profile can be seen in Figure 5.2. For a full description of the graph's features, please check the `gprof2dot` page. For this analysis, though, it's sufficient to consider only the percentage number in each node and edge. The number without parenthesis in the nodes is the percentage of the total time spent in the said function and its call tree. The number in parenthesis excludes the call tree contribution. And the number in the edges is the percentage of the total time transfered from the callee to the caller.

In both Figures 5.1 and 5.2 it's visible how object reading has a big influence in the elapsed time, confirming the previous hypothesis. A special attention must be given to `oid_object_info_extented()`, which, as presented in the previous chapter, is the main entry-point for object reading. Together with its callees, this function is responsible for over one third of the total execution time, in both profiles. Another highlight goes to `unpack_entry()`, the function which performs object decompressing and delta-reconstruction, accounting for approximatelly 23% of the total execution time. In its call tree, `patch_delta()` gets the first place in time consumption. With all this information, targeting the CPU Bound tasks of object reading (especially delta reconstruction) seemed very promissing for a good optimization.

What we can't see in these profiles, though, is the time spent inside shared libraries, such as `zlib` (it's a `gprof` limitation). That's probably why we see `git_inflate()`, the wrapper for `zlib` decompression, taking such a small section of the total execution time in the said profiles. And since it was also a candidate for parallelization alongside delta reconstruction, another profiller had to be used to evaluate it's potential. For that, we used `perf`.

Git was recompiled with `-g -O0`, to append the necessary information for `perf` and, again, disable compiler optimizations. Then, the same command as before was executed but prefixed with `perf record -F 99 -g --`. This generates a `perf.data` file which was then processed by `perf script`. To vizualize the resulting information we used `FlameGraph`[4]. The resulting image for both single thread and 8 threads can be seen, respectively, in Figures 5.3 and 5.4. These images show us the call stack profile, with each rectangle representing

[3] https://github.com/jrfonseca/gprof2dot
[4] http://www.brendangregg.com/flamegraphs.html

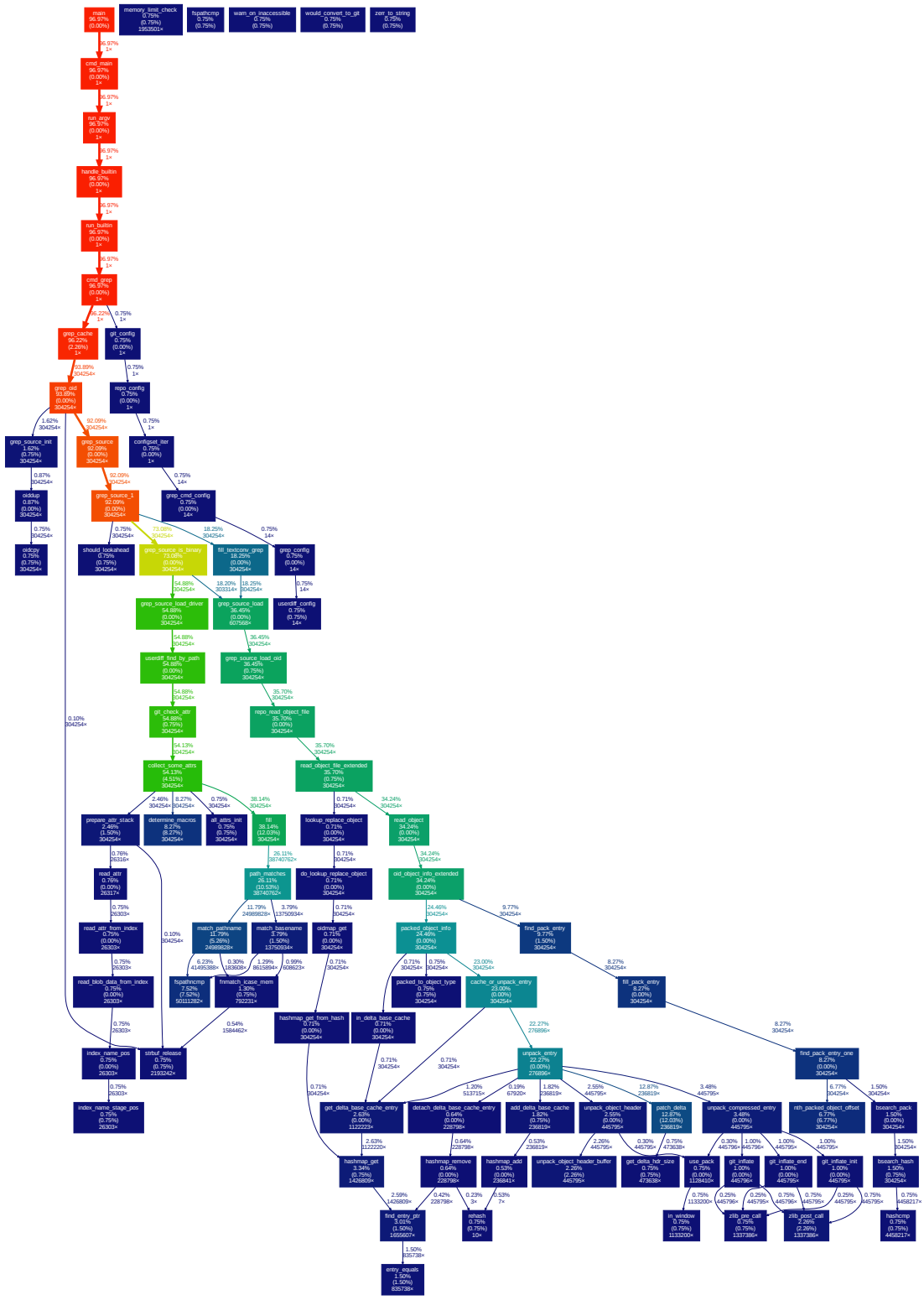

Figure 5.1: *Vizualization of* gprof*'s profile for cached* `git-grep` *in chromium's repository with a single thread.*

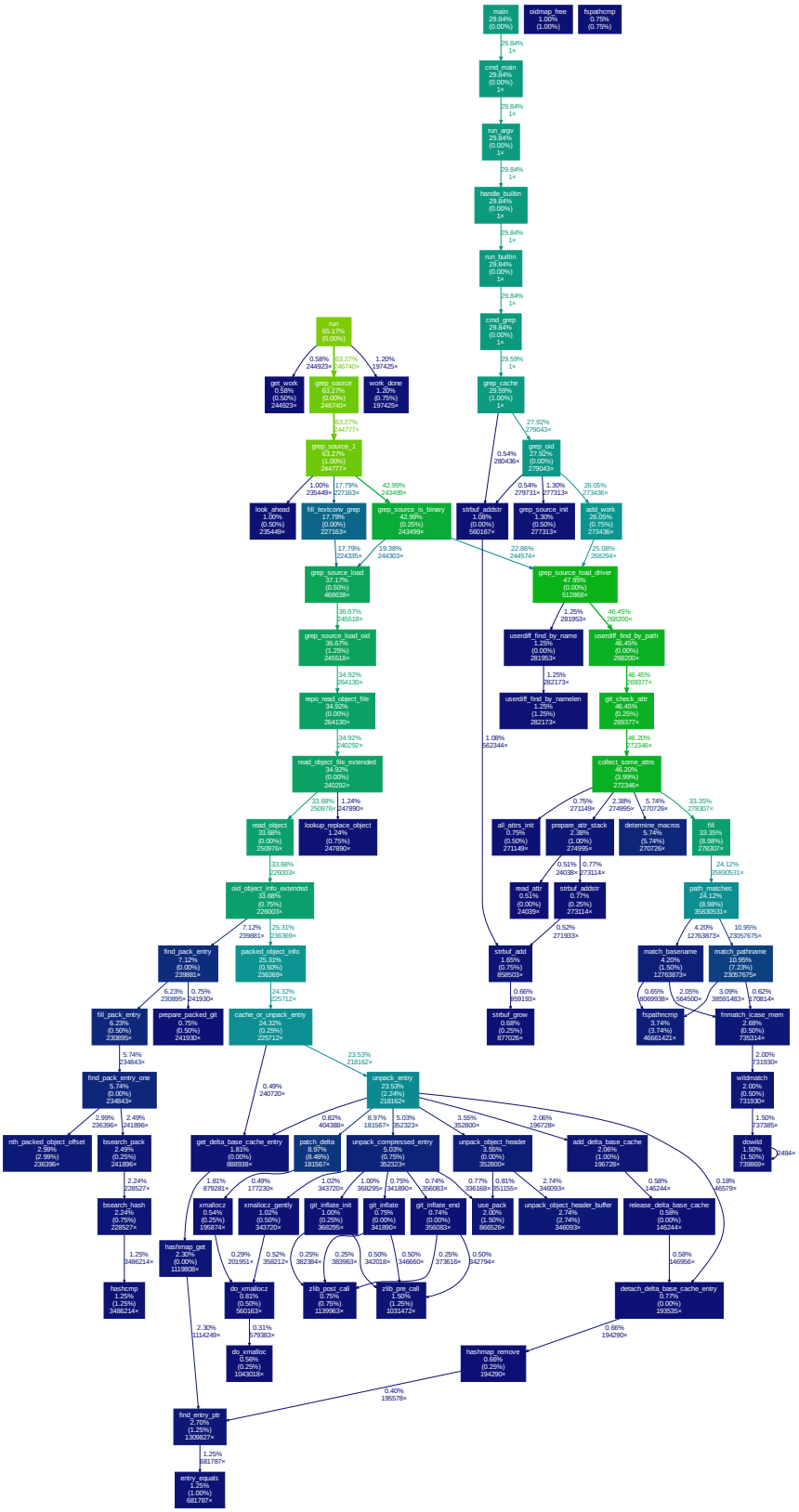

Figure 5.2: *Vizualization of* `gprof`*'s profile for cached* `git-grep` *in chromium's repository with 8 threads.*

a stack frame. The larger the rectangle, the more time the process spent on it. The y-axis is ordered in caller-callee fashion (from bottom to top) and the x-axis is alphabetically ordered. We also colored the figures to highlight some sets of functions of interest.

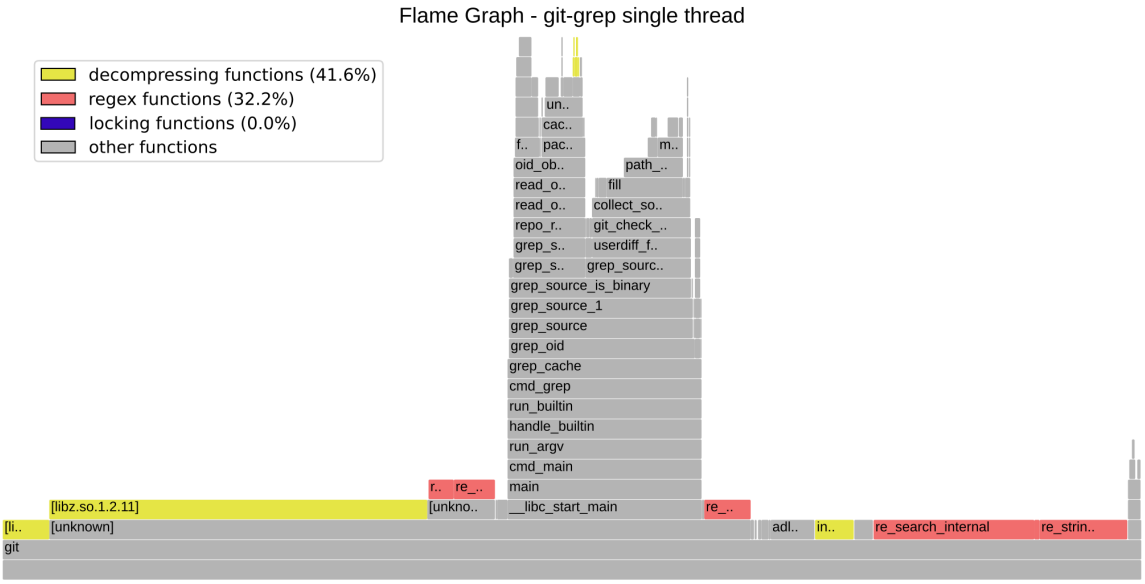

Figure 5.3: *Vizualization of* `perf`*'s profile for cached* `git-grep` *in chromium's repository with a single thread.*

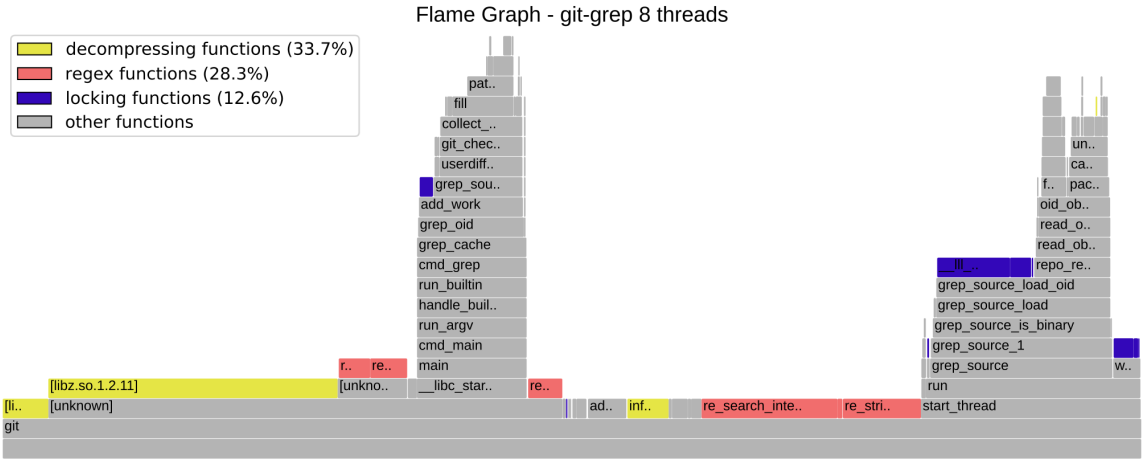

Figure 5.4: *Vizualization of* `perf`*'s profile for cached* `git-grep` *in chromium's repository with 8 threads.*

Figures 5.3 and 5.4 show that an impressive percentage of time is spent in decompressing routines. In both of them, this set of operations alone accounted for more than one third of the program's total execution time. Additionally, although zlib decompression is currently being performed serialized across `git-grep`'s threads, this operation is thread-safe (see Roelofs *et al.*, 2010). And the work in different decompressing streams is, in theory,

pretty paralelizable. These characteristics made decompression one of the most interesting candidates for parallelism.

Another interesting point to note in Figure 5.4 is the considerably long time spent in locking functions. One reason for that could be lock contentions, which should also be reduced if we allow more work to be performed in parallel.

Chapter 6

Development

As already discussed, threads were disabled in `git-grep` for the object store case. However, the threaded implementation wasn't removed from the code. So the development idea for this project was to, first, improve the object reading functions so that they could safelly and efficiently be performed in parallel. Then, take advantage of the already implemented threads machinery in `git-grep`, and allow it to work threaded in object store again, but this time, with a hopefully better performance. In this process, we would remove the lock used by `git-grep` to call object reading functions as they would then be already thread-safe. Concerning the locks, `git-grep` used to have three `mutexes` in its code: `grep_attr_mutex` to protect the thread-unsafe userdiff API (more on this API later in this chapter); `grep_read_mutex` to protect object reading operations and other function calls that would otherwise be in race condition with object reading; and `grep_mutex` to control the operations on the producer-consumer queue. The only one we planned to remove during this process is the `grep_read_mutex`.

6.1 First Approach: Protect Only Global States

The `git-grep` profiles from Chapter 5 showed that, in theory, a parallel access to object reading could indeed bring some good speedup. Additionally, delta reconstruction and zlib decompression seemed to be the sections that would benefit the most from this (the latter being the most time consuming and thus, the most promissing). Because of that, we choosed to target zlib decompression. One might wonder, however, why targeting one segment for parallelism? Why don't remove the `grep_read_mutex`, protect the operations on global variables that were now left unprotected and let everything else run in parallel? In fact, this was also our first idea, as described in the initial GSoC proposal. This design, however, turned out to be very hard. Since object reading is pretty complex, locating and

protecting each and every global state is not an easy task (because they might be used in different levels of abstraction). Also, this approach may not seem as such a big change in code, but in fact it is. And thus, it is probably much safer if done in incremental stages over some period, making sure the expected behavior is preserved.

But why could these changes be so dangerous? By allowing threads to work in parallel in some segments, we are also opening holes in a big critical section that used to wrap the whole object reading execution. Even if we could locate and protect all reading and writting operations on global variables, it possibly wouldn't be enough to avoid race conditions. Just by splitting this big critical section in two, we risk having global resources being changed in between a thread's execution of the two halves. And if the second half expects a certain resource, not guarding itself against the possible change, this might lead to a failure. In other words, the splitting of the critical section would break a previously guaranteed invariant that ensures correctness.

To illustrate this problem, take a look at the diagrams in Figure 6.1, which represent the work done by the `git-grep` threads over time. The one in the left corresponds to the original code, with totally sequential object reading. In the right side, though, we pretended to have added a single parallel region inside the object reading code (creating one hole in the critical section). Now, a global resource that was, for example, prepared by the section A_1 to be used by A_2, might not be available when A_2 executes, because B_1 might have changed it in the meantime. If we fail to find this kind of race and protect the affected segments while introducing the parallel regions, we might also add a non-deterministic bug. And sometimes requiring specific circumstances to happen, this kind of bug could be quite hard to detect. This is the reason why, even though we could theoretically achieve a greater speedup with more parallel regions, we decided to take a simplified path focusing on a single region. Of course this kind of problem can still ocour as we cannot avoid splitting the critical section somehow if we want to perform any segment in parallel. However, by choosing a single region to tackle for now, we can focus on a smaller change set and mitigate the risks of introducing such bugs. As previously said, future incremental improvements can then be made to add more parallelism.

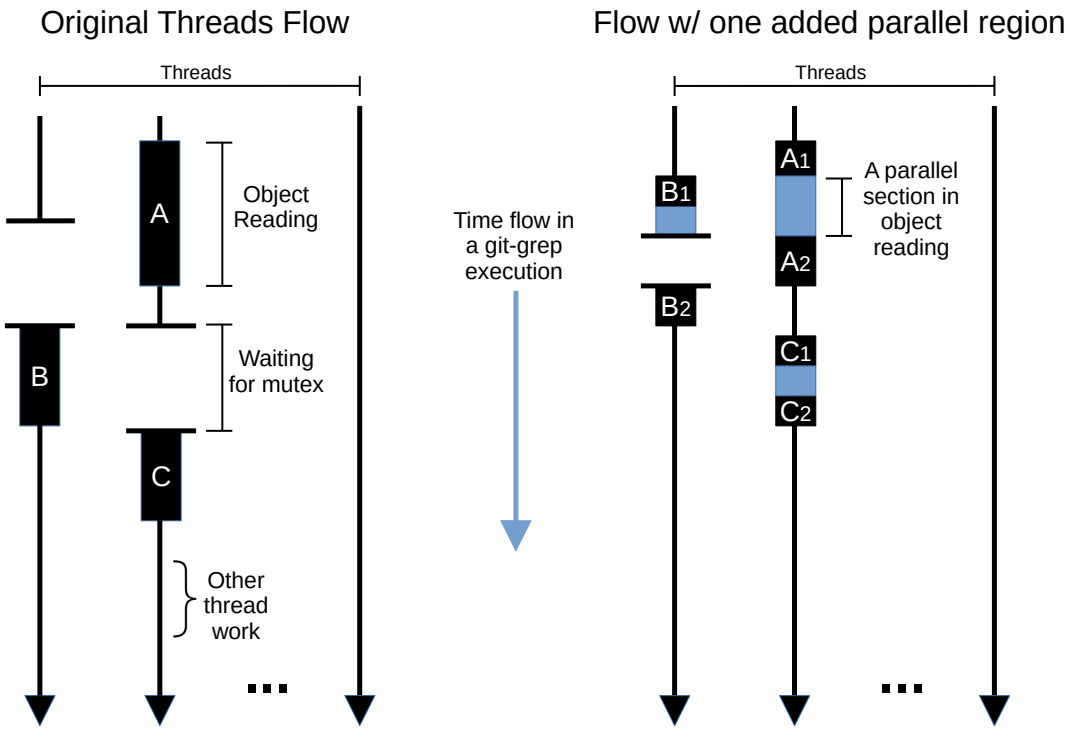

Figure 6.1: *Diagrams representing the* `git-grep` *threads flow. The left side corresponds to the original code. The right side corresponds to a modification adding one parallel region inside object reading. Note how, in the right,* B_1 *might execute between* A_1 *and* A_2*, possibly changing global variables and breaking invariants that were previously ensured in the original code.*

6.2 Next Approach: Parallel zlib Decompression

The next approach was to select a single section to execute in parallel. By the results from Chapter 5, zlib decompression was the obvious candidate. So we would continue protecting the whole object reading machinery but leaving zlib decompression unprotected, to run in parallel (and taking care to avoid the possible races described in Section 6.1). To do so, we would replace `git-grep`'s `grep_read_mutex` with an `obj_read_mutex` inside object reading, that would be released when performing decompression. One of the extra benefits of adding this lock is that the parallel access to object reading could then be used by other commands as well, without the need of using external locks. More explicitly, this `mutex` would be:

1. Acquired at the beginning of `read_object_file_extended()` (which, then, seemed to be the topmost object reading function in `git-grep`'s calls);
2. Released before performing decompression, i.e. a call to `git_inflate()`, anywhere in `read_object_file_extended()`'s call tree (to do it in parallel);
3. Reacquired right after; and

4. Finally released again in the end.

With adjustements, this general idea was what ended up being implemented in the end. Being so time consuming, parallelizing only decompression already brings a great speedup while promoting a somehow controlled change set. It's important to mention that adding the said mutex could slow down the sequential code that uses the object reading API (because of the locking/unlocking overhead). So instead of just blindly adding the lock, we provided some extra functions for the API users to enable or disable the internal lock usage as needed.

One of the first challenges of replacing the `grep_read_mutex` was the fact that it was not only protecting object reading operations but also other operations that could be in data race with them. So removing this lock and protecting just the former wouldn't be enough. These protected non-object-reading operations were mainly executed by the code related to the options `--recurse-submodules` and `--textconv`. The former is used to propagate the search to the submodules[1] of the repository. And the latter to use the text conversion[2] options for the paths with the respective attribute in the `.gitattributes` file. Once more, to honor the idea of making small incremental changes, we decided not to deal with this problem in the first implementation, leaving it for a second version (or patches on top of the first version). Of course we also couldn't just ignore the problem and let `git-grep` fail. So we disabled threads whenever any of these two options were given for an object store grepping, making the added improvements only affect the code outside these options, for now.

6.2.1 Race Condition at Delta Base Cache

The early tests for this first version, using the `gentoo` repository as data, showed a promissing time reduction. But when trying to test with larger repositories such as `linux` or `chromium` the code failed with a `Segmentation Fault` error. Using `GDB`[3] we found out that the error came from an invalid access in the delta base cache. Although this global cache was protected by the added object reading lock, this could be the result of a race condition in the format illustrated in Figure 6.1.

To understand the problem, we have to understand first how this LRU cache is implemented. It must provide the basic operations of insertion, retrieval and removal. Besides, it must keep track of which entries were least recently used, so that they can be removed when more space is needed. To satisfy these requirements and keep a good time complexity,

[1] https://git-scm.com/book/en/v2/Git-Tools-Submodules
[2] https://git.wiki.kernel.org/index.php/Textconv
[3] https://www.gnu.org/software/gdb/

the cache is implemented with the conjuntion of a hashmap and a doubly linked list. The former is used to fastly get entries based on a key, which is a (packfile, offset) pair, as described in the Chapter 4. And the latter is used to keep track of the LRU order: new entries are always added in the tail; and everytime an entry is used, the API expects its user to remove and re-insert the entry to update the list order.

In Git, both the hashmap and the linked list are implemented as intrusive data structures. This means that the data structure and its payload are constructed in the same composition. In the list, for example, "links are embedded in the structure that's being linked" (Yerburgh, 2019). Some advantages of this method, as described in the just quoted blogpost, are less memory allocations and caching thrashing. With that in mind, the definition of a delta base cache entry ends up as being something like this:

Program 6.1 Simplified structure of a delta base cache entry.

```
1   struct delta_base_cache_entry {
2       void *data;
3       struct delta_base_cache_key key;
4       struct hashmap_entry h;
5       struct list_head l;
6   };
7
8   struct delta_base_cache_key {
9       struct packed_git *packfile;
10      off_t offset;
11  };
12
13  struct list_head {
14      struct list_head *next;
15      struct list_head *prev;
16  };
17
18  /* Hashmap entries are stored in a 'struct hashmap_entry *' array where
19     the index is given in function of their hashes. */
20  struct hashmap_entry {
21      unsigned int hash;
22      /* entries with same hash are stored in a linked list rooted in
23         the array position they would fall in. */
24      struct hashmap_entry *next;
25  };
```

Note that, with the above definiton, we only handle a `struct list_head` when traversing the list. But it's, in fact, possible to retrieve the associated `struct delta_base_cache_entry`

making use of the `container_of()` macro. This very handy operation, gives us the address of the struct from one of its fields (see Yerburgh, 2019 for more info). The hashmap usage might seem a little counterintuitive at first, but it is, in fact, quite similar: to retrieve an entry, we initialize and pass a `struct hashmap_entry` and a `struct delta_base_cache_key` to the hashmap retrieval function. The former is used to tell the hash of the entry we are looking for and the latter to disambiguate between entries with the same hash (this implementation also allows the insertion of two entries with the same hash and key, in which case just the first found is returned). The function returns a `struct hashmap_ent` which can, then, be passed to `container_of()` to get the container cache entry. Note that the given `struct hashmap_entry` and `struct delta_base_cache_key` might not be the ones that were declared inside the `struct delta_base_cache_entry` we are looking for! In fact, in the common case it is not, because if it were, we wouldn't need to fetch the entry, at first place.

With the basic knowledge of how the cache is implemented, let's get back to the race condition problem. Using `Valgrind`[4] with `memcheck` to debug our previous `Segmentation Fault`, we discovered that a thread was trying to access the `struct hashmap_key` of an already free'd `struct delta_base_cache_entry`. But how could that be possible? Firstly, we have to remember the code from the `unpack_entry()` function (described at the Pseudocode 4.1). This function is present in the object reading call chains, so a thread would have the `obj_read_mutex` once it get to this code. However, every uncompress operation on this function would be performed in parallel (by releasing the lock right before and re-acquiring right after). This way, we opened some holes in the critical section in which this function was inserted. Thus, it is no longer guaranteed that the function will run from top to bottom uninterruptedly. Instead, it might be called again by other thread before the previously started execution ends. If this function is not reentrant, we might have just introduced an error. And in fact, that's exactly what happened. Allowing the above situation to happen, we also allowed for the same key to be accidentally added twice in the cache, through a race condition. This can happen, for example, in this case:

1. Thread A is performing the decompression of a base O (which is not yet in the cache) at PHASE II. Thread B is simulteneously trying to unpack O, but just starting at PHASE I.

2. Since O is not yet in the cache, B will go to PHASE II to also perform the decompression.

3. When they finish decompressing, one of them will get the object reading mutex and go to PHASE III while the other waits. Let's say A got the mutex first.

[4] http://valgrind.org/

4. Thread A will add O to the cache, go throught the rest of PHASE III and return.
5. Thread B gets the mutex, also add O to the cache (because it thinks O wasn't there yet) and returns.

Although the hashmap implementation allows the insertion of two entries with the same key, something odd will happen in this current situation. If we try to add another entry to the cache and there is no space for it, the code will traverse the LRU list removing entries until there is sufficient space. The removal action includes three operations: remove the respective `struct hashmap_ent` from the hashmap, remove the respective `struct list_head` from the list and, finally, free the container `struct delta_base_cache_entry`. To remove the list entry, we just have to redirect the pointers from the `prev` and `next` nodes. But to remove the hashmap entry is a little more complex. Just like the insertion operation previously explained, for removal we also have to pass the `struct delta_base_cache_key` and `struct hashmap_ent`. This time, though, we don't need to initialize new instances, we just give the ones from the `struct delta_base_cache_entry` we have (the one we got from the `container_of()` call with a `struct list_head`). The hashmap removal function will then go through the same procedure of finding a stored `hashmap_ent` with the same hash of the `hashmap_ent` we gave and, possibly, disambiguating with the given key. And here is where our problem resides: having two `struct hashmap_ent`'s associated with two `struct delta_base_cache_key` that contains the same value makes it possible to remove the wrong one. But even worst, we end up keeping the one we did wanted to remove. And since the container struct of the left entry is free'd, a later fetch in the hashmap might try to read an address in a free'd memory section, leading to the `Segmentation Fault`.

Fortunately, although the process required to locate and understand the error was quite complicated, the solution was really simple. We just had to check if an entry was already in the cache before insertion, and skip redundantly inserting it again if so. At this point we had `git-grep` executing with parallel decompression, good threaded speedup and aparently no race conditions. So the first version of the patch set was sent for review. It was composed of:

- **[1/4]**: object-store: add lock to read_object_file_extended()
  https://public-inbox.org/git/052de4c139bf4962182e6cb8f4aa315aa6130124.1565468806.git.matheus.bernardino@usp.br/
- **[2/4]**: grep: allow locks to be enabled individually
  https://public-inbox.org/git/235de7de2874bd089b106be75121e1616308ed55.1565468806.git.matheus.bernardino@usp.br/
- **[3/4]**: grep: disable grep_read_mutex when possible

https://public-inbox.org/git/d2e3f4eac24d26210f8962ebd82fd24a99c91fdf.1565468806.git.matheus.bernardino@usp.br/

- **[4/4]**: grep: re-enable threads in some non-worktree cases
  https://public-inbox.org/git/8c26abe9156e069ad4d19e9f0ce131cd1453f030.1565468806.git.matheus.bernardino@usp.br/

This series' cover letter can be seen at: https://public-inbox.org/git/cover.1565468806.git.matheus.bernardino@usp.br/.

6.2.2 Dealing with `--textconv` and `--recurse-submodules`

With the first version working, the next step was to take care of the code related to the `--textconv` and `--recurse-submodules` options. As previously mentioned, we disabled threads in the object store `git-grep` whenever any of these two options were used, because without `grep_read_mutex` (that was replaced for the internal `obj_read_mutex`) the code related with those options would be in race conditions with object reading. For a second version, though, it would be nice if `git-grep` could also safelly perform threaded with them and take advantage of parallel decompression. To do that, we first tried to follow the call chains originated at the functions related to these options to find and protect the places that operated on global states also read/modified by object reading.

One of the first things we noticed was that, considering the submodules code, `read_object_file_extended()` wasn't the topmost object reading call in `git-grep`, as we previously expected. So we would have to move the `obj_read_mutex` down in the call chain in order to have all calls protected. As described in Section 4.3 `oid_object_info_extended()` is the common back-end for all main object reading entry points (besides being one itself). So it would make sence to move the `obj_read_mutex` down to this function. And if the other entry points were already thread-safe except by their calls to this one, using the lock at it would make all of them thread-safe.

This hypothesis would be correct if it weren't for two small sections in `read_object_file_extended()` that would become thread-unsafe if removing the mutex from it. Fortunatelly, we could just keep using the mutex in this function, but now leaving the segment that would lead to `oid_object_info_extended()` out of it (because this function would already lock the said mutex internally, to protect its code). That is almost what we did. For one of those two sections, we could use a better approach: it was not thread-safe because of a call to `lookup_replace_object()` that, in turn, calls the lazy initializer `prepare_replace_object()`. Both of these functions were part of the `replace-object.h` API. And inspecting the code, we found out that it would be pretty

simple to make this API thread-safe with a single extra mutex. This was great because we were reducing the critical section taken care by the `obj_read_mutex` and allowing more work to be done in parallel. With these changes done, all the previously mentioned object reading entry points could now be called in threaded code.

We were making good progress with the manual call chain analysis, but it was a slow process. So seeking a faster method, we tried a more empirical approach. We crafted an artificial repository with two reasonably sized submodules and forced many text conversions, marking the `cat` command as the text conversion tool for every C file. The idea was that running `git-grep` on this repo with the above options we could hopefully see and correct the possible race conditions. Then we would repeat the process until no more race conditions were visible. And to detect them, we once more used `Valgrind`, but this time with other tools: `Helgrind` and `DRD`. Unfortunatelly, though, `Valgrind` makes the program execution very slow. So `ThreadSanitizer` was also used, by compilling Git with `-fsanitize=thread`.

With this process, we got to a version where both options could be used with threads and neither `Valgrind` nor `ThreadSanitizer` would accuse race conditions. But unfortunatelly, `--textconv` would run pretty slow. Besides, we realized this empirical process wouldn't be safe enough to use in production code. Just because we couldn't spot race conditions with these tools, it didn't mean we were really free of them. So we got back to manually looking the call chains. But this time, we focused on the code related to a single option at a time (once again, striving simple small improvements), starting with the `--recurse-submodules`. This was easier as the code was confined in the `grep_submodules()` function from `builtin/grep.c`. The calls previously protected by the `grep_read_mutex` in this function were: `is_submodule_active()`, `repo_submodule_init()`, `repo_read_gitmodules()`, and `add_to_alternates_memory()`. So, in order to keep the same behavior, we just had to protect the code, somehow, against concurrent calls of any of these functions and also against concurrent calls of these functions and object reading ones. But to do so, we would have to understand why they could conflict with object reading. That's when we came across this commentary in `grep_submodule()`:

```
/*
 * NEEDSWORK: submodules functions need to be protected because they
 * access the object store via config_from_gitmodules(): the latter
 * uses get_oid() which, for now, relies on the global the_repository
 * object.
 */
```

Unfortunatelly, though, it seemed to be a little outdated, since the said func-

tion no longer used `get_oid()`. And inspecting the code we saw that it now called `add_to_alternates_memory()`, which adds the submodule's object directories to the in-memory alternates list. Basically, this is a list of additional directories from which Git should "borrow" objects if they are not found in the main repository being processed (represented by the global `struct repository *the_repository` variable). There is a persistent on-disk version of the list, and a non-persitent in-memory version. In either case, the entries are loaded in memory to a linked list at `the_repository->objects->odb`. Because this list is global and it might be used by object reading functions, the concurent call to `add_to_alternates_memory()`, which writes to the list, could cause data races. Additionally, there was another explict call to this function in `grep_submodules()` with this commentary:

```
/*
* NEEDSWORK: This adds the submodule's object directory to the list of
* alternates for the single in-memory object store.  This has some bad
* consequences for memory (processed objects will never be freed) and
* performance (this increases the number of pack files git has to pay
* attention to, to the sum of the number of pack files in all the
* repositories processed so far).  This can be removed once the object
* store is no longer global and instead is a member of the repository
* object.
*/
```

Removing these two uses of `add_to_alternates_memory()` would help reducing the necessary critical section in `grep_submodules` besides apparently improving performance and memory usage. We tried this idea and, once more, made good progress, but there were some complications. The worker threads were always working in `the_repository`, but they could find the submodules' objects exactly because they were added to `the_repository`'s in-memory alternates list. If we were to remove the second `add_to_alternates_memory()` call, we would have to pass on the `struct repository` of the submodules to the worker threads and adjust their code to use it instead of `the_repository`. It might seem simple at first, since many functions called by the threads already take a `struct repository` parameter. But there're also many functions which don't, and just use the global `the_repository` internally, so we would have to make them capable of working on arbitrary repositories, first.

Besides, we would have to find another way of handling the `repo_clear()` call for each submodule. This function is responsible for freeing the associated resources of a given `struct repository` and it's currently being called by `grep_submodules()` right before

returning. It can do so because the submodules' object directories were already added to the global alternates list at that point, so the submodule struct is no longer needed. But once we stop adding them to this list and start passing the subrepo reference on, we would have to rethink where to perform the cleanup. Probably we would have to implement a task counter mechanism to know when all tasks in a submodule were completed to safelly free it. These complications made us consider that the required changes could, potentially, became too complex for a single patch set. So we decided it would be better to try this improvement of removing the `add_to_alternates_memory()` calls in another future patch set, and try to solve our current problem on this one through simpler paths. Also, as we later found out, the submodule functions also call object reading functions internally. So the above changes still wouldn't be enought to make `git-grep --recurse-submodules --cached` benefit from parallel decompression without race conditions.

That's when the main idea for the version 2 of the patch set came out. To recap, the problem we were facing was as follows: `grep_read_mutex` used to protect two sets of operations: object reading calls and other function calls that could conflict with object reading. We replaced `grep_read_mutex` by the internal `obj_read_mutex`, but it only protected the first set. One possible solution to maintain the previous behavior would be to expose this mutex externally, and use it to protect the second set, as well. But these two sets were in different abstraction layers, and the second set even contained calls to functions from the first. So how could we protect them both with the same lock, without getting relocking errors? Searching the threads API, we found the recursive mutex implementation. The specification says:

> "A thread can relock a recursive mutex without first unlocking it. The relocking deadlock which can occur with normal mutexes cannot occur with this type of mutex." - IEEE and The Open Group, 2018

This is exactly what we needed. So we turned `obj_read_mutex` into a recursive lock and created a public API to lock and unlock it. The idea of this API was to expose the lock externally to be used to protect places that cannot execute concurently with the object reading machinery. The API then guarantees mutual exclusion between this machinery and the code being protected. These lock and unlock functions were used to wrap the code related to `--recurse-submodules` and `--textconv`.

It's important to highlight that the usage of the recursive mutex comes with a cost. The specification says that "multiple locks of a recursive mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex." (IEEE and The Open Group, 2018) In our case, this means that, when the lock is used externally to protect sections that call object reading functions internally, those won't run decompression in

parallel. This happens because, during decompression, the thread would still be holding the mutex with a lock counter of at least one. However, expecting that the areas we would use this mutex outside object reading wouldn't be many (nor much time consuming ones), it would still be worthy to use this method. It also solved another problem we didn't previously noted when implementing the version 1 of this patch set: the call graphs originated at `read_object_file_extended()` contain recursive calls to this function. So we already had the risk of relocking problems. (These recursive calls were found using a call graph tool which will be mentioned in Subsection 6.2.3.)

6.2.3 Analysing Call Graphs

At this point, it seemed we were very close (if not already at the point) of having thread support for both `--recurse-submodules` and `--textconv` with good performance and no race conditions. We did wanted, however, to make sure all code was safelly protected before re-enabling threads in all `git-grep` cases. But manual analysis was getting too laborious. So we decided to try some tools to generate the code call graphs for us, so that we could later filter out the paths we know to be protected and look for unprotected ones left behind. We tried cally[5], GNU cflow[6] and even writting a custom tool using `GDB` in the back, called callpath[7]. But none of them were able to give the desired output. They either showed only paths of an specific execution; or struggled with compile-time macros; or couldn't handle multiple valid functions with the same name (in different files). This problem caught the attention of Giuliano Belinassi, a graduate student at the University of São Paulo and `GCC` contributor. Seeking to help, he wrote a patch that made GCC dump the call graph for the program being compiled, during the Link Time Optimization pass (see Annex A). His implementation overcomed all of the previous problems we had with other tools.

Compilling Git with the patched GCC, would generate a file containing the call graph in dot format[8]. Then we wrote a python script to read the file and filter only the paths from a given set of functions to another. There was also an option to exclude paths going through certain nodes. With the filtered output, we tried to find paths departing from `cmd_grep()` (the function responsible for the `git-grep` command) and `run()` (the start routine for the worker threads) and leading to any of the thread-unsafe functions we knew to be present in object reading's call chains. For example, `parse_object()`. The generated call graphs were huge. But incrementally eliminating paths we knew to be thread-safe (either because

[5] https://github.com/chaudron/cally
[6] https://www.gnu.org/software/cflow/
[7] https://github.com/matheustavares/callpath
[8] https://www.graphviz.org/doc/info/lang.html

they were executed before thread spawning or because they were protected by locks), we were left with much smaller subgraphs. These would then indicate the paths that were unprotected and, thus, racy. This technique turned out to be very helpful in the process of ensuring that we had all call paths protected before re-enabling threads in the object store case.

More than that, this technique has allowed us to find some already threaded sections in `git-grep`'s code (prior to our modifications) that could, in fact, lead to race conditions; even for executions of the working tree grep. As an example, we have the following call chain (which is performed by the producer thread after the consumers have already been spawned):

```
cmd_grep() > grep_objects() > deref_tag() > parse_object()
```

Like the above one, we found other five racy spots in `git-grep`'s current code: the call to `gitmodules_config_oid()` again in `grep_objects()`; two function calls in `grep_submodule()` there were outside the critical section of the previous `grep_read_mutex`; and the calls to `userdiff_get_textconv()` and `userdiff_find_by_path()`. These last two were guarded by the `grep_attr_mutex` but their call trees had object reading operations, so they should be guarded by `grep_read_mutex` as well, to avoid races with the spots protected by this lock. The fixes for these six race conditions were perfomed in the first three patches of this series' second version: (the patches' links and the full patch list for this series' most recent version will be shown later in this chapter)

- **[1/11]**: grep: fix race conditions on userdiff calls
- **[2/11]**: grep: fix race conditions at grep_submodule()
- **[3/11]**: grep: fix racy calls in grep_objects()

These patches were sent as the first ones in the series because they strive to fix possible problems in the current code. Because of that, they still used the `grep_read_mutex`. In the following patches, though, when we replace this mutex by the internal `obj_read_mutex`, we also took care of using the added API for this lock to protect this places. One of them no longer needed the external protection. `userdiff_find_by_path()` was only thread-unsafe because of the previously thread-unsafe object reading. Now that the latter became safe, so did the former. Finally, all other non-object-reading function calls that used to be surrounded by the `grep_read_mutex`, were also protected by `obj_read_mutex` in these following patches.

The possibility to generate call graphs permitted us to also see a call path that wasn't racy but could easily become, with future changes to the codebase. We are talking about

the `prepare_packed_git()` lazy initializer at `packfile.c`. It is used to initialize some fields in a `struct repository`. Although this function is present in the call stack of `git-grep` threads, all paths to it are currently protected by the `obj_read_mutex`. Besides, the main thread usually indirectly calls this initializer before firing the worker threads, so there's no risk. However, since this lazy initializer also works like an initialization checker, it is used in many places. Therefore, future modifications in the codebase could easily accidentally allow paths to the initializer from `git-grep`'s code, introducing a race condition. So to prevent future headaches, we also forced this initializer to perform eagerly when setting git-grep up. The downside is that we lose the feature of only performing intialization when (and if) needed. But the overhead added in the cases where it wasn't needed shouldn't be very noticeable. The initializer is also called at `reprepare_packed_git()`, which, as the name suggests, performs reinitialization. This function is also present only in protected paths inside `git-grep`'s code but it could also easily become a problem in the future. So again, to avoid headaches, we protected its code with the `obj_read_mutex`.

6.2.4 Allowing More Parallelism on Submodules Functions

With the object reading functions protected, the submodule initialization calls at `grep_submodule()` were pretty close to being thread-safe. So seeking an even better performance, we tried to make the small changes needed to remove some of these calls from the critical section (allowing more parallel work). The `repo_submodule_init()` call was already thread-safe, so we just removed it from the critical section. The `submodule_from_path()` and `is_submodule_active()` calls still needed to be protected only because they call `repo_read_gitmodules()` which contains, in its call tree, operations that would otherwise be in race condition with object reading (for example calls to `parse_object()` and `is_promisor_remote()`). The objective of `repo_read_gitmodules()` is to read the repository's `.gitmodules` file. So we could force an eager reading of this file during `git-grep`'s setup, and make `submodule_from_path()` and `is_submodule_active()` skipping calling the thread-unsafe function if the file was already read. This way, we could safelly move those two calls out of the critical section. This was done through the following two patches: (again, these patches' links will be presented later in this chapter)

- **[7/11]**: submodule-config: add skip_if_read option to repo_read_gitmodules()
- **[8/11]**: grep: allow submodule functions to run in parallel

Some calls to submodule functions couldn't be removed from the critical section because it would require more complex changes (which would be better done in their own patch set). For these, a commentary was left in the code mentioning what needs to be done for such a future improvement.

6.3 Additional Improvements to git-grep

At this point, we already had threads re-enabled for git-grep's object store case with a good speedup, as we will see in the next chapter. But before that, in this section, we will go over two additional improvements made to the code of git-grep during this project: the first is also related to performance, reducing the code protected by grep_mutex to increase the parallel work; the second is a fix for a bug reported in the mailing list regarding sections of the code we were already working on.

6.3.1 Removing Thread-Safe Code from Critical Section

The function add_work() at builtin/grep.c is called by the producer thread to add a new task for the consumer threads. While studying this code to re-enable threads in the object store case, I noticed that this function calls grep_source_load_driver(). The latter is used to load the userdiff driver for a given file or blob in the repository.

A diff driver is a set of options that can be enabled per file in a repository. Initially, it was used to control how diffs would be presented by Git, for specific files in a repository. However, as the Git project evolved, diff drivers started to be used in other contexts as well. Between other options, a driver might specify, for example: an external command to run instead of Git's internal diff algorithm; a command to run in the affected files to perform text conversion on them; and whether to treat the affected files as if they were binaries. Git already ships some pre-defined drivers such as the python driver and tex driver, suitable for the respective languages. But users might define their own drivers in one of the Git configuration files. These user defined diff drivers are what we intuitivelly call "userdiff drivers". To use a driver (user defined or pre-shipped), the user must specify for what files they want the driver to be considered. And this is done in the .gitattributes file of a repository. A typical use case would be to include a line like this: *.tex diff=tex.

Between other usages, the diff driver is required by git-grep to know for which files it must perform text conversion (when --textconv is given), and to decide which files are binary (when --text is not given, which makes git-grep treat all files as text). In the last case, if a file doesn't have an associated diff driver (or the driver doesn't set the binary option), Git will fall back to manual detection. In the past, the diff driver loading would be performed by the worker threads. But they would load the drivers in a non-deterministic order while the underlying code was optimized to handle paths in sequential order. So the code was changed to pre-load the driver by the main thread in commit 9dd5245 ("grep: pre-load userdiff drivers when threaded", 2012-02-02). To do that, a call to grep_source_load_driver() was added to add_work(). However, this call is being

performed inside the critical section of the `grep_mutex`. And, by what we saw in Figure 5.4, this is a relativelly time consuming function. So it could probably be promoting lock contention for the consumer threads. Since `grep_source_load_driver()` is already thread-safe, the call can be safelly moved out to allow more parallel work and, hopefully, help reducing the large ammount of time spent in locking functions (as also seen in Figure 5.4). The change is shown at Patch 6.2, which is part of this project's main patch set (links will be provided later in this chapter).

Program 6.2 Patch moving the thread-safe `grep_source_load_driver()` call out of the critical section, for better performance.

```
1   diff --git a/builtin/grep.c b/builtin/grep.c
2   index 163f14b60d..d275b76647 100644
3   --- a/builtin/grep.c
4   +++ b/builtin/grep.c
5   @@ -92,22 +92,22 @@ static pthread_cond_t cond_result;
6
7    static int skip_first_line;
8
9   -static void add_work(struct grep_opt *opt, const struct grep_source *gs)
10  +static void add_work(struct grep_opt *opt, struct grep_source *gs)
11   {
12  +       if (opt->binary != GREP_BINARY_TEXT)
13  +               grep_source_load_driver(gs, opt->repo->index);
14  +
15          grep_lock();
16
17          while ((todo_end+1) % ARRAY_SIZE(todo) == todo_done) {
18                  pthread_cond_wait(&cond_write, &grep_mutex);
19          }
20
21          todo[todo_end].source = *gs;
22  -       if (opt->binary != GREP_BINARY_TEXT)
23  -               grep_source_load_driver(&todo[todo_end].source,
24  -                                       opt->repo->index);
25          todo[todo_end].done = 0;
26          strbuf_reset(&todo[todo_end].out);
27          todo_end = (todo_end + 1) % ARRAY_SIZE(todo);
28
29          pthread_cond_signal(&cond_add);
30          grep_unlock();
31   }
```

With this simple change, we got time reductions of up to 34% in some test cases. The full timings and results will be shown in Section 7.1.

6.3.2 Bugfix in submodule grepping

On July 8th, a Git user reported an undesired behavior with `git-grep` and submodules, in the Git mailing list (Zaoui, 2019): Even when `--cached` was not used and no revision was given, `git-grep --recurse-submodules` would still grep the submodules' objects and not their working trees. Since the problem was in the same code I was working on, I decided to take a look and try to fix it during this project. After successfuly replicating[9] the reported situation, the next step was to confirm that it was indeed a bug, and not an intentional behavior change. This was done looking for a past commit in which the code was working as the user expected and, then, the commit that changed this – which was `f9ee2fc ("grep: recurse in-process using 'struct repository'", 02-08-2017)`. Since `f9ee2fc` didn't mention about the behavior change, it was probably indeed a bug, introduced during the process of converting the previous `--recurse-submodules` code to run in the same process instead of spawning new ones.

Analysing the code, the problem was found at this line inside `grep_submodule()`, the function responsible for propagating the search to the submodules: `hit = grep_cache(&subopt, pathspec, 1)`. The function being called, although called `grep_cache()` is also used to grep in the working tree. It is named like that because the function traverses the cache (in fact, the Git index), to get the paths of the files Git is currently tracking, and then dispatch the search on them. To control whether this search should read the working tree files or the respective objects, `grep_cache()`'s receives a boolean as its third parameter, named "cached". At the line of code previously quoted, `grep_cache()` is always called with `cached=1`, thus, ignoring the working tree even when it shouldn't. To fix that, an aditional cached parameter was added to `grep_submodule()` so that it would known the desired behavior and call `grep_cache()` properly. To avoid possible future regressions, two test cases were also added to Git's test base, testing if `git-grep --recurse-submodule` respects the presence and absence of `--cached` in submodules. The final patch containing the bugfix and test additions was already merged into master, being part of the Git release 2.24.0:

- **[1/1]**: grep: fix worktree case in submodules
  https://public-inbox.org/git/ba3d8a953a2cc5b4ff03fefa434ffd7bd6a78f15.1564505605.git.matheus.bernardino@usp.br/

[9] With the series of commands presented at: https://matheustavares.gitlab.io/posts/week-10-a-bug-in-git-grep-submodules#grep-submodules-bug-ignoring-worktree

6.4 Current State

At the time this document is being written, the most recent iteration of this project's main patch set is version 3. The reason why we have been focusing on describing the second version, though, is that the main code changes were presented in it. Version 3 stayed pretty similar, only adding a few extra improvements to documentation and code. Although these new changes are very important, they are much simpler, so there should be no problem going through them at a faster pace. The new changes are:

1. Fixed typos in version 2's commit messages.
2. Added a patch to adjust the default number of threads in `git-grep` according to the number of logical cores available. (Previously, it would always use eight threads when the user did not specify otherwise. This fixed number could be too many for machines with fewer cores, and too little for machines with more cores.)
3. Added documentation about how we ensure thread-safeness for the reading of packed objects (more on this later).

The last item is probably the most important, so it deserves some additional information. During recent reviews of the patch set, some concerns were raised about the possibility of a race condition when reading packed objects with multiple threads (Tan, 2019). To understand the problem, we will need to go into some context first: When reading from a packfile, Git does not read the whole file at once, as it can be larger than the available memory. Besides, it would be wasteful as typically only a small section of the packfile is needed. So, instead, the file is mapped into memory using `mmap`[10], which works by fetching the file's content on demand, while exposing it as the whole file was already available in memory. Because of the way this operation is internally implemented, though, files over 4GB cannot be mmapped entirely in 32-bit architectures. To solve that, the Git community came up with the idea of subdividing the process, and mmaping only what is needed at a time. This mmapped sections of a packfile became known as pack windows.

To decompress a packed object, Git will open and pass a pack window as input to the zlib decompression function. This window must remain valid throughout the whole operation. However, since decompression would now be performed in parallel, we have to ensure that no other thread will invalidate the window, in the meantime[11]. In particular, window disposal can happen for several reasons, such as reaching the maximum number of opened windows.

[10] http://man7.org/linux/man-pages/man2/mmap.2.html

[11] In regarding to the reading of loose objects, the memory sections sent as input to decompression are local to the stack of the executing thread. So there is no risk of they suffering a race condition.

Further investigating the code, however, it was found that the window functions already use a thread-safety mechanism. The C struct that defines a pack window contains a integer field called "`inuse_cnt`". This counter is incremented before window reading operations and checked before window disposal, only proceeding if the value is found to be zero. So no disposal should occur if another thread has already expressed a reading intention.

Another concern regarding threaded pack reading was the concurrent calls to `close_pack_fd()`, which can close packs even with in-use windows. However, as the `mmap()` documentation (Linux man-pages project, 2019) states "closing the file descriptor does not unmap the region". So there should be no problem here, as well.

Finally, we also considered the calls to `reprepare_packed_git()` (see Subsection 6.2.3), which resets some fields in a `struct repository`; specially some related to packfiles. But we found that the function called by this one to handle packfile opening, `prepare_pack()`, won't reopen already available packs. Therefore, the opened windows should remain intact.

Since these are important considerations regarding the thread-safeness of packed object reading, it was suggested to re-roll the series including them. That was what most motivated the creation of version 3. The above explanations were inserted as code comments and/or part of the relevant commit message(s).

6.4.1 Links for the patch set

Without further ado, bellow is the list of patches that composes the third version of this project's main patch set, i.e. "`grep: improve threading and fix race conditions`":

- **[01/12]** grep: fix race conditions on userdiff calls
  https://public-inbox.org/git/e2f3d377f5408d3d9365b8ac1b785d6d3f0437a9.1579141989.git.matheus.bernardino@usp.br/
- **[02/12]** grep: fix race conditions at grep_submodule()
  https://public-inbox.org/git/6f0899701b88e255bae68e16e11a978488c0b1cd.1579141989.git.matheus.bernardino@usp.br/
- **[03/12]** grep: fix racy calls in grep_objects()
  https://public-inbox.org/git/5295c892ee12eb4f8a2fab2cd7e419dc04b18203.1579141989.git.matheus.bernardino@usp.br/
- **[04/12]** replace-object: make replace operations thread-safe
  https://public-inbox.org/git/d7f739bc57b6f59cab7c718300c28b8c6b0a61a8.1579141989.git.matheus.

bernardino@usp.br/

- **[05/12]** object-store: allow threaded access to object reading
  https://public-inbox.org/git/b72e90f229dbf7d5be016fd6251a9b3ef76f2431.1579141989.git.matheus.bernardino@usp.br/
- **[06/12]** grep: replace grep_read_mutex by internal obj read lock
  https://public-inbox.org/git/fc1200bb07f749420dad044d39dfe30ae73ad640.1579141989.git.matheus.bernardino@usp.br/
- **[07/12]** submodule-config: add skip_if_read option to repo_read_gitmodules()
  https://public-inbox.org/git/d39d2ce9c4c4975969a7b99cbe1ee6c8abb586c1.1579141989.git.matheus.bernardino@usp.br/
- **[08/12]** grep: allow submodule functions to run in parallel
  https://public-inbox.org/git/af8ad95d413aa3d763769eb3ae9544e25ccbe2d1.1579141989.git.matheus.bernardino@usp.br/
- **[09/12]** grep: protect packed_git [re-]initialization
  https://public-inbox.org/git/0ccf79ba863a1a512506cc3aae4cc523d64ab8ae.1579141989.git.matheus.bernardino@usp.br/
- **[10/12]** grep: re-enable threads in non-worktree case
  https://public-inbox.org/git/6c09e9169dfb21fc2cd3f69700316d3a87e72019.1579141989.git.matheus.bernardino@usp.br/
- **[11/12]** grep: move driver pre-load out of critical section
  https://public-inbox.org/git/2f72f3034118432381f3c9378e70a65d27e3dfbb.1579141989.git.matheus.bernardino@usp.br/
- **[12/12]** grep: use no. of cores as the default no. of thread
  https://public-inbox.org/git/a5891176d7778b98ac35c756170dd334b8ee21c7.1579141989.git.matheus.bernardino@usp.br/

The cover letter for this version can be found at: https://public-inbox.org/git/cover.1579141989.git.matheus.bernardino@usp.br/. The patches were already merged into the "Proposed Updates"[12] branch of the Git project. They should remain there until it is decided that they are ready to be merged into the `next` branch (and later into `master`), or until any problem is found; in which case a re-roll might be needed.

[12] https://git.kernel.org/pub/scm/git/git.git/log/?h=pu

Chapter 7

Results and Conclusions

7.1 Results

7.1.1 On Grenoble

The final results in `grenoble`, using the tests described at Section 3.2, can be seen in Figures 7.1 and 7.2. These plots compare the execution times of the original code with the code after our improvements. Note that the original code didn't allow threads for object store, but we enabled them with the Patch 1.1, for comparison.

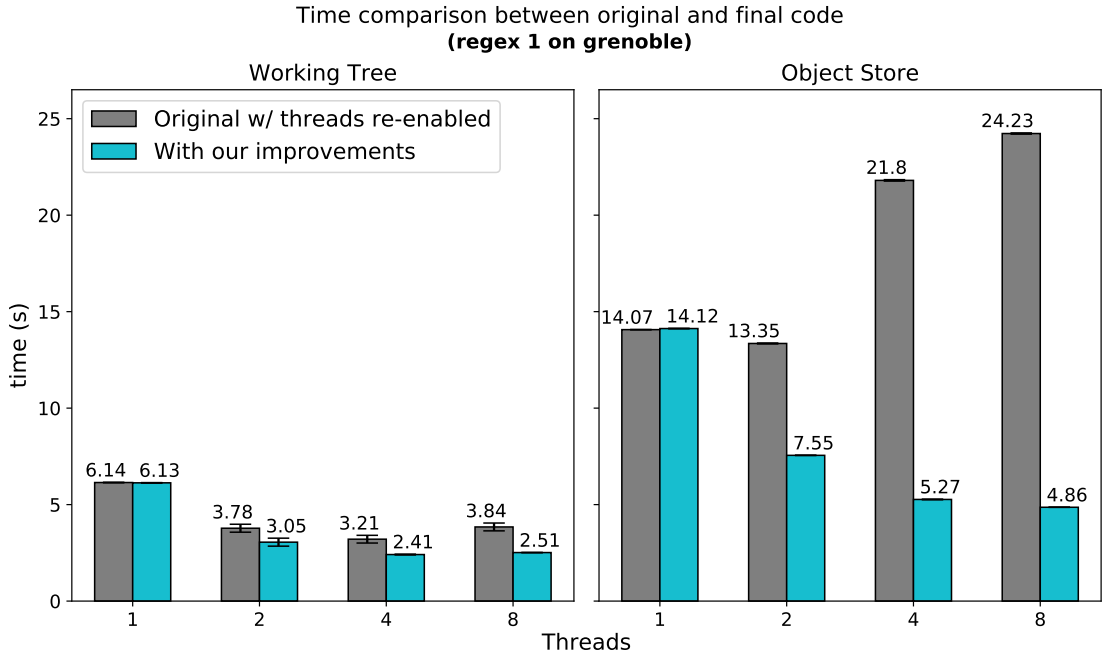

Figure 7.1: *Time comparison between the original git-grep code and the final code with the patches from this project. Tests executed with regex 1 ("abcd[02]") on machine grenoble (see Appendix A).*

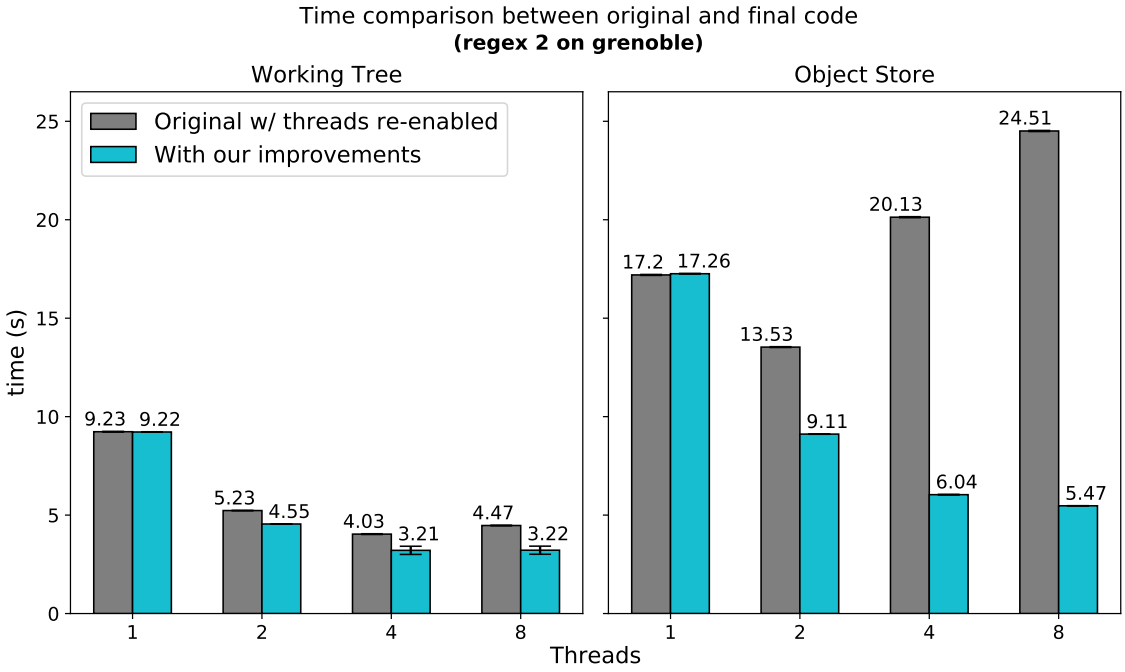

Figure 7.2: *Time comparison between the original git-grep code and the final code with the patches from this project. Tests executed with regex 2 ("`(static|extern) (int|double) \*`") on machine grenoble (see Appendix A).*

These graphs show that the code with out improvements reached speedups of up to 3.15x over the sequential code, in the object store grepping. Compared to the threaded code without parallel decompression, our version reached speedups of up to 4.99x, using the same number of threads. We can also see some time reductions in the working tree grepping, which will be discussed in Subsection 7.1.3.

Observing the big time reductions with 2 and 4 threads in contrast to the sequential code, one might find it odd that the jump from 4 to 8 threads only produced a small performance boost. However, this is totally expected, given the hardware used. Note that both `mango` an `grenoble` have 8 logical cores but only 4 physical ones. The doubled number of logical cores happens through Intel's Hyper-Threading technology, which allows more than one simulteneous thread execution in a single physical core. Although it does boost parallel performance, it cannot reach the full processing power of actually having twice the number of physical cores. And that is one of the reasons why we don't see such a big performance improvement from the executions with 4 to 8 threads in these tests.

7.1.2 On Mango

As previously mentioned, the problem we tackled in this project is very closely related to I/O. So it is also important to validate the performance improvements in machines with

SSD. Figures 7.3 and 7.4 show the results for the same tests of the Subsection 7.1.1 but on `mango`.

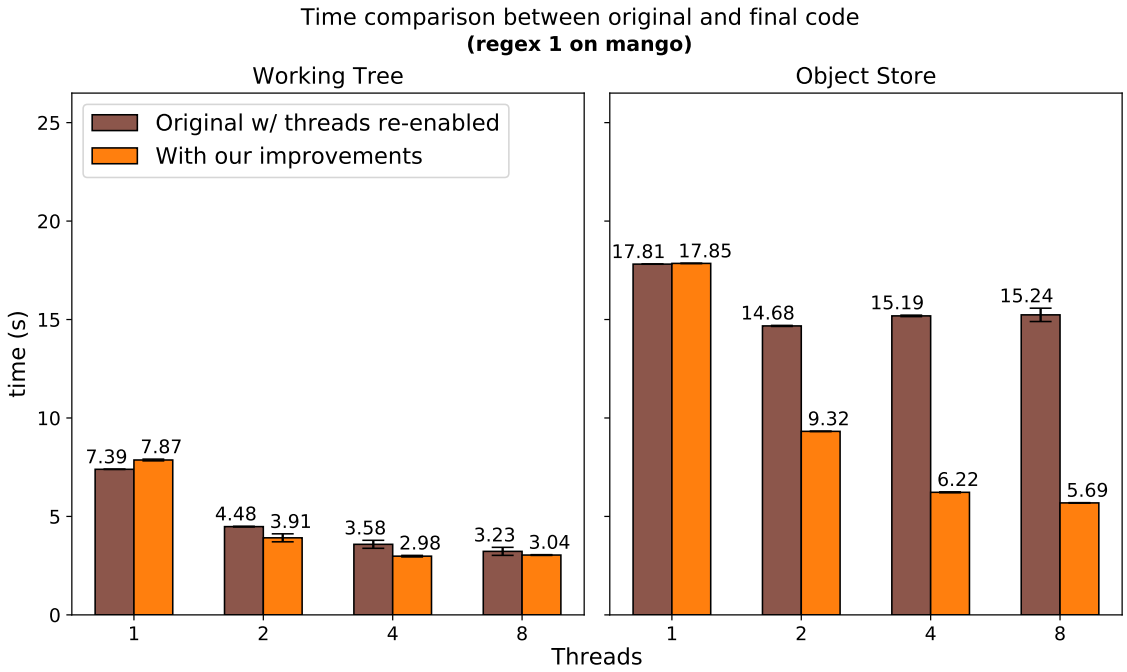

Figure 7.3: *Time comparison between the original git-grep code and the final code with the patches from this project. Tests executed with regex 1 (*"abcd[02]"*) on machine mango (see Appendix A).*

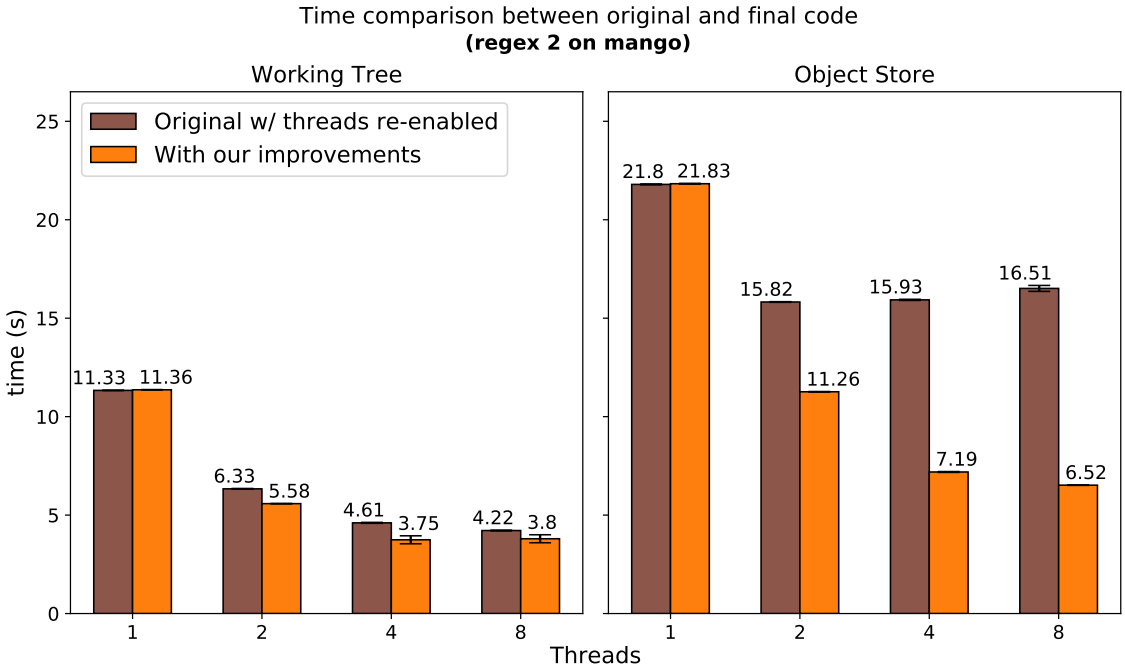

Figure 7.4: *Time comparison between the original git-grep code and the final code with the patches from this project. Tests executed with regex 2 ("`(static|extern) (int|double) \*`") on machine mango (see Appendix A).*

These plots show an even greater speedups over the sequential code, reaching a factor of up to 3.34x. It is also interesting to notice how the addition of threads behaved different between `grenoble` and `mango`, regarding the original code. While the former used to have an increasing slowdown with more threads, the latter used to keep the execution times in a plateau. This could be due to a variety of reasons. One hypothesis is that the SSD allowed for greater performance when threaded reading, in comparison to the HDD. But it could also be other hardware differences, such as the CPUs' clock speeds when working multithreaded. Nevertheless, the improved code managed to successfully harness the available parallel power with good speedups in both machines.

7.1.3 Patch 6.2 and Working Tree Speedup

In both `mango`'s and `grenoble`'s plots, we can see some time reduction in the working tree case, reaching speedups of up to 1.52x over the original threaded code. We didn't really expect to see this difference in this git-grep case, since it normally wouldn't benefit from parallel decompression. And indeed, further investigating the test executions, we saw that neither `oid_object_info_extended()` nor `git_inflate()` were called. So the speedup couldn't be comming from parallel decompression. Therefore, we were lead to believe that it was a result of the critical section reduction patch (6.2).

To measure how this small modification, alone, contributed to the observed performance, we generated the graphs shown in Figures 7.5 and 7.6. Since the said patch was at the tip of the development branch (i.e., it was the last modification made), to observe its contribution, we just had to repeat the tests in the parent commit (which has all the previous changes we made, but not this one). We have choosen to only show the results on `grenoble` since they are sufficient to explain why we observed the said speedup.

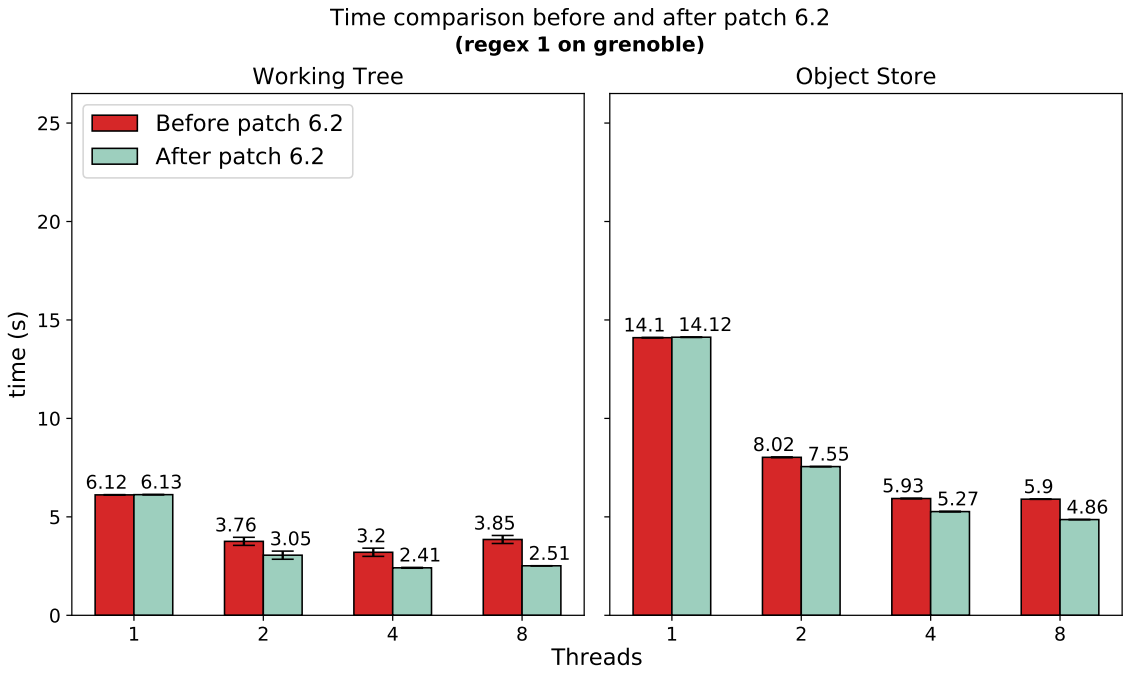

Figure 7.5: *Time comparison of git-grep's code before and after Patch 6.2. Note that the "before" version already has parallel access to decompression. Tests executed with regex 1 ("*`abcd[02]`*") on machine grenoble (see Appendix A).*

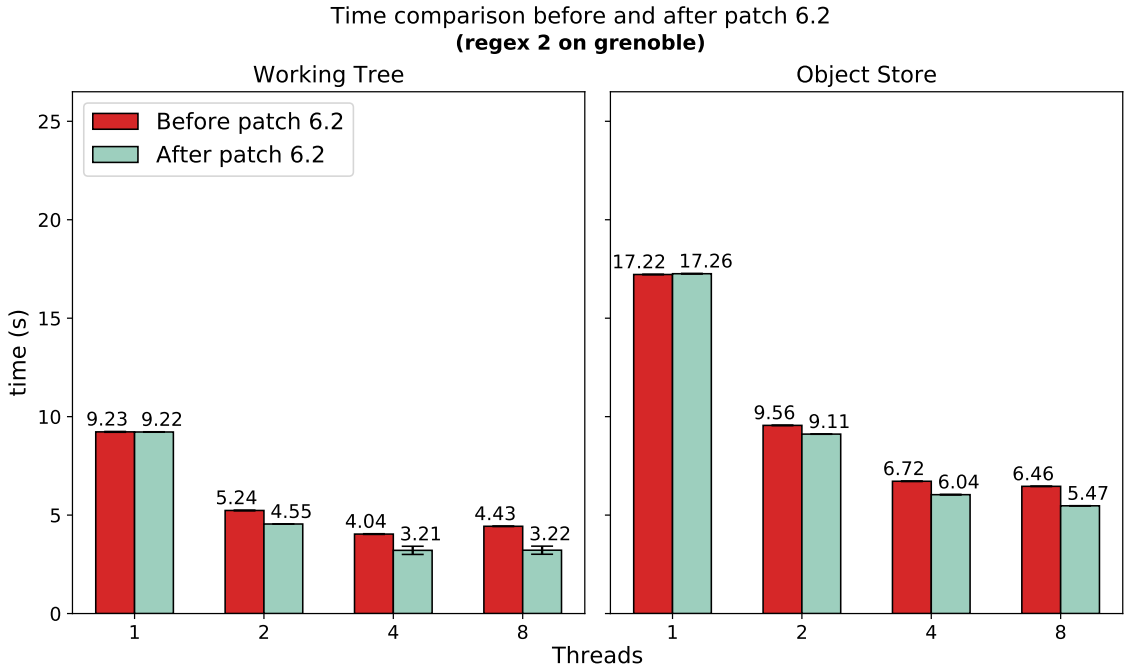

Figure 7.6: *Time comparison of git-grep's code before and after Patch 6.2. Note that the "before" version already has parallel access to decompression. Tests executed with regex 2 ("*`(static|extern) (int|double) \*'`*") on machine grenoble (see Appendix A).*

Comparing Figures 7.4 and 7.6, we can see that parallel decompression without the said patch didn't change the elapsed times for the working tree grepping. Therefore, all the speedup observed in this case (in Figure 7.4) did, in fact, come only from the Patch 6.2. It is impressive how such a small change could be so effective in increasing performance. Alone, it was responsible for time reductions of up to 17% in the object store case, and 34% in the working tree case.

7.2 Conclusions

From the observed results, we can conclude that the proposed changes successfully allowed for thread re-enabling in `git-grep`'s object store case, increasing performance by a large factor: we observed speedups of up to 3.34x over the original code and almost 5x over the threaded code without the improvements. Additionally, the changes also allowed `git-grep` to take even more advantage of parallel processing power in the working tree case as well, with observed speedups of up to 1.53x.

Besides the performance improvements in `git-grep`, the proposed changes can also benefit future optimization projects in Git: Many Git commands and features depend on the object reading machinery which, until now, had to be accessed in a serialized way. But in this project, we created an API that allows thread-safe object reading with

good performance (thanks to parallel decompression). Since object reading can sometimes be a bottleneck, this API has a great potential to be used for future optimizations in the codebase; both to introduce threads in currently sequential sections and to improve threaded performance in already parallel ones.

One observation that could be pointed out as a drawback in the proposed changes, is the performance drop in the single-threaded executions of `git-grep`. This could have happened due to the forced eager initialization of previously lazy initializers and the locking/unlocking overhead of the added mutex to protect the `replace-object.h` API (mentioned in Subsection 6.2.2). Nevertheless, in almost all cases, the slow down is so small that it can be neglected: apart from a single test case, the time differences in all other measurements are only visible in the second decimal place.

Regarding the GSoC program, in which this project was participating, we have successfully reached the end of the last evaluation and received the approved status. With all the learning acquired throughout this process, I decided to write an unofficial first-steps guide on how to contribute to Git (Tavares, 2019a). The idea was to have a place for my own reference, but also, to hopefully help future newcomming contributors. Members of the community also supported the idea, already sending suggestions.

7.3 What is next

For future improvements, the `--textconv` code could be refactored for even better performance. We noticed that the multithreaded git-grep can get slow in the object store case when `--textconv` is used and there're too many text conversions to perfom. Probably, the reason is that the `obj_read_lock` is used to protect `fill_textconv()`, and therefore there is a mutual exclusion between textconv executions and object readings. Because both are time consuming operations, not being able to perform them in parallel can cause performance drops.

There are also some side tasks that I got to know during this project and I want to work on in the near feature. The first is related to the in-memory alternates list. As mentioned in Section 6.2.2, the object directories of submodules are currently being added to this list by `git-grep` so that future threaded calls to object reading functions can find them. But adding to this list can be bad for both memory and performance, so it might be good to avoid doing that and, instead, pass the submodules `struct repository *` data down to the threads. Another place where this is done is at the function `config_from_gitmodules()` during submodule configuration. It should be possible to avoid adding to the alternates list in this case as well. Additionally, these changes might also bring the possibility of

performing more operations in `git-grep`'s `grep_submodule()` function in parallel.

Besides these tasks related to the alternates list, there are some consistency issues yet to be solved: The call graphs we generated for git-grep (see Subsection 6.2.3) showed some call chains originated at `parse_object()`, containing functions that receive a `struct repository *` parameter, but call others that use `the_repository` internally. These inconsistent uses of `the_repository` are not currently problematic, but it might be a good idea to pass the repository down for future uses.

And finally, there are other possible optimizations to work on for `git-grep`. One of them can be seen at Hamano, 2019. Basically, there is a cache for the text converted blobs which is stored in the object store of the superproject (for both its blobs and submodule's blobs). This does not perform well, however, if the user performs a `git-grep` on the superproject and then, to further inspect, on the submodule, because the cache is then lost. So one possible improvement is to store the cache in the repository to which each blob belongs. Pedro Souza, a computer science student at the University of São Paulo, has already started an effort towards solving this issue.

Chapter 8

Personal and Critical Assessment

Working on this project has been a very challenging but also rewarding experience. It gave me the chance to learn a lot more about software development, CPU parallelism, collaborative working and the open-source world. I'm also very grateful for being able to work and interact with so many amazing developers in the Git community. And I hope I can continue contributing to Git and learning from its code and the community.

The idea of contributing to an open-source project during my capstone project came in late 2018. At that time, some colleagues and I started contributing to the Linux Kernel, as part of a course work. Rodrigo Siqueira, our mentor during the course, always encouraged us to be part of the open-source community. And to embrace the oportunity of learning from so many successful projects which the code is available for us to study. With that in mind, and intrigued by how Git would store all that information that we programmers generate, I decided to give it a try studying its code. I quickly marveled at the beauty of Git's objects concept and the very optimized way they were stored. Having the chance to conciliate my will to learn more about Git's internals with the possibility of participating in GSoC was great! Rodrigo Siqueira was the one that first mentioned I should try making it my capstone project as well. And that is how it all started.

8.1 Main Difficulties

The first difficulty I had in this project was the process of learning the code (or, in fact, the sections of the code I worked on). Git is a big project with many features. So without the knowlegde of where was what, I felt kind of overwhelmed and lost at the beginning.

Also, since my project involved going trought a lot of call chains, I had to take some time to understand what each function was doing and if it could possibly have another function in its call chain that changed global variables. But with some code studying and documentation reading, in addition to the support from the community, I eventually overcomed this initial barrier and got much more comfortable with the repository. The lesson I took is that, although starting on a big project might be difficult in the beginning, once the basics are learned, it gets much easier to infeer new ideas and develop further knwoledge and intuitions.

Writting this final essay was also a great challenge. I'm used to writting small technical blog posts and emails, but I had never written such a big and formal document before. It was quite challenging. Especially the attention required with the verbal voice and grammatical person used.

Besides these two previously mentioned personal difficulties, most of the remaining challenges were in the technical side. From here to the end of this section, I will try to list some of them, together with the solution we applied. Most of them relate to the process of ensuring thread-safeness for a particular function (and its call tree). That's a crutial task while parallelizing new sections, to avoid introducing race conditions. And yet, it can be very challenging and demand some code navigation and studying. As many of the challenges that arise during this process seem to be common between parallelization projects, I hope these comments also serve as a possible reference for similar projects.

Git's codebase contains some amazing techniques for performance, memory management, and even "meta" techniques to make the code cleaner. Unfotunately, some of them are, a priori, not thread-safe, which can complicate the parallelization process. One of them is the use of lazy initializers. This is a very interesting technique because it avoids initializing variables that won't be needed in a particular execution. Besides saving memory, this is also good for performance. The negative side is that, as oposite to pre-initializing everything before spawning threads, the lazy initializers are usually not thread-safe. The solution was to protect them or, if we knew the resource would be needed, force an eager initialization.

Another nice feature that is unfortunatelly not thread-safe is the return of static buffers in function calls. I.e., instead of returning a dynamic allocated data, a function can have a static variable in its scope (which will last throughout the whole execution) and return its address. The biggest advantage is that the callers don't have the responsability of freeing the memory after using it. But two parallel calls to the function may overwrite the result from one another. In this project, the said functions were protected with a mutex. But another solution would be to make them hold an array of static data, where each position

would be "owned" by a different thread. Then, to differenciate which thread is calling and use the proper memory, the thread private storage could be used.

And finally, the codebase makes use of some strategical global variables to avoid passing down the same data in lots of call stacks. One good example is the variable `the_repository` which holds information from the repository in which the Git process is operating. Unfortunatelly, though, this can sometimes complicate the process of evaluating thread-safetness. Especially since this variable can be used in different abstraction levels, one might think a function is safe for not using global states but, in fact, it contains another function in its call tree which does. The solution used in this project was not particularlly fast: we manually scanned the call trees for global variables usage, with some help from call graph generators, `ctags` to easily jump between symbols, and custom scripts to filter out known safe paths from the call graph. There seems to be a much easier way, though, using `GCC` with plugins[1]. This wasn't used in this project simply because we didn't know the possibility back then.

8.2 Extra Activities

Working on this project and interacting with the Git and open-source community has allowed me to participate in amazing activities/events, for which I am very grateful. In August, me and Renato Geh, who is another student from the University of São Paulo, presented at the linuxdev-br[2] conference, talking about object-oriented techniques in C (Tavares and Geh, 2019). The goal of our presentation was to show real use cases of the said techniques in code snippets from Linux and Git. I focused mostly on Git's dir-iterator API, which I worked on as my first contribution (see Section 2.3). I also participated in this year's Git Summit, a virtual meeting to discuss current ideas being developed in Git and future plans for the project. As a new contributor, I participated mostly as a spectator, but it was an amazing learning experience.

Finally, I also want to mention the local community we are forming at University of São Paulo, to help other students start contributing to Git as well. This is an initiative inside FLUSP[3], a group of students at the University of São Paulo focused on contributing and promoting the usage of FLOSS. This year, some of us started meeting weekly to exchange Git learnings and tips on how to contribute to it. As part of the group's value of mutual knowledge sharing, I paired up with some colleagues to pass on what I've learned during this project and help some of them sending their first contributions. This local community

[1] https://gcc-python-plugin.readthedocs.io/en/latest/working-with-c.html#finding-global-variables
[2] https://linuxdev-br.net/
[3] https://flusp.ime.usp.br/

have a great potential and, hopefully, will grow, in the following years, as a great first point of contact to the FLOSS world, for many students.

Appendix A

Machines Used in Tests

Mango Processor: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Cores: 4
Threads: 8
RAM: 16GB
Storage: PCIe SSD SAMSUNG MZVLB512HAJQ-000L2
OS: Manjaro Linux, Linux kernel 4.14.149-1-MANJARO
Topology: Figure A.1

Grenoble Processor: Intel(R) Xeon(R) CPU E3-1230 V2 @ 3.30GHz
Cores: 4
Threads: 8
RAM: 32GB
Storage: HDD Seagate Barracuda ST1000DM003-1ER162, 7200 rpm, SATA 3.1 [1]
OS: Debian 10.0, Linux kernel 4.19.0-5-amd64
Topology: Figure A.2

[1] This machine also has a SanDisk SDSSDA120G SSD with SATA 3.2 bus which holds the system root partition. However, all test files (which include the Git binaries and testing data) were stored in the home partition which is at the mentioned HDD.

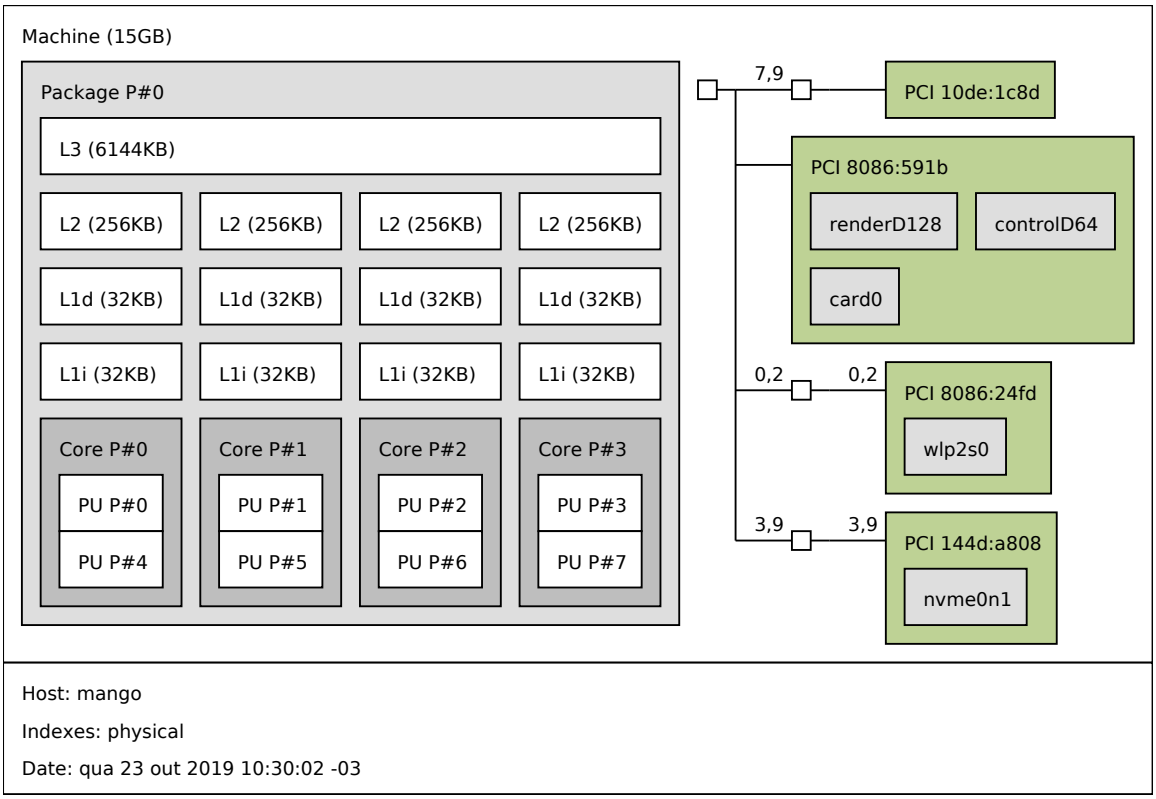

Figure A.1: *Topology of the machine mango.*
(Generated with lstopo from the hwlock package.)

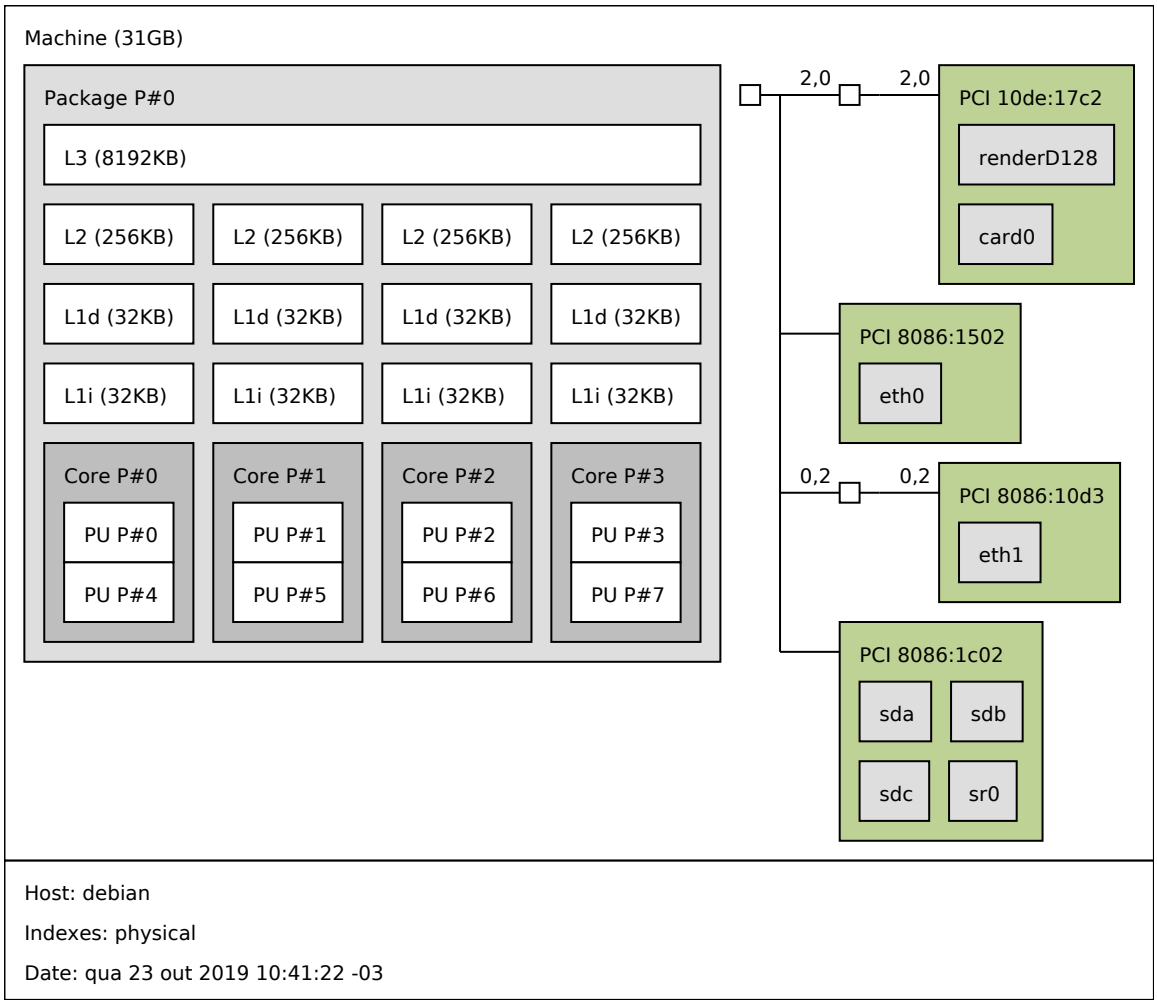

Figure A.2: *Topology of the machine grenoble.*
(Generated with lstopo from the hwlock package.)

Annex A

GCC patch for call graph dumping

Following is the initial patch wrote by Giuliano Belinassi[1] to make GCC dump the compiled binary's call graph during the Link Time Optimization (LTO) pass. It is appliable over commit `0198bef ("PR c++/91868 - improve -Wshadow location.", 2019-09-24)`. To make use of this feature, the patched GCC must be invoked with the flag `-flto`. It is also important to compile only one binary at a time and move the generated `/tmp/lto_cgraph.dot` file between compilations. Otherwise, the graph description resulting from the second compilation will be appended to the output of the first, resulting in a corrupted file.

This initial version was written as a temporary solution, since the dumping wasn't controlled by any flag or option. But later, Giuliano wrote a more polished version (composed by the patches Belinassi, 2019a and Belinassi, 2019b), which was sent to the GCC mailing list and already accepted. The feature should be part of GCC version 10. Also, with Giuliano's assistance, I later wrote a GCC plugin[2] combining this feature with the function filtering options described in Subsection 6.2.3.

Program A.1 GCC patch for call graph dumping

```
1  diff --git a/gcc/cgraph.c b/gcc/cgraph.c
2  index 331b363c1..9c3a52d28 100644
3  --- a/gcc/cgraph.c
4  +++ b/gcc/cgraph.c
5  @@ -2142,7 +2142,7 @@ cgraph_node::dump_graphviz (FILE *f)
```

cont ⟶

[1] giuliano.belinassi@usp.br
[2] https://github.com/matheustavares/gcc-callgraph-plugin

⟶ *cont*

```
6         {
7           cgraph_node *callee = edge->callee;
8
9     -       fprintf (f, "\t\"%s\" -> \"%s\"\n", name (), callee->name ());
10    +       fprintf (f, "\t\"%s\" -> \"%s\"\n", dump_name (), callee->dump_name ());
11        }
12     }
13
14    diff --git a/gcc/cgraphunit.c b/gcc/cgraphunit.c
15    index cb08efeb5..d54054ea1 100644
16    --- a/gcc/cgraphunit.c
17    +++ b/gcc/cgraphunit.c
18    @@ -2617,6 +2617,13 @@ symbol_table::compile (void)
19          timevar_start (TV_CGRAPH_IPA_PASSES);
20          ipa_passes ();
21          timevar_stop (TV_CGRAPH_IPA_PASSES);
22    +
23    +     if (in_lto_p)
24    +     {
25    +         FILE *dump_file = fopen("/tmp/lto_cgraph.dot", "w");
26    +         symtab->dump_graphviz (dump_file);
27    +         fclose (dump_file);
28    +     }
29        }
30        /* Do nothing else if any IPA pass found errors or if we are just streaming LTO.  */
31        if (seen_error ()
```

References

[Belinassi 2019a] Giuliano Belinassi. *Patch "[PATCH v2] Make lto-dump dump call-graph in DOT format"*. 2019. URL: https://gcc.gnu.org/ml/gcc-patches/2019-07/msg00145.html (visited on 12/04/2019) (cit. on p. 64).

[Belinassi 2019b] Giuliano Belinassi. *Patch "Fix incorrect merge of conflictant names in 'dump_graphviz'"*. 2019. URL: https://gcc.gnu.org/ml/gcc-patches/2019-10/msg01534.html (visited on 12/04/2019) (cit. on p. 64).

[Chacon 2008] Scott Chacon. *Introduction to Git*. 2008. URL: https://www.youtube.com/watch?v=xbLVvrb2-fY (visited on 10/16/2019) (cit. on p. 17).

[Chen 2018] Erik Chen. *Issue 18: "git blame" is slow on large repositories*. 2018. URL: https://bugs.chromium.org/p/git/issues/detail?id=18 (visited on 12/03/2019) (cit. on p. 6).

[Chacon and Straub 2014] Scott Chacon and Ben Straub. *Pro Git*. Apress, Nov. 2014 (cit. on pp. 1, 7, 15, 17).

[Deutsch 1996] L. Peter Deutsch. *RFC 1951: DEFLATE Compressed Data Format Specification version 1.3*. 1996. URL: https://tools.ietf.org/html/rfc1951 (visited on 12/03/2019) (cit. on pp. ii, 14).

[Eastlake and Jones 2001] Donald E. Eastlake and Paul E. Jones. *RFC 3174: US Secure Hash Algorithm 1 (SHA1)*. 2001. URL: https://tools.ietf.org/html/rfc3174 (visited on 12/03/2019) (cit. on p. 14).

[Git Project 2014] Git Project. *Pack heuristics*. 2014. URL: https://git-scm.com/docs/pack-heuristics (visited on 10/16/2019) (cit. on p. 18).

[Git Project 2018] Git Project. *Pack Format*. 2018. URL: https://git-scm.com/docs/pack-format (visited on 10/16/2019) (cit. on p. 18).

[Git Project 2019] Git Project. *Git's Repository Documentation*. 2019. URL: https://github.com/git/git/tree/master/Documentation (visited on 12/04/2019) (cit. on p. 7).

[Google Trends 2019] Google Trends. *Google Trends: Comparing Git, Subversion, Mercurial, Perforce and CVS*. 2019. URL: https://trends.google.com/trends/explore?date=2004-01-01%202019-10-28&q=%2Fm%2F05vqwg,%2Fm%2F012ct9,%2Fm%2F08441_,%2Fm%2F08w6d6,%2Fm%2F09d6g (visited on 10/28/2019) (cit. on p. 2).

[Hamano 2019] Junio C Hamano. *Email "Re: [RFC PATCH 0/3] grep: don't add subrepos to in-memory alternates"*. 2019. URL: https://public-inbox.org/git/xmqq36gt5qhr.fsf@gitster-ct.c.googlers.com/ (visited on 12/02/2019) (cit. on p. 56).

[IEEE and The Open Group 2018] IEEE and The Open Group. *The Open Group Base Specifications Issue 7, 2018 edition, section "Thread Extensions"*. 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/xrat/V4_xsh_chap02.html#tag_22_02_09_08 (visited on 11/26/2019) (cit. on p. 39).

[Linux man-pages project 2019] Linux man-pages project. *MMAP(2) Linux Manual Page*. 2019. URL: http://man7.org/linux/man-pages/man2/mmap.2.html (visited on 01/18/2020) (cit. on p. 47).

[Nguyen 2019] Duy Nguyen. *Snippet "hacky parallel grep"*. 2019. URL: https://gitlab.com/snippets/1834613 (visited on 12/05/2019) (cit. on p. 8).

[Roelofs *et al.* 2006] Greg Roelofs, Jean-loup Gailly, and Mark Adler. *zlib Technical Details*. 2006. URL: https://zlib.net/zlib_tech.html (visited on 12/03/2019) (cit. on p. 14).

[Roelofs *et al.* 2010] Greg Roelofs, Jean-loup Gailly, and Mark Adler. *zlib FAQ*. 2010. URL: https://zlib.net/zlib_faq.html#faq21 (visited on 11/26/2019) (cit. on p. 27).

[Shaffer *et al.* 2019] Emily Shaffer, Junio C Hamano, and Derrick Stolee. *Email Thread "[Git Developer Blog] [PATCH] post: a tour of git's object types"*. 2019. URL: https://public-inbox.org/git/5dab3dc6-3942-422e-d29d-3e8682ebc4df@gmail.com/T/#mf6c760fd87f7416d39e5ac54e9e33df9d835be87 (visited on 12/01/2019) (cit. on p. 15).

[Spajic 2018] Zvonimir Spajic. *Understanding Git - Index*. 2018. URL: https://hackernoon.com/understanding-git-index-4821a0765cf (visited on 11/22/2019) (cit. on p. 7).

[Stack Exchange, Inc. 2018] Stack Exchange, Inc. *Stack Overflow Developer Survey Results 2018*. 2018. URL: https://insights.stackoverflow.com/survey/2018#work-_-version-control (visited on 10/28/2019) (cit. on pp. i, 2).

[Tan 2019] Jonathan Tan. *Email "Re: [PATCH v2 05/11] object-store: allow threaded access to object reading"*. 2019. URL: https://public-inbox.org/git/20191112025418.254880-1-jonathantanmy@google.com/ (visited on 12/01/2019) (cit. on p. 46).

[Tavares, Couder, *et al.* 2019] Matheus Tavares, Christian Couder, *et al. Email Thread "Questions on GSoC 2019 Ideas"*. 2019. URL: https://public-inbox.org/git/CAHd-oW7onvn4ugEjXzAX_OSVEfCboH3-FnGR00dU8iaoc+b8=Q@mail.gmail.com/t/#u (visited on 12/04/2019) (cit. on p. 8).

[Tavares 2019a] Matheus Tavares. *First Steps Contributing to Git*. 2019. URL: https://matheustavares.gitlab.io/posts/first-steps-contributing-to-git (visited on 10/28/2019) (cit. on pp. ii, 55).

[Tavares 2019b] Matheus Tavares. *GSoC Proposal: Make pack access code thread-safe*. 2019. URL: https://matheustavares.gitlab.io/assets/Matheus_Tavares_GSoC_Proposal.pdf (visited on 12/04/2019) (cit. on p. 8).

[Tavares 2019c] Matheus Tavares. *Matheus Tavares' GSoC Blog*. 2019. URL: https://matheustavares.gitlab.io/gsoc/ (visited on 12/04/2019) (cit. on p. 8).

[Tavares, Chen, *et al.* 2019] Matheus Tavares, Erik Chen, and Harry Cutts. *Email Thread "Git blame's performance on chromium"*. 2019. URL: https://groups.google.com/a/chromium.org/d/topic/chromium-dev/oYe69KzyG_U/discussion (visited on 11/30/2019) (cit. on p. 12).

[Tavares and Geh 2019] Matheus Tavares and Renato Geh. *Object Oriented Techniques in C: A Case Study on Git and Linux*[3]. 2019. URL: https://www.youtube.com/watch?v=x0ELqk2lCcI (visited on 12/04/2019) (cit. on p. 59).

[Torvalds 2005] Linus Torvalds. *Email "Re: Kernel SCM saga.."* 2005. URL: https://lkml.org/lkml/2005/4/8/34 (visited on 11/30/2019) (cit. on p. 2).

[3] Slides available at https://matheustavares.gitlab.io/assets/oop_git_and_kernel.pdf

[Yerburgh 2019] Edward Yerburgh. *Intrusive linked lists*. 2019. URL: https://www.data-structures-in-practice.com/intrusive-linked-lists/ (visited on 11/24/2019) (cit. on pp. 33, 34).

[Zager 2014] Stefan Zager. *Email "Make the git codebase thread-safe"*. 2014. URL: https://public-inbox.org/git/CA+TurHgyUK5sfCKrK+3xY8AeOg0t66vEvFxXJiA9wXww7eZXQ@mail.gmail.com/ (visited on 11/30/2019) (cit. on p. 12).

[Zaoui 2019] Daniel Zaoui. *Email "Weird behavior with git grep --recurse-submodules"*. 2019. URL: https://public-inbox.org/git/20190708111459.135abe50@zen/ (visited on 12/05/2019) (cit. on p. 45).