Parallelizing Git Checkout: a Case Study of I/O Parallelism on Desktop Applications

Matheus Tavares Bernardino

Thesis presented to the
Institute of Mathematics and Statistics
of the University of São Paulo
in partial fulfillment
of the requirements
for the degree of
Master of Science

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

This project was funded by Amazon and partially funded by CNPq

São Paulo
July 13, 2022

Parallelizing Git Checkout: a Case Study of I/O Parallelism on Desktop Applications

Matheus Tavares Bernardino

This version of the thesis includes the corrections and modifications suggested by the Examining Committee during the defense of the original version of the work, which took place on July 13, 2022.

A copy of the original version is available at the Institute of Mathematics and Statistics of the University of São Paulo.

Examining Committee:

Prof. Dr. Alfredo Goldman Vel Lejbman (advisor) – IME-USP
Prof. Dr. Leonel Augusto Pires Seabra de Sousa – IST/TU Lisbon, INESC-ID
Prof. Dr. Guido Costa Souza de Araújo – IC-UNICAMP

Bernardino, Matheus Tavares
Parallelizing Git Checkout: a Case Study of I/O Parallelism on Desktop Applications / Matheus Tavares Bernardino; orientador, Alfredo Goldman. - São Paulo, 2022.
136 p.: il.

Dissertação (Mestrado) - Programa de Pós-Graduação em Ciência da Computação / Instituto de Matemática e Estatística / Universidade de São Paulo.
Bibliografia
Versão corrigida

1. Programação paralela. 2. Git. 3. Sistemas de controle de versões. 4. Sistemas de arquivos em rede. 5. Paralelismo em E/S. I. Goldman, Alfredo. II. Título.

Bibliotecárias do Serviço de Informação e Biblioteca Carlos Benjamin de Lyra do IME-USP, responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2: Maria Lúcia Ribeiro CRB-8/2766; Stela do Nascimento Madruga CRB 8/7534.

Acknowledgments

As iron sharpens iron, so one person sharpens another.
— Proverbs 27:17

I am very grateful for all the amazing people I have met throughout my life, and without whom this work would not have been possible. First of all, I thank God for His outstanding love and care, at all times. My parents, Osorio and Lauriceia Bernardino, for their continuous friendship and support, which keeps pushing me forward. My grandparents, who are some of my greatest inspirations of love and altruism. Natalia Ferreira, who always does her best to see me happy (btw, her carrot cake is simply the best!). Thanks to all my uncles, aunts, cousins, and the whole family. I am very blessed for having you guys!

I want to thank the Git community for their amazing work and all the support throughout this project. In particular, I would like to thank: Jeff Hostetler and Nguyễn Thái Ngọc Duy, who I co-developed the parallel checkout feature with; Christian Couder, who not only mentored me during GSoC 2019 but also reviewed the parallel checkout patches and assisted me in my professional life; Junio C Hamano, Derrick Stolee, Christian himself, and all others who dedicated their time to review the parallel checkout code; Jeff King, Elijah Newren, Ævar Arnfjörð Bjarmason, Johannes Schindelin, and so many other amazing developers who I had the immense pleasure to work with in the Git community!

I also thank my advisor, Prof. Alfredo Goldman, for all the personal and technical assistance and motivation. Prof. Marco Gubitoso, one of my first inspirations to start working with parallel programming. Geert Jansen, for trusting in my work and sponsoring me in this project. And Rodrigo Siqueira, a great friend and mentor, who introduced me to the FLOSS development world and with whom I have learned so much!

I own a big thanks to all my friends who kindly let me use their computers to run performance tests and checkout benchmarks: Lucas Oshiro, Giuliano Belinassi, Pedro Sousa, Gabriel Demetrius, Rodrigo Ribeiro, Felipe Caetano, Spoonm, and others. Thank you so

much for your help.

All my friends and colleagues from the FLUSP group and the Systems Laboratory at IME, including (but not limited to) Rodrigo Siqueira, Giuliano Belinassi, Nelson Lago, Dylan Guedes, Tallys Martins, Melissa Wen, Marcelo Schmitt, Lucas Oshiro, Renato Geh, and many others. It was a pleasure to embark on this journey with you!

Finally, thanks to the “Patos” group and all my school friends, who I am blessed to still hang out with and share so many moments!

Resumo

Matheus Tavares Bernardino. **Paralelizando o Git Checkout: um Estudo de Caso sobre Paralelismo de E/S em Aplicações Desktop**. Dissertação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Sistemas de controle de versões (SCV) são ferramentas que monitoraram e gerenciam as alterações feitas em um conjunto de arquivos ao longo do tempo. De forma mais abrangente, SCVs também podem contribuir para moldar e gerir fluxos de colaboração, encontrar e corrigir *bugs*, relembrar as motivações por trás de determinada alteração de código, etc. Apesar de tipicamente poderem monitorar qualquer tipo de dados, sistemas de controle de versão trazem benefícios importantíssimos para projetos de software e, com isso, se tornaram prática padrão neste campo. Dentre as ferramentas de SCV disponíveis atualmente, o Git é o mais popular entre desenvolvedores. A ferramenta é utilizada hoje para versionar desde pequenos projetos pessoais, com alguns megabytes de tamanho, até repositórios corporativos massivos com mais de 300 GB e 3,5 milhões de arquivos. Por esse motivo, velocidade e escalabilidade estão entre as principais prioridades para a comunidade de desenvolvimento do Git. No entanto, o desempenho da ferramenta por vezes se encontra aquém do desejado em sistemas de arquivos em rede (NFS), onde operações de entrada e saída (E/S) costumam ser mais custosas. Em particular, uma operação do Git que sofre com estes custos é o *checkout*, que é responsável por restaurar arquivos de versões específicas de um projeto. Diversas otimizações foram empregadas em códigos relacionados à operação de *checkout* ao longo do tempo, mas o processamento sequencial dos arquivos ainda trazia uma penalidade de tempo grande para NFS, além de ser subótimo para sistemas de arquivo locais em SSDs. Neste projeto, trabalhamos para paralelizar o maquinário de *checkout* do Git, resultando em *speedups* de até 4,5x em NFS e 3,6x em SSDs. Também estudamos como o paralelismo afeta as tarefas de E/S realizadas pela operação de *checkout* em diferentes máquinas e dispositivos de armazenamento. A funcionalidade de *checkout* paralelo foi incorporada ao repositório *upstream* do Git e disponibilizada para todos os usuários da ferramenta na sua versão 2.32.0, que foi lançada em Junho de 2021.

Palavras-chave: Programação Paralela. Git. Sistemas de Controle de Versões. Sistemas de Arquivos em Rede. Paralelismo em E/S.

Abstract

Matheus Tavares Bernardino. **Parallelizing Git Checkout: a Case Study of I/O Parallelism on Desktop Applications**. Thesis (Master's). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

A version control system (VCS) is a tool that tracks and manages the changes made to a set of files over time. More broadly, VCS tools can also help to shape and manage collaboration flows, find and fix bugs, remember the motivations behind a given code change, etc. Although these tools can typically track any type of data, version control systems bring huge benefits to software projects and, as a result, have become standard practice in this field. Among the VCS tools available today, Git is the most popular among developers. This tool is currently being used to version control a variety of repositories, from small personal projects of a few megabytes in size to massive corporate repositories with more than 300 GB and 3.5 million files. For that reason, speed and scalability are among the top priorities for the Git development community. However, the performance of the tool sometimes falls short of what is desired on networked file systems (NFS), where input and output (I/O) operations tend to be more costly. In particular, one Git operation that suffers from these costs is checkout, which is responsible for restoring files from specific versions of a project. Various optimizations were employed on code related to the checkout operation over the years, but the sequential processing of files still carried a large time penalty for NFS, as well as being suboptimal for local file systems on SSDs. In this project, we worked to parallelize the Git checkout machinery, resulting in speedups of up to 4.5x on NFS and 3.6x on SSDs. We also study how parallelism affects the I/O tasks performed by the checkout operation on different machines and storage devices. The parallel checkout feature was incorporated into the upstream Git repository and made available to all users of the tool since version 2.32.0, which was released in June 2021.

Keywords: Parallel Programming. Git. Version Control Systems. Network File Systems. Parallel I/O.

List of Figures

List of Tables

List of Programs

Contents

Appendices

Chapter 1

Introduction

1.1 Version Control Systems

A version control system (VCS) is *"a system that records changes to a file or set of files over time so that you can recall specific versions later"* (Chacon and Straub, 2014). These systems manage and store the history of a project as it evolves, not only allowing the future study of historical changes but also the present coordination of parallel changes that are made by different authors. Among the many features that a VCS can provide today, some of the core functionalities are: the creation of a new version with some description of the changes, the retrieval of one or more file(s) contents as they were at a specific version, the summary of differences among any two versions, and the visualization of the project's history.

The files of a project versioned by a VCS usually live in what is called a repository, where many other important metadata is also stored. Repositories can typically hold (i.e. version control) any type of data, not limited to source code. Nevertheless, version control tools are often associated with software engineering as it has become widely used in this field (Nurlisa *et al.*, 2018), (Spinellis, 2005) (Otte, 2009). So much so that VCS tools like Git are also referred as a source control management (SCM). Version control systems constitute a very important element of the software development tool chain, and paramount when working with a large team of developers, as it *"supports a collaborative framework that makes it easy for software developers to work together effectively"* (Nurlisa *et al.*, 2018). Nevertheless, version control has many benefits for small teams and individual contributors too. Being able to recall when a code change was made (and more importantly: "**why**" a change was made) helps developers on their decisions about future changes in the same area, as well as locating and fixing bugs. The history of a project as provided by a VCS can also be of great assistance to future developers, as they can study how specific pieces of code came to be.

There are two major types of VCS: centralized version control systems (CVCS) and decentralized version control systems (DVCS). The first category includes the systems where there is one centralized repository on a server and the users interact with it through the network. The second category defines the systems where each user has its own full

copy of the repository locally. Some examples of well known CVCS are CVS (CVS, 2022), Subversion (Apache, 2022), and Perforce's Helix Core (Perforce, 2022), while popular DVCS tools include Git (Git Development Community, 2022), Mercurial (Mercurial, 2022), and Bazaar (The Bazaar Team, 2022). Each model of version control system has its own advantages, but DVCS are commonly acknowledge for the robustness of not having a single point of failure. On the other hand, many users of DVCS still employ some kind of centralization by pointing one repository as the single "source of truth".

1.2 Git

Git is a decentralized open source version control system. It was created in 2005 by Linus Torvalds to fill a gap inside the Linux kernel community after the proprietary VCS that they had been using until then, BitKeeper, got its free license revoked. Linus and other contributors started looking for alternatives, but none of them seemed to fully meet the requirements of the project and its contributors at that time. In the meantime, Linus decided to work on a set of scripts that would serve him as *"a fallback [...] if nothing out there works right now"* (Torvalds, 2005). With about one day, the "distribution and archival mechanism", as Linus first put it, was able to host itself, and about 10 days later it was used to make the first Linux commit (The Linux Foundation and Torvalds, 2014). With the help of other contributors who liked the idea and engaged in the early development, Git got its v1.0.0 released about eight months later.

As mentioned in the Pro Git book (Chacon and Straub, 2014), some of the initial goals for the software were:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Git has quickly gained popularity and, by 2006, other open source projects were using it to version control their code, like: Cairo, Gnumeric, Wine, xmms2, and the X.org X server (Hamano, 2006). In the following years many more projects adopted Git as their VCS, and thanks to services like GitHub (launched in 2008) and Gitlab (launched in 2011), it became very easy for anyone to host their Git repositories online. Today, Git has become the most popular VCS for source code management. 2018's Stack Overflow Developer Survey showed that almost 90% of the interviewed developers were versioning their code through Git (Stack Exchange, Inc., 2018a). 2021's edition showed that over 93% of the respondents have done extensively development work in Git during the past year and about 89% want to work with Git in the next year (Stack Exchange, Inc., 2018b).

To this date, approximately 1900 people have contributed to the Git project with about 50000 accepted patches. Junio Hamano has been the maintainer since 2005.

1.3 Motivation and Goal

Checkout is the Git operation responsible for restoring specific versions of one or more files in a repository. It is a core operation for a version control system, allowing its users to switch between different snapshots of the project, discarding unwanted changes, testing alternative versions, etc. Besides the homonymous `git checkout` command, the checkout machinery is used by many other Git commands like: `clone`, `reset`, `switch`, `restore`, and `merge`.

Checkout is usually a fast operation when updating a small number of files, but performance can become a problem as the workload grows. This is specially critical for Git users over networked file systems; which, due to the higher I/O latencies, may experience some checkout commands taking up to 50x or even 130x more time than local file systems. To put it into perspective, a full checkout of the Linux repository (which contains over 70K files), takes around 8 seconds on a local Linux machine with SSD, but it can take 5 to 15 minutes on network file systems. Furthermore, Linux is not even the largest repository versioned through Git: the Chromium repository has over 300K files, and Windows has over 3.5M (Harry, 2017).

With that in mind, our goal with this project is **to parallelize the working tree update phase of checkout to improve its performance for large workloads, specially over NFS**. During this work, we aim to answers the following research questions:

- **RQ1**: Which are the most time-consuming operations at checkout, and why?
- **RQ2**: How does the underlying storage type (SSD, HDD, and NFS) influence sequential and parallel checkout performance?
- **RQ3**: What performance gain can we get by parallelizing I/O-heavy desktop code, like git checkout?
- **RQ4**: How does parallelizing checkout impacts memory usage?

We hope that the discussions around these questions will help other developers working with parallelism of programs that have intensive I/O requirements, like checkout. Our main contribution with this work is the implementation of the parallel checkout framework, which resulted in speedups of up to 3.6x on SSDs and 4.5x on NFS in our checkout benchmarks using the Linux repository. The feature was merged into the upstream project and released to all Git users on June 2021, as part of Git 2.32.0. In this process, we also found and worked to fix a checkout vulnerability in Git, which allowed for remote code execution attacks when cloning a maliciously crafted repository. Finally, we discuss the challenges faced during this project and the design decisions taken, as a case study on how I/O operations from real world desktop applications can be optimized for better performance on NFS and SSDs.

1.4 Document Structure

This document is organized as followed: Chapter 2 presents the necessary theoretical concepts about Git and its internal structures, including the division between `gitdir` and

`working tree`, the different states a file might be categorized as, the object store, and the index. Then it uses these concepts to give an overview of how the checkout process works. Chapter 3 shows some results for profile tests on a checkout operation, discussing which steps take more time on each type of storage. Chapter 4 enumerates the challenges to a parallel checkout implementation, including different race condition issues that can cause erroneous results and even security vulnerabilities. Chapter 5 discusses other works related to this project, focusing on three previous approaches to parallelizing the checkout machinery. Chapter 6 presents the development process and the design decisions taken, discussing how they helped us overcome the challenges from Chapter 4. It also gives an overview of the parallel checkout implementation, with a step-by-step description of the code flow. Finally, Chapter 7 presents the results for the performance tests and concludes with a discussion on current limitations and future possibilities.

In this document, we will reference commits from Git repositories in the following format: `abbrev-hash ("commit title", yyyy-mm-dd)`. Where, "abbrev-hash" correspond to the first 10 hexadecimal digits from the commit's SHA-1 hash.

Chapter 2

Background

This chapter discusses the key concepts of Git's internal storage system and presents a detailed overview of each step taken during a checkout operation. Sections 2.1 and 2.2 give an introduction to Git's terminology and basic functioning. Section 2.3 discusses how Git classifies the different files present on the working directory of a repository. Section 2.4 briefly explains what is the index, the different types of Git objects, and how they get stored on disk. Section 2.5 presents the anatomy of a checkout operation with a step-by-step description. Finally, Section 2.6 presents the `lstat()` cache, an important structure which is used during checkout and, as we will discuss later, should be handled with great care throughout the parallelization process.

2.1 Commits and References

In a very simplistic definition, Git stores snapshots of a set of files over time. These different versions of a project (also referred as *revisions* by other VCSs) are called `commits` in Git. Commits are full snapshots, but they are stored in a very efficient format, which allows repositories to scale. When a new commit is made, Git also registers what was the previous commit to log the changes over time. This allows to later create on-the-fly *diffs*, which helps to see what modifications were introduced at each version.

On a simple historical flow, the commits may form a linear chain. But through the use of *branching* and *merge* operations, the commit history can in fact become non-linear. In the more general scenario, the commit history is a direct acyclic graph, where each node corresponds to a commit, and each commit can have zero or more arcs departing from it and arriving at its predecessor(s). Figure 2.1 illustrates a Git repository history with multiple commits and branches. The commits are represented by gray rounded shapes and identified by their SHA-1 hash (which was abbreviated to six characters).

A commit with no predecessors (or *parents*), like commit *a02f3d* in the figure, is called a `root commit`. And a commit with multiple predecessors, like *73ba2d*, is called a `merge commit`, because it unifies two or more previously parallel history lines, called `branches`. In Git, branching is said to be a lightweight mechanism. The identification of a branch, called the `branch head`, is simply a pointer to the commit at the tip of the branch, asso-

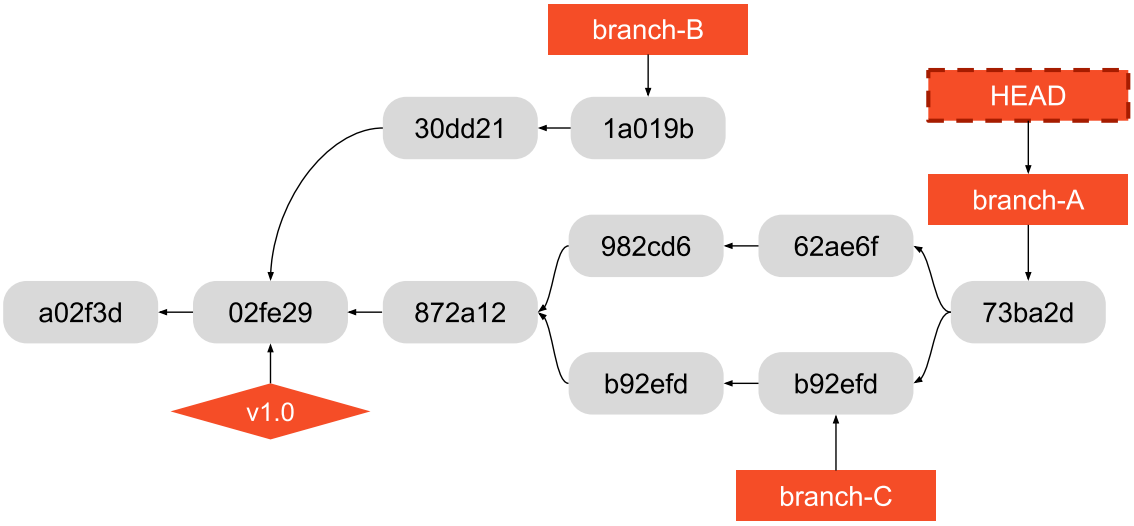


Figure 2.1: *Diagram representing Git commits and references.*

ciating a textual name (i.e. the branch name) to a commit. This pointer moves on as new commits are created at that branch or when the user wants to reset the tip of the branch to a given commit. Figure 2.1 contains three branches, represented by the orange rectangles with no outline. Note that `branch-C` was merged into `branch-A`, whereas `branch-B` remains unmerged relative to `branch-A`.

Branch heads are one example of `references` in Git, or `ref` for short. Other types of references include `tags`, which can be similar to branches, with the difference that they do not move around, and remote branches, which are used to track branches on a remote repository. There are also special references called `symbolic refs` which can point to either a Git object (like a commit) or other references. The prime example for this is the `HEAD` ref which, roughly speaking, points to the currently checked out version of a project. At Figure 2.1, we have one tag named "v1.0", which is represented by a diamond. The orange rectangle with a dashed outline is the `HEAD`, which is pointing at `branch-A`. So that would be the current state of the repository in this example. In total, this diagram shows five references.

2.2 Repository Layout

Physically, Git repositories are usually divided in two main parts: the `working tree`, or `working directory`, and the `git directory`, or `gitdir`. The first corresponds to the root directory of a project, containing all the user files that are being version controlled, possibly alongside any others that are not (like build artefacts or new files that were not yet added to Git). The `git directory`, on the other hand, stores the data from previous versions of the project files as well as other important metadata about the repository and its contents. These two parts of a repository are illustrated by Figure 2.2.

The `gitdir` usually resides inside the `working tree`, as a non-version-controlled directory called *.git*. But that is not a hard rule: for some repositories, `.git` is a regular file

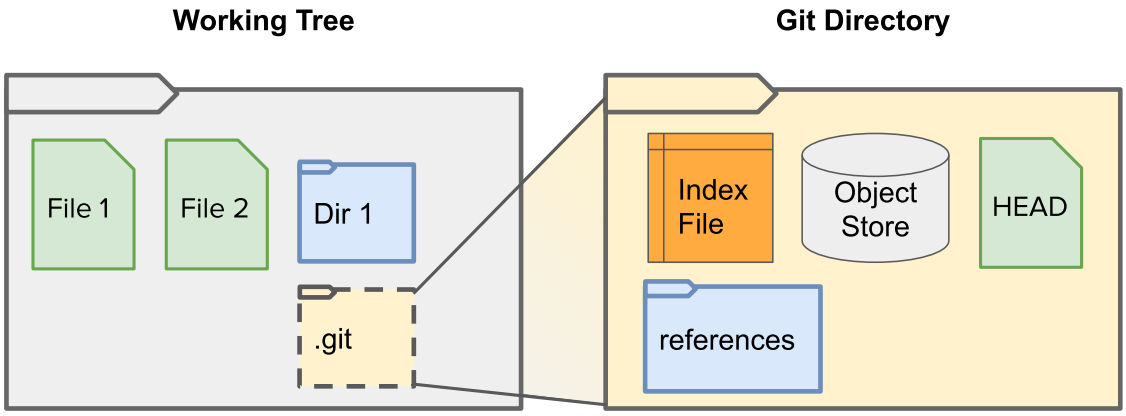


Figure 2.2: *Most common layout of a non-bare Git repository.*

containing the path to a directory that lives somewhere else. Some repositories may even have a `gitdir` only, with no associated `working tree`. These are called `bare repositories`, and they are typically used on cloud and/or shared environments to publish repositories that others can pull from or push to. Furthermore, Git allows its users to create and manage multiple `working trees` within the same repository[1], using the `git worktree` command. Thus, it is possible to associate `working trees` to `bare repositories` (although that is not the typical use case for them).

2.3 Working Tree

Working tree files can be in one of two states: `tracked` or `untracked`. The first category refers to the files that are being version-controlled by Git, and the second corresponds to files that are not. Untracked files may either be new files that were not added to Git yet or files that the user purposely does not want to put under version control. In this case, users can additionally mark the files as `ignored`, which will make Git avoid adding them to upcoming commits and showing them in status reports.

In relation to the current `HEAD`, i.e. the commit currently checked out in the working tree, *tracked* paths can be further categorized into three states: `clean`, `modified and staged`, and `modified and unstaged`. The first refers to tracked paths which have no modifications regarding the currently checked out revision. `Staged` paths contain modifications which git was told to include in the next commit. And `unstaged` paths refers to modifications that should not be commited yet. The last category also includes `unmerged` paths, which are paths that could not be automatically merged during a `git merge` operation (due to textual conflicts) and now require user intervention to resolve the conflicts. Modified files (either `staged` or `unstaged`) are also called `dirty`, meaning that they have uncommited changes.

Note that the same path may sometimes appear in more than one category. For example the user may modify a tracked file and `stage` it, but them further modify the file

[1] In a repository with multiple `working trees`, parts of the `gitdir` are shared among all `working trees`, while some structures (like the index and HEAD files) are private.

without adding the new changes to Git. This way, the file will have both `staged` and `unstaged` modifications. Figure 2.5 illustrates this, also showing how each of these described states relates to the index file, which will be explained in a following section.

2.4 Git Directory

Among other data, the `gitdir` holds the `reference storage`, `object storage`, `hooks`, configuration files, and the `index` file. In this section, we will focus on the two structures that are more relevant to this work: the object storage and the index.

2.4.1 Git Objects

Object are immutable units of storage in Git. They are compressed using the DEFLATE algorithm, and referenced by the SHA-1 hash[2] of their contents. Objects are one of the most important structures in Git; they are responsible for storing the different versions of the project files, as well as other metadata like the structure of the directories, authors and dates of each commit, etc.

There are currently six types of objects in Git, each following a different format: blobs, trees, commits, tags, ofs-deltas, and ref-deltas. The first four have a direct semantic mapping to user-related contents, while the last two serve a "lower level" purpose of optimizing disk space used by the others on packfiles (more about that in the next section). We will not go into too much detail about the structure of each object type, as that is not relevant for this work, but we will briefly discuss what each one of them represents and what kind of information they store.

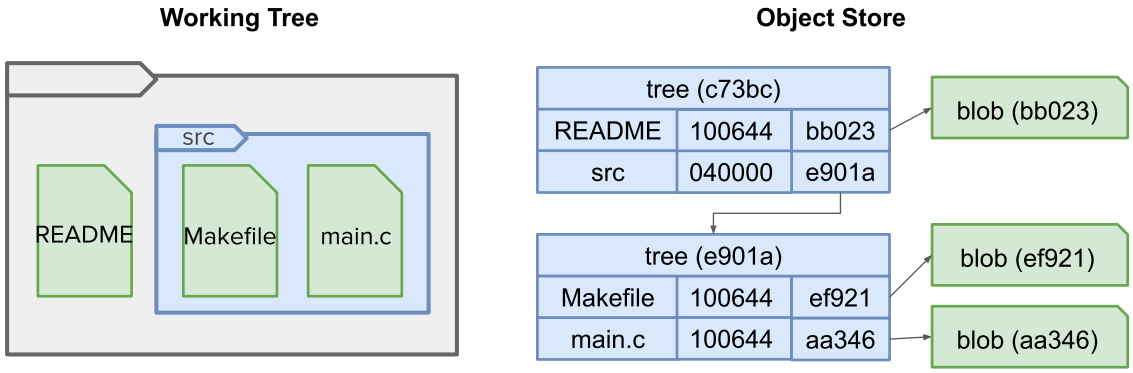


Figure 2.3: *Example of working tree files and their respective Git objects.*

The blob is the most flexible object type, holding an arbitrary stream of bytes. Blobs are used to save the contents of the tracked files on a project. Going up a level, we find the trees, which are used to represent directories. A tree object stores a set of file names, together with their types, their permissions, and the hashes of the objects that hold their contents. The type and permission can be represented by a single numerical value (e.g.

[2] At the time of writing, the Git project is transitioning from SHA-1 to SHA-256. Support for SHA-256 is currently experimental.

100644 constitutes a non-executable regular file) and the referenced object can be: a blob, for regular files and symbolic links; a tree, for sub-directories; or a commit, for submodules (another repository nested inside this repository). Figure 2.3 showcases how some working tree files get stored as Git objects.

One more level up there are the commits, which as already mentioned, represent the different registered versions of a project. Each commit object holds metadata like: the commiter's name and email, the date of creation, and a user-given message, which is normally used to describe the changes introduced by that commit. Together with this metadata, the commit object holds one object ID (i.e. the SHA-1 hash) of a tree object, which corresponds to the root directory of the project in that specific commit. Each commit object can also reference one or more `parent` commits, making up the project's history and logging how the project evolved over time.

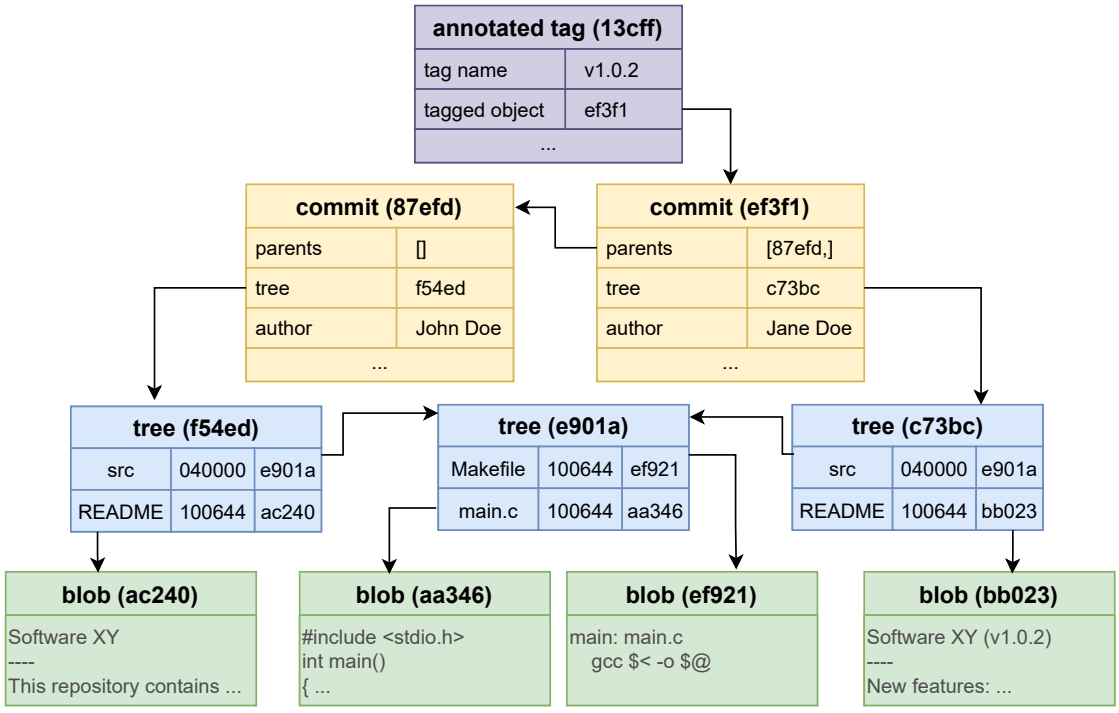


Figure 2.4: *Main Git objects and how they interact with each other.*

Finally, the last non-delta object to be mentioned is the annotated tag. We have already discussed about tags in general, but we have not distinguished the two different types of tags: lightweight and annotated. The first is only composed by a ref (that is why it is called lightweight), while the second also have a full Git object associated with it. This allows annotated tags to store more information besides a tag name and the ID of the tagged object (which is normally a commit, but can also be another object type, including a tag). This includes, for example: a message, date, and the tagger's name and email. Figure 2.4 shows a complete example of how the four object types mentioned in this section might interact with each other.

2.4.2 Object Storage

Git objects can be stored on disk in two formats: loose and packed. Both formats use the DEFLATE compression, but packed objects allow even better space optimizations through *deltification*. Loose objects are individually stored in their own binary files, which gets named by the object's SHA-1 hash. The packed format, on the other hand, allows multiple objects to be stored in a single file, which is called the `packfile`. These files are paired with a pack index file, which helps Git commands navigate the associated packfile more quickly.

As previously mentioned, packed objects can also take advantage of deltification; a mechanism which aims to minimize the storage of redundant data. When two or more objects are similar in their contents, Git may store only one of them wholly (called the `base`), and store the other ones as `deltas`, which basically contain a reference to a base and instructions on how to reconstruct the full object represented by the delta with modifications to the said base. Note that the base may be a delta itself, producing what is called a "delta chain". Of course, there must be a non-delta object at the end of a delta chain, otherwise it would not be possible to reconstruct the delta objects in it.

Normally, when first creating an object, Git stores it in loose format. But occasionally, Git's garbage collection kicks in and packs the loose objects if their number surpasses a configurable threshold (6700 by default).

2.4.3 The Index

The `index` primarily holds information about the current state of the working tree, also serving as an "interface" between the working tree and the object store. Its main section is composed by a list, which contains one entry for each file in the present commit, and potentially some extra entries for merge conflicts and new files that the user wants to add in an upcoming commit. Each index entry is composed by: a pathname, file metadata (like the last modification time, file mode, size, and inode number), a set of flags, and the hash of an object, which represents the contents of the file when it was last staged or updated in the index (e.g. due to a checkout).

When a file is "staged" (e.g. with `git add <file>`), it either gets a new index entry (if it is not already present in the index) or its index entry gets updated with the new state of the file. That is, its latest contents and metadata. In both cases, if there are any modifications in the file's contents, a new object is created at the `Object Store` and the index entry receives its reference. The metadata is usually collected with one of the functions from the `stat()` family. This information allows Git to tell whether the working tree file associated with a given entry has any new "unstaged" changes, without having to read and compare the whole file and the corresponding `blob`, at every `git status`. Figure 2.5 illustrates the different states of working tree files in regards to the index and HEAD (i.e. the commit/tree that is currently checked out in the working tree).

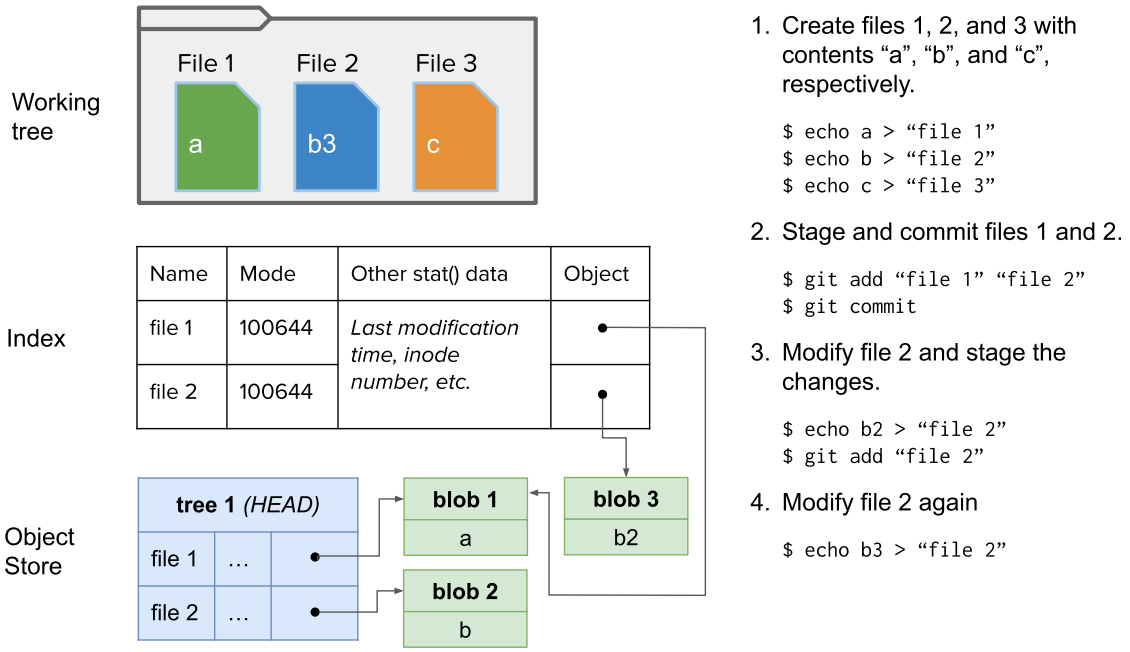


Figure 2.5: *Example of different working tree file states with regards to the index and HEAD: “file 1” is tracked and clean; “file 2” is tracked and dirty, with both staged (b2) and unstaged (b3) modifications; and “file 3” is untracked. The commands on the right showcase how such scenario was produced.*

2.5 Checkout Anatomy

`Checkout` is the Git operation that materializes files from specific versions into the working tree, also updating the index accordingly. The first Git command that comes to mind when discussing this operation is the homonymous `git checkout` command, but the checkout machinery is also used by many other commands like `git reset`, `git clone`, `git merge`, `git sparse-checkout`, etc. Checkout-related commands usually perform the following “top-level” steps:

1. Read the index file and load its data into memory. Each index entry is stored in a `struct cache_entry` (which we will reference by either “cache entry” or “index entry” from now on). These entries are placed in an array which is sorted in ascending order on the pathname field, just like the on-disk index file[3];
2. Update the in-memory index according to the goal of the running command. This normally involves loading different data into the index by creating, modifying, and/or removing index entries to achieve a certain end result (e.g. a branch switch, a merge, etc). Although working tree files are typically not modified at this step, it may be necessary to inspect the working tree to warn about unsaved changes that could be lost (and possibly abort the operation);
3. Reflect the changes made to the index entries in the working tree, creating and deleting files as required. The contents of the new files are loaded from the objects

[3] For more information, see: https://github.com/git/git/blob/v2.32.0/Documentation/technical/index-format.txt

storage;

4. Save the metadata from the newly written files in the in-memory index entries;
5. Write the new index to the disk.

Not all of the commands that use the checkout machinery perform all of the steps above, and some of them even do additional work, but this list summarizes the key points. To help visualizing how these steps can be executed by a Git command, consider the following example: suppose we have a Git repository with two branches, named B1 and B2. The commit at the tip of B1 contains the files X and Y, while the commit at the tip of B2 contains X and Z. The contents of the file X differs between the two branches. Now consider that B1 is currently checked out and all files are clean (i.e. there is no local modifications in the working tree or the index with regards to B1). Figure 2.6 ilustrates this proposed scenario showing how the branches, working tree, and index would look like. Note that the file X from B1 and B2 point to different blobs.

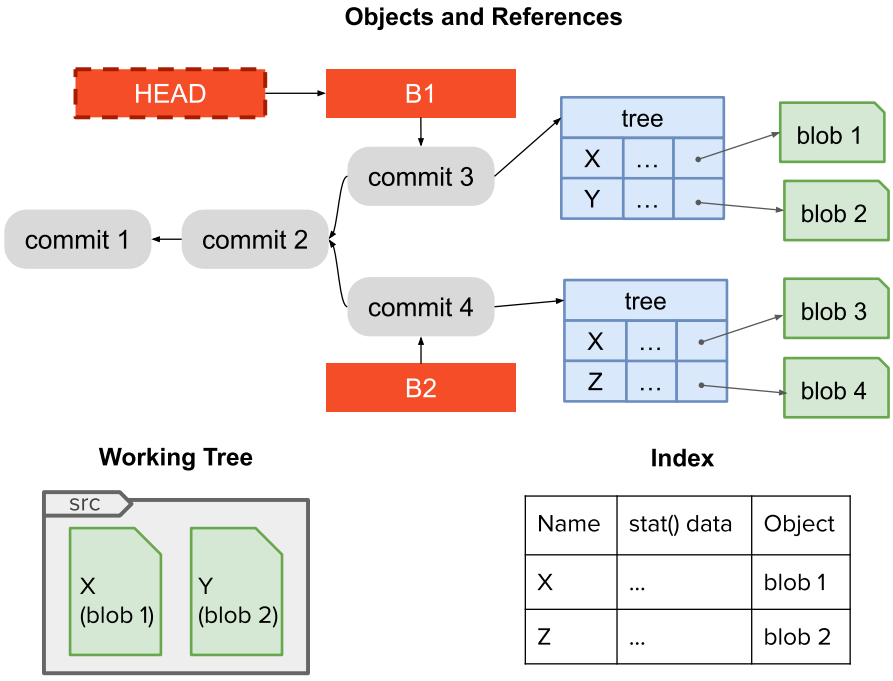


Figure 2.6: *Repository representation for a checkout example (pre-checkout).*

In order to switch to branch B2, the user may run the command `git checkout B2`, which will do the following: First, the index will be loaded from disk. The file currently contains two entries: X and Y. Then, Git will load the commit at the tip of branch B2, as well as the tree pointed by it, to update its in-memory index entries (i.e. Step 2). During this process, Git will mark the entry Y to be removed, update the entry X, and create a new entry for Z (all in the in-memory index). This leads to Step 3, which will update the working tree. At this point, Git will replace the old X file with the new one (which has different contents), remove Y, and create Z. Then, after saving the metadata from files X and Z in the new in-memory index, this in-memory structure will be written to disk, replacing the old index file.

The main focus of our parallelization effort is Step 3, which is responsible for updating the working tree. This step — and also the one after it — are performed by the `checkout_entry()` function, which receives one cache entry and produces the necessary changes in the working tree to make it up to date in respect to that entry. Normally, checkout users that call this function directly will do so in a loop to update multiple paths. One example of such user is `git checkout-index`, which does not need to perform any prior changes to the index, only write out the paths that were requested by the user.

However, it is more common for a checkout-related command to perform index modifications first, and then check out the modified entries in the working tree. While the specific type of index change may depend on the running command, a very common operation is to load one (or more) tree object(s) into the index, like we saw in the previous example. This process is the main goal of `unpack_trees()`, which, after updating the index, can also be optionally instructed to perform the necessary `checkout_entry()` calls and update the working tree to make it match the new modified index. Consequently, the `unpack_trees()` API can take care of steps 2 to 4 all by itself.

These two functions make up the main "entry points" to the checkout machinery; the higher level `unpack_trees()` API and the lower level `checkout_entry()`. The following two subsections will discuss each one of them with more details, but focusing on the latter, which is the main focus of this work.

2.5.1 The `unpack_trees()` Function

`unpack_trees()` traverses one or more tree objects while performing a "trivial merge" among them and the index. The inner workings of this process depends on the number of trees and merging function. We will not go into too much details about it here, but you can read more at the Appendix B and Git's "trivial merge" documentation (Git Development Community, 2021d). The `unpack_trees()` machinery also updates the working tree, if requested by the caller, and checks for unsaved data that could be lost in the process to properly warn the user.

While `unpack_trees()` prepares the new candidate index, it marks the entries that were changed so that the proper updates can be made in the working tree at the end. This is done using two temporary flags: CE_UPDATE, to indicate paths that must be created or updated, and CE_WT_REMOVE, to indicate paths that must be removed. Note that, while some Git commands show information about file renames, the checkout machinery handles them like two distinct operations when updating the working tree: a file removal followed by a file creation. This means that there will be two entries in the in-memory index: one for the pre-rename file, which will be flagged with CE_WT_REMOVE; and one for the post-rename file, which will be flagged with CE_UPDATE.

The two sets of flagged entries are handled separately in two consecutive loops. First, `unpack_trees()` iterates through the CE_WT_REMOVE entries to delete the respective files from the working tree. This is done by `unlink_entry()`, which first checks that the leading components of the path are real directories, to avoid following symbolic links and potentially removing the wrong file. (Git handles symlinks in the working tree by tracking the links themselves, not the target files; so it should **not** follow symlinks when removing

or checking out entries.) The check is done using `lstat()`, and some results are cached for better performance. We will describe more about this cache later, in Section 2.6. At this step, if a directory gets empty after removing one or more files, it is also removed. Then, the process is recursively repeated at the parent directory.

After processing the CE_WT_REMOVE entries, `unpack_trees()` loops through the CE_UPDATE calling `checkout_entry()`, which creates the necessary files in the working tree.

2.5.2 The `checkout_entry()` Function

The `checkout_entry()` function implements the core procedures to materialize the files referenced by the new candidate index into the working tree. Given one index entry, it does the following steps:

1. **Check the CE_WT_REMOVE flag.** Remember that, although `unpack_tree()` processes these entries in a prior separated loop, some users of the checkout machinery call `checkout_entry()` directly, so it was extended to check for this flag and call `unlink_entry()` when it is set. If the flag is set, the function return early (after removing the entry).

2. **Check the leading components of the file**. This is done by calling `lstat()` on each dirname component and making sure that it is a real existing directory. Like the dirname check done when removing old files, some `lstat()` results are also cached at this step, as we will discuss in Section 2.6.

   Missing directories are created. If one of the components already exist but not as a directory (i.e. as a regular file or symbolic link) Git either removes and replaces it by a directory, or aborts the operation to avoid overwriting unsaved data from the user (where "unsaved data" can be either untracked files or unstaged modifications). This behavior is controlled by an internal flag, passed to `checkout_entry()`.

3. **Check the file existence and status.** This is done by `lstat()`-ing the file and, if it exists, comparing the returned data with the data cached in the index entry to see if they match. In some specific circumstances the stat data may be insufficient to tell whether the file is up to date. So it may also be needed to read, convert, and hash the file's contents to compare it with the cached hash. But this is not needed in general as the stat data is typically sufficient.

   The comparisons can lead to three different states:

   (a) **The file is already up to date.** There is nothing to do in this case, so the function returns early with a success status.

   (b) **The file exists but it does not match the index entry.** In this case, Git either removes the old file to make room for the new one or aborts the operation to avoid deleting user data. Once again, this decision is made through an internal flag passed to `checkout_entry()` by its caller.

We can also reach this case due to path collisions. E.g. when trying to check out case sensitive files like `FILE` and `File` on a case-insensitive file system, the two paths will correspond to the same file on disk. The collision is treated exactly like any other "existent but not up to date file". The only exception is on clone, when the path gets added to a list of collisions which is later reported to the user. Using the currently implemented mechanics, Git can only do this report on clone, because it is certain that the working tree was empty before checkout and, therefore, finding an already existent file when trying to check out an entry can only mean that there was a collision. However, such mechanics would not work on e.g. a branch switch.

Note that, instead of removing the old file just to create the new one, Git could simply open the existing file and overwrite its contents. However, it is much easier to recreate it, as the file system will already set the right permissions for the new file, taking the current umask value into consideration.

(c) **The file does not exist yet.** This is the most straightforward case as there is nothing to be removed, only created.

4. **Write the file.** This step is handled by a lower level function which: loads the associated `blob` object from the object store (i.e. reads, unpacks, and delta-reconstructs it); passes the blob to the required conversion filters (which might include running external commands); creates the file in the working tree, and write its contents. For large blobs, this process is done using a streaming mechanism to avoid handling the blob in memory all at once. The streaming interface loads one section of the blob at a time, converts it, and write the result, repeating this until the entirety of the blob is processed.

   The disk reading is performed through memory mapping, both for loose and packed objects (as well as for the packfile idx and the Git index itself). This is a form of lazy I/O that maps sections of files on the disk to pages in the virtual memory. The mapping is initialized with a `mmap()` call (which is likely to execute very fast), and then the application can use the returned pointer as if the whole data was already available on memory at that address (although it is actually not). When the application attempts to access the data, it generates a page fault, making the kernel read the respective section on the file and load it into the page that the application wants to access. All of this is done transparently to the application, which can direct access the address returned by `mmap()` without even knowing what is happening under the hood.

   Regarding the conversion phase, users can specify zero or more filters to be applied upon the stored blobs before writing them to the working tree. These are called smudge filters. There also exists a filtering mechanism for the other way around, i.e. to convert the contents of working tree files before adding them to the object store. These are called clean filters. Some filtering operations are performed by the running Git process itself, like end-of-line conversions and re-encodings, but users can also specify external filters to be invoked by Git. These can appear

in two variations: one-shot filters and long-running process filters. The first only processes one blob per process, receiving the blob's contents through `stdin` and delivering the result to `stdout`. The second requires a more robust protocol, as the same process is capable of filtering multiple blobs. Long-running filters are usually invoked only once during checkout and persist throughout the whole operation, receiving blobs from Git and handling back the results.

5. **Retrieve the file metadata from the just written path.** This includes file attributes like the size, mode, last modification time, etc. I.e. the data gathered with any function from the `stat()` family. The metadata is then saved in the in-memory index entry associated with the just written file.

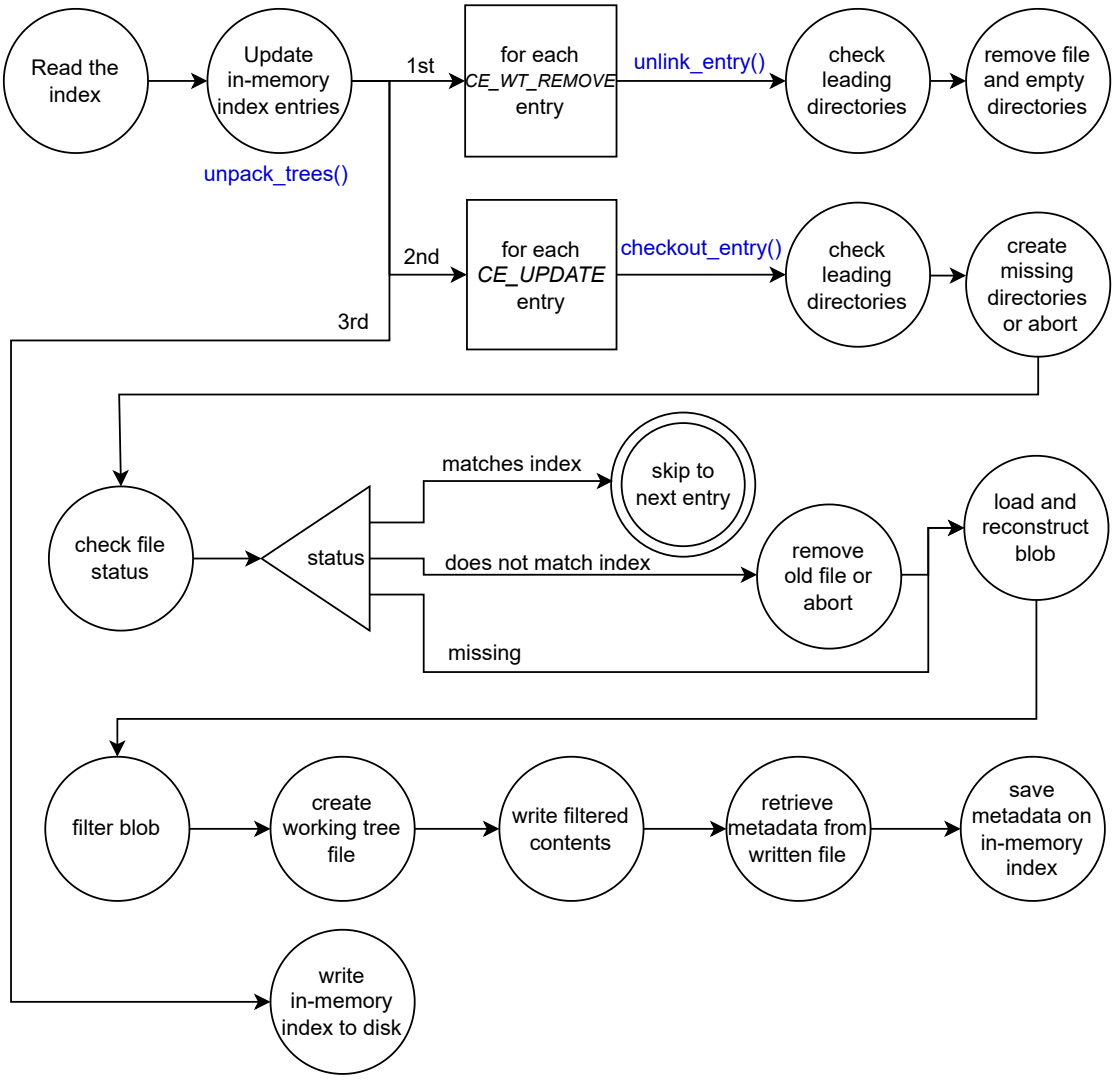


Figure 2.7: *Diagram of the different steps commonly taken during checkout.*

Figure 2.7 summarizes the different steps described in this section in a simplified diagram, with emphasis on the `checkout_entry` tasks.

Returning to the example from the beginning of this section (Figure 2.6), we can now

expand on the working tree update phase. Remember that we analysed a checkout operation that switches from branch B1 (with files X and Y) to branch B2 (with files X and Z). Also remember that X had different contents at the two branches. Skipping directly to the working tree update phase, we would first iterate over the `CE_WT_REMOVE` entries, which in this case is only Y, and remove the associated files from the working tree. Then we would look at the `CE_UPDATE` entries, i.e. X and Z. Starting with X, we would look at its leading directories. Since it has none, there is nothing to do at this step. Then we would check the file status, which shows that the file is present but does not match the index (because the in-memory index is already updated to reflect B2, while the working tree file still has the contents from B1). So Git would remove the file. Finally, it would load the blob that has the contents for X at B2, create a new empty X file in the working tree, and write the data. (There is no filter specified in this case, so the data read from the object store is ready to be written in the working tree.) This process is then repeated for Z, with the difference that there is no Z file in the working tree, so there is nothing to be removed.

There are a few important caveats to how we described the `checkout_entry()` steps. First, we focused the explanation around entries associated with regular files, which should be the vast majority of files on most repositories. Nevertheless, symbolic links and submodules follow a similar process. For symbolic links, the major difference is that Git does not apply any smudge filters and neither stream write them, as their contents are much shorter then regular files. As for submodules, the checks at Step 3 are a little different, and the submodule's checkout is handled in a separated child process. But the main process spawns the subprocess and waits for its completion before continuing with the other entries.

Another consideration worth mentioning is how `unpack_trees()` and `checkout_entry()` end up duplicating some working tree checks (e.g. to evaluate whether a file is up to date). One reason for this is that not all users of the checkout code use the higher level `unpack_trees()`, so the checks must be at `checkout_entry()` too. But this is only part of the story, as even the ones that do will also need the checks at the lower level function to detect and react to some scenarios. The duplication is obviously not good for performance, so there were propositions on how to remove them. We will discuss more about it at Section 5.4.

Finally, although it was important to study how `unpack_trees()` works to make sense of `checkout_entry()`'s role during a checkout, our goal with this work is to improve the performance of checkout operations that need to create many files in the working tree. Therefore, as previously mentioned, our main focus is at the `checkout_entry()` calls, which we wish to spread among multiple workers. It is beyond the scope of this project to accelerate other sections like the tree walk and merging code or the index parsing/writing, for example.

2.6 The `lstat()` Cache

As described in Section 2.5, all paths modified during checkout first have their leading components checked with `lstat()`, to make sure that they are all real existing directories.

If not properly handled, this step can incur in an enormous amount of `lstat()` calls. Furthermore, since many files share the same leading directory (or part of it), most of these calls will be redundant. For example, the checkout of tag v5.12 in an empty working tree of the Linux repository incurs in the creation of about 72 thousand regular files distributed over 4700 directories. If we were to call `lstat()` for every directory component of each one of these 72 thousand files, we would make over 906 thousand calls, which is way more than the number of directories itself. To optimize this process, Git uses an internal `lstat()` cache, which is capable of reducing the number of calls in this example to roughly 13 thousand. There are still redundant calls, but the cache leads to an impressive 70x reduction. The caching is very efficient as many checkout operations will process the index entries in order, so paths within the same directory are adjacent to each other.

The cache does not cover all uses of `lstat()`, though, just the ones issued through one of the functions provided by the cache's API. Some examples are `has_symlink_leading_path()`, which returns true if and only if one of the directory components of a path is a symbolic link, and `has_dirs_only_path()`, which returns true if and only if the full path only contains real directories. Each function from this API caches different things, but all of them use the same global storage[4], which basically consists of a string buffer and some flags. Reusing our two API examples, `has_symlink_leading_path()` caches both directory and symlink entries, while `has_dirs_only_path()` only caches directories. The cache works by storing the last longest path checked and known to be one of a specific type that the function wants to cache. For example, if we are caching symlinks and directories, at some point the cache might contain the string `X/Y/Z` and the flag `FL_SYMLINK`, indicating that `Z` is a symlink. Note that, in this case, we also get the extra information that `X` and `X/Y` are real directories.

During checkout, the `lstat()` cache is used in three places: first during the tree walk and merging phase of `unpack_trees()`, then for file removals (the `unlink_entry()` calls), and finally for file creations (the `checkout_entry()` calls). The last two both need to check whether the leading part of a checkout path contains real directories, but they do so with slightly different mechanics, so they use different functions, which cache different things. To give an example of how the checkout code use this cache, pretend we need to check out paths `A/B/C` and `A/B/C2`. We will focus only on file creations, for now, because that is the main goal of the parallel checkout effort. At this phase, the code caches directory entries and nothing more. Starting with `A/B/C`, we would pass `A/B` to the `has_dirs_only_path()` function, which would either `lstat()` both `A` and `A/B` or retrieve the information about these components from the cache. Say `A` is cached, but `A/B` is not. The function knows `A` is a directory, and now it only has to `lstat() A/B`. If it is also a directory, the function stores this information in the cache and returns true. Now suppose the next entry to be checked out is `A/B/C2`. We know `A` and `A/B` are real directories because we have cached this information, and now `has_dirs_only_path()` can return fast without even calling `lstat()`.

[4] With the exception of some thread-safety variations which allow each thread to pass its own cache state as an argument.

Chapter 3

Profiling

For a successful parallelization, it is important to understand what are the hotspots in the code, and whether they are well parallelizable or not. Otherwise, the developer might end up parallelizing tasks which are not critical for performance or investing too much time in places that cannot really be parallelized in an efficient way.

Conversely, some problems are well parallelizable but not in the way that the current implementation is designed. In this sense, the early study of performance can also assist developers to plan what refactoring can be made in order to allow a greater degree of parallelism in the code, before actually implementing any parallelism.

3.1 Tools

Profiling tools can be of great help at this stage. But it is important to understand what *exactly* each tool is tracing or sampling, in order to choose the appropriate one for each use case. The code we were aiming to parallelize performs many I/O operations: reading the objects from the database, creating files and directories in the working tree, writing their contents, querying the file system for the metadata, etc. For this reason, relying *only* on a CPU time profiler, like `gprof` would not give us the right intuition as to which role I/O and other off-CPU waiting time play during checkout.

Another requirement for our choice of profiler was that it should capture both system call times as well as userspace calls, and let us see exactly which functions on the second group lead to the ones on the first group. This is very important in checkout's case because some application functions may in fact spend most of their time in lower level system code, doing apparently unrelated tasks, and we would not be able to visualize that otherwise. The most obvious example where this can happen is during disk reading: as Git reads the objects and the index file using `mmap()`, the actual disk I/O only happens when Git tries to use the data and generates a page fault. This makes profiling more challenging since the object reading times are mixed with the actual usage of the data during the decompression phase. If our choice of profiler was not able to identify and separate system routines from application routines and show us the caller-callee relationships, it would not be possible to see what percentage of time was actually spent on I/O vs. decompression code.

Furthermore, to have the full picture of the call chains, we also wanted our profiling tool to be able to report functions from both static and dynamically linked libraries, like zlib and glibc. This gives us a greater level of detail in the profile results, and allow us to better visualize what are the real bottlenecks in each case, if any.

The `bcc-tools`[1] project has exactly what we needed to fulfill these requirements. Besides being a development toolkit for Linux eBPF[2] programs, the project also contains a collection of tools using this technology to monitor stack traces, disk usage, network statistics, and other aspects on Linux machines, including profiling. Furthermore, because eBPF programs run in the kernel context (sandboxed), eBPF-based profilers are more efficient and cause less overhead to the application being profiled.

The two `bcc` tools we used are `offcputime` and `profile`. The first gives a summary of the time spent off-CPU per call stack, and the second is an on-CPU sampling-based profiler. Combining the resulting data from both tools we can see the full picture of where the wall clock time is spent during a checkout operation.

To make sure these tools would be able to resolve the function addresses into the proper symbols, we compiled the libc, zlib, and Git using *-g -fno-omit-frame-pointer*. Additionally, due to an issue in the `bcc-tools`[3] the symbols cannot be resolved if the application being profiled ends before the profiler itself. To solve that, we modified Git to receive the PIDs of the `profile` and `offcputime` processes and signal them to terminate right before Git itself finishes.

After collecting the data from both profilers, we used Brendan Gregg's `FlameGraph`[4] tool for visualization. The output from `offcputime` is already in time unit, but `profile` (the on-CPU sampler) only gives the number of samples it took at each function. So we divided its output by the configured sample rate (which, in our case, was 49 Hz) to have the same unit for the on-CPU and off-CPU data.

3.2 Profile Results

As previously mentioned, the checkout machinery is used by many Git commands. But since our primary goal was to speedup operations with many file creations, we chose to profile a command where this is the main bottleneck: a `git checkout .` execution on an empty working tree of the Linux kernel repository (version 5.12). This is similar to the first checkout users see right after cloning the repository. It is responsible for the creation of over 70 thousand files.

The Linux repository is frequently used for Git performance tests because it is one of the largest repositories publicly available. Besides its 70 thousand working tree files, the

[1] https://github.com/iovisor/bcc

[2] https://ebpf.io/

[3] https://github.com/iovisor/bcc/issues/2883#issuecomment-617465106

[4] https://github.com/brendangregg/FlameGraph
At the time of writing, the latest tagged release of FlameGraph is version 1.0 from 2017. We used a more recent version from the Git repository, at commit `1b1c6deede (``more java matching'', 2019-02-16)`, with a few coloring modifications to support our I/O-and-CPU format.

commit history has about 8.2 million objects, today, putting the total size of the repository at around 4.5 GiB (about 3.5 GiB comes from the gitdir alone).

Figures 3.1, 3.2, 3.3, and 3.4 show the results respectively on an SSD, two different HDDs, and an NFS setup. The hardware description for all of the machines used in tests for this work can be found at Appendix C. For convenience, a summary of the machines that appear in this chapter is also provided at Table 3.1.

Machine	Storage	CPU	Memory
Mango	SSD PCIe NVMe v1.2, ext4	Intel(R) Core(TM) i7-7700HQ, 2.80GHz, 8 logical cores	16 GiB (2 x 8GiB) DDR4, 2133 MT/s
Cicada	HDD 5400 rpm, SATA 2.6, ext4	Intel(R) Core(TM) i5-3317U, 1.70GHz, 4 logical cores	6 GiB (1x 4GiB 1x 2GiB) DDR3, 1333 MT/s
Wall-e	HDD 5400 rpm, SATA 3.0, ext4	Intel(R) Core(TM) i5-4210U, 1.70GHz, 4 logical cores	8 GiB (1 x 8GiB) DDR3, 1600 MT/s
NFS EBS gp3 * Note: client and server on AWS EC2 instances	NFS v4.1 backed by a EBS gp3 volume (SSD) with xfs. (NFS Server)	Intel(R) Xeon(R) Platinum 8124M, 3.00GHz, 4 logical cores (NFS client)	10 GiB (NFS client)

Table 3.1: *Summary of the hardware used for profile tests. All machines running a Linux distribution. Refer to Appendix C for the full hardware and software descriptions.*

In these graphs, each rectangle corresponds to a function call, and its width represents the time spent on it. The y-axis is ordered in caller-callee fashion from bottom to top, thus showing the call stack depth. The x-axis represents the proportion of run time used by each function. The graphs are also color coded: blue and turquoise correspond to off-CPU time, while red and yellow correspond to on-CPU time.

Additionally, the lighter colors on top (,) represent kernel functions, which are also separated from Git functions through gray "-" rectangles.

For research purposes, we also profiled this operation from a version of the Linux repository containing only the objects from v5.12 expanded to loose format (see flamegraphs at Appendix A). This repository was created using the `--depth=1` option of `git clone` and then expanding the generated packfile with `git unpack-objects`. This resulted on a repo with about 76 thousand objects totaling 260MiB (or about 446MiB of disk space[5], without the working tree. Note that such a repository is **very unlikely** to exist naturally since Git packs the loose objects when their number surpasses the 6700 threshold. Our artificially created repository has almost 12x that number, so we had to disable automatic garbage collection with `git config gc.auto 0` to avoid packing

[5] Files are allocated on disk in clusters, which defines the minimum allocation unit of the file system. If the data is smaller than the cluster size, the file will still occupy a full cluster on disk. Therefore, there can be discrepancies between the data size and the size on disk.

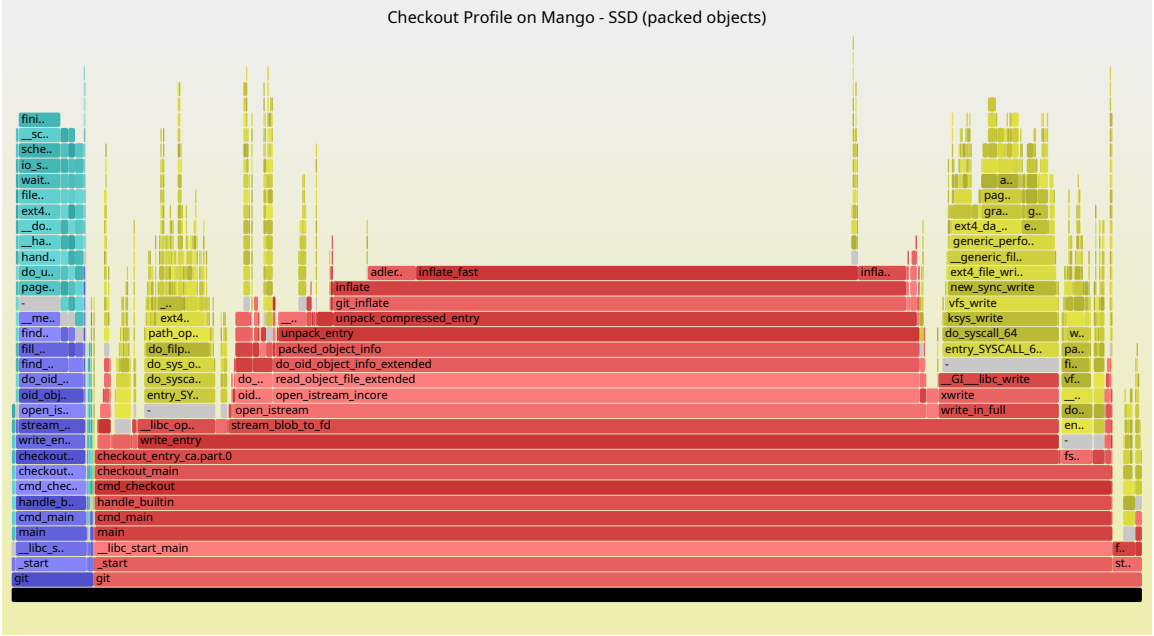


Figure 3.1: *Checkout profile flamegraph on machine Mango - SSD (packed objects case).*

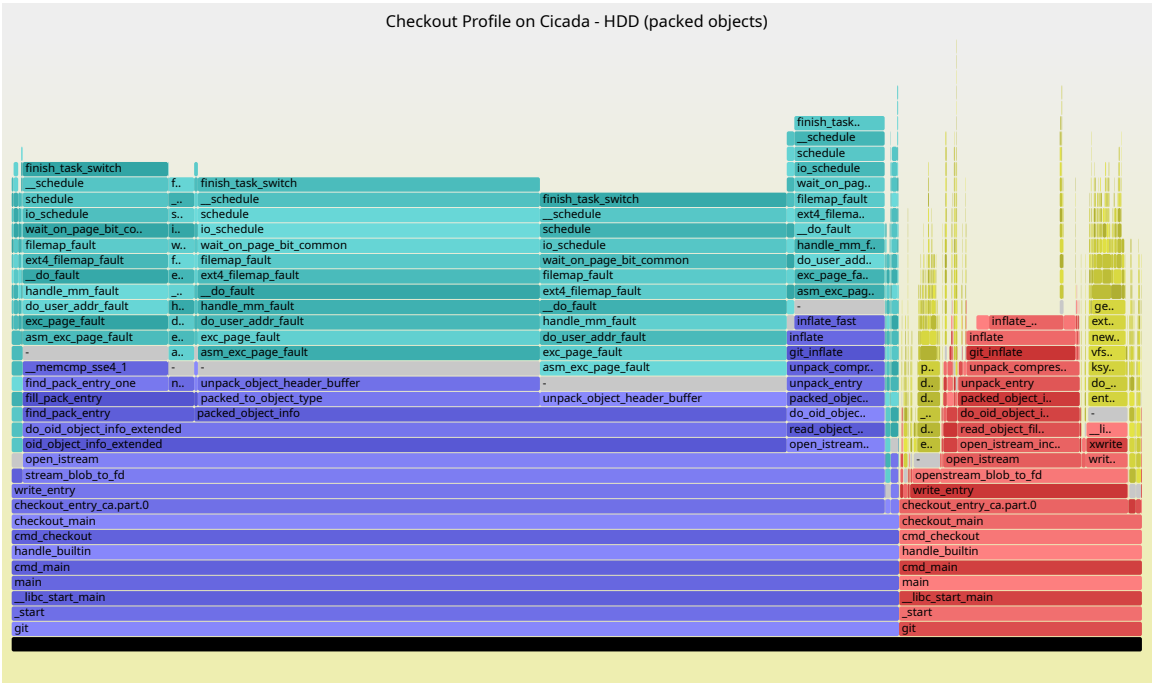


Figure 3.2: *Checkout profile flamegraph on machine Cicada - HDD (packed objects case).*

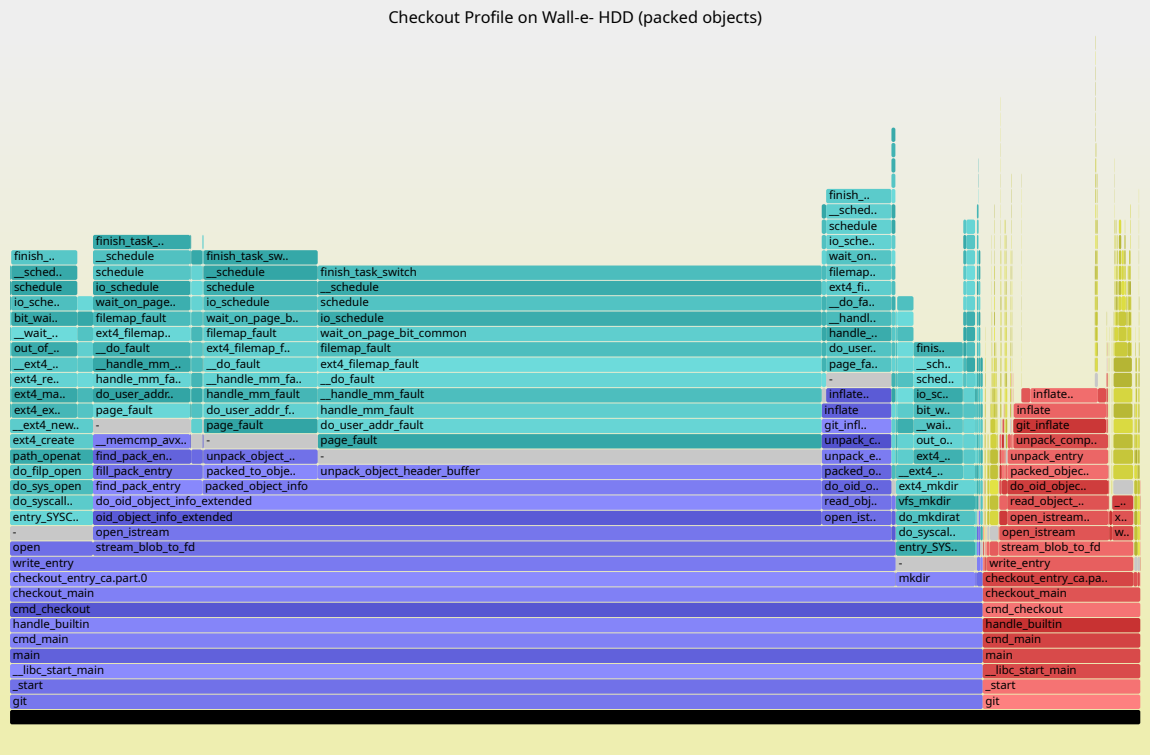


Figure 3.3: *Checkout profile flamegraph on machine Wall-e - HDD (packed objects case).*

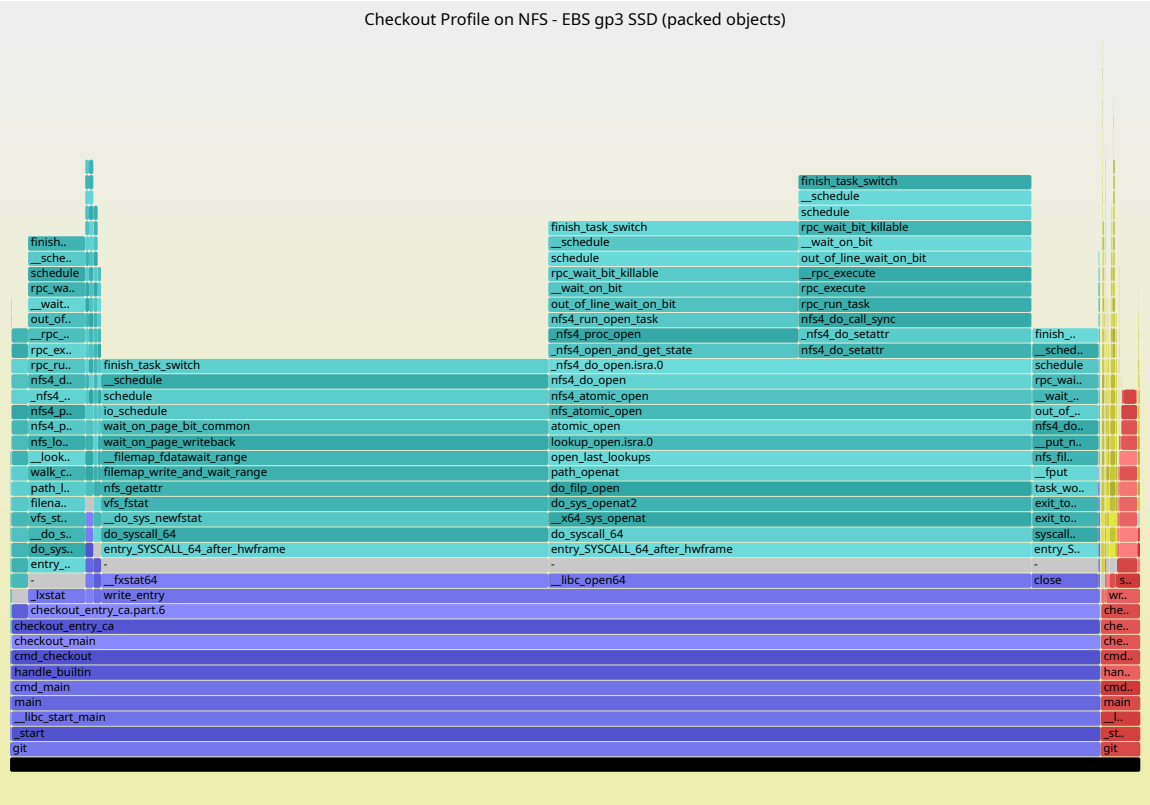


Figure 3.4: *Checkout profile flamegraph on NFS - EBS gp3 - SSD (packed objects case).*

while running the profile and benchmarks. Since this is an artificial example, we merely used it to compare how checkout performs with loose and packed objects. **The results from the profiling and benchmarks using loose objects should not be interpreted in any more general context besides experimentation.**

The flamegraphs have a substantial amount of information and it can be tricky to collect insights directly from them. So, in the following sections, we will summarize the most time consuming operations for each machine and dissect the data on a case-by-case basis.

3.2.1 SSD

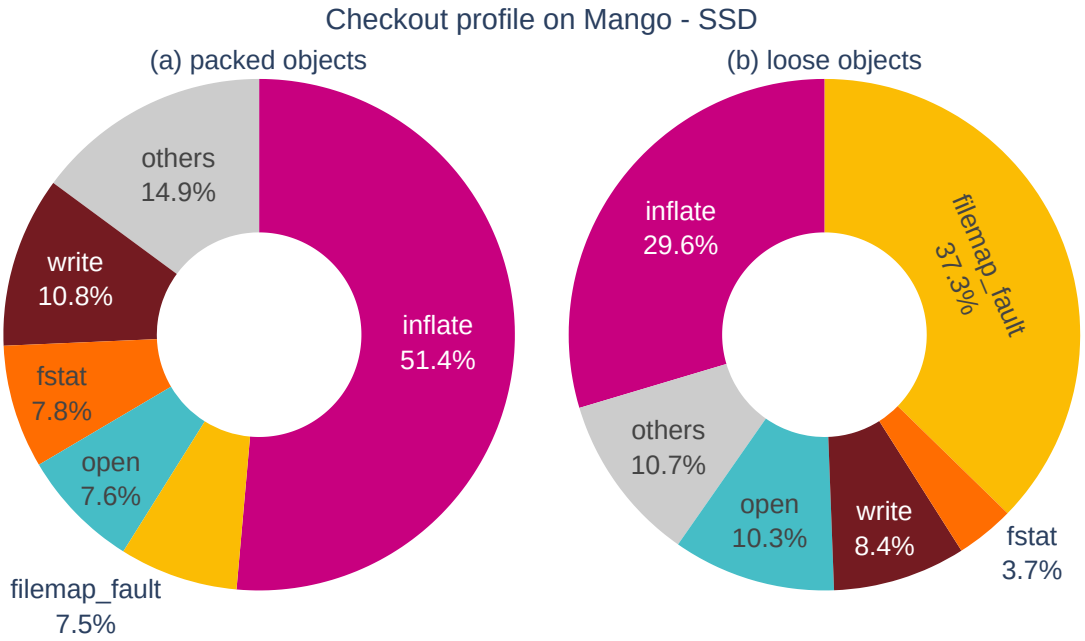


Figure 3.5: *Summary of checkout profile on machine Mango - SSD*

Starting our analysis with the packed objects case (Figure 3.5a), we can see that more than half the runtime was spent in the decompression routines alone. I.e. zlib's `inflate()` function. That is great news for parallel checkout, since inflation is a thread-safe CPU-bound operation and we have to call it for each object, meaning that it can be highly parallelizable. We also see a good amount of time spent on I/O, with `stat()`, `write()`, `open()`, and `filemap_fault()`. The last one in this group is the kernel function responsible for reading data from a memory mapped file during a page fault. Since the SSD is very fast, reading does not take so much time and CPU becomes the bottleneck.

Nevertheless, parallelism should increase the number of reading requests arriving at the I/O queue, giving the operating system a better chance to reorder and merge them in a way that further increases the throughput. Furthermore, SSDs are known for having good internal parallelism (Chen, Lee, *et al.*, 2011) (Chen, Hou, *et al.*, 2016) so the concurrent reading requests should be parallelizable to some extent.

Looking at the loose objects case (Figure 3.5b), we can see that `filemap_fault()`

takes a much bigger chunk of the total runtime. Besides, the average runtime (for 15 samples) went from 8.6 seconds to 15.7 seconds. It is worthy noticing that the reading patterns from these two checkout cases are quite distinct: with packed objects, we read multiple different sections of a single large (3.2GiB) file, the packfile; whereas on the loose objects checkout, we actually read 76K small (~6.1KiB) individual files.

Due to the higher compression level of packfiles, one could hypothesize that the overall reading size must be smaller in this case. To test this idea, we used the `vmtouch` tool[6], which allows us to see how many pages of a file are resident in memory (i.e. physical memory). Generally, this is not a precise test as some of the pages that were loaded during checkout may be evicted from memory before we get to run `vmtouch` (or even during the Git execution itself). To mitigate this issue, we ran the test when the system was mostly idle, and choose a machine (Mango) that has enough RAM to avoid premature page evictions. To verify this, we loaded all files from both clones of the Linux repository (i.e. the packed and the loose repo), and the machine was able to hold all the 6.1 GiB in the page cache simultaneously.

Running `vmtouch` after the checkout execution we are profiling (also taking care to drop the file system caches before running Git), we found that 436MiB of the `.git` directory were resident in memory after the loose objects checkout. Whereas on the packed objects case, the resident size was 551MiB, which shows that our initial hypothesis is incorrect. What we did not consider was that, in the packed case, Git also had to read many sections from the packfile index. (If we only look at the objects themselves, the resident size is about 354MiB.)

Another hypothesis is that the packed object checkout takes better advantage of read ahead, thus increasing its reading performance. Read ahead is a mechanism by which the kernel will read more bytes than what was actually issued on a read request, on the premise that the application is likely going to require those extra bytes later. Since the packfile and its index are **way larger** than the loose object files, reading operations could be making better use of the read ahead windows on those large files. On the other hand, those two files are read with a mix of sequential and random patterns, and sequential readding is usually what benefits the most from read ahead. Nevertheless, packfiles still have good locality, and the SSD machines where we ran the tests have enough RAM to avoid the need for swapping out packfile pages during the checkout execution. This means that even though the sections loaded through read ahead mechanics might not be needed right away (as if we were doing sequential reads), Git can still benefit from them being in the cache since it may later need to process an object in that vicinity.

To test this new hypothesis, we ran a benchmark checking out v5.12 on both Linux repositories (i.e. the packed and the loose versions) with and without read ahead. We disabled read ahead by running "`blockdev --setra 0`" on the SSD device of machine Mango, and re-enabled it with "`blockdev --setra 256`", which sets the read ahead to 256 blocks of 512 bytes (totalling 128 KiB). This was the default read ahead size on this device. To be extra careful, we also modified Git to issue `posix_fadvise()` calls on opened packfiles, idx files, and loose object files, giving the `POSIX_FADV_RANDOM` hint.

[6] https://hoytech.com/vmtouch/

This informs the kernel that the application will not be reading the data sequentially, thus allowing it to decrease or disable the read ahead window for that file. Table 3.2 shows the result of this test, using a sample size of 15 executions for each test case. As we can see, the affect of disabling read ahead is indeed much more prominent on the packed objects case, where the runtime was reduced by more than half.

	With read ahead	No read ahead	Speedup
Packed Objects	8.681 s ± 0.296 s	18.910 s ± 0.385 s	0.46 ± 0.02
Loose Objects	15.544 s ± 0.148 s	17.622 s ± 0.197 s	0.88 ± 0.01

Table 3.2: *Checkout benchmark on machine Mango with and without read ahead.*

3.2.2 HDD

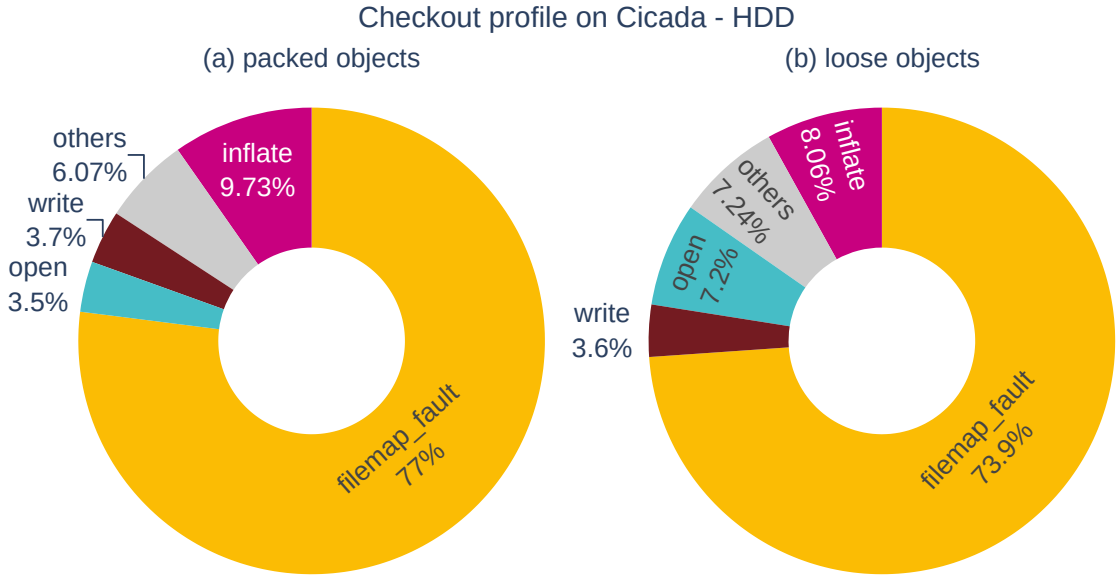


Figure 3.6: *Summary of checkout profile on machine Cicada - HDD*

On both the HDD machines (Figures 3.6 and 3.7), we see `filemap_fault()` dominating the runtime for both packed and loose objects, with over 70%. The higher reading times were already expected, as HDDs rely on spinning disks and mechanical arms to read the data, making them significantly slower than SSDs. So much so that I/O becomes the major bottleneck and zlib inflation only takes 6~10% of the time. This might not be a good scenario for parallelization, as HDDs do not usually process random requests very well. As Boito *et al.*, 2018 mentions: *"HDDs are known for presenting the best performance when accesses are done to sequentially positioned blocks instead of randomly because it minimizes seek time."* Note that the authors use "sequentially" in terms of physical disposition of the data (i.e. "contiguously"), not the number of threads or processes. Thus, the sequential checkout likely already suffers from some amount of random access, as the required objects may be somewhat apart from each other (both the loose object files, which can be

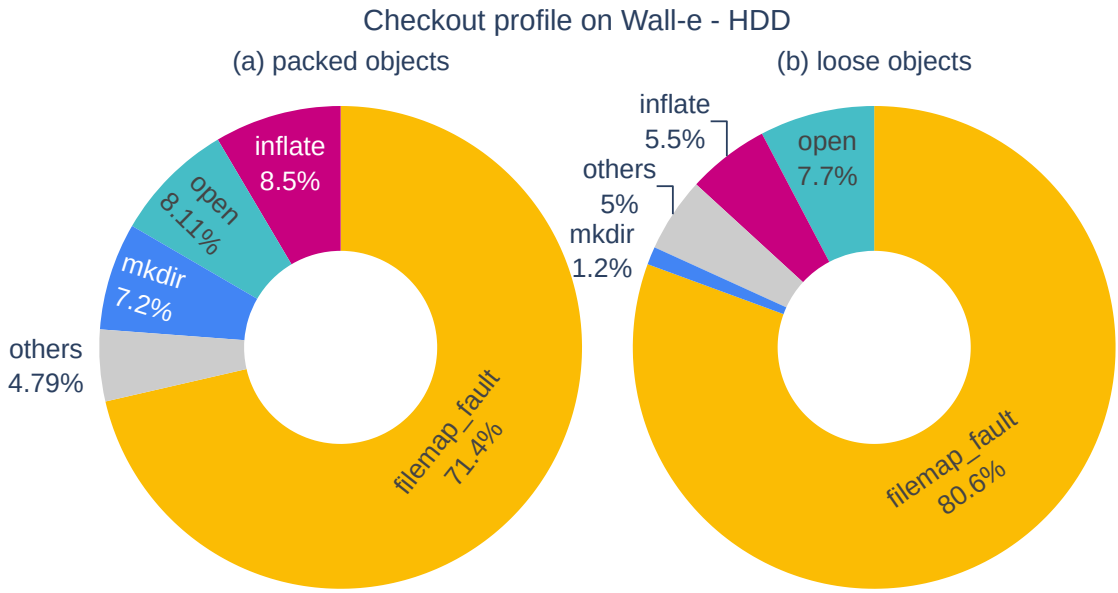


Figure 3.7: *Summary of checkout profile on machine Wall-e - HDD*

scattered on the disk, and the packed objects which, despite the good locality of packfiles, might not be that close either). Nevertheless, parallel checkout may further increase the randomness in the access patterns, increasing the seeking time and the I/O latency.

In some cases, however, the increased number of outstanding reading requests on the I/O queue might allow better optimizations to be employed by the I/O scheduler, as we mentioned at Section 3.2.1. Nevertheless, note that the HDD will not benefit from this effect for internal parallelism, as the SSD does, due to the differences in their mechanics. We will discuss these topics further on Section 4.5.

Still looking at the HDD summary plots, one could ask why the reading operations take so much time while writing is actually very fast. One of the key reasons behind this difference in performance is that writes are usually cached/buffered on Linux, thus deferring the actual disk operations. This is called write-back caching, and it makes writing much faster. The application can of course override this behavior by, for example, using `fsync()` to flush the data before closing the file, or opening the file with the `O_SYNC` flag (which is similar to calling `fsync()` after each `write()` call). However, Git does not use any of these mechanisms when writing to the working tree.

To see what impact the synchronous data flush would produce on the HDD checkout, we repeated the profile tests using two modified versions of Git, each employing one of the techniques we just mentioned above. For the "fsync-on-close" modification, checkout took 1h04m on the packed case (54x times slower than plain Git) and 1h13m on the loose case (44x slower than plain Git). On both cases, `fsync()` was responsible for 94~97% of the total run time. For the `O_SYNC` modification, the Git command took 1h37m on the packed case (82x slower than plain Git) and 1h54m on the loose case (69x slower than plain Git). Furthermore, `write()` was the dominant function with 96~98% of the checkout

run time.

3.2.3 NFS

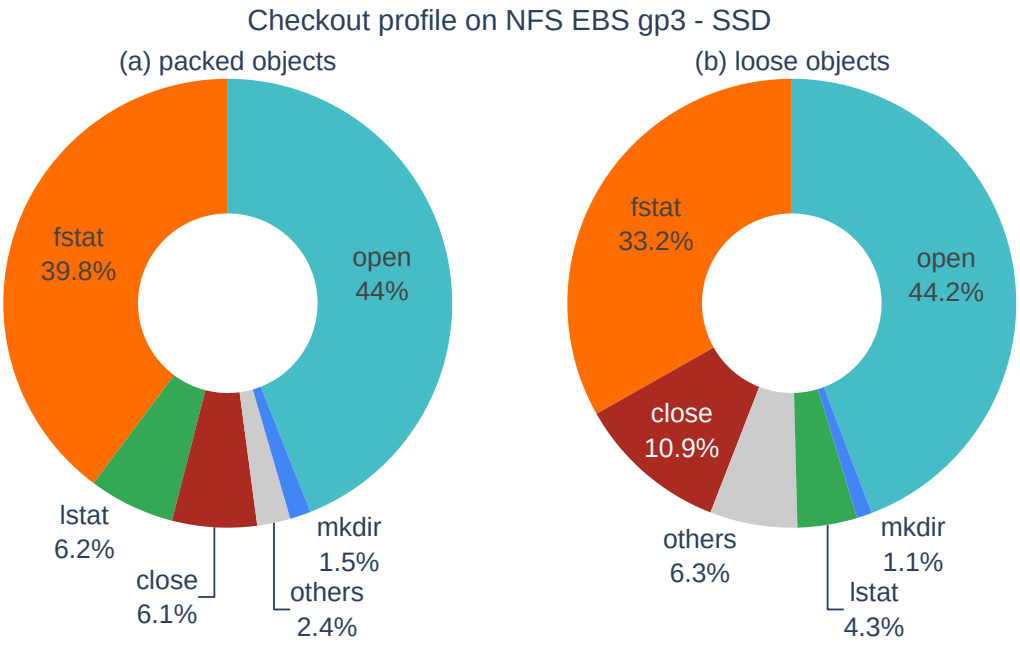


Figure 3.8: *Summary of checkout profile on NFS - EBS gp3 - SSD*

The NFS profile (Figure 3.8) is quite different from the previous ones, but it is expected since the I/O mechanics are also very different there. First of all, notice that the relative time taken by `inflate()` and `filemap_fault()` is now so small, that the functions no longer appear in the pie plot[7] (their contributions fall inside the "others" section). The two most time consuming functions on NFS were `open()` and `fstat()`, in that order. Both these calls require one (or more) round-trip(s) to the server, which is subject to network latency. The round-trip costs may only be a small overhead when writing a large file, for example, but in our checkout workflow where Git needs to create lots of small files, the costs add up and end up taking plenty of time.

For `open()`, we can see on the flamegraph that the most time consuming calls are the ones creating new files in the working tree. Git creates these files with the `O_CREAT` and `O_EXCL` flags, which means that the server must use exclusive creation semantics. This implies extra work to check for the file existence and also to ensure exclusivity for that particular NFS client. Under some circumstances, the exclusivity is implemented by storing a verifier in the space used for some of the new file's attributes, so the client needs to perform one additional network request, after the file creation, to properly set the initial values for its attributes. This means that the client must perform two network round-trips, one for the OPEN operation of the NFS protocol, and another for SETATTR.

[7] Just for comparison, the time taken by `inflate()` on NFS corresponds to about 1.4% of the total runtime on the packed case, and 1.01% on the loose case. For the `filemap_fault()` calls, these percentages are respectively 0.1% and 4.4%.

And we can use the flamegraph again to see that this is indeed what is happening in our case. See how the runtime of the `_nfs4_do_open()` kernel function is equally divided between another open subroutine and the `nfs4_do_setattr()` function.

Before talking about `fstat()` we must briefly discuss close-to-open cache semantics. We already mentioned that Linux typically uses write-back caching by default. On a local file system, the cache is shared among all the processes that are running in that machine, so they call all see the file modifications even before they make their way to the persistent storage device. However, on a networked file system, each client has its own local cache, which is of course not shared with the others. Therefore, to be able to use caching, which is so important for performance, while keeping some kind of consistency, the NFS protocol provides what is called "close-to-open" cache semantics. Roughly speaking, this means that machine B will be able to see the changes to a file made by machine A, as long as machine B opens the file after machine A closes it. This design allows NFS clients to perform `write()` calls locally on their caches (again, if the application does not request otherwise), but they must flush all written data to the server on `close()`. For this mechanics to work, the clients must also revalidate their caches on `open()`, by fetching file attributes from the server and comparing them to what is cached.

Because of the close-to-open cache consistency, we were expecting that `close()` would be very time consuming in our checkout profile, as it has to flush all the locally written data to the server. Furthermore, modern NFS servers normally use synchronous mode by default, which means that the server itself must flush the operations to its own disk before responding. So a client `close()` has an associated overhead of both the network flush and the synchronous I/O on the server. However, `close()` does not even appear in our plot of the most time consuming functions during checkout on NFS. We were initially quite intrigued by this observation, but then we inspect the client's sockets during checkout using `tcpdump` and `wireshark`, and found out that `fstat()` was triggering the kernel to perform an NFS WRITE operation. I.e. the kernel was flushing the previous local writes to the server on `fstat()`, and therefore, there was nothing more to be flushed at the time Git called `close()`, which made it much faster than what we expected. This also explains why `fstat()` was so time consuming. We can even see this fact at the flamegraph: see how `vfs_fstat()` spends almost the totality of its time on `filemap_write_and_wait_range()`.

With that in mind, it is in fact quite easy to understand why `fstat()` produces this behavior. This function is used to retrieve metadata about an open file, including but not limited to its size, mode, and last modification time. In order to get the correct value for some of this fields, the NFS client must flush the written data to the server beforehand. Otherwise, the value could be outdated. Confirming this hypothesis, there is a comment in the Linux kernel code at the `nfs_getattr()` function, right before calling `filemap_write_and_wait()`, which says "Flush out writes to the server in order to update c/mtime".

Having an understanding about how and why checkout spends its time on NFS, we can try to access whether a parallel implementation would be beneficial in this case. For both `open()` and `fstat()`, the two most time consuming functions, practically the entire time is spent off-CPU, either sending requests to the NFS server or waiting on its responses.

Since NFS servers are typically able to process multiple connections simultaneously, parallel checkout can be an effective way to amortize the network latency and also promote parallelism in server work associated with these operations.

In summary, the results from this chapter shows us that parallel checkout is quite promising for NFS, and likely to local SSDs as well, where there is a lot of CPU work that can be parallelized. (We can achieve even greater speedups if the SSD is able to benefit from the concurrent readings too.) However, local HDDs are still an open question. At Section 4.5 we will expand the discussion around the different aspects of SSDs and HDDs in regards to parallelism, theorizing which role this may play when running parallel checkout locally.

Chapter 4

Challenges

Parallelizing software is a complex endeavor. It usually requires a good amount of prior refactoring work, specially to remove or reorder task dependencies, expose parallelism, protect resources that must be shared, and duplicate the ones that do not. These are some of the most common obstacles in a parallelization effort. While overcoming them does not guarantee a well-performing parallel code, failing to do so can certainly limit the degree of parallelism achieved and, more dangerously, even lead to erroneous results in case of unnoticed race conditions. Most of the challenges faced during this project involve two of these issues: task reordering and shared resources. In this section we will describe some of these obstacles. Some of them only apply to multi-threading, and some apply to both thread- and process-based parallelism.

4.1 Accessing the Object Store

For each file that needs to be written, the checkout machinery must first load its associated blob from the object store. An internal API takes care of all the work associated with locating the object (which may be either in loose or packed format), reading the data, inflating, and delta-reconstructing it. This is a considerable amount of work, and it is pretty time-consuming, as previously discussed; so it is in our best interest to perform the loading of different blobs in parallel. However, there are many global resources in the object reading code:

- **The list of packfiles**: a global list of packfiles which may be opened and clone throughout the execution of the Git process.
- **Delta base cache**: Objects stored in packfiles are allowed to be deltified. On the process of reconstructing the delta objects, the same base might be required by more than one delta. To save time, Git uses a LRU in-memory cache of already read bases, which is shared among multiple threads.
- **Pack windows**: Packfiles hold many objects in a single binary file. To read these objects, Git uses memory mapped sections called `pack windows`. These can be invalidated for a number of reasons, such as when Git needs a new window and the

maximum number of opened windows was reached. The list of windows is global, which means that the usage and disposal must be protected in multi-threaded code.

- **Other global variables**: The object reading code keeps track of some statistics like the number of mmap calls and the maximum number of simultaneously open mmap windows. This data is stored in global variables, which are read and written during checkout. Finally, there are also function-scope static variables, which are persistent across multiple function calls and also shared among the threads. These variables are commonly used in Git either to avoid the cleanup section in functions that would otherwise need to release the memory before returning or to return data that the caller itself does not need to release after the use.

In a previous work (Bernardino *et al.*, 2020) we identified that the object decompression phase (i.e. reading the data from disk and inflating it, but not including the delta reconstruction) was one of the most time consuming steps in `git grep`. Some experiments using the Chromium repository as data showed that decompression accounted for up to one third of the total execution time, followed by regex functions with 28% and locking functions with roughly 13%. At that time, we introduced[1] a very coarse-grained mutex to the object reading code, which made it possible for multiple threads to safely read objects concurrently with a good performance. The catch was to protect **everything** but the zlib decompressing calls, which are already thread-safe. This approach allowed for a significant performance boost on `git grep` without risking to introduce race conditions. A by-product of that work is that we now have an API that could be used to read objects with parallel threads and overcome the challenges listed above.

However, the big lock added in that work only covers object reading functions. Checkout also uses global shared resources in other parts of the code that are not yet protected. Another problem is that some of the global resources accessed by the object reading machinery are also used by other parts of the codebase. One of the best examples is the global struct `the_repository`, which stores many important information about the repository where Git was invoked. This means that it is not sufficient to wrap the object reading functions with locking mechanisms, but we also need to evaluate what other parts of the code use the same global resources and make sure to also acquire the lock before executing them. Without this, the object reading lock **cannot fully accomplish its function**. When working at `git grep`, this evaluation was sort of easy as the code already had higher-level locks which we just needed to replace with the new object reading one. But this would be a whole different effort on checkout, which does not have any lockings or previous indications of thread-unsafe function calls.

4.2 Path Collisions

This refers to another form of resource sharing: the file system. Path collisions happen when the file system folding rules make different path names end up corresponding to the same file on disk. It can happen, for example, with case-sensitive files (like "`FILE`" and "`file`") on a case-insensitive file system. However, case-sensitivity is not the only

[1] 31877c9aec ("object-store: allow threaded access to object reading", 2020-01-15): https://github.com/git/git/commit/31877c9aec21e0824fd4fcf415069cf8dfae4b72

mechanic that can lead to collisions. See, for example, Unicode normalization: the Unicode character "ä" can be represented with two different sequence of bytes: `0303 0244` (in octal), which is the code for "Latin Small Letter A with diaeresis" or `0141 0314 0210` which is "Latin Small Letter A" plus the "Combining Diaeresis". Some file systems will convert the Unicode paths to a standard format and thus, these different representations of "ä" will be considered the same.

Some folding rules are even more complex. For example, the FAT file system on MS-DOS used 8.3 filenames also known as "short names" (Microsoft, 2021). This format allowed for a maximum of eight non-period and non-space characters, optionally followed by a period and one to three other non-period and non-space characters (which would be the file extension). For backward-compatibility with legacy applications, modern NTFS file system on Windows still support the 8.3 filename format. When this is enabled, the applications can refer to a file using either its long name or the automatically generated 8.3 name. This conversion from long to short name can be a source of path collisions. For example, the file ".txt" is not a valid short name, so it would be automatically assigned the short name "txt~1". Therefore, if we try to write "txt~1" after ".txt", the two would collide on disk. Furthermore, some Windows versions will ignore trailing spaces and periods too[2], so ".txt." and "txt~1." would also refer to the same file of ".txt" and "txt~1".

Path collisions bring many challenges to a parallel checkout implementation. Multiple threads/processes could be working on different paths without knowing that they correspond to the same file on disk, which could lead to a variety of race condition problems, like:

- The threads/processes could end up racily writing to the same regular file, thus producing a dirty result.
- A thread/processes could fail because a path it expects to be missing in the working tree was in fact already populated by another thread/process.
- Two threads/processes might compete in removing and creating files at the same paths. Depending on the order of operations, this could lead to errors on Windows, where applications cannot remove files with open descriptors.

These examples may cause erroneous results or even failures, but there are even worse possibilities when path collisions are not handled properly. **Some of which can imply in security risks.** If a symbolic link collides with the leading directory of another file being checked out and we fail to detect that, Git may end up following the link and writing the file at the wrong place. The path can possibly fall outside the repository, and even overwrite user data. Even worse, a maliciously crafted repository could abuse this oversight in the checkout implementation to drive remote code execution (RCE) attacks, by making the symlink point to "`.git/hooks`" and writing a "post-checkout" hook. The "hooks" mechanism is a way for users to specify custom scripts that should be executed upon specific Git events. The "post-checkout" hook is executed write after checkout, so if Git can be tricked to write repository files at "`.git/hooks`" during checkout, it would be possible to run a malicious script without user consent. This could even be used during a

[2] https://github.com/git/git/commit/1d1d69bc52dcc7def5b2edbd165cc0a4e3911c8e

clone operation, which is one of the core features of Git. We will discuss more about this specific security challenge in Sections 6.2.1 and 6.2.2 and in Appendix D.

Path collisions are specially challenging because the system does not provide a universal API to detect whether two path names will collide on disk without actually writing them to check. Of course, applications can make some inferences based on the file system type and operating system, but this approach is not robust as there can always be a file system that the application is unaware of. It would not be feasible to implement all folding rules both extensively and safely.

In Chapter 2 we mentioned that the sequential checkout handles path collisions just like any other "present but not up-to-date" file. To recap, before creating the file, `checkout_entry()` uses `lstat()` to check that there is no file at the same path in the working tree and that the dirname components are all real existing directories (not symlinks). If the second check fails, due to missing components, the function creates then and proceed with the checkout. However, if any of these checks fail due to an already existing file, it either aborts the operation or removes the file that is preventing the entry from being checked out and moves on.

In other words, when there is a path collision among two or more entries to be checked out, the sequential code simply check out each one of them, removing the previous version from the working tree and writing the new one. This may sound like unnecessary work (and it possibly is), but we do not expect path collisions to happen that often. Furthermore, since collisions may leave the repository in a state that is not really useful for its users, the colliding files are usually renamed when the owners of the repo want to support file systems that may experience such collisions.

It is worth mentioning, that while the sequential checkout approach for colliding paths may be bit expensive, it has a nice perk regarding the state of the index after checkout: because all colliding files **are in fact written** (although only the last one survives on disk), all of them will be `stat()`-ed and the associated index entries will be populated with info like file size and mode. By filling these metadata fields, we save time on later `git status` operations as they do not need to read the files' contents to check that they are dirty. But we will discuss this with more details later.

Path collisions are not much of a "special case" in the sequential checkout code. That is, Git would already have to `lstat()` the leading directories and the basename before creating the file, so collision handling almost comes as a free bonus from these mechanics. However, for parallel checkout, the `lstat()` check would not be sufficient as there is no guarantee that another worker will not have created the file in between a positive check that the path was empty and the actual file creation. The same goes for leading directories: one worker could successfully evaluate that all components are real directories and proceed to the file creation, without knowing that another worker replaced a directory with a symlink in the meantime.

4.3 Using Stale Data From the `lstat()` Cache

In Section 2.6 we discussed how Git caches some `lstat()` calls during checkout and the importance of this for performance. Caching is indeed a very important technique, but it must be used with great care. One could ask how Git can be sure that the cached leading directories are still valid in case of path collisions during checkout. That is, the checkout machinery may remove previously created directories if they collide with another entry it needs to check out, so how does it keep the cache consistent with the working tree changes and avoid later uses of stale information? One of the ways that the sequential checkout protects itself against a cache misuse is by checking out entries in the index order. As the documentation of the Git index format says, *"index entries are sorted in ascending order on the name field, interpreted as a string of unsigned bytes (i.e. `memcmp()` order, no localization, no special casing of directory separator '/')."* (Git Development Community, 2021a) Therefore, if the entries are checked out following the index order, it is still possible that the information from the cache becomes outdated, but we can be sure that no outdated information will ever be used, as checkout will not interleave entries from different directories. In other words, the ordered checkout can rely on the invariant that, once it start checking out files from a new directory, it will never go back to any previously visited directory to write more files[3]

It may be a little tricky to visualize this property, but it becomes clearer with a concrete example: pretend we have the files `A/B`, `A/C`, and `a`. In the index, they will be sorted in the same order that they were introduced here. Notice that if we check them out in this order on a case-insensitive file system, we will eventually replace the directory `A` with the regular file `a`, while still caching the information that `A` is a directory in memory. However, that is completely fine as, at the time this happens, the checkout machinery will have already checked out all files from the `A` directory, and it is not going back to it.

Of course using the index order to prevent cache corruption only works because the cache lookup code use exact matching to compare the query path with the cached components. If it were to perform case-insensitive comparisons, or even normalize Unicode characters, we could in fact end up using invalid data from the cache. To give an example, imagine the checkout of files `AA/b`, `Aa`, and `aa/c` (already presented in index order). Since the lstat cache uses exact matching, when we call `has_dirs_only_path()` for the last path, it will not say that `aa` is a directory, even though it has cached the (now incorrect) information that `AA` is. And again, it is fine that it caches this because we will not check out anything with the `AA/` prefix for the rest of this execution.

Another case to consider is the removal of files that were in the index but should no longer be after the current checkout. Typically, the high level `unpack_trees()` will remove these files in a separated loop before it starts processing the file creations. However, the lower level `checkout_entry()` API can process both cases, offloading the removals to `unlink_entry()`. This capability is used by the `git checkout` command when it is

[3] As we later found out, the classic sequential checkout may in fact write entries out of order under two circunstancies. In these cases, the code was susceptible to invalid uses of the `lstat()` cache. See Appendix D for more information about this finding and the fix.

given the `--no-overlay` option. Since `unlink_entry()` will also remove leading directories when they are left empty after a file removal, it seems possible that a subsequent file creation might end up using outdated cache information regarding a path that was a directory before being removed. However, this cannot really happen because `checkout_entry()` and `unlink_entry()` use different functions from the `lstat()` cache API, and those cache different types of entries; and as a safety measure, the cache gets reset whenever the caching options changed between uses.

We have shown that, if checkout follows the index order, it will not use stale data from the `lstat()` cache. However, the order assumption is hard to satisfy on a parallel checkout implementation. If we do not take care to invalidate the cache when appropriated, or avoid using outdated information, the parallel workers might end up trusting that a path is a valid directory when it is not. This can lead to the same kind of problems described in Section 4.2. If the cached directory gets removed or replaced with a regular file (due to a path collision), the checkout worker will just fail to write out the entry. However, if it gets replaced with a symlink, the worker will blindly follow the link and write the file at the wrong place. The security implications described in Section 4.2 also apply here.

4.4 Filters and the Attributes Machinery

As mentioned in Section 2.5, tracked files can be filtered when checking in (by "clean" filters) and out (by "smudge" filters). The specification about which filters must be applied to each path are defined in a file called `.gitattributes`. Each directory of the project can have its own `.gitattributes`, which will have higher precedence than the ones before when evaluating the attributes for the files in that directory (and below it). There are also global and system-wide files, which have the lowest precedence of all, and a non-versioned repository-specific file at `.git/info/attributes`, which has the highest precedence.

The filters applied when checking in files are called `clean filters` and the filters applied when checking out blobs are called `smudge filters`. Regarding the way in which each filter is executed, there are three classes: in-core filters, which are performed by Git itself, without needing to execute a child process; external "one shot" filters, which are commands/scripts invoked by Git to process a single blob or file; and long-running filters, which can process many blobs with a single invocation. The last class requires the filtering command/script to implement a protocol by which Git will feed it the blobs and collect the results over time. One of the advantages it holds against one shot filters is that there is no need to `fork()` and invoke the filter for every blob during checkout. But, of course, this performance gain comes with a more complex implementation.

Long-running filters can also make use of the `delayed checkout` feature. This is a mechanism by which filters can delay the response to Git, letting the checkout process continue while the filter performs the conversion in the background for better performance. Later, after all entries have been processed, Git will ask the filter again for the entries that it required to be delayed. One of the most widely-used filters that make use

of this capability is `Git-LFS`[4].

Filters must be handled carefully for a successful parallel checkout implementation. Below we enumerate some of the challenges they impose. Most of these issues were originally raised by Jeff Hostetler, a Microsoft Engineer and Git contributor who had also worked on another parallel checkout approach before. His original findings on the subject, as well as other challenges, can be found at his parallel checkout technical document[5].

1. Like the object reading code, the attributes reading and filtering sections also make use of global resources which would need to be protected in a multi-threaded implementation. In particular, the data loaded from the different `.gitattributes` files during checkout is kept in a global stack, which is not thread-safe. Merely protecting the stack with a lock would probably not be a good approach: the attributes machinery is optimized to handle paths in sequential order, and the concurrent requests for paths in different directories could impact the performance.

2. Besides the blob smudging part, `checkout_entry()` might sometimes needs to clean a working tree file to compare it with its associated blob and check whether the file has unsaved changes. This operation uses the attributes code too, so it would also have to be protected on a multi-threaded checkout.

3. Sometimes, Git may need to load the `.gitattributes` contents from its object store, instead of the working tree (e.g. when checking out files from a directory that was not in the working tree yet). In this case, the checkout machinery needs access to an index state from where it will retrieve the hash of the `.gitattributes` blob. However, on a multi-process parallel checkout implementation, the workers might not have access to the modified in-memory index state of the main process. If the implementation does not handle this, the workers might end up using outdated hashes from the on-disk index.

4. External filters might have their own locking and/or non-concurrency assumptions which we could break if Git starts spawning them in parallel. E.g. a filter could be logging its operation on a shared file, expecting that no more than one instance of the filter process will be running at any given time.

5. Long-running filters should persist throughout the whole Git execution. Supporting entries that require such filters on parallel checkout would impose additional communication and synchronization challenges for the workers. We would either have to synchronize their interactions with the filter to avoid races or elect a single worker to communicate with the filter and propagate the results to the others.

6. Long-running filters may also use the delayed checkout capability, in which case they can postpone the reply to some blobs. In this case, besides the communication challenges, the workers would have the additional challenge of managing a queue of delayed blobs and re-consulting the filter latter.

[4] https://git-lfs.github.com/

[5] https://github.com/jeffhostetler/git/blob/0d97a158e274db3c59b3a6e382ec91b08d44fa8c/Documentation/technical/parallel-checkout.txt#L336

4.5 Parallel I/O

When we started this project, we already expected that parallel checkout would be beneficial on networked file systems, where parallel connections can amortize the latency. Furthermore, previous tests from 2008 (see Section 5.1) had already reported that parallelizing the I/O operations on checkout could lead to a 4.5x speedup on NFS. However, we did not know which gains could be expected on local file systems, specially on HDDs. Furthermore, many aspects can affect I/O performance, which makes it difficult to predict — with sufficient generality and certainty — how parallel checkout will behave on arbitrary repositories and machines. Some of these elements that influence performance are: the access pattern (including number and size of working tree files and object files, deltification, and compression level), the directory hierarchy in the working tree, the fragmentation of the files, the physical proximity of the different requests on disk, the type of file system, mounting flags, and so on. Therefore, whether or not the I/O patterns from parallel checkout would really produce any significant improvement on local file systems was also a challenge in itself.

One of the effects that parallel checkout can produce on the system is to increase its I/O queue depth. As Ghodsnia *et al.*, 2014 defines, *"I/O queue depth is the average number of outstanding I/Os in the I/O queue at any point of time. The I/O queue depth can be increased by issuing multiple I/O requests at the same time or issuing I/O requests with a rate faster than the rate of handling I/O requests by device."* The ability to send multiple I/O requests to a storage drive is called command queueing, and there are different technologies for it, such as the Native Command Queueing (NCQ) for SATA — which allows a queue depth of up to 32 jobs — or the technology present on NVM Express, which can achieve way higher queue depths.

As we already discussed, increasing the I/O queue depth is very interesting for NFS as the queued network requests can be processed in parallel and increase performance, but what is the effect locally? For both the SSD and the HDD, increasing the I/O queue depth may allow the I/O scheduler to reorder and/or merge the different requests into more efficient access patterns and produce a higher throughput (e.g. a pattern that requires less seeking). However, increasing the I/O queue depth produce even larger effects on the SSD. As Chen, Hou, *et al.*, 2016 says *"NCQ enables SSDs to achieve real parallel data accesses internally. Compared to hard drives, NCQ on SSDs not only allows the host to issue multiple commands to the SSD for better scheduling but, more importantly, it also enables multiple data flows to concurrently exist inside SSDs."*

This has to do with the architectural organization of the device. As Chen, Hou, *et al.*, 2016 describes, the design of SSDs promote a lot of potential for internal parallelism: Their storage is usually split across an array of flash memory packages, which are connected to flash memory controllers by multiple channels. Each channel can be used independently. The packages can also be operated individually, allowing multiple packages connected thought the same channel to interleave their operations. Inside the packages, there are usually multiple chips (or dies), which can also execute commands in parallel to the other chips. Finally, the chips typically contain two or more planes, and a given command can be executed in multiple planes at the same time. This design allows the operations at each one of these levels to be parallelized or interleaved, producing great

opportunities performance-wise. HDDs, on the other hand, usually have a single actuator to read and write the data from the platters, and it typically executes a single operation at each time.

Although increasing I/O depth is likely to help SSDs, Chen, Hou, *et al.*, 2016 showed that mixing reads and writes may produce a negative performance interference on the SSD. The authors enumerate many possibilities for what may cause this performance penalty, such as the competition for critical hardware resources on the SSD and the dispatch of background tasks (like read ahead and asynchronous write back) which could impact the foreground operations. In their tests, they saw a bandwidth reduction of 4.5x on sequential writes when performed concurrently with random reads (which might happen on packfiles, although they tend to have great locality). It remains to be seen whether this may happen with parallel checkout, but it should be noted that, in that test, the I/O operations were issued directly on the SSD through block devices (with no partitions or file systems) and they used synchronous mode. Git checkout, on the other hand, does not perform synchronous writes, so it can achieve faster writing bandwidths by taking advantage of the file system cache in memory. For the same reason, it is possible that we do not see the read/write interference in the checkout workflow.

Still discussing the results from Chen, Hou, *et al.*, 2016, we see that the parallelism on SSD can also negatively impact the detection of sequential data access and the read ahead mechanism, which prefetches data in order to make them available for the application later. As we saw on Table 3.2, such mechanism is quite important for the packed object case on the SSD. However, the study shows that the performance impact of parallelism on read ahead only happens for a small concurrency level. For higher concurrency levels, the benefit from parallel requests surpasses the losses from an impaired read ahead.

Another consideration to keep in mind is the operating system where Git is running at. Git was originally created to run on Linux, as its initial purpose was to version control the code from the Linux kernel itself. But over the time it was quickly ported and adapted to run on Windows, MacOS, BSD, and other systems. Some of these follow the POSIX specification, so the adaptation was easier. On others, however, it was necessary to implement small wrappers to emulate functions that were already being used by Git. These could either be functions that were not implemented in the same way on the OS that Git was being adjusted for, or even functions that did not exist natively. One of the downsides of this adaptation is that some operations ended up being a bit more expensive on some of these operating systems. One commonly mentioned example in the Git mailing list is the `stat()` wrapper for Windows. Therefore, the parallelization of checkout may also produce varying results for the same machine using different operating systems.

Finally, there were two additional I/O concerns related to the file creation order. First, the directory locking. The kernel needs to acquire an exclusive lock at the directory level when creating a file inside it. The concurrent creations from parallel checkout could thus produce and be slowed down by lock contention on the kernel. However, there is a simple mitigation for this: we can distribute the paths to the workers in contiguous chunks, so that it minimizes the chance of having more than one worker creating files on the same directory. The second concern comes from a thread at the Git mailing list (Beller, 2016). In 2016, it was conjectured whether the arbitrary file creation order from parallel check-

out could slow down some subsequent read-intensive operations. However, as another developer named Nguyễn Thái Ngọc Duy later replied (Duy, 2016b), this is to close to the file system implementation, and modern file systems should already use mechanisms to speedup the pathname lookup (such as b-trees and indexes), making the creation order have no relevance to performance.

4.6 Portability

We already mentioned that Git implements some wrappers to emulate functions that are not available on a given operating systems. Still in the topic of portability, we should avoid using non-portable functions (without conditional macros or wrappers for other systems), as well as functions that do not behave the same on different systems. For example, at some point during development we attempted to auto-detect whether the underlying file system is an NFS mount and auto-enable parallelism if so. However, this proved to be quite trick to do in a portable way, and even when we tried to limit the feature for Linux systems only, we found out that the function we were using, `getmntent()`, had behavior divergences between different implementations of the C standard library, namely GNU's glibc[6] and musl[7].

Another consideration related to portability is the use of the `fork()` syscall to create a copy of the running process. Subprocesses are a good alternative to overcome the thread-unsafe issues on a parallel checkout implementation, even though they require extra work for communication. Nonetheless, child processes created with `fork()` without a subsequent `exec()` inherit the memory pages from their parent, so the initial set of tasks could be easily retrieved from memory. Furthermore, Linux forking works with "Copy-on-Write" mechanics, so it puts off duplicating a shared page until a process needs to write on it, which saves both time and memory. However, `fork()` is not available on all systems supported by Git. A parallel checkout implementation that depends on the above mentioned mechanics would either have to fallback to threads where `fork()` is unavailable or disable parallel support completely. Fortunately, Git has an internal API to create and manage subprocesses, which uses `fork()` followed by `exec()`, on systems that support these functions, or an alternative subprocess creation routine in others that do not. However, note that by using this API, we cannot make use of the shared "Copy-on-Write" memory mechanics from `fork()` as it is not portable. So the child processes will start with a clean image and the data must be sent to them in another way, such as through pipes.

As we have seen in this chapter, parallel checkout comes with important challenges, from race conditions on shared resources to parallel performance on I/O devices and limitations due to portability. At the next chapters, we will refer to these challenges when discussing previous parallel checkout approaches and, then, the final version that we submitted to the Git community.

[6] https://www.gnu.org/software/libc/

[7] https://musl.libc.org/

Chapter 5

Related Work

The idea of parallelizing checkout is not new. In fact, we can see discussions about it in the Git mailing list from as old as 2008, when Git was only three years old. A lot has changed in Git's code base since then, with over 30K commits and 1M new lines of code, test, and documentation. Over the years, parallel checkout was suggested in at least two other moments, with different approaches to the same idea: one in 2016 and another one in 2020. A lot of the code and solutions developed for these two previous efforts was re-used to compose the final version we present at this work. Therefore, parallel checkout is in its nature a collaborative endeavor. In this chapter, we will go over some of the details from all the three previous visions of parallel checkout since 2008, and also highlight a few other checkout related optimizations that were merged upstream during these years.

5.1 Approach I: 2008

The first parallel checkout approach had mostly the same motivation that we had for this work: to make Git faster on NFS. The series (Pickens, 2008) submitted by James Pickens as an RFC (i.e. *Request for comments*), was very short: only composed of two patches, adding a total of 165 lines.

James distributed the to-be-checked-out index entries among multiple threads and used a `mutex` to protect the directory creation, the blob loading, the attributes gathering, and the filtering sections. The author reported a 4.5x speedup on NFS (best run) when cloning the Linux kernel repository (which was around version v2.6.28 at that time). However, the author also pointed out a 0.73x slow down when repeating the same test on a local setup (Pickens, 2008). This might have happened because I/O was not the main bottleneck locally, and the locks degraded performance due to contention overhead. Or perhaps I/O was the bottleneck but the operation overlapping had a negative effect of stressing and slowing the drive. This shows, in practice, some of the challenges we described on Section 4.5.

At the time this version was proposed, the `lstat()` cache was quite young and yet to see many of the improvements it would receive in the next years. The `checkout_entry()` API did not make use of it yet, so 2008's parallel checkout implementation did not have to

deal with the challenges described at Section 4.3. However, since this version performed file creations and removals in parallel and also did not employed any mechanism to detect path collisions, it was subjected to the problems described in Section 4.2, including the security implications. Regarding the blob loading and the attributes reading and filtering phases, this version was mostly safe from the issues raised at Sections 4.4 and 4.1. These code sections were protected by the added lock, filters were spawned one at a time, and there was no support for long-running filters when the RFC was developed. However, the lock was not acquired before calling the function that checks the state of the path relative to its index entry, and as mentioned in Item 2 of Section 4.4, this function might also make use of the attributes and filtering machinery. Like this function, there could be other spots that used global resources without acquiring the added lock. Unfortunately, due to the thread-unsafe nature of many Git functions, it gets quite difficult to ensure that all global resources are protected when the threaded code uses different subsystems of the codebase, and access different levels within them.

Linus Torvalds, the creator of Git and Linux, replied to that series with some concerns about the design. He suggested to replace the locking strategy by one of these two options: a work queue for the file write-outs, allowing everything else to run sequentially and without locks while dispatching only the queued writes to the threads (which was the main target of that parallel implementation anyway); or working to make the object handling code thread-safe and implement a broader parallelism without any queue nor locking, which could possibly improve performance even further by allowing not only I/O parallelism but CPU too. This alternative had the potential to produce speedups on local file systems as well.

The original author acknowledged the elegance of the lockless suggestion, but also pointed out that it was, unfortunately, significantly more complex and time-consuming to implement. Furthermore, they worried that the refactoring option (to make more code thread-safe and allow greater parallelism) could possibly produce only a small improvement in performance which would not be worth the magnitude of the required changes. Regarding the work queue alternative, Linus himself later raised another issue: Git has to update the index after each file checkout, and it has to access global states to do so. Therefore, this phase would not be able to run locklessly. There was no further development on that series after these discussions.

5.2 Approach II: 2016

The second approach to parallelize checkout came in 2016, by Nguyễn Thái Ngọc Duy. He first shared a prototype in the Git mailing list (Duy, 2016a), but later refined it into a more complete series of patches (Duy, 2016d). This implementation was considerably more voluminous than the first parallel checkout effort, with 12 patches and 730 added lines, but it was very elegant. The author used multiple child processes to overcome the thread-unsafe issues in the object reading code and the attributes loading code. As he pointed out, this approach comes with some drawbacks, such as the inter-process communication cost, higher memory usage, and not sharing some in-memory caches among the workers (like the delta base cache). Still, it was way less intrusive (and cumbersome)

than making all the necessary changes to overcome the thread-unsafety of the object reading code.

This implementation took a similar approach to what Linus suggested in 2008 with the "work queue". Steps like checking the working tree for unsaved changes, removing the old file, and creating the leading directories were still performed sequentially; but the blob loading, filtering, and the file writing were delayed, using a queue. That is, when the code reached these steps for any particular entry, it checked if the entry is eligible for parallel checkout and enqueued it for later if so. Otherwise, checkout proceeded as normal. At the end of this "pre-processing" phase, the enqueued entries could then be offloaded to the worker processes for the actual write out. The implementation considered all regular files eligible for parallel checkout, but symlinks and submodules were not.

As for communication, the main process creates a pipe for each worker process and uses a `poll()` loop to monitor when the pipes becomes available for reading or writing. Depending on the direction, the main process either sends more work or reads the status report from the worker. The data sent to the workers are the SHA-1 hash and the pathname of each file that needs to be checked out, and the status report received from the workers contain how many successful tasks it completed and/or if there was any failure. Note that the workers do not send any additional information about the written files, so the main processes still needs to `stat()` them in order to update the index. Parallelizing the `stat()` calls could perhaps improve performance even further on NFS but, of course, it would increase communication costs. Finally, the messages are exchange using Git's pkt-line format which is "a variable length binary string" (Git Development Community, 2021c).

This version allowed more parallelism than the first one. Furthermore, as it also parallelized the object decompression phase, which can be one of the main bottlenecks on local file system checkouts, it could potentially benefit more users besides the ones on NFS. In fact, the author reported an approximately 2x speedup doing a full checkout of Linux-2.6 in his local Linux machine, with ext4 file system.

Regarding blob conversions, this parallel checkout implementation did not have to deal with long-running filters, as they were not yet supported back in 2016. However, one-shot filters were allowed to run concurrently in this version. Another consideration to make is about the attributes loading: as mentioned at Item 3 of Section 4.4, this operation requires access to an in-memory index state to get the hashes of the `.gitattributes` blobs whenever they must be loaded from the object store instead of the working tree. However, the worker processes do not have access to the main process' index and, instead, end up using the version that was on-disk before checkout, which would likely be outdated by that time. If the series had been further developed, this could be fixed by sending the necessary data to the workers. This is exactly what the Approach III did, as we will discuss later.

Some of the path collision issues we raised in Section 4.2 were present in this version: to handle collisions, the code relied on the `stat()` checks from the "pre-processing" phase and performed no additional checking during the parallel phase. This means that parallel workers designated to write paths that collide with each other would end up racily fighting for the file creation. Since regular files are created with the `O_EXCL` flag, one of the

workers would succeed and the others would simply fail and display an error message. This behavior would diverge from what the sequential checkout does, but it does not impose any security problems. The more serious case would be on dirname collisions. The workers trust that the main process created all the leading directories of the enqueued entries before spawning the workers, so they do not check again if each leading path is valid. If a symbolic link — which is not eligible for parallel checkout, and thus, is created before spawning the workers — replaces one of the leading directories of an entry in the parallel queue, the worker which gets assigned this entry will blindly follow the link. Therefore, the checkout code would be vulnerable to the security issues mentioned in Section 4.2. We must highlight, however, that this version was a prototype which ended up not being submitted to the Git mailing list. If the author had decided to continue its development, these issues would likely have been further analysed and tackled in a safe manner.

After working more on parallel checkout, Duy found that while it reduced the Linux-2.6 checkout time by almost half in one of his laptops, it actually made the same benchmark slower in another laptop (Duy, 2016c). He conjectured that the second laptop probably had a slower disk, which makes I/O the major bottleneck during checkout; and spreading the I/O operations over many processes could be stressing the I/O scheduler instead of helping. Because of these results, Duy decided not to push the series forward, but he left the patches in his public Git repository for anyone interested in trying it out.

5.3 Approach III: 2020

The third parallel checkout approach, authored by Jeff Hostetler, was under development in early 2020 (Hostetler, 2020). He did not finish and submit the patches to the mailing list, but plenty of his work (and Duy's) was used in the version we submitted later.

Like the 2016 approach, Jeff also decided to split the index entries among multiple child processes to work around the thread-safety issues in the object reading code and other parts of the codebase. Jeff wrote a technical document[1] about his parallel checkout implementation, where he also enumerated all threading issues he found associated with the checkout machinery to justify the decision about using child processes for the work. This version also used the pkt-line format for inter-process communication. However, the protocol was a bit more complex, requiring an initial handshake between the main process and the workers to establish the supported commands by which they can later communicate. The main process is always the one to start a message exchange, issuing the desired command.

This version focused on the `unpack_trees()` code, parallelizing the loop over the CE_UPDATE cache entries. So it did not support direct `checkout_entry()` callers that do not use `unpack_trees()`, like `git checkout-index` and `git checkout <pathspec>`. Like Duy, Jeff also divided the entries in eligible for parallel checkout and ineligible. But instead of classifying them inside `checkout_entry()`, he added a sequential loop over

[1] https://github.com/jeffhostetler/git/blob/0d97a158e274db3c59b3a6e382ec91b08d44fa8c/Documentation/technical/parallel-checkout.txt

the `unpack_trees()` entries to collect the eligible ones in an array for later partitioning. These entries exclude symlinks and submodules, but also regular files that require external smudge filters (of any kind). With this limitation he avoided breaking non-concurrency assumptions from the filters and the difficult synchronization issues regarding long-running filters and the delayed checkout queue. This already solves three of the challenges enumerated in Section 4.4. The entries that are ineligible for parallel checkout are sequentially written **after** the eligible ones. An interesting aspect that comes out of this decision is that is not possible for symbolic links to replace already created directories of parallel-eligible entries. So one cannot "trick" the workers to follow the symlinks, as these are only written after the workers finish.

Jeff also took a clever approach at the evaluation of the attributes for each path. Remember that the attribute rules are loaded from the `.gitattributes` files from the different directories of the project and cached in a global "attributes stack". Then, for each path, the conversion machinery looks at the stack, removes and adds layers as necessary (e.g. if the path is inside a different directory than what was previously loaded) and returns a `struct conv_attrs` for that path. This struct contains all the information about which filters must be applied to that path, as collected from the attributes files. Jeff's parallel checkout implementation makes use of this struct in two moments: first to decide whether an entry is eligible for parallel checkout, and then to smudge the blobs during the execution of the parallel workers. However, the attributes lookup code and the smudging code were coupled together, so Jeff separated them and used the lookup code during his first sequential loop to access the eligibility of an entry, taking care to save the `struct conv_attrs` together with the entry. This way, we can lookup the attributes only once, sequentially, and them send them to the workers for smudging. This is important because: (1) it solves the problem describe in Item 3 of Section 4.4, which is that workers cannot properly collect the smudging attributes by themselves, as this task requires access to the index state of the main process; and (2) it has a positive impact on performance as the attributes can be collected only once and in sequential order, which the attributes machinery is already optimized for (addressing Item of Section 4.4).

The only question is how to send these extra bytes to the workers. One possibility would be to convert each `int` and `enum` to strings and parse those in the other end, but that adds extra complexity. Instead, Jeff packed all the fieds he wanted to send to the workers in a C struct, casted it to a byte stream (more specifically, to a char array), and send it through the pipe wrapped in a pkt-line. The cleveness of this approach is that it requires no extra parsing on the workers, instead, they can directly cast the payload back to the C struct.

After the first classification loop, the array of eligible entries is partitioned across the worker processes, which are then able to load the blobs, smudge the contents, and write the results all by themselves. Jeff also charged the workers with the task of `stat()` or `lstat()`-ing the written paths and sending the results back to the main process. This addition relative to the previous approaches strives to improve performance on systems where `stat()` is more expensive.

Jeff divided the worker processes into multiple threads:

- One **command-and-control thread**, responsible for communicating with the

main process and returning data. (On failure, the workers communicate the error to the main process which retries writing the entry with more context.)

- One **blob preload thread**, to sequentially load the blobs that will be required later (and optionally smudge them).
- One or more **writer thread(s)**, which write(s) the blobs to the working tree (smudging them if the blob preload thread did not do it already). As we will see next, it only makes sense to use more than one writer thread when using the "asynchronous" mode.

This version of parallel checkout was able to operate in two modes: synchronous or asynchronous. In synchronous mode, the checkout code path stays mostly the same as the classic sequential code; the major difference is that the blobs are loaded and smudged in parallel. In this mode, each worker process has only one writer thread, and the writes are coordinated by the main process, done one at a time at the regular index order. The loop over the CE_UPDATE entries from `unpack_trees()`, in the main process, still calls `checkout_entry()` for each entry and all steps are performed as usual, except by the file writing (Step 4 from Section 2.5.2). At this point, the code path is deviated to a function which signals the worker process assigned with that entry to properly perform the writing. Note that the working tree checks are all performed by the main process. Jeff mentions that this mode should be appropriate for all current users of the `unpack_trees()` API — but those that start with a clean working tree may take more advantage of the next mode.

On asynchronous mode, the main process does not use `checkout_entry()` and instead let the workers directly create the files in the working tree without coordination with the main process. The workers also use multiple writer threads to create the files. Note that this mode does not perform any working tree checks (to see, for example: if there is an already existing file which should or should not me overwritten; if the path is already up to date; if there was a path collisions; etc.). The workers assume the path is clean and try to create the file (and its leading directories); if this is not the case, they report an error and let the main process retry it sequentially later. Due to these mechanics, this mode can only be used when the working tree is being populated for the first time, e.g. on `git clone`. Otherwise, since the workers do not check each component of the paths to make sure they are real directories, they may follow symlinks when writing the files which, as previously mentioned, can lead to errors and security issues. (Note, however, that this only refers to symlinks already present in the working tree; new symlinks are ineligible for parallel checkout and, thus, are checked out after the eligible entries.)

On both modes, the reading threads load the blobs in "the background" to have them ready for the synchronous or asynchronous writing threads when needed. Because of these mechanics, this implementation cannot read the blobs using the streaming interface, so large blobs will have to be held at memory in their entirety, increasing the memory footprint[2]. Nevertheless, Jeff mentions in his technical document that, with some refac-

[2] Note that the sequential checkout already reads packed blobs without streaming when their uncompressed size falls below the `core.bigFileThreshold` setting (512MB by default). Only packed blobs above this size (uncompressed) and loose blobs are read by streaming.

toring, it should be possible to perform streamed smudging and writing, which would already save some memory.

This version of parallel checkout was significantly more complex than the two previous ones, with a total of 36 patches, divided into API changes (to the filtering and unpack-trees subsystems), thread-safety conversions, instrumentation, and the actual parallel checkout implementation and tests. In total the patches changed 32 files with 4917 line additions and 76 deletions. Although this version was more robust, the size of the patches could unfortunately also make it harder to review and maintain the code. The robustness also came with a higher configuration complexity, with four exposed settings: the size of the blob preload buffer, the number of writer threads, the number of worker processes, and a threshold for the minimum number of files to use parallel checkout. The checkout mode (sync or async) defaults to synchronous, but Jeff had plans to use different defaults for each command/situation.

Regarding our goal to optimize checkout on networked file systems, we have seen on Section 3 that the file creation and the data flush (either through `fstat()` on an open descriptor or `close()`) are the two dominant bottlenecks. However, this parallel checkout approach unfortunately only parallelizes these steps on async mode, which has a very limited usage. Additionally, pathspec-limited checkouts (i.e. `git checkout -- <pathspec>`) and `git checkout-index` would not benefit from parallel checkout, as they call `checkout_entry()` directly.

5.4 Other Checkout-Related Optimizations

All parallel checkout implementations mentioned in this chapter, as well as this work, focus on the working tree update phase of checkout. As already mentioned, it is beyond the scope of this project to optimize other tasks involved in a checkout operation, like the tree walk and index parsing phases. However, these sections of the code base also have been receiving performance improvements over the years, in different fronts. In this section we will showcase some of the performance work developed in these areas:

In 2007, Linus Torvalds optimized [3] some index operations at the tree merge code, used by `unpack_trees()`, to avoid removing entries in the in-memory index array (which requires shifting the remaining ones) when the entry would have to be re-added later with updated contents (thus, moving memory again to make space for the entry). By eliminating these unnecessary movements in the index array, the patches lead to over 9x speedups in operations with large trees (50K+ paths) where this code was the main

[3] See patches:

- 288f072ec0 ("Optimize the common cases of git-read-tree", 2007-08-10): https://github.com/git/git/commit/288f072ec033cf917eed949119428db3626ddc71
- d699676dda ("Optimize the two-way merge of git-read-tree too", 2007-08-10): https://github.com/git/git/commit/d699676dda5fdf0996601006c3bac2a9c077a862
- 566b5c057c ("Optimize the three-way merge of git-read-tree", 2007-08-10): https://github.com/git/git/commit/566b5c057c452d04605805ea2f7af210c6fb9b59

bottleneck (e.g. a branch switch in a repo with lots of files but with very few differences among the two branches).

Two years later, Kjetil Barvik optimized[4] yet another index operation in `unpack_trees()`'s call chain: after the tree merge, the function calls `check_updates()` to update the working tree, add stat data in the index for the newly written files, and remove index entries which were marked to be removed together with their working tree copies. The last step used to be performed by calling `memmove()` in a loop, shifting entire portions of the index array to the left in order to overwrite a single entry. This was done for each entry that had to be removed. Kjetil replaced this code by a smarter loop which uses two pointers to accumulate the entries that should be kept in the array at its left portion, without needing to repeatedly move larger chunks of memory.

Index operations were not the only target of performance optimizations in the past years. In 2018, for example, Nguyễn Thái Ngọc Duy optimized[5] the tree walk using the cache-tree index extension. The idea behind it was to compare the OID of a tree with the information on the cache-tree extension (stored in the index file of a Git repository) and avoid walking the trees for which the contents are already known in the index. This saves I/O and CPU time by avoiding to read and decompress many tree objects (trees and subtrees). Duy reported that a "`git checkout -`" on the Webkit repository (275k files) went from 2.56 seconds in his machine to 1.94 seconds after the patch. With a follow up patch[6] to reuse still valid cache-tree data, Duy further reduced the 1.94 seconds to 1.61 seconds.

Other examples of performance optimizations that `unpack_trees()` received over the years include:

- The lstat cache was improved to also cache the last full directory path detected (until then it only cached symlink entries), speeding up checks on deep directory structures; as the components would no longer have to be lstat-ed again for each path. (Linus Torvalds, 2008).
  https://github.com/git/git/commit/c40641b77b0274186fd1b327d5dc3246f814aaaf
- The one-way merge code was taught to avoid calling `lstat()` (and comparing the results with the associated index entries) when it is not asked to update the working tree (Martin von Zweigbergk, 2012).
  https://github.com/git/git/commit/686b2de0ceb2c5a1fb6c8822a8aceb8a05e2fc76
- File system monitor support was added, to speed up detecting of file changes (Ben Peart, 2017).
  https://github.com/git/git/commit/883e248b8a0fd88773cb902ab8e91273eb147d07

Finally, as we briefly mentioned in Chapter 2, that some file system logic was dupli-

[4] 36419c8ee4 ("check_updates(): effective removal of cache entries marked CE_REMOVE", 2009-02-18): https://github.com/git/git/commit/36419c8ee41cecadf67dfeab2808ff2e5025ca52

[5] b4da37380b ("unpack-trees: optimize walking same trees with cache-tree", 2018-08-18): https://github.com/git/git/commit/b4da37380b7774248086f42bcd59397a44e1ac79

[6] 836ef2b69f ("unpack-trees: reuse (still valid) cache-tree from src_index", 2018-08-18): https://github.com/git/git/commit/836ef2b69f3a8668c35a537715cf3bbc08fdcf39

cated around `unpack_trees()` and `checkout_entry()` in regards to working tree checks. These duplications, unfortunately, cannot be easily resolved as there are call paths that do not pass through `unpack_trees()`; and even the ones that do, need the checks from `checkout_entry()` for a variety of other reasons: to handle path collisions, to create missing directories, to remove outdated files, etc. The duplication is still bad for performance, though, so it was previously suggested[7] to separate the tree merge process in two steps, doing every necessary operation on the index first, and only then inspecting the working tree for unsaved changes. This refactoring was not yet implemented in `unpack_trees()`, and it is beyond the scope of our project.

It is worth mentioning, though, that a similar optimization was implemented in the merge machinery code, in 2020, as part of the new merge strategy called "merge-ort". This strategy produces a tree as a result and delays updating both the index and the working tree to after the merge has been produced. Although this code is not used by the "common" `git checkout` workflow (unless using `--merge`), this is a very interesting project which deserves a mention not only for its performance benefits during merge, but also new future possibilities that it brings, such as rebases on bare repositories (i.e. repositories without a working tree). The merge-ort strategy was developed by Elijah Newren, and more information can be found in the author's posts at: https://blog.palantir.com/optimizing-gits-merge-machinery-6-7bf887a131d8.

[7] https://lore.kernel.org/git/20110222192632.GB4881@localhost/

Chapter 6

Parallelizing Checkout

We have already discussed the theoretical concepts that surround the checkout machinery in Git, the challenges associated with its parallelization, and previous approaches to this endeavour. In this chapter, we will tie these discussions together and present our development process for the parallel checkout feature. We will review the most important design choices for this implementation, discuss how we overcame the previously presented challenges, and give an overview of the new code flow. The chapter is divided in three sections. The first one presents the initial prototype we worked on, which used thread-based parallelism, and discusses the issues that made us switch to process-based parallelism. The second shows what changes and improvements were made to the first prototype, specially around path collisions and the `lstat()` cache. Finally, the last two sections discuss a little bit about correctness tests for the new parallel checkout feature, and the main contributions from each previous approach, as well as this work.

6.1 Our First Prototype: Multi-Threaded Version

Early in development, we invested some time to study the latest sequential code as well as the previous parallel checkout approaches shared in the Git mailing list. The goal was to understand what were their key features, what worked the best in each version and what not, what designs decisions were taken to overcome the challenges, and how the current state of the code base would receive such changes. Using that knowledge, we could then try to combine the best elements from each version, looking for good performance, robustness, and easiness to review and maintain, while also avoiding any potential downside that an individual approach might have.

We decided to use version II from 2016 as a basis for our implementation, because it presented a good balance between code simplicity and robustness. Additionally, the design adopted in this version supported all callers of `checkout_entry()`, not only the users of the `unpack_trees()` API, and it always performed both blob loading and file creation in parallel. Both these features are important for our goals because: 1) we can extend the performance benefits of parallel checkout to more commands; and 2) the parallel file creation is essential for NFS, where this is one of the main bottlenecks, while the parallel blob loading is a great opportunity to optimize checkout on local file systems too

(specially SSDs, where inflation is the bottleneck). Approaches I and III focused specifically on `unpack_trees()`. Also, approach I only parallelized file creation and approach III parallelized both operations only in async mode, which was available to a limited number of Git commands.

However, version II had some issues with smudge conversions and path collisions. Regarding the smudging, it allowed external filters to run concurrently and the workers did not use the correct index information when loading the Git attributes for a path to be smudged. Also, there was no support for long-running filters at that time, so we would need to deal with this case ourselves when rebasing that version on top of a more recent Git commit. To handle these conversion issues, we ported some of Jeff's patches from version III, which: separated the attributes lookup and filter application; used the attributes to classify which regular files are eligible for parallel checkout; and, finally, saved the already loaded attributes to make them available later for the workers.

Our criterion for parallel-checkout eligibility excluded submodules and regular files that require external filters (being them long-running filters or one-shot filters). This first prototype allowed for symbolic links to be checked out in parallel, but we noticed that this would make it harder to parallelize directory creations in the future, so we latter disallowed symlinks on the parallel phase (like Duy and Jeff had already done in their implementations). This change should have a negligible impact on performance as regular files are usually much more numerous than symlinks. Furthermore, the contents of symlinks are quite small when compared to the contents of regular files, and they do not accept conversion filters, therefore it is usually less work to check out a symlink, anyway.

Also note that, because each submodule is checked out in a separated Git process, their entries can also be written using the parallel checkout framework. The fact that the submodule itself is classified as ineligible only means that we do not include it (or its entries) in the parallel checkout queue of the superproject. Instead, we let the child process spawned by the checkout machinery decide whether the submodule's entries will be written in parallel, using its own parallel checkout queue and resources. It is important to highlight that, for each submodule, the checkout machinery will wait for the spawned child process to finish before continuing to the next index entries. Therefore, even though submodules may use the parallel checkout framework, we do not create more than the configured number of workers at a time.

Regarding path collision, we took a similar approach to what Jeff had implemented in version III, detecting the collisions in the workers and skipping the entry so that the sequential code path could retry writing them later with more context (it would then be able to properly add the entry to the list of collided paths on clone, overwrite it, etc.). The collision detection code we implemented would originally inspect the error code from a failed file creation and consider the following errors as collisions: `EEXIST`, `ENOENT`, and `ENOTDIR`. This worked for basename collisions but it did not cover all cases of dirname collisions. Therefore, we ended up improving this system for the final parallel checkout version. We will discuss more about the improve mechanics at Section 6.2.1.

All of the design decisions mentioned so far made it to the final implementation that was merged into the upstream Git repository. There was, however, one major change

in design from our first prototype to the final version: the use of processes instead of threads. We initially decided to implement the parallelism using threads because they come with many benefits for the programmer: the memory is shared[1], so communication and synchronization protocols are much easier to implement and the code is shorter and easier to maintain; the startup cost is less expensive on Windows (compared to processes); the memory footprint is smaller; Git's in-memory caches (like the delta-base cache) are shared so there is no unnecessary work and memory duplication to load and cache these entries among multiple workers; etc.

However, threads also come with a high number of challenges, and one of its greatest benefits, the easy shared memory access, is also a major weakness. Global resources are visible and modifiable by all threads and, without proper locking and synchronization, there can be racy usage of the resources leading to errors and unwanted outcomes. In fact, some of the challenges we described in Chapter 4 arise specifically from the unsafe use of threads. Nevertheless, we still decided to take a chance with threads because we now had a tool to overcome the thread-unsafeness of the object reading code, which was one of the highest barriers to use threads in the previous parallel checkout approaches. As mentioned in Section 4.1, this is the object reading lock, introduced in the beginning of 2020. The locking mechanics allow concurrent accesses to the object reading routines with good performance by promoting parallelism in the object decompression phase.

Our first threaded version[2] seemed to be producing correct results, so we shared it with Jeff, who had most recently worked in parallelizing checkout. The performance benchmarks were also showing promising results. Using hyperfine[3], a command-line benchmarking tool with statistical analysis, we varied the parallel checkout parameters from both our threaded version and approach III to find the best settings for a Linux clone on Machine `Mango`. For approach III with async mode these were: 16 processes, 2 writer threads (per process), and 10 slots for preloaded blobs (per process). For approach III with sync mode: 2 processes, 1 writer thread, and 30 slots. For the in-process threaded prototype, the optimal setting was 10 threads.

Then, using these settings, we ran a benchmark, switching from tag v2.6.15 to v5.6-rc2 in the Linux repo (about 77398 files changed) and also cloning v5.6-rc2. The results are show in Table 6.1.

Although the initial timings showed 2x to 2.3x speedups, another test showed that up to 20% of the average thread runtime could be spent in mutex functions, which suggests locking contention on the object reading code. This is shown at Table 6.2, whose data was gathered by timing the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions on a full checkout of Linux v5.12. Switching to a multi-process implementation could probably help with that, as the different processes would be able to read the objects locklessly, but there would still be some amount of serialization in the kernel — for the I/O operations — and in the actual drive as well, specially HDDs. Furthermore, the cost of

[1] It is also possible to set up a writable shared memory for multiple processes, but it would not be as straightforward as it is for threads. Specially considering the portability challenges.

[2] https://github.com/git/git/compare/master...matheustavares:parallel-checkout

[3] https://github.com/sharkdp/hyperfine

	v2.6.15 to v5.6-rc2	clone of Linux-v5.6-rc2
Original	7.739 s ± 0.024 s	7.920 s ± 0.112 s
Approach III (sync)	5.389 s ± 0.028 s	5.727 s ± 0.036 s
Approach III (async)	4.022 s ± 0.043 s	3.968 s ± 0.101 s
Threaded prototype	3.711 s ± 0.115 s	3.479 s ± 0.044 s

Table 6.1: *Run times for two checkouts in the Linux repository using four different checkout versions: the original upstream code, parallel checkout Approach III, using sync and async modes, and the threaded prototype version. Each value shows the mean run time and standard deviation for 30 cold-cache executions on Machine `Mango`*

inter-process communication and in-memory cache duplication could also make the net effect invisible.

Threads	Average thread runtime	Average time spend on lock/unlock	Proportion
2	5.127 ± 0.133 s	0.185 ± 0.017 s	3.62%
4	3.568 ± 0.145 s	0.359 ± 0.014 s	10.05%
8	3.338 ± 0.061 s	0.627 ± 0.005 s	18.77%
10	2.884 ± 0.045 s	0.585 ± 0.011 s	20.29%

Table 6.2: *Average per-thread runtime and mutex locking time for a `git checkout .` of Linux v5.12 on Mango (SSD), using the threaded parallel checkout prototype. Values are the confidence interval from 15 executions with 95% of confidence.*

Nevertheless, we still decided to abandon the threaded prototype; not so much for performance reasons but for correctness and safety. Although the initial tests did not show any errors, the threaded call chains were quite long and we were not able to safely attest that all its functions were free from thread-unsafe operations; specially since they used different subsystems in Git. In fact, we later found out that the streaming interface may access global resources from the object reading subsystem through low level functions that do not acquire the object reading lock. Such functions are called to stream non-delta packed objects whose uncompressed size exceeds the value from the `core.bigFileThreshold` setting (which defaults to 512MB). In this scenario, the threaded parallel checkout would be susceptible to different cases of race condition, some of which would lead to segmentation faults.

We also found some static function-scope variables in low level routines[4], which are used all throughout the code base (usually on error messages). For the threaded parallel checkout, a race involving these functions would probably be less problematic than the object reading race, but it could still cause flaws on Git error messages, for example. We invested some time trying to make some of these functions thread-safe[5], but ended up

[4] https://lore.kernel.org/git/CAHd-oW5zh=BG29O0Z-M7R26Lgd=RHECMV2+qByF+vU6PmrEn_Q@mail.gmail.com/t/#u

[5] https://lore.kernel.org/git/cover.1593115455.git.matheus.bernardino@usp.br/t/#u

dropping the patches due to some portability issues with their use of the thread-local storage[6].

These are just a few examples of thread-unsafe functions that we found our threaded checkout prototype was wrongly using. However, even if we fixed them would probably be insufficient to ensure the thread-safety of parallel checkout under the many different scenarios it can be used. As two Git contributors commented in 2018, when it was suggested to parallelize other regions of `unpack_trees()`:

> "I'm generally terrified of multi-threading anything in the core parts of Git. There are so many latent bits of non-reentrant or racy code."
> - Peart, 2018

> "Multi-threading anything in git is fraught with challenges as much of the code base is not thread safe."
> - King, 2018

That is not to say that process-based parallelism is always a better choice, and it also comes with its own set of challenges. However, it can be a more viable alternative on large pre-existing codebases that present many thread-unsafe routines. The decision to switch to multi-processes did pay off in our case as it made us more confident against race conditions. Nevertheless, the development of the threaded prototype was not in vain as the code was reused for the process-based implementation.

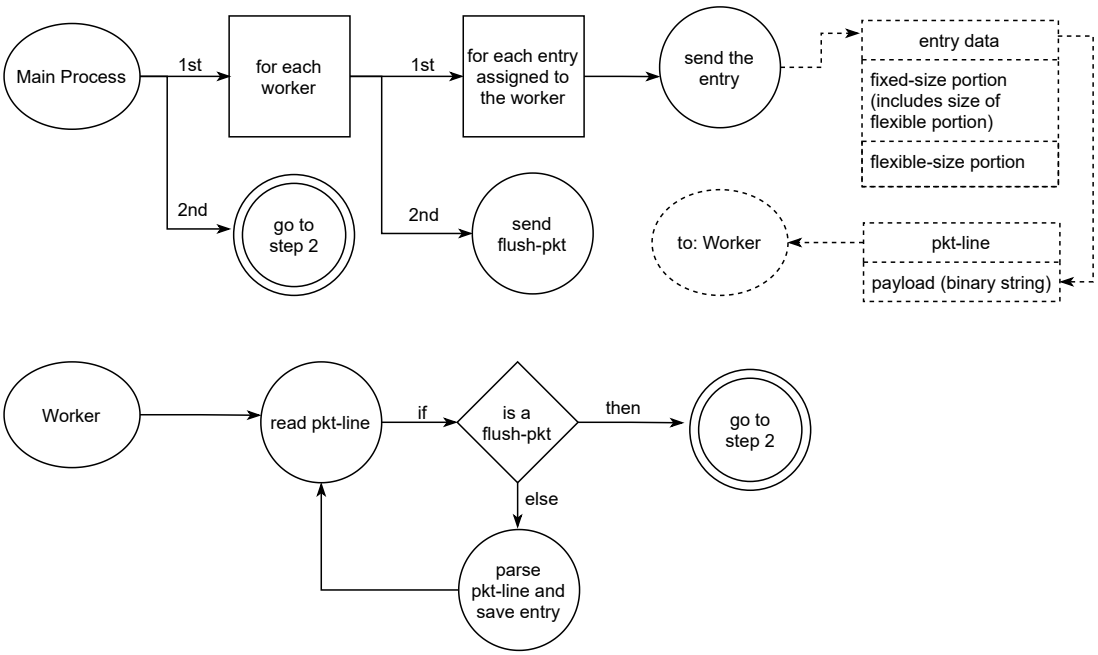
6.2 Final Version

To make the change from threads to child processes, we first had to define a protocol by which the main process would communicate with the workers. We choose to use a very simple protocol which mixed both Duy's and Jeff's designs: The main process starts the communication by sending the assigned entries to each worker. The data sent for each entry is composed by two elements: a C struct, containing all the fixed-size fields; and a flexible-size part containing two strings, the entry's path and encoding name. These two parts are allocated together (i.e. contiguously), casted to a binary string, and wrapped in a pkt-line to be transmitted to the workers. On the other end, the workers can directly cast the fixed-size portion of the binary string back to the original C struct, and then copy the two additional strings (whose sizes are specified in the fixed-size portion). After sending all entries to each worker, the main process sends a pkt-flush message (a special pkt-line with length zero) to indicate end of input from its side. Then its time for the workers to process the received entries and send back the results, also terminating with a pkt-flush message. The results for each entry are packed in another C struct, which includes: the ID of the entry, a status code and, if the entry was successfully checked out, the `stat()` data collected by the worker after writing the entry. To avoid busy-waiting, the main process receives the data from the workers using `poll()`, which waits until any of the open pipes is ready for reading. Figure 6.1 is a visual representation of how this interactions between the main process and the workers were implemented.

[6] https://en.wikipedia.org/wiki/Thread-local_storage

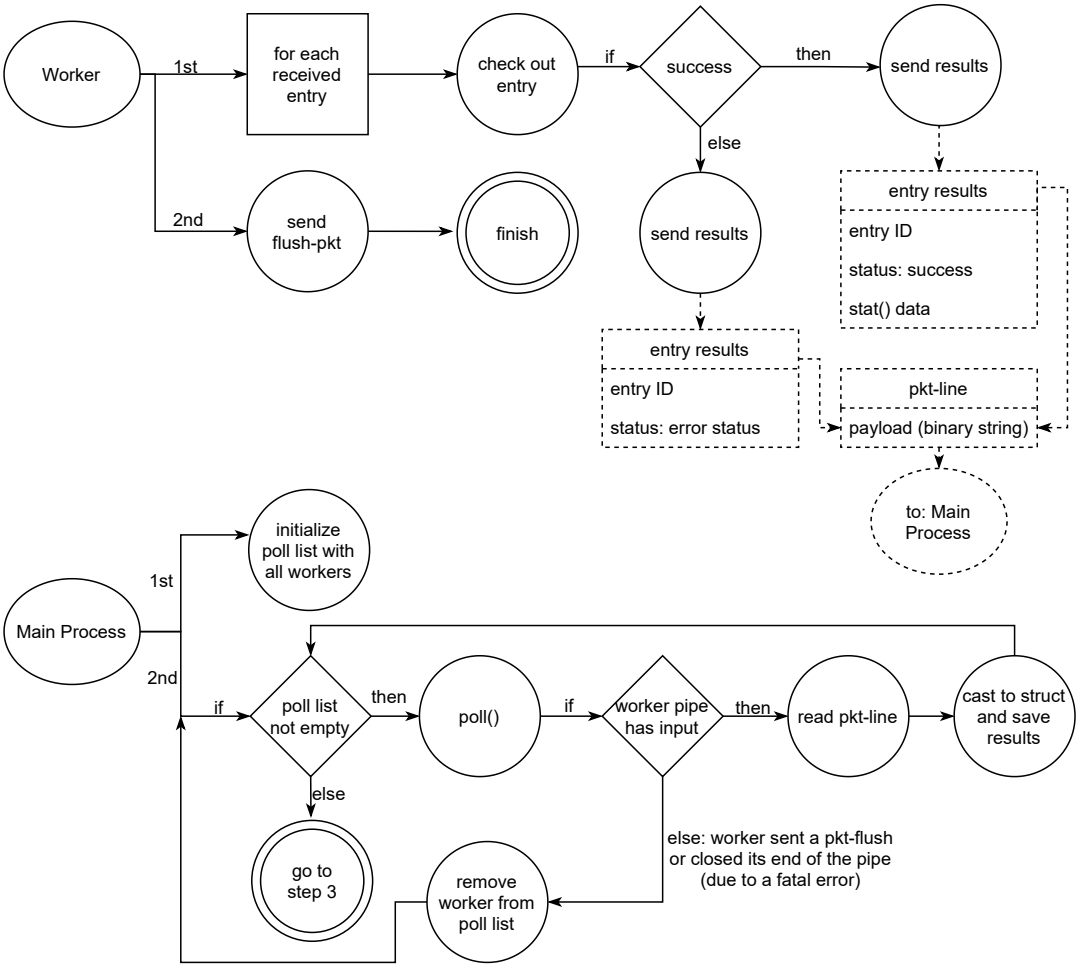
Step 1

Send parallel checkout entries to the workers.



Step 2

Collect the results.



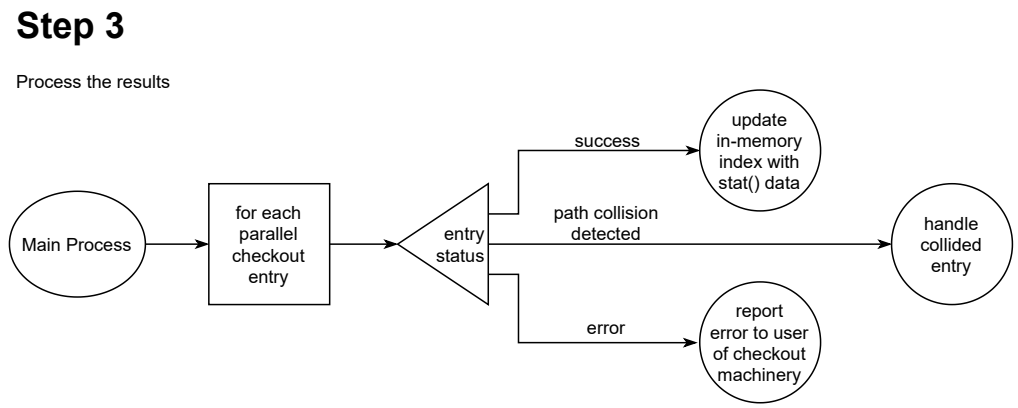


Figure 6.1: *Diagram of the interactions between the main process and the workers during parallel checkout.*

The communication protocol was not the only modification made since our threaded prototype. Another import decision change was to duplicate some of the code from the checkout machinery so that we could specialize it for sequential or parallel mode. In the first prototype, we wanted to avoid duplication as it is usually bad for both readability and maintainability. We learned, however, that this attempt ended up trading off duplication for higher code complexity: since we tried to handle both checkout modes in the same function, we needed more code to check for specific behaviors of each case. For example, the sequential checkout would only try to create the file when it was about to write it (i.e. after the blob was already loaded and filtered). This is not a good strategy for parallel checkout as the worker could end up loading and filtering the blob only to later discover that the work was wasted because the path already exists in the working tree (and thus the collided entry should be skipped). It would be more interesting to the worker to try creating the file first and skip all the work if it already exists. Furthermore, we also needed to signal the collision to the main process when the parallel mode was active, but handle the `open()` error differently on sequential mode. Because of the high code complexity that came with handling both cases together, we decided to split the two modes and accept some code duplication in favor of a simpler and easier to read code.

The final version also included improvements and corrections regarding path collisions and the use of stale data from the `lstat()` cache. The threaded prototype had problems in both these areas, incurring in some of the issues we described at Sections 4.2 and 4.3. Although the worker threads did catch basename collisions when failing to create a file, they did not handle dirname collisions very well. The threads would only recognise a dirname collision when `open()` failed with `ENOENT` or `ENOTDIR`. This would correctly cover the cases where a directory was replaced with a regular file or simply removed, but not the case of a directory beings replaced with a symbolic link. Therefore, the threads could erroneously follow symlinks when creating the files.

Finally, the threaded version also did not account for the risks of using invalid data from the `lstat()` cache when performing unordered checkouts. In particular, that version would incur in this problem when sequentially writing the collided entries after the

parallel phase. In the two following subsections, we will describe how we handled each of these issues for the final process-based implementation.

6.2.1 Path Collisions

As we showed in Section 4.2, unhandled path collisions during checkout are really dangerous as they might lead to errors, data loss, and even security breaches. Therefore, we took a great deal of care in the final parallel checkout implementation to make sure that all collision cases would be properly detected and handled. When a parallel-eligible entry collides with another entry, being it parallel-eligible or not, the assigned worker detects it, marks the entry with the "`PC_ITEM_COLLIDED`" code, and skips it. This is all done as soon as the workers starts processing an entry to make sure that no work is lost in case the entry is found to collide with another.

The idea behind the collision detection code is quite straightforward. It basically relies on two mechanisms: the creation of files using the "O_EXCL" flag, which acts as a exclusive-access lock for that path in the file system; and a call to "has_dirs_only_path()" in the workers, to make sure that the leading components of the entry are all real directories. This only works because the main process removes the old file from the working tree (if present) and creates all leading directories of each entry before enqueuing them for parallel checkout. Furthermore, the workers can only remove (regular) files on a single scenario: when they encounter an error that prevents an entry from being checked out after they have already created the file. In this case, the worker removes the file it had previously created to avoid leaving it empty in the working tree (thus reproducing the behavior from sequential checkout). Therefore:

- If there is a basename collision (e.g. "file" and "FILE"), only one worker will be able to create the file with "O_EXCL", and the others will fail with an EEXIST or EISDIR error. So we check for this errno values on an `open()` failure and consider that a collision. In the rare case that a worker manages to create the file but due to a checkout error decides to remove it, another worker can still try to check out another entry that collides with that path. Nevertheless, all entries that should be retried later will be marked with the collision flag (the one that was removed should not be retried, as the worker would have already printed a message with the error it encountered when trying to write it).
- If there is a dirname collision (e.g. "one/file" and "ONE"), there are two possible outcomes:

    1. If "one/file" is handled by the main process **before** "ONE", then the directory "one" is removed to make room for the checkout of the regular file (or symlink) "ONE". (It does not matter if "ONE" is parallel-eligible or not.) In this case, the worker assigned with the "one/file" entry will check if "one" is still a valid directory and fail, either because it is missing in the working tree (i.e. it was removed for the checkout of "ONE"), or because it is no longer a directory (i.e. the file "ONE" was already checked out at that path). In any case, we consider the directory checking failure a collision.
    2. If "one/file" is handled by the main process **after** "ONE", then the file "ONE"

is removed to create the directory "one". In this case, the worker assigned with the "ONE" entry will fail to create the file with "O_EXCL", falling in the basename collision case described earlier.

The more attentive reader may question what happens in case of submodule collisions, as the submodule may collide with the dirname of another entry, thus leaving the collided component as a valid directory (e.g. the submodule "SUB" and the entry "sub/file"). This is indeed a bit trickier. In this case, parallel checkout will only detect a collision if the submodule also has an entry named "file" in it, or a pathname that collides with "file". Otherwise, "sub/file" will end up written inside the submodule. (Except on clone, where the submodule is checked out after the superproject.) This is certainly not ideal, but it is nonetheless how the sequential checkout deals with this case.

Note that sequential and parallel checkout might produce divergent results if the regular file appears before the submodule in the index (e.g. the file "SUB/file" and the submodule "sub"). That is because sequential checkout will process the file first and parallel checkout will process the submodule first, as it is not parallel-eligible. This is unfortunate, but it is not an error as Git does not make any guarantees about which file will be present after checkout in case of collisions. Furthermore, this is already a quite specific corner case, where the working tree is likely to be unusable (or at least partially broken) for users on file systems that produce such collisions. Therefore, it would not be worth the effort to unify the two modes regarding this specific outcome.

After all parallel-eligible entries are processed and the workers finish their execution, the main process loops through the entries marked with `PC_ITEM_COLLIDED` and retries to check them out. This will essentially emulate the behavior that the classic sequential checkout implements upon path collisions, writing each one of the colliding entries on forced checkouts (although only one will actually survive on disk), or erroring out with a message on non-forced checkouts. The sequential checkout does not have any special code to handle path collisions among checkout entries (besides the reporting message on clone), so it applies the same logic used for paths that were already in the working tree before checkout (`lstat()` the path, check if it is clean, remove it if doing a forced checkout or abort otherwise). However, the parallel checkout can distinguish path collisions from this other case, so in theory, we could adopt a different behavior (e.g. just ignore the colliding entries). We ended up choosing to retry checking them out for a few different reasons:

- It is the same behavior implemented by the sequential checkout, so we avoid having dissonant outcomes that may surprise users in some way. Note, however, that there is still a slight difference regarding which colliding file will survive at the end of checkout. For sequential checkout, it is always the last one, while for parallel checkout, this is not deterministic.

- It allows us to use already implemented code to report the collisions during clone. (Remember that the sequential checkout only reports them on clone because that is the only scenario when the working tree is known to be empty before checkout. On parallel checkout, we could report collisions in any operation, but we did not do so to avoid the difference in behavior.)

- There can actually be a performance benefit for future Git operations if we write all colliding entries during checkout. Although only one entry will survive in the end, writing all of them allows the checkout code to `lstat()` the files and save the metadata in the index. Without this, we would have NULL `lstat()` fields for all entries except one in the colliding group, and future operations that need to refresh the index would have to read the working tree file to make sure that these entries are dirty, as it would not be possible to do so using the `lstat()` fields. This means that for a colliding group of N entries, every subsequent `git status` would have to read, filter, and hash the written file $N - 1$ times, to compare it with the hash of each entry in the group and discover that they are dirty. By writing all of them during checkout, like the sequential code does, we can pay this overhead only once and save time in future commands. Although a working tree with many colliding entries is probably not very useful in the first place, so the user will probably resolve the collisions or maybe use a `sparse checkout` to fix the performance problem we described here.

Of course, even though the first two items are more important, none of the three are permanent, and both the parallel and sequential checkout can be changed, in the future, to ignore colliding entries or even report them at any checkout operation. Nevertheless, these were the reasons why we choose to adopt the behavior we implemented for now, and we considered it to be a stable option for the first parallel checkout patches. Specially because it tries to approximate the parallel version to the classic sequential version, so it is less likely that users will find problems with it. Furthermore, reusing the code makes the implementation easier and less likely to contain new bugs. However, as we already mentioned, the behavior can be easily changed in the future if necessary, as the collision detection code is decoupled from the collision resolving code.

6.2.2 Avoiding Misuse of the `lstat()` Cache

In the previous section we discussed how parallel checkout detects path collisions when concurrently writing entries in the working tree, to avoid errors and security issues. However, we have not yet addressed how we avoid using possibly stale data from the `lstat()` cache in case of such collisions. Since we use the cache to check whether the leading components of the to-be-written entries are all valid directories, a collision that replaces the directory with another file type, could lead to the same security issues mentioned earlier.

As we have discussed in Section 4.3, following the index order during checkout does not prevent the cache from becoming stale, but it does prevent the stale data from ever being used in checkout. This happens because, once the code leaves a directory, it never goes back to it to write more entries, and the cache can only become stale after leaving a directory to write an entry at a higher path level (which can collide with the said directory or one of its parents). On parallel checkout, however, we cannot guarantee this order, so we had to study other alternatives to protect the code from a cache misuse. In this process, we found out that even the classic sequential checkout already had some cases where it did not follow the index order and, thus, could end up using stale data from the cache. Namely, these were the `git checkout-index` command, and the code path for delayed

checkout, a feature used by some long-running smudge filters (see Section 4.4). As we later discovered, these two code paths were in fact vulnerable to the security issues described in Sections 4.2 and 4.3. Delayed checkout was more critical as it could be exploited during clone to drive remote code execution (RCE) attacks via a maliciously crafted repository (i.e. by tricking Git to write a malicious script in the "hooks" directory, so that it gets executed after checkout). We discuss more about this vulnerability and how we worked with the community to fix it in Appendix D.

As for parallel checkout itself, our study showed us that the workers were not vulnerable to this bug, and also could not use any outdated cache info. The workers start with a clean instance of the `lstat()` cache, and from the time they start running until they finish processing the assigned entries, neither them nor the main process removes any directories. These two invariants are enough to ensure no worker will use outdated information from the cache and end up creating files at wrong places, since the cached directories will be valid throughout the whole parallel execution. However, the main process was vulnerable in two situations:

- **Falling back to sequential mode**: after "pre-processing" all entries to be checked out and building the parallel checkout queue, is time to check whether the number of enqueued entries surpasses the configured threshold for parallelism, and start the workers if so. Otherwise, the entries are written sequentially by the main process itself. In the second case, the `lstat()` cache may already contain invalid data (because directories might have been removed during the "pre-processing" phase), and although the queue follows the index order, it likely will not start from where the "pre-processing" phase left off. So the invalid cached data may be erroneously used at this point.

  Here is an example: say we have the entries `A/a`, `A/b`, `A/c`, and the symlink `a` in the index. Now suppose the first two entries are not eligible for parallel checkout because they require external filtering. The pre-processing phase will then check out `A/a`, `A/b`, and the symlink `a` (which is also not parallel-eligible), leaving `A/c` in the parallel checkout queue. Now we proceed to the parallel phase. At this point, the `lstat()` cache of the main process considers `A` as a directory, but the path was in fact replaced by a symlink in the working tree. The checkout queue is to small to distribute among multiple workers (it has only one entry), so the main process decides to sequentially write the queue itself. Now it will check out `A/c` following the symlink `a`, while thinking it is a real directory.

- **The handling of collided entries after parallel checkout**: this is more or less similar to the first case, but this time we actually start the workers and run into the cache vulnerability after they finish. If the workers find a path collision, they mark that entry to be checked out later by the main process and, when that time comes, the `lstat()` cache of the main process may have already outdated information from the pre-processing phase, leading to the bug.

  We can use the same example from the previous item to illustrate this one, but we need to add one more parallel-eligible entry (`A/d`) and set the threshold for parallelism to 2. Now, the parallel checkout queue will contain tow entries (`A/c`

and `A/d`), so the checkout machinery will try to check them out using two child processes. However, the workers will find that `A` is not a directory and deduce that a path collision must have happened, so they will mark the entries with `PC_ITEM_COLLIDED` and let the main process handle them later. When that happens, the main process will retry writing the two entries, but its own `lstat()` cache will be outdated. So it will follow the symlink `a` and end up writing the two files at the wrong place.

These two vulnerable spots in the parallel checkout framework would in fact be solved once we had the fix for the more general bug, which was that the `lstat()` cache could become stale when there are path collisions. By protecting the cache from ever getting into this state, we would no longer need to worry about unordered checkout cases, which includes parallel checkout. A complete discussion about the patches that fixed this vulnerability, as well as the development process, can be found in Appendix D. To summarize it, we decided to manually reset the whole `lstat()` cache whenever the Git process removes any directory (through the `rmdir()` call) and also when a child process finishes. It is easy to see how these mechanics can indeed fix the two examples given above, by not allowing the `lstat()` cache to become stale in the first place.

6.2.3 A General Overview of the Final Code

To put all the pieces together, we will here show a brief overview of the step-by-step code path taken by parallel checkout. Using a top-bottom approach, we will start from the parallel checkout API, which was developed at the same level of `checkout_entry()`, to support the largest number of users of the checkout machinery. The API designed is easy to integrate for current sequential users of `checkout_entry()`, requiring a minimum amount of changes. The `Patch` 6.1 shows an example of how these changes should look like for a generic `checkout_entry()` user.

Note that `checkout_entry()` transparently decides whether the entry is parallel-eligible or not and properly enqueues it for the later `run_parallel_checkout()` call when appropriated. To do that, the function received a small modification, represented in italic in the simplified workflow below (adapted from Git Development Community, 2021b):

1. Check whether there is any untracked or unclean file in the working tree which would be overwritten by this entry, and decide whether to proceed (removing the file(s)) or not;
2. Create the leading directories;
3. *Load the conversion attributes for the entry's path;*
4. *Check, based on the entry's type and conversion attributes, whether it is parallel-eligible. If so, enqueue the entry and the loaded attributes for parallel checkout. If not, write the entry right away, using the default sequential code.*

The function was also changed to optionally receive the conversion attributes, instead of having to load them itself. This is used by the main process both when retrying to check out entries that the workers could not write due to path collisions and also when

Program 6.1 Example of a patch adding parallel checkout support to an existing `checkout_entry()` user. Adapted from Git Development Community, 2021b.

```
diff
@@ -1,6 +1,20 @@
 ...

+int pc_workers, pc_threshold, err = 0;
+struct checkout state;
+
+get_parallel_checkout_configs(&pc_workers, &pc_threshold);
+
+/*
+ * This check is not strictly required, but it
+ * should save some time in sequential mode.
+ */
+if (pc_workers > 1)
+ init_parallel_checkout();
+
 for (each cache_entry ce to-be-updated)
   err |= checkout_entry(ce, &state, NULL, NULL);

+err |= run_parallel_checkout(&state, pc_workers, pc_threshold, NULL, NULL);
+
 ...
```

the checkout machinery falls back to sequential mode after enqueuing entries for parallel checkout (more on that later).

After the `checkout_entry()` loop, `run_parallel_checkout()` spawns the workers and distributes the entries among them if the size of the queue is above a set threshold. Otherwise it just writes the entries sequentially. This is because, for a few entries, the parallelization gains may not outrun the cost of process spawning and communication. (By default the threshold is 100 entries, but this value can be configured through the `checkout.thresholdForParallelism` setting.) The entries are distributed in contiguous chunks to minimize the chance of having more than one worker simultaneously writing entries at the same directory, which could incur in lock contention in the kernel. We also experimented with other forms of work distribution like work stealing, but we did not observe any significant performance changes when using it.

For each entry, the workers perform the following steps (adapted from Git Development Community, 2021b):

1. Checks if there is any non-directory file in the leading part of the entry's path or if there already exists a file at the entry's basename. If so, mark the entry with `PC_ITEM_COLLIDED` and skip it;
2. Creates the file (with `O_CREAT` and `O_EXCL`);
3. Loads the blob into memory (inflating and delta reconstructing it);
4. Applies any required in-process filter, like end-of-line conversion and re-encoding;

5. Writes the result to the file descriptor opened at Step 2;
6. Calls `fstat()` or `lstat()` on the just-written path, and sends the result back to the main process, together with the end status of the operation and the item's identification number.

Steps 2 to 5 are very similar to the ones taken by the sequential checkout path, but as previously mentioned, they use a different function to accommodate the specificities of parallel checkout. Furthermore, when possible, these steps are combined by the streaming machinery to avoid loading the whole blob into memory all at once, just like sequential checkout does.

As mentioned earlier, if the worker encounters an error, it is allowed to remove the file it has already created to avoid leaving it empty in the working tree. This is the **only** time a worker can remove any file, and this is paramount to avoid race conditions and also to ensure our collision detection mechanics work correctly.

After the last worker finishes its assigned entries, the main process (still at `run_parallel_checkout()`) inspects the exiting codes and proceeds to handling the results, in two steps (once more, adapted from Git Development Community, 2021b):

1. First, it updates the in-memory index with the `lstat()` information sent by the workers. (This task must be done first as this information might me required in the following step.)
2. Then it writes the items which collided on disk, i.e. items marked with `PC_ITEM_COLLIDED`. (This time, `checkout_entry()` does not need to reload the conversion attributes as we already have them in the parallel checkout queue.)

6.3 Correctness and Regression Tests

The Git repository includes its own shell-based testing framework and a test suite containing about 932 test files with a total of 13892 individual test blocks as of v2.32.0 (note that this includes setup blocks, which are not properly tests). Developers are encouraged to write new tests whenever they add new functionality or fix bugs, both to check that the modified code behaves as expected and to serve as a regression-detecting mechanism for future changes. The test suite contains mostly functional tests for the different commands, options, and configurations, but there are also white-box tests of Git's internals and on-disk data structures. It also includes a few C helpers that are compiled against the Git code base both to perform some unit tests and to help on test setup.

The parallel checkout feature promoted many changes to the checkout machinery, which affects many commands. Such changes obviously could not be submitted without a good testing coverage. To do that, we ported the tests written by Jeff Hostetler in his parallel checkout implementation, and contributed with some additional tests for collision detection, checkout of submodules, forced checkouts, symlink detection on leading directories, and parallel checkout support on other commands. The parallel checkout tests are divided in three files, each focusing on a different aspect of the feature:

- **t2080-parallel-checkout-basics.sh**[7]: Basic tests to ensure parallel checkout produces the same result as the classic sequential mode. These tests run `git clone` and `git checkout` with the sequential mode, parallel mode, and fallback-to-sequential mode, which happens when parallel checkout is enabled but the number of entries is below the configured threshold for parallelism. Then it checks the output of `git status` and the difference between the tree blobs and the index to make sure that both the index and the working tree were constructed correctly. After all modes have been executed, we also directly compare the working trees. The checkout operation chosen for this test is interesting because it contains many file type changes (e.g. from the symlink "d" to the file "d/d"), content changes for each type of entry (regular file, symlink, and submodule), new files, and finally, removed files. So we get to see how checkout behaves on many scenarios. This test file also contains test cases for symlink detection on leading dirs, and the `--force` option.
- **t2081-parallel-checkout-collisions.sh**[8]: This file includes tests to check that the workers can correctly detect collisions both at the basename and at the dirname of to-be-written entries. We also explicitly test that the workers do not follow symlinks when they collide with the dirname of an entry and that the collisions are properly reported on clone (like the sequential checkout does).
- **t2082-parallel-checkout-attributes.sh**[9]: This file exercises parallel checkout in cases where different smudging filters are required. The goal is to make sure that all conversion attributes are correctly passed from the main process to the workers, and that they have all that is needed to convert the entries by themselves. We also check that entries requiring external filters (thus ineligible for parallel checkout) are properly written at the same operation that writes parallel-eligible entries. This includes a test for the interoperability of parallel checkout with delayed checkout.

In addition to the tests specifically targeting parallel checkout, we also modified one of the automatic test rounds that the Git project runs at the GitHub Actions infrastructure to execute all tests with parallel checkout enabled. Not all tests will actually use parallel workers in this test round — since they must check out at least two files (in the same operation) to allow for any parallelism — but this approach is still interesting as we can at least test the fallback-to-sequential mechanics on different commands.

6.4 Main Contributions

Parallel checkout was truly a collaborative endeavour, with the participation of different developers throughout a large time span. Without the different contributions, the feature would not have achieved the level of maturity it has now. In this section, we are going to highlight the key aspects originated from each of the parallel checkout implementation of 2016 and 2020 that were incorporated into the final version merged upstream. In this process, we are also going to enumerate the main new contributions from this

[7] https://github.com/git/git/blob/v2.32.0/t/t2080-parallel-checkout-basics.sh

[8] https://github.com/git/git/blob/v2.32.0/t/t2081-parallel-checkout-collisions.sh

[9] https://github.com/git/git/blob/v2.32.0/t/t2082-parallel-checkout-attributes.sh

particular work.

Table 6.3 shows some of the main general differences from each parallel checkout approach, including design choices, features (marked with "✓"), and possible issues (marked with "X").

The general structure for the parallel checkout framework merged upstream comes from the approach of 2016. This includes elements such as: the general API, the way to handle the progress meter, the `poll()` loop (to avoid busy-waiting), and the entry enqueuing strategy. The structure from the 2016 parallel framework supports all checkout commands, can use the streaming interface, and parallelizes both I/O and CPU (object decompression), which makes it interesting for both NFS and local SSDs. As mentioned earlier, it also provides a good trade-off between code complexity and flexibility/robustness.

To handle the attributes loading and filtering issues enumerated at Section 4.4, we resorted to the mechanics implemented in the approach of 2020. This includes the classification of entries that require external filters as ineligible for parallel checkout (to avoid breaking non-concurrency assumptions), and the changes to preload the entries' attributes and send them to the workers. This is important because only the workers do not have access to the in-memory index of the main process, which may be required when loading the attributes. From the 2020 version we also incorporated parts of the interprocess communication protocol (with some simplifications, such as removing the initial handshake), the suite of parallel checkout tests, an adjustment to the path collision report on clone to make it work with parallel checkout, and parts of the design document.

As for this work, our main contributions to the parallel checkout implementation were:

- The analysis of the previous parallel approaches, applying the best features of each one for the final version. The goal was to achieve a fast and robust implementation with as little patch complexity as possible.
- The study of the `lstat()` cache and its possible misuses during unordered checkout. This analysis not only ensured that parallel checkout would be protected against stale uses of the cache, which could result in serious security risks, but also allowed us to find and fix a remote code execution vulnerability which was already in the code (more about it at the Appendix D).
- The expansion of the parallel checkout tests. The added tests include checks for collision detection, checkout of submodules, forced checkouts, symlink detection on leading directories, and parallel checkout support on other commands. We also employed a continuous integration rule to execute a full round of the Git test suite with parallel checkout enabled, to potentially catch any odd interactions that the specifically design tests might have missed.
- The capability to safely write in parallel on a non-empty working tree. Parallelizing writes is very important for NFS, where that was the main checkout bottleneck. The previous approaches were all able to write in parallel, but with either security or usability limitations. The implementations from 2008 and 2016 could not

	2008	2016	2020	2022
Parallelism Technique	Threads	Processes	Processes & threads	Processes
Focus of parallelism	Writing	Reading, writing and decompression.	Reading and decompression. *Also writing, on clone.	Reading, writing and decompression.
Patchset complexity	2 patches 165 added lines	12 patches 730 added lines	37 patches 4840 added lines	22 patches 2058 added lines
Support all checkout cmds		✓		✓
Uses streaming interface	*N.A. (no streaming interface)*	✓		✓
final stat() in parallel	✓		✓	✓
Adds tests			✓	✓
Work Redistribution		✓		
May load attributes using wrong index		X		
Path collision detection not fully adapted for parallelism	X	X	(Only if async mode is used on non-empty working trees)	
Race conditions on file creations and removals	X			
Race conditions on attributes reading and filtering	X			
Concurrency on external filters		X		

Table 6.3: *General design and feature differences from each parallel checkout approach.*

reliably detect collisions among two entries being checked out in parallel. And the approach from 2020 could only write in parallel when running under the async mode, which requires a clean working tree before checkout and, thus, can only be used by `git clone`. (Otherwise, the code could miss collisions between an entry being checked out and an untracked symlink in the working tree.) We proposed a mechanism which allows us to obtain the performance benefits from parallel writes on all checkout cases, without missing any type of path collision among regular files and symlinks.

- The performance and memory benchmarks conducted (more about them at the following chapter), which allowed us to understand in which types of storage parallel checkout is most effective in order to properly advise Git users and choose the default configurations.

Chapter 7

Results

In this last chapter, we will present the results from our performance and memory benchmarks on the parallel checkout feature, which was submitted and merged into the upstream Git project. We will also discuss its current limitations and what we consider to be some possibilities for future improvements, wrapping up with a conclusion about the main contributions of this work.

7.1 Performance Benchmarks

To test the parallel checkout performance, we benchmarked the same operation we used for profiling: a checkout execution on an empty working tree of the Linux kernel repository, which requires the creation of over 70 thousand files. This is an operation that represents well what we were seeking to optimize with parallel checkout, which is the file creation process. Nevertheless, for those interested in seeing how parallel checkout performs with other checkout operations that require more non-parallelized work, such as index manipulations and file removals, please refer to Appendix E.

Once again we repeated the test on a shallow clone of the Linux repository containing only loose objects; but remember that these tests are mostly for research purposes, as no repository should have these many loose objects in natural use cases. All benchmarks show mean runtime of 15 executions and the limits for a confidence interval of 95%. The chosen sample size of 15 executions produced, in these performance benchmarks, a reasonable trade off between a sufficiently small error margin and a shorter time to run all the experiments.

7.1.1 SSD

Figures 7.1, 7.2 and 7.3 show the results on SSD. The speedups range from 2x to 3.6x on the packed objects case, and 2.3x to 7.2x on the loose objects case. On all three machines[1], we got the best overall results at around 16 to 32 workers for the packed objects, and 64 workers for the loose objects. But notice that we start seeing diminishing returns around

[1] Refer to Appendix C for the specifications of the machines used to ran the tests.

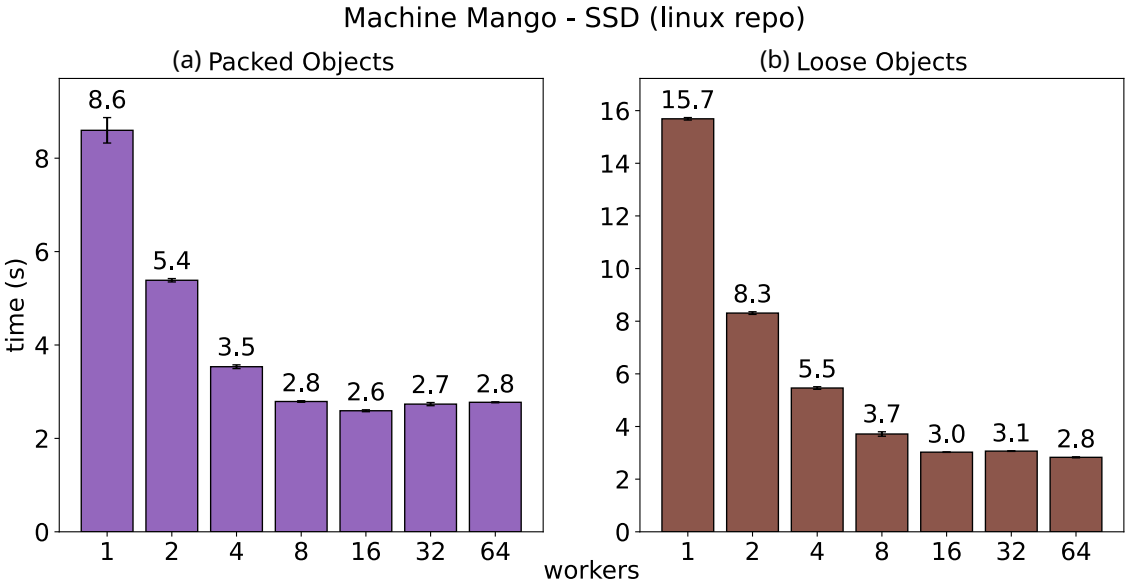


Figure 7.1: *Checkout benchmark on machine Mango - SSD*

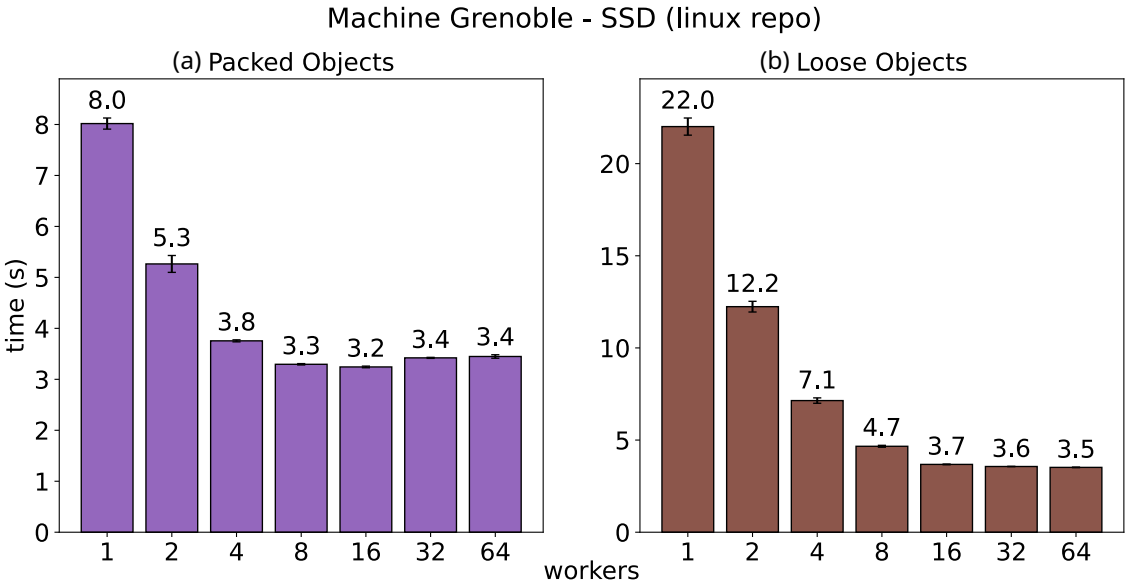


Figure 7.2: *Checkout benchmark on machine Grenoble - SSD*

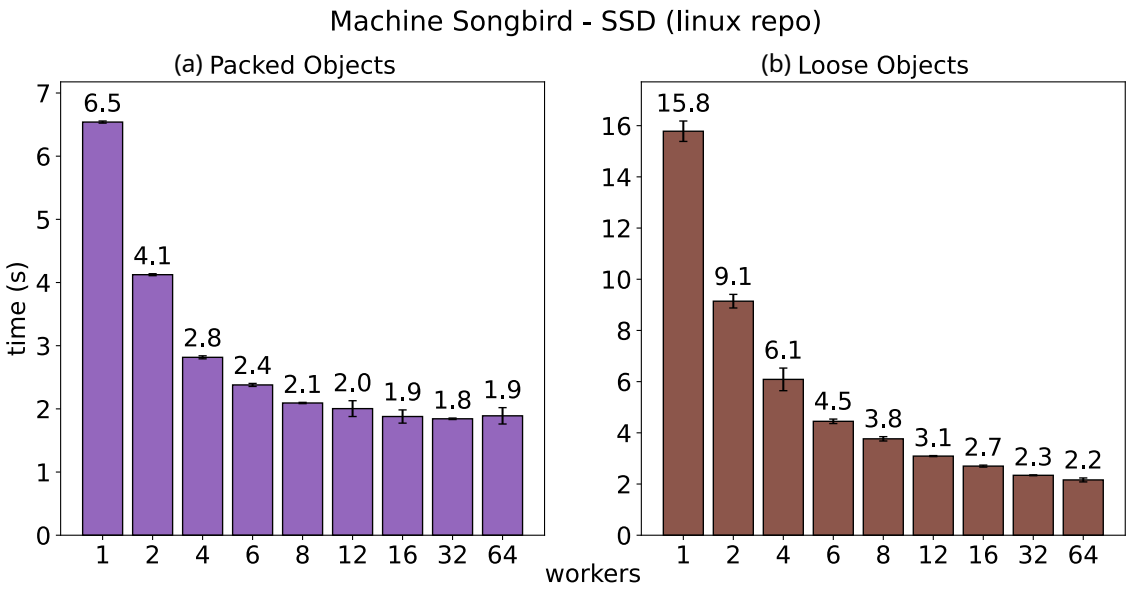


Figure 7.3: *Checkout benchmark on machine Songbird - SSD*

8 workers. Thus, values higher than that may not be worth the additional usage of system resources (like RAM and I/O bandwidth).

The speedup in the packed object case was already expected as the CPU-bound (and well parallelizable) inflation operation dominated the run time for the sequential checkout. But it is interesting to see the even greater speedups in the loose object case, where object reading was the most time consuming task. Using *iostat*[2] we could see that the percentage of read request that get merged together during the loose objects checkout goes from 7.9% with one worker to 10.4% with 64 workers on Mango. Since this is only a small increase, the speedup likely comes, in its majority, from the efficiently internal parallelization on the SSD (and perhaps a better usage of the I/O bandwidth as well).

7.1.2 HDD

The HDD results are shown in Figures 7.4, 7.5 and 7.6. As seen on these results, parallel checkout was not very effective on the HDDs. All three machines achieved some improvement in the packed objects case, but the overall speedup was too small. The only exception is Wall-e (Figure 7.4a), where we did see an interesting 2.1x speed boost. Furthermore, there seems to be some variation regarding the optimal number of workers for the packed objects case in each machine. Some values even produce performance degradations. As for the loose objects case (Figures 7.4b, 7.5b, and 7.6b), parallel checkout was in fact significantly slower than the sequential version on all three machines. (But remember that this is an artificial case, that should not happen much in practice.)

The time reductions on the packed objects case likely come from a combination of two factors: the overlapping of I/O and computation, and the deeper I/O queue depth together with a favorable pattern for optimization by the scheduler (i.e. request reordering and/or

[2] https://man7.org/linux/man-pages/man1/iostat.1.html

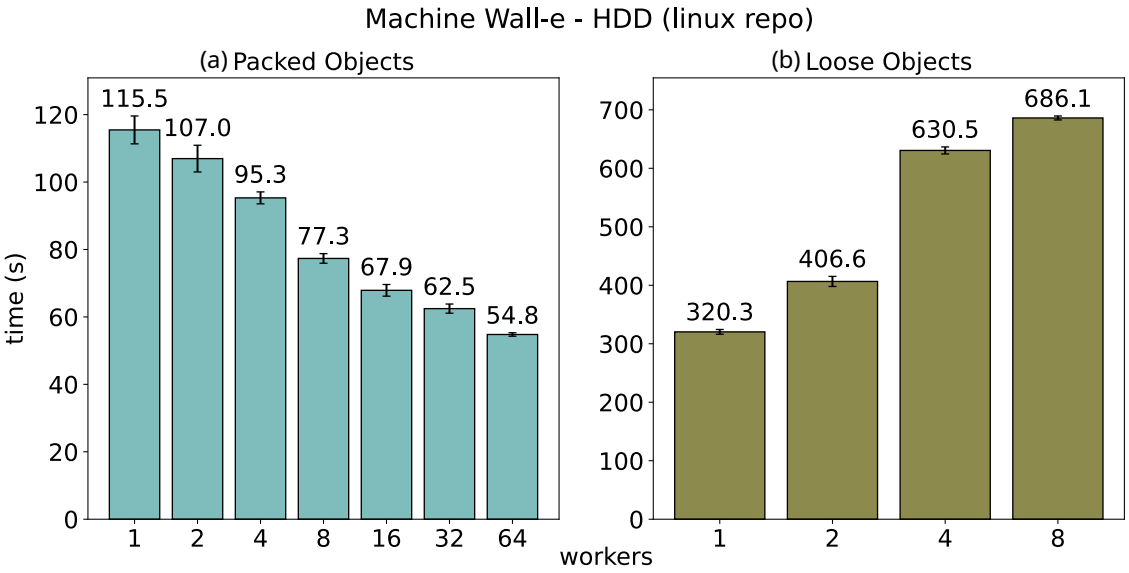


Figure 7.4: *Checkout benchmark on machine Wall-e - HDD*

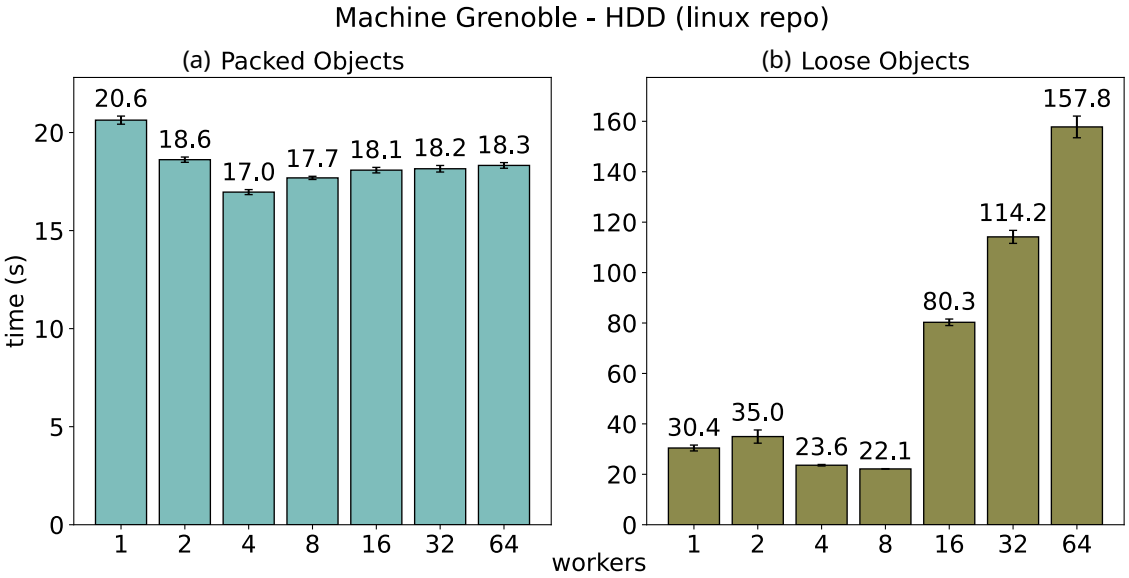


Figure 7.5: *Checkout benchmark on machine Grenoble - HDD.*

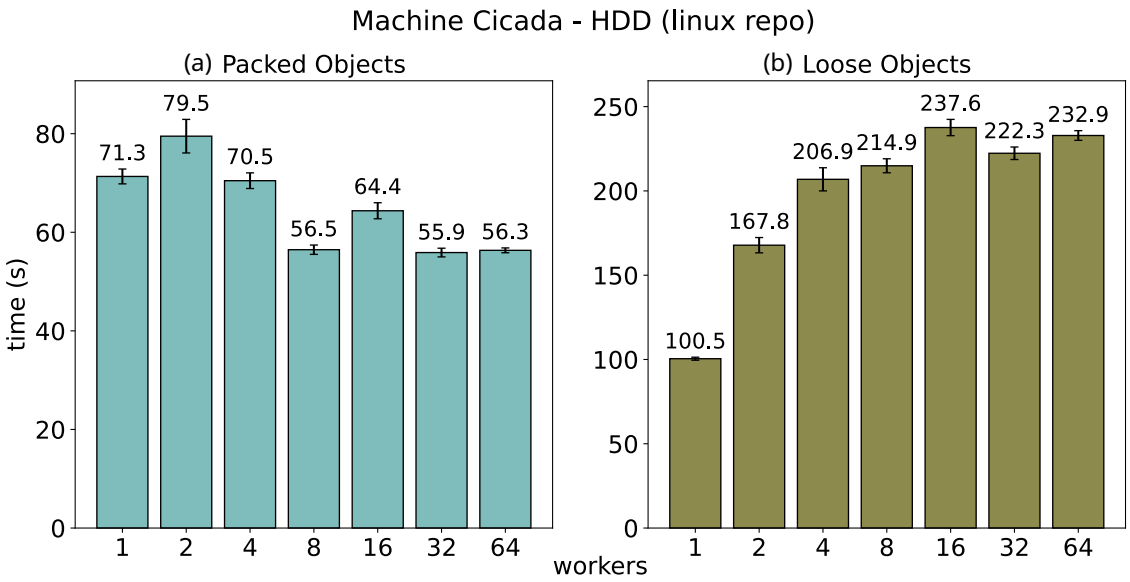


Figure 7.6: *Checkout benchmark on machine Cicada - HDD.*

merging). Thus, the HDD can probably serve the requests in a way that lowers the disk seeking time.

The odd case among the packed objects plots is machine Wall-e (Figure 7.4a). On this machine, we keep seeing speedups as we increase the number of workers, and the improvements are significantly better than the other two machines. To inspect that, we used the `filefrag`[3] tool, which showed that the 3.2 GiB packfile from our tests was handled differently by the file system of each machine: on Cicada, the file was fragmented into 4 extents[4]; on Grenoble, 23 extents; and on Wall-e, 245 extents. Probably Wall-e's HDD was more occupied, so the system could not find large contiguous chunks, forcing it to further fragment the file.

Reading a fragmented file is bad for performance as it requires more seeking on disk and read ahead is less effective. However, the increased I/O queue depth, leading to higher optimizations opportunities for the scheduler, likely compensated for the performance issues of fragmentation and reduced the disk seeking time. That is our hypothesis as to why parallelization was so effective in this particular case, producing the larger speedups from Figure 7.4a. Conversely, the less fragmented files from Cicada and Grenoble could benefit more from read ahead mechanics (as more blocks are packed together on disk). So sequential checkout was already faster in comparison to Wall-e, and the reordering and merging of the concurrently-issued read requests was likely less efficient. That is, newly requested bytes could have already been loaded into memory on a previous request thanks to read ahead.

As for the loose objects benchmarks, the observed behavior is a bit harder to understand. This time, any gains from overlapping I/O and computation are likely outweighed by a performance degradation from the parallel reads. It is quite possible that the reading

[3] https://man7.org/linux/man-pages/man8/filefrag.8.html

[4] Each extent is a contiguous range of data blocks.

patterns produced by the parallel workers did not provide good opportunities for the I/O scheduler, even with a deeper queue depth. One of the hypothesis we considered is that the loose files could be too scattered over the disk, making it harder for the scheduler to perform effective merges on the incoming read requests (specially since the queue of outstanding requests has a limited size). As we mentioned before, files with some amount of fragmentation, like Wall-e's packfile, may actually be good candidates for parallel reading, but this effect likely does not scale indefinitely as the number of fragments grow. And if the dispersion of the loose files are **way too big**, it would not be surprising if it produced the opposite results on parallel performance.

Using the `filefrag` tool once again, we collected the physical offsets for each loose object file on machine Cicada's HDD. The aggregated data for the ~76K files — each composed by a single extend — were split in 15547 discontinuous chunks of blocks. (Where each block corresponds to 4KiB.) This large number of chunks could make it rarer to find multiple outstanding I/O requests that are close together and, thus, candidates for merging. Furthermore, we measured how the parallel workers change the percentage of read request getting merge during checkout, showing that the parallelism was indeed much more effective in increasing this measurement in the packed case than in the loose case. These results can be seen at Figure 7.7.

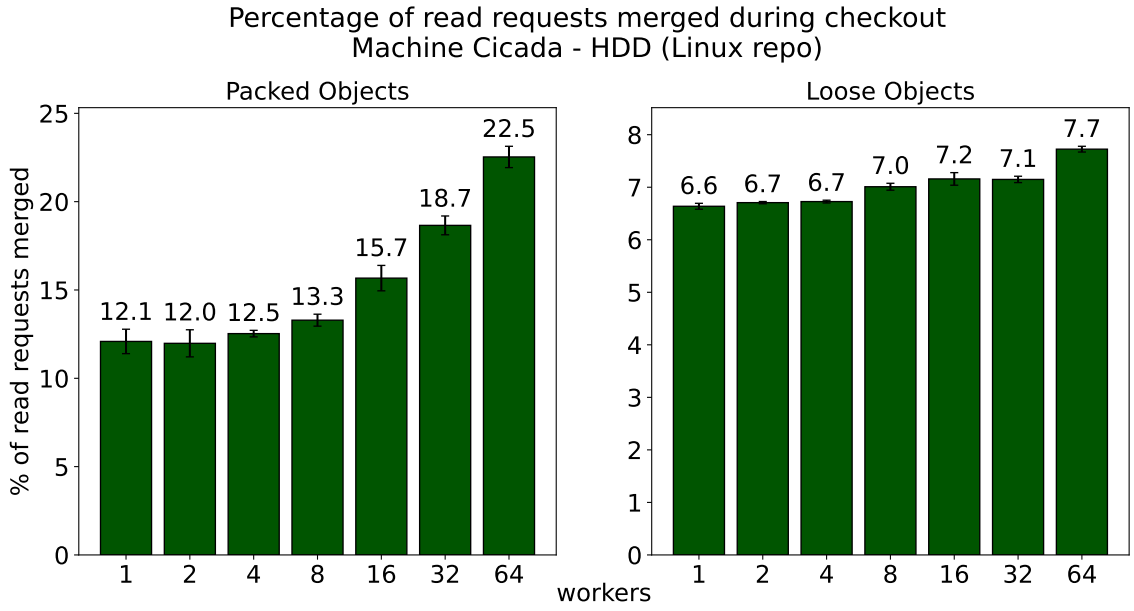


Figure 7.7: *Percentage of read requests merged together before being sent to the device during checkout on Machine Cicada (HDD). Plotted values are the average of 15 samples.*

Finally, it is worth mentioning that, without enough optimization opportunities for the I/O scheduler, the effect of parallelism on the HDD could lead to performance degradation due to the dispute over critical resources and locking contention on the concurrent reads.

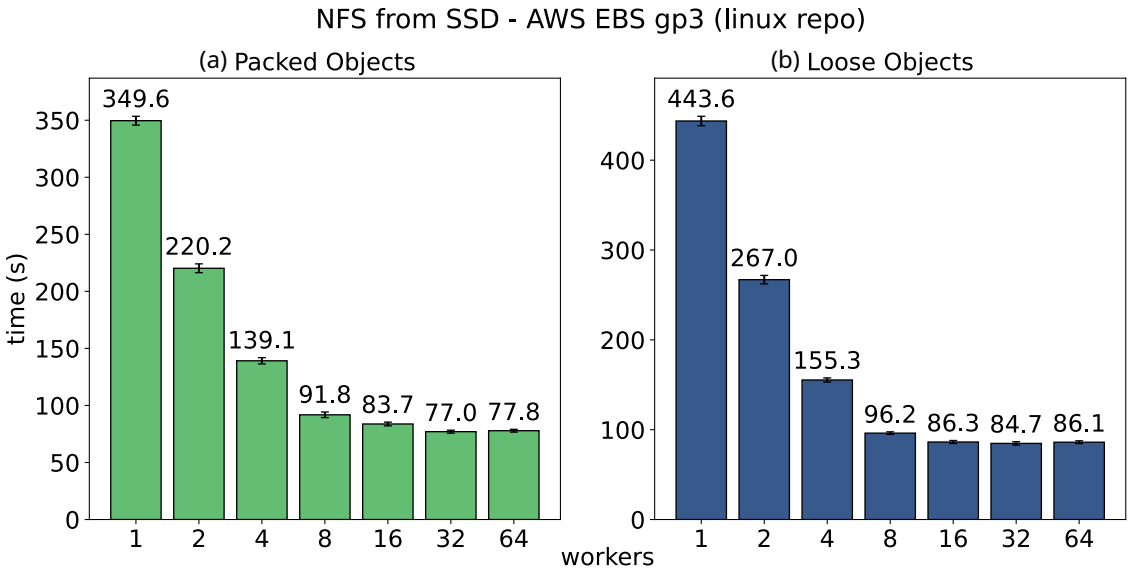


Figure 7.8: *Checkout benchmark on NFS - AWS EBS gp3 (SSD)*

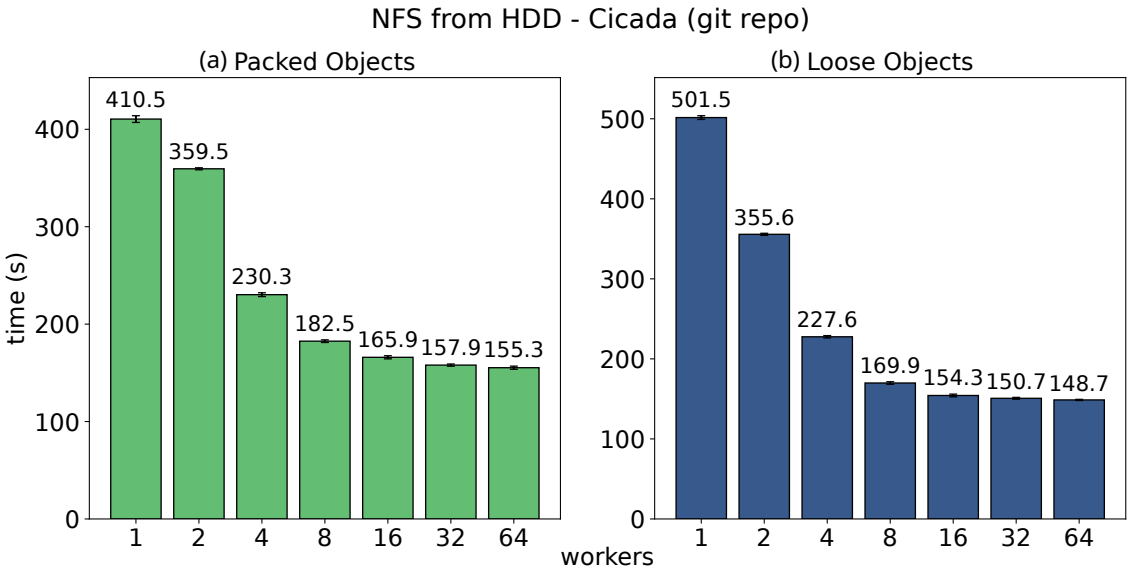


Figure 7.9: *Checkout benchmark on NFS Cicada - HDD*

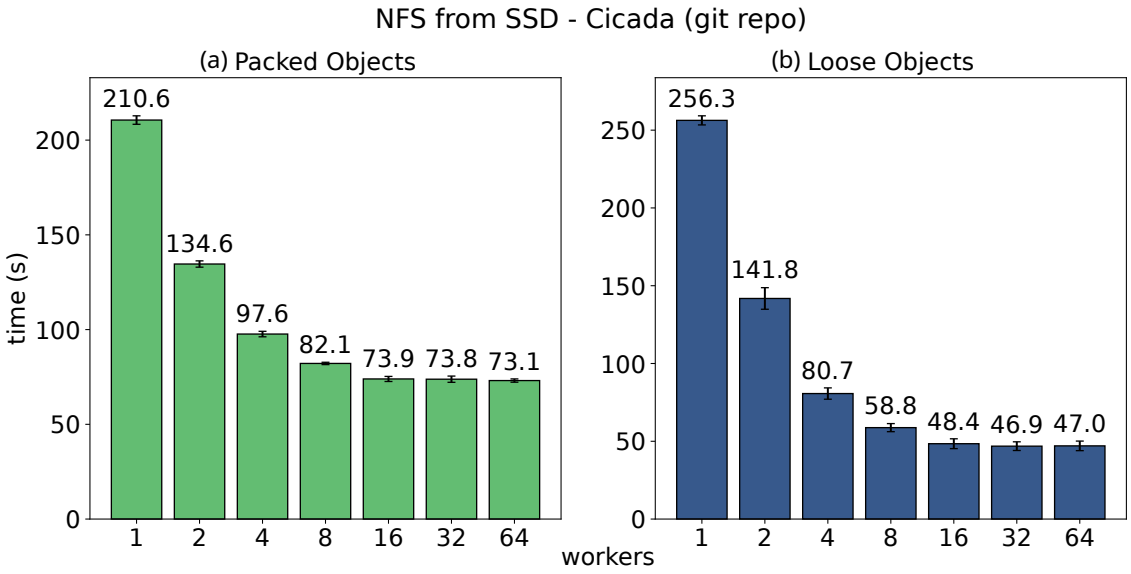


Figure 7.10: *Checkout benchmark on NFS Cicada - SSD*

7.1.3 NFS

Figures 7.8, 7.9, and 7.10 show the results on NFS. On the "NFS Cicada" setup (Figures 7.9 and 7.10), the checkout benchmark with the Linux repository was taking too much time to run: almost an hour for a single execution of the sequential checkout on the SSD and over two hours on the HDD (that is, both over NFS). So we decided to use the Git repository instead. This repo contains about 4 thousand files at tag v2.32.0. The packed and loose repository versions were created exactly like the Linux ones. The first ended up with around 310 thousand objects, totaling 172 MiB (of both size and disk space); and the second ended up with around 4 thousand objects, totaling 14 MiB of size (or 26 MiB of disk space).

We obviously cannot compare the times from Figures 7.8 and 7.9 as they are using different repositories, hardware, configurations, and etc. But we wanted to see what difference an SSD makes on a parallel checkout over NFS, so we repeated the tests on "NFS Cicada" using its "caching SSD", a small 20 GiB SSD meant for caching and accelerating the Windows boot. Beware that this device is not designed for general storage[5].

The best results on NFS were at around 32 to 64 workers. However, we start to get diminishing returns from 8 workers, so this seems to be a good value for NFS mounts as it can achieve good performance without overusing the system's resources. As discussed in Chapter 3, two functions dominate the execution time of our checkout operation on NFS: `open()`, with 44% of the total runtime, and `fstat()`, with 33~40%. Both these functions spend most of their time off-CPU on network request/response operations and other I/O waiting time. The parallel checkout implementation amortizes the network latency associated with these expensive calls and allows more parallel work on the server.

We also ran the benchmark on a simulated single-core setup for both the NFS server

[5] For completeness, we also benchmarked parallel checkout locally on the caching SSD, using it as a general storage device. The results are show at Section E.3.

and client, to see how the parallelism would affect performance when we do not have multiple cores available. This was performed using the Linux CPU Hotplug feature (Linux Development Community, 2021), which allows us to disable some CPU cores by writing "0" to the special files `/sys/devices/system/cpu/cpu<number>/online`. For this benchmark, we disabled all cores but one (core 0), on both the NFS server and client of the NFS Cicada setup. Results are shown at Figures 7.11 and 7.12. Once more, these are average run times and confidence intervals from 15 samples with a confidence level of 95%. The results were very similar to the ones from the multi-core setup, with the optimal configuration around 8 to 16 workers. This suggests that the performance gains we get from the parallelism on NFS are also applicable for single-core machines.

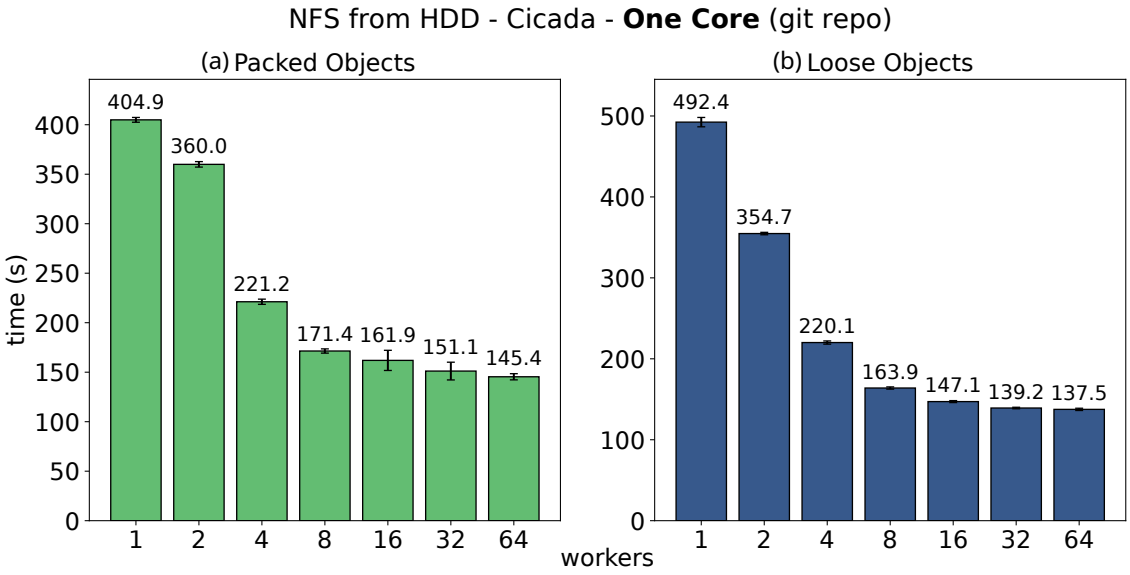


Figure 7.11: *Checkout benchmark on NFS Cicada - HDD, with a single-core setup on server and client.*

7.1.4 Windows

It would be impracticable to repeat our performance tests in every system and machine architecture where Git is used. However, we still wanted to see how parallel checkout behaves on a different operating system, so we ran the local benchmarks on Microsoft Windows as well. It has a large user base and an active development community on Git. Besides, it is not a UNIX-like system (differently then macOS, BSD, and Linux), so it is a good choice to complement the benchmarks we already have on Linux. We ran these tests using the `Git for Windows SDK`[6], which comes with the `mintty` terminal emulator and the `git-bash` shell. Since the `Git for Windows SDK` is a subset of `MSYS2`[7], it was possible to use the benchmark scripts we already had for Linux, with a few adjustments.

There are a couple caveats for checking out the Linux repository on Windows, though. First, the v5.12 tree contains some paths that only differ in case, such as

[6] https://github.com/git-for-windows/git/wiki/Technical-overview

[7] https://www.msys2.org/

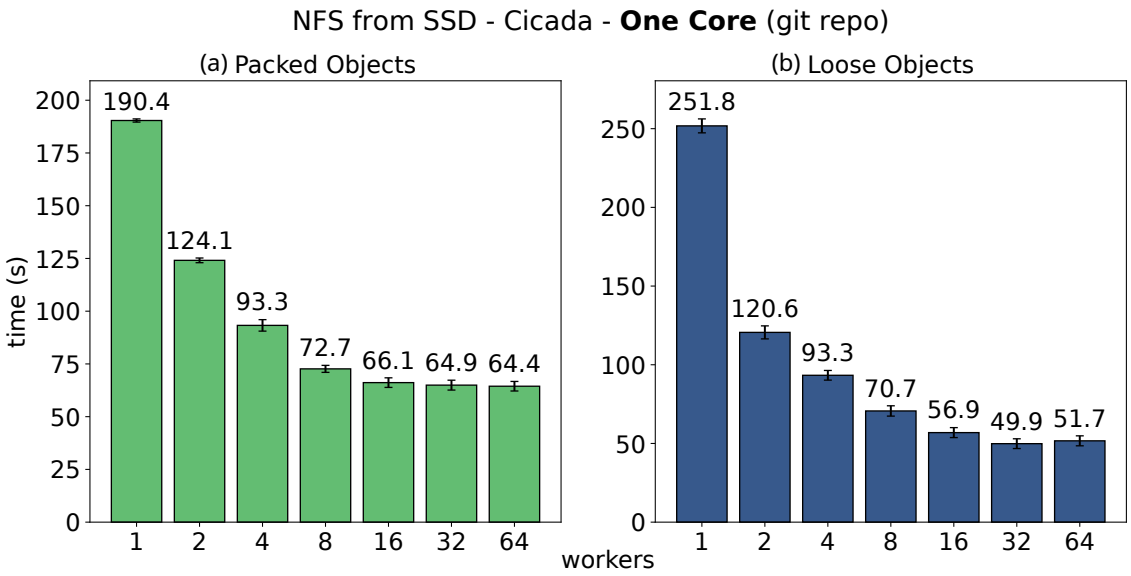


Figure 7.12: *Checkout benchmark on NFS Cicada - SSD, with a single-core setup on server and client.*

"`net/netfilter/xt_DSCP.c`" and "`net/netfilter/xt_dscp.c`". These paths collide during checkout on Windows, but they will not prevent the operation from completing successfully. Therefore, even thought a subsequent `git status` execution will show these files as dirty, there is really no impact in our benchmark. A more delicate case is caused by paths that use reserved named on Windows, such as "`aux`". The Linux v5.12 tree has three of such paths and, unlike case collisions, the reserved names would cause an error during checkout making the operation end unsuccessfully. To overcome this problem, we renamed the "`aux`" files, prepending them with a "`_`" character, so that they could be properly checked out on Windows.

Figure 7.13 shows the benchmark results on SSD (Machine Mango, Windows 10 partition), and Figure 7.14 shows the same benchmarks on HDD (Machine Cicada, Windows 10 partition). Both plots show the confidence interval of 15 samples (95% confidence level). Unfortunately, the HDD benchmark produced too much variance, even though Cicada was running a fresh Windows installation, without any additional apps. So we decided to try again with the Git repository (the upstream version, not Git-for-Windows). The packed and loose repositories were prepared just like it was described on Section 7.1, for the NFS Cicada tests. The results using the Git repository are shown at Figure 7.15.

Looking at Figure 7.13, we can see that the behavior of parallel checkout on Windows was similar to what we observed on Linux for the Mango machine (SSD). Although the speedup factor was a little smaller on Windows — with 2.22x on the packed objects case, and 3.75x on the loose objects case — the optimal number of workers remained the same, at around 8 to 16.

We start to see higher differences on machine Cicada (HDD). For the packed objects case, parallel checkout was marginally better than the sequential mode when checking out the Linux repository (Figure 7.14a). This is consistent to what we saw on the Linux system at the same machine. When checking out the Git repository, however, parallelism increased the run time for the packed objects case (Figure 7.15a). It is difficult to pinpoint

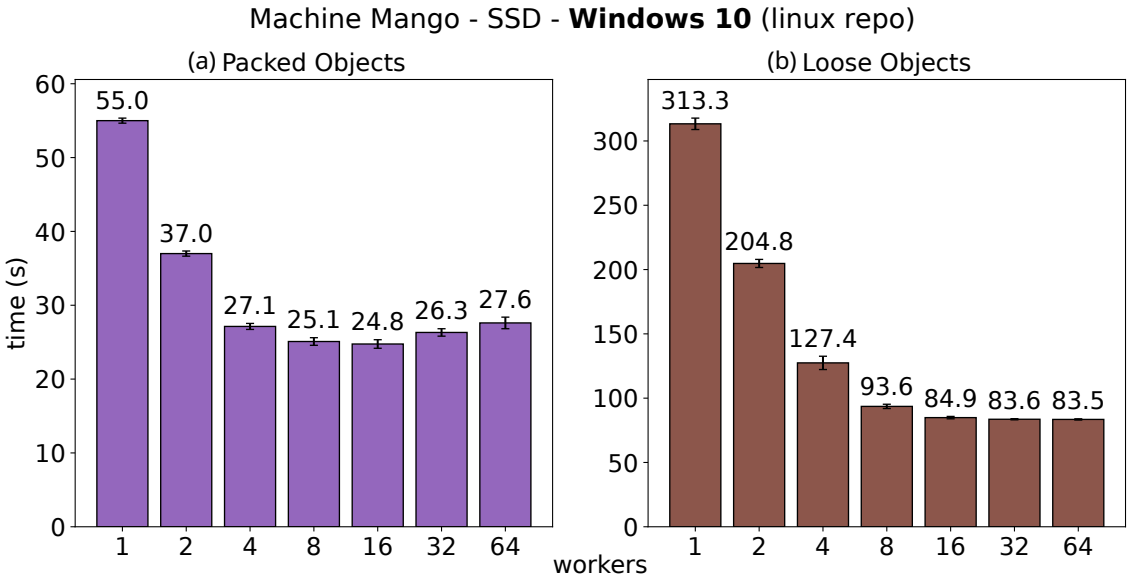


Figure 7.13: *Checkout benchmark on machine Mango - SSD, on Windows.*

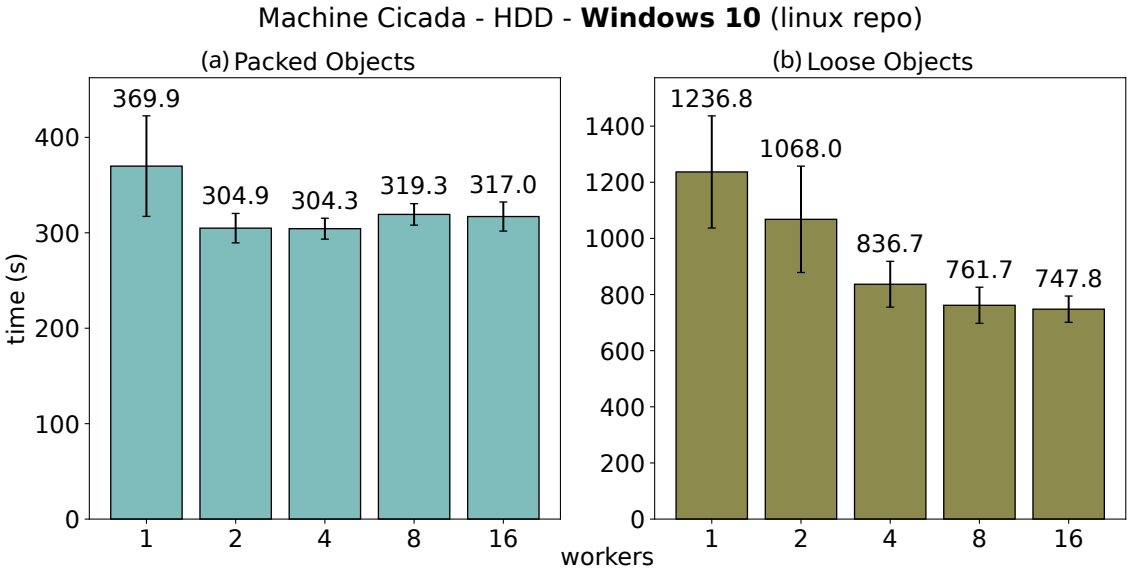


Figure 7.14: *Checkout benchmark on machine Cicada - HDD, on Windows.*

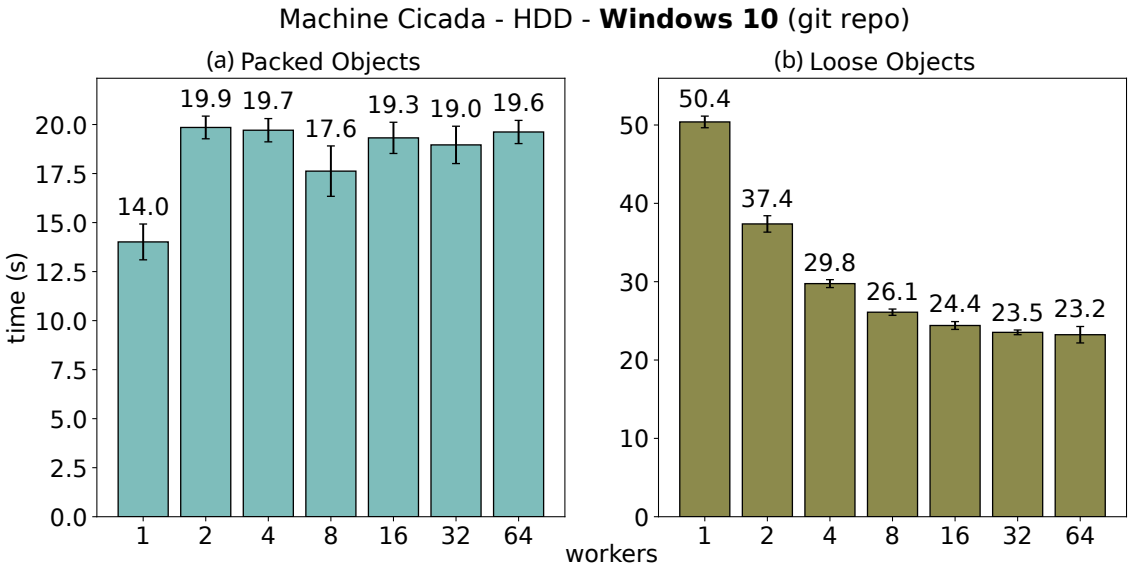


Figure 7.15: *Checkout benchmark on machine Cicada - HDD, on Windows.*

the reason behind this difference, but it might be related to the size and/or fragmentation of the packfiles. The Linux repository has a packfile of about 3.2 GiB, while Git's packfile is only 5% of that size. So there might not be enough workload to efficiently read the objects from it in parallel.

Regarding the loose objects case on the HDD, we see time reductions in both repositories (Figures 7.14b and 7.15b), with a 1.65x speedup on the Linux repo (although the variance was quite high), and 2.17x on the Git repo. These results are very different to the ones we got on the Linux system, where parallel checkout generally slowed down the loose objects checkout. One hypothesis for this discrepancy is that NTFS, the file system we used on Windows, could be storing the loose object files more closely together than ext4; which, in turn, would result in higher merging opportunities for the I/O scheduler.

7.2 Memory Benckmarks

Memory usage is an important factor to keep track when dealing with parallelism. A high memory usage can lead to performance degradation not only for the parallel program itself, but also for other processes running on the user's machine, so it is not something to take lightly. Furthermore, our implementation uses multiple processes with their own memory spaces (because we call `exec()` after `fork()`, replacing the children images). So, unlike threads or forked process with Copy-on-Write mechanics — i.e. without a subsequent `exec()` call — our workers do not share the heap, BSS, and other segments of memory (although they can share `mmap()`-ed file regions, as we will discuss later).

There are many ways to monitor memory usage, and just like profiling, it is important to understand what each tool really measures to decide which is most appropriate for each case. During parallel checkout, many files (or file regions) get mapped into memory to read the objects, and we want to measure the memory used by these too. So, as powerful

as a tool like `massif` is, it would not be suitable for our needs: By default, massif measures the heap memory allocated with `malloc()`, `realloc()` and similar functions. It can also optionally include the stack memory. However, it does not include pages mapped with `mmap()`, nor the size of the code, data, and BSS sections. These can be profiled with the `--pages-as-heap` option, but then we would be measuring the virtual memory, not what is actually on RAM. The distinction here is important because, not all the pages `mmap()`-ed during checkout necessarily get into the physical memory, as they are lazily loaded by the kernel when (and if) needed. The mapped windows from packfiles may in fact contain many objects that are not needed, and thus, the pages are not even read from disk. So, ideally, we would want to measure only the physical memory usage.

On the other hand, the RSS (Resident Set Size) measurement, which focus on physical memory, may also be misleading for our analysis as it includes shared memory (e.g. shared/dynamic libraries), which is not exclusively owned by a single process. Furthermore, since we are running multiple checkout worker processes, these values would be counted more than once. Instead, to understand how much memory parallel checkout uses, we decided to monitor the PSS (Proportional Set Size) of the Git processes over time. The PSS concept, proposed by Matt Mackall (Corbet, 2022), is an interesting way to understand how much physical memory a process is really using on Linux, as it attempts to better distribute the usage of shared memory. PSS is calculated by summing two values: the unshared memory of a process and its proportion of shared memory, obtained from the division of the total shared memory by the number of processes using it. To give an example, if a process is using 100 MiB of unshared memory and also sharing 100 MiB with three other process, its PSS is 125 MiB.

The PSS value of a process at a given time can be retrieved thought the procfs interface. More specifically, the `/proc/<PID>/smaps` file. We wrote a small memory profiler that stops[8] the `git checkout .` process and its children at a given frequency, inspects the relevant smaps files, and prints out the sum of their PSS together with a timestamp. We ran Git checkout through our profiler with different numbers of workers, and repeated each experiment 15 times to produce an interval of confidence for the average PSS usage over time. We collected approximately 40 samples per second. However, because there was some variance on the total number of samples and sampling rate over the 15 repetitions, we (linearly) interpolated the 2D points from each of them and re-sampled the resulting curves at fixed time intervals. With this processed data, we took the average PSS at each new X point and plotted the results together with the confidence interval, for a confidence level of 95%. These plots are shown at Figures 7.16 and 7.17.

Finally, at Table 7.1 we also present the average of the peak PSS in each profiling case. Note that the values on the table deviate a bit from the plots. This is expected since, the peak PSS on each of the 15 repetitions fell on different timestamps and, on the plots we took the average PSS value at each chosen X timestamp. In other words, the peak PSS on the plots is timed-based while the peak PSS on the table is time-independent.

[8] We also tried to collect the PSS data without stopping the processes, but that significantly reduced the sample rate; specially with higher numbers of checkout workers, which start competing with our profiler for system resources. Stopping the processes also helps increase the sample rate on short-running checkouts.

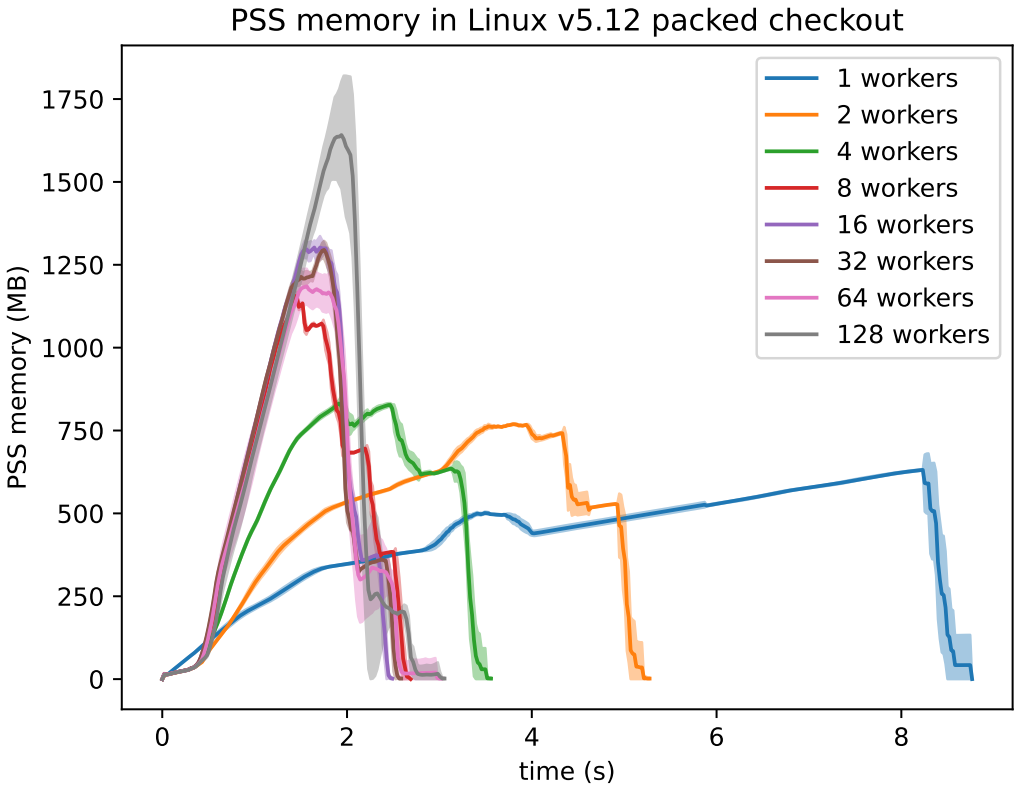


Figure 7.16: *Memory usage (PSS) on packed checkout of Linux v5.12. Machine Mango - SSD. Each curve represent the average of 15 executions, plotted with a confidence interval of 95%.*

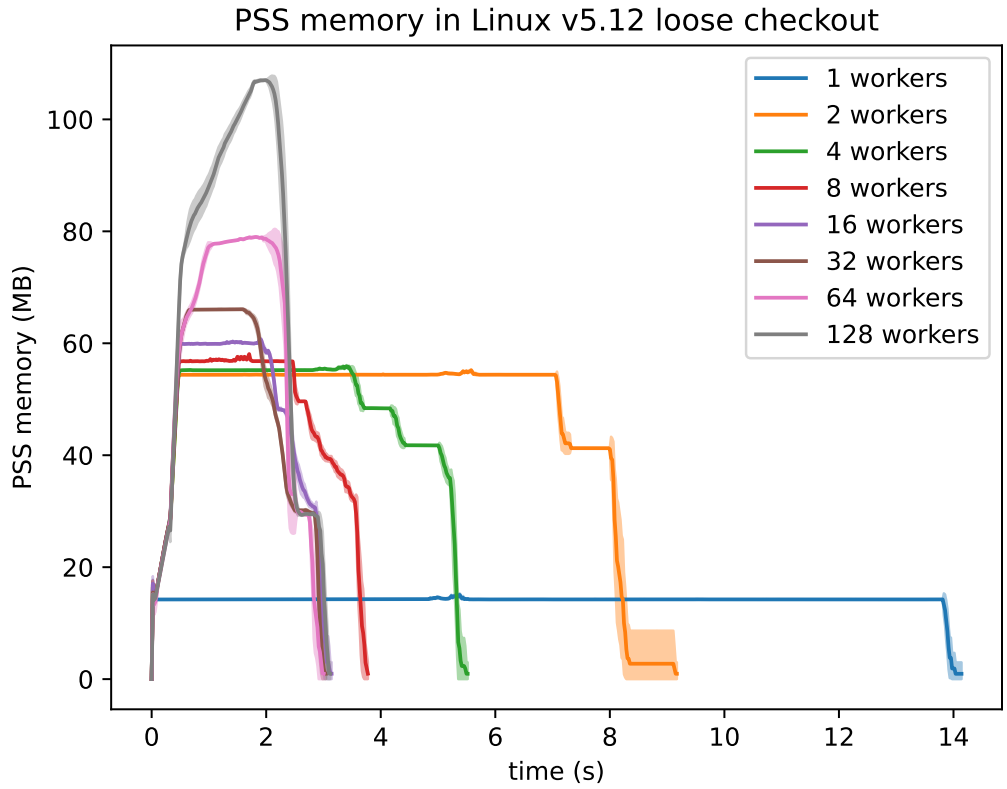


Figure 7.17: *Memory usage (PSS) on loose checkout of Linux v5.12. Machine Mango - SSD. Each curve represent the average of 15 executions, plotted with a confidence interval of 95%.*

Workers	Packed Checkout	Loose Checkout
1	636.10 ± 0.56	17.05 ± 0.82
2	777.62 ± 1.20	55.68 ± 0.08
4	863.44 ± 11.99	56.45 ± 0.08
8	1168.17 ± 4.39	58.12 ± 0.07
16	1372.67 ± 12.75	61.03 ± 0.08
32	1331.62 ± 19.65	66.27 ± 0.05
64	1243.71 ± 47.79	79.31 ± 0.05
128	1826.53 ± 104.15	107.35 ± 0.10

Table 7.1: *Peak memory usage (PSS) in MB of a Linux v5.12 checkout on machine* `Mango`*. The offsets show the confidence interval of 95% using a sample size of 15 executions.*

As expected, memory usage increases as the number of workers grow, but fortunately it is way below a linear growth. On the packed case (Figure 7.16), this is probably largely due to the packfile being mapped as read-only, which allows the pages to be shared among the multiple processes. On the loose case (Figure 7.17), where there is no intersection between the memory mappings of the different processes, the explanation might come from the fact that each mapped object is unmapped right after being checked out, and

each individual worker process a single blob at a time. Therefore, on the loose case, the order of simultaneously mapped object files is limited by the number of workers. This also explains why we see plateaus in the loose checkout plot, at Figure 7.17, and increasing slopes with no plateaus in the packed case, at Figure 7.16 (as well as the major difference on peak PSS usage). For packfiles, Git usually only unmaps a window when the total number of bytes mapped reaches the `core.packedGitLimit` threshold. (The windows chosen for unmapping are the ones least recently used.) The default value for this setting is 32 TiB[9] on 64 bits systems, which is considered "effectively unlimited", as mentioned on the documentation.

Because packfile windows are kept active during checkout, it is possible to run into situations where Git ends up mapping way more bytes than what is available on the physical memory. In this case, the system needs to swap-out pages to make room for newly needed pages, which might impact performance. This can already affect the sequential checkout mode, but the result will probably be more dramatic on parallel checkout: the different workers may be concurrently requesting pages that are not on RAM, which can lead to memory thrashing. This was not the case in our memory benchmarks, as the physical memory of `Mango` was sufficient to hold all bytes needed during the Linux checkout (even though Linux is not a small repository and the benchmarked operation has to load many objects). However, the memory thrashing possibility is something to keep in mind for machines with small RAMs and/or when checking out massive repositories, with many large packfiles, like the Windows repository (Harry, 2017).

7.3 Current Limitations

Parallel checkout proved to be very efficient on SSDs and over NFS mounts, which was the main goal of this project. However, it did not produce a significant improvement for HDDs in most of the testing cases. For this reason, we decided to still use the sequential mode as the default.

Furthermore, some files are currently not eligible for parallel checkout. As earlier described, these are symbolic links and regular files that require external smudge filters, and the limitation exists to prevent race conditions and avoid breaking non-concurrency assumptions from external filters. Because of this, repositories that contain many of such files may not harvest the full potential of parallel checkout. With that said, most repositories probably have a higher number of regular files without filters than any of those two other categories.

Finally, all file removals and directory creations are still performed sequentially, so parallel checkout is not efficient when the operation requires many more of these tasks than file creations.

[9] This value was defined in 2017, to fix a `git fetch` failure. See be4ca29057 ("Increase core.packedGitLimit", 2017-04-20): https://github.com/git/git/commit/be4ca290570f9173db64ea1f925b5b3831c6efed

7.4 Future Work

It would be interesting if Git could choose its default checkout mode based on what would bring the best performance for the machine it is running at. This could help many users on SSD and NFS mounts that might not be aware of the parallel checkout mode otherwise. This "smart default" selection could be performed in many ways. Some alternatives are: 1) Git could try to detect whether the working tree is on an SSD/NFS and auto-enable parallelism in this case; 2) Git could run a micro-benchmark using both checkout modes and decide which is best.

Option 2) is interesting as it does not make any assumptions on the file system or environment, but implementing it at checkout time would probably add a significant overhead to the operation. Specially because the benefit of parallel checkout is only noticeable with a higher number of files.

We tried to work on option 1) for a while, but we quickly found that it is considerably difficult to detect the file system and storage device types in a portable way. To exemplify this, see GNUlib's `read_file_system_list()`[10], which returns a list of mounted file systems, but has to use a very large amount of macros and complex conditional code to be portable.

With that in mind we limited our focus on Linux, which is the system where we had already verified that parallel checkout produces good results for NFS mounts. The auto-detection could then be incrementally expanded to other systems in the future as the community sees fit. However, even after we decided to limit the initial implementation to Linux, we found many challenges when trying to reliably infer the file system type:

- `statfs()`: besides being non-POSIX, the Linux Standard Base project (LSB) marked the library calls to `statfs()` and `fstatfs()` as deprecated and suggested using `statvfs()` and `fstatvfs()` instead.
- `statvfs()`: is POSIX but, unfortunately, does not return the file system type.
- `getmntent()`: the function presents behavior differences among its different libc implementation: the glibc version unescapes the mount paths, whereas musl leaves them as they appear on the mount table. Note that it is not possible to know whether the path is escaped or not just by looking at it. For example, the escaped string "\134" evaluates to "\", but if we receive "\134" without knowing that it is escaped, we could also say that it is the unescaped version of the string "\134134".
- `libmount`: this is an interesting option, but it adds a new dependency to Git, and it would not make much sense to add this overhead to the build process, packagers, and users for a small use in Git.

We discarded these options because of their issues for the Git project and, instead, chose to manually parse the `/etc/mtab` file, which contains information about the currently mounted file systems. Historically, `/etc/mtab` was not the most reliable source of information as it was managed by userland applications (`mount` and `umount`), so it

[10] https://github.com/coreutils/gnulib/blob/dd0af10fa597a95ffe5f4f110ef5edefc2f680bc/lib/mountlist.c

did not work reliably with namespace, containers, and other kernel features. However, Linux procfs system added the `/proc/mounts` file, which is managed by the kernel itself and contains the same information in a more reliable method. Since them, many Linux distros replaced `/etc/mtab` with a symbolic link to `/proc/mounts`. Therefore, reading `/etc/mtab` is a relatively good method regarding reliability and portability, as it will usually be a symlink to the kernel managed `/proc/mounts` file, but still be useable on systems that do not provide this file. In this case, we accept somewhat outdated information without any risks because this is being used only to drive the decision about whether to enable parallel checkout or not. That is, in the seldom case of getting unreliable mount information, we will, at the worst scenario, get a slower checkout; but with no effect on correctness. Additionally, we chose to use `/etc/mtab` only when the libc did not provide the `_PATH_MOUNTED` which, when defined, should evaluate to the path of the mounts file as defined in the library.

Finally, we also copied[11] the `decode_name()` function from glibc's `misc/mntent_r.c` file[12] to unescape the paths found in the mtab file (this was the behavior that was missing in musl's implementation of `getmntent()`).

We sent the prototype patch set[13] to the Git mailing list[14], asking for comments about the general idea or possibly alternative suggestions. Ævar Arnfjörð Bjarmason replied saying that he was not intrinsically opposed to the approach we have taken, as a stopgap, but that we should think of a better alternative for the long run. He mentioned that hardcoding the default value of workers on NFS mounts is probably not a good idea because, being a protocol, there might be some NFS implementation which does not benefit from parallelism in the same way. Instead, he suggested to take a more sustainable approach which would work for other file systems and environments, besides producing a better custom result for each user/machine: using the recently added `git maintenance` command to perform background micro-benchmarks from time to time and automatically adjust performance settings with values optimized for the repository. Besides parallel checkout itself, he mentioned `core.untrackedCache` as another example of setting that could take advantage of such framework. Ævar's suggestion follows the idea we enunciated earlier, but it solves the checkout overhead problem by offloading the actual benchmarks to background processes. Of course this could also produce more noise in the results as the user might be running other processes that are competing for the system resources. Another consideration is discoverability: `git maintenance` is not schedule by default and less proficient users might not know about the performance benchmarks.

We did not proceed with the fs-detection or micro-benchmarking ideas, but these would be interesting options for the future of parallel checkout. Other possibilities for

[11] The original glibc implementation is licensed under LGPLv2.1, and it can be relicensed in Git to GPLv2, as seen in https://www.gnu.org/licenses/gpl-faq.html#AllCompatibility

[12] https://sourceware.org/git/?p=glibc.git;a=blob_plain;f=misc/mntent_r.c;hb=75a193b7611bade31a150dfcc528b973e3d46231

[13] https://github.com/matheustavares/git/compare/8619b75483df916b63f7ff96b6c7b4e462061cc5...matheustavares:2e2c787e2a1742fed8c35dba185b7cd208603de9

[14] https://lore.kernel.org/git/9c999e38-34db-84bb-3a91-ae2a62b964b5@jeffhostetler.com/t/#ma5fcf209d1a99646b565472856633de16501022b

further improvements are:

A **Remove files in parallel.** This might not make a huge difference on local SSDs, but it probably does on NFS mounts. The effect would not be seen on our Linux checkout benchmark, as that operation does not require any file removals, but it would be interesting for branch switches. Specially those with many more removals than creations, like checking out the tag v2.6.15 from v5.8 on Linux, which removes almost four times the number of files it creates.

However, the risk in such optimization is to not handle path collisions properly or forget to reset the `lstat()` cache in the parallel processes. To minimize the danger, it would be better to avoid mixing file creations and deletions on the same parallel operation. The implementation could also focus on the file removals from `unpack_trees()` (i.e. the loop over the CE_WT_REMOVE index entries), although this would miss the removals at `checkout_entry()` itself. But even using these limitations for safety measures, there are still other challenges to overcome: how to deal with submodules, how to handle path collisions when removing files, and how to remove the directories that are left empty after all the files within it get removed (remember that the current code schedule them for removal and check if they can be removed when switching the dirname, but that would have to be adapted for a parallel execution).

B **Create leading directories in parallel.** This also has a great potential for repositories on NFS mounts. We attempted to implement this idea and got promising results, but we did not pursue it further because our implementation required a few changes to the `lstat()` cache code and we were not confident that these would not open spaces for any potential vulnerabilities like the one we had just fixed. In particular, we had to make `checkout_entry()`'s usage of the cache also store "missing component" data (i.e. when the path is found to be missing on disk). This information can turn stale easily and we certainly would not want to risk introducing a bug in this code right after fixing a similar serious problem. Still, it may be possible to implement this optimization without the cache changes. Our use of the "missing component" caching was to remove any existing non-directory file in the dirname of the entry before enqueueing it to allow the workers to create the directories (or proper detect path collisions during checkout when the path is not missing anymore).

C **Avoid duplicated working tree checks.** As mentioned in Section 5.4, there is some duplicated logic in `unpack_trees()` and `checkout_entry()` for the inspection of working tree files regarding their existence and status. Removing this duplication might lead to some improvement, specially on systems where `stat()` is more expensive.

D **Avoid duplicated `lstat()` calls in checkout workers.** To make sure the leading directories created by the main process are still valid (i.e. there was no path collision which made them disappear), the checkout workers must `lstat()` each component before writing a checkout entry. This process does use the `lstat()` cache to minimize the number of calls, but perhaps we can avoid all calls by delaying the checkout

of symbolic links, and only writing them **after** the workers finish. This way, path collisions could be detected by a simple `open()` failure and there would be no risk of wrongly following symlinks.

E **Skip checking working tree on clone.** Although the working tree should be empty on clone, we still need to `lstat()` each path and all their directory components to detect path collisions, avoid following symlinks, and create missing directories. The parallel workers are already able to detect path collisions by themselves. If the Items D and B get implemented, it should be possible to entirely skip the `checkout_entry()` code, on clone, for entries that are eligible for parallel checkout.

F **Do not `stat()` entries after checkout when not needed to**. Currently, the checkout workers always `stat()` the files the have written and send the returned data to the main process. In some cases, like `git checkout-index --prefix`, Git will not update the index after checkout so the `stat()` information is useless. We can avoid the function calls and inter process communication costs by communicating that fact to the worker processes.

G **Work redistribution.** Some index entries may take longer to check out because of larger blobs or slower filtering. In such cases, it could be interesting to implement some kind of work redistribution or scheduling, in order to minimize the waiting time for a potentially slower checkout worker. We tested a scheduling mechanism on parallel checkout but ended up not including it in the final patches because there was no significant performance improvements in our tests. However, there may be cases and repository shapes where it could in fact produce a higher optimization, so it should be interesting to test this further and acess its value.

7.5 Conclusions

We parallelized the checkout machinery in Git, leading to up to 4.5x speedups on NFS and 3.6x on SSDs, in a Linux checkout benchmark with packed objects. HDDs did not benefit as much from parallelism, seeing 1.2x to 2.1x speedups in the same benchmark, but also regressions in other cases. The parallel checkout feature was accepted and merged into the upstream Git project, being released as part of Git 2.32.0 (from Jun 6th, 2021).

The feature was developed in a total of 22 patches, divided into three series. The individual commits and the source code can be seen at:

- Part 1: preparatory API changes:
  https://github.com/git/git/compare/a5828ae6b52137b913b978e16cd2334482eb4c1f...ae22751f9b4bbbebcd0366a48a118b5a575af72d
- Part 2: parallel checkout implementation:
  https://github.com/git/git/compare/a0dda6023ed82b927fa205c474654699a5b07a82...68e66f2987724a639c896e7996ea347be62ef578
- Part 3: parallel checkout tests and extended support to other commands (`git`

`checkout-index` and `git checkout <pathspec>`):
https://github.com/git/git/compare/68e66f2987724a639c896e7996ea347be62ef578...87094fc2daa9613c2fad454dbb068a8f23ce8de8

Working in the checkout machinery and looking for issues with the new parallel framework also allowed us to find two pre-existing bugs related to the `lstat()` cache. One of which could be used to drive RCE attacks during the clone of a repository. Both the bugs were fixed and released to the Git users. More information about them can be found at the Appendix D.

Appendix A

Loose Object Checkout Flamegraphs

Figures A.1, A.2, A.3, and A.4 show the performance profile flamegraphs for the loose object checkout on different machines, as described on Chapter 3.

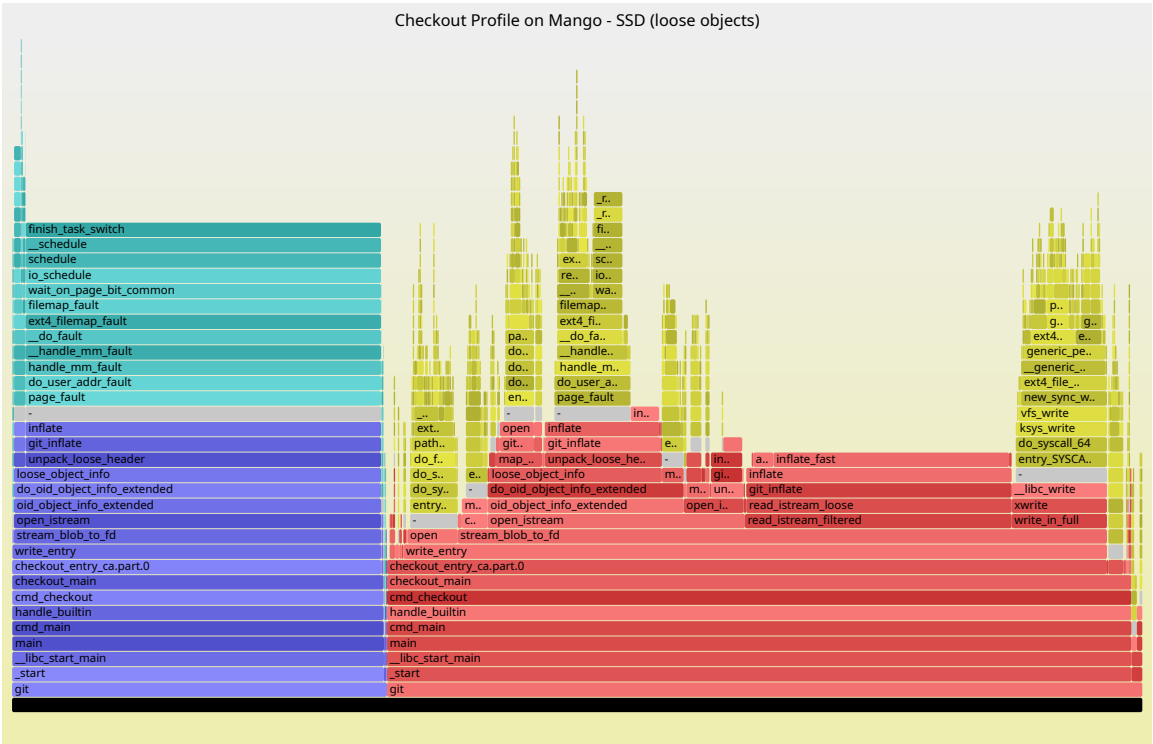


Figure A.1: *Checkout profile flamegraph on machine Mango - SSD (loose objects case).*

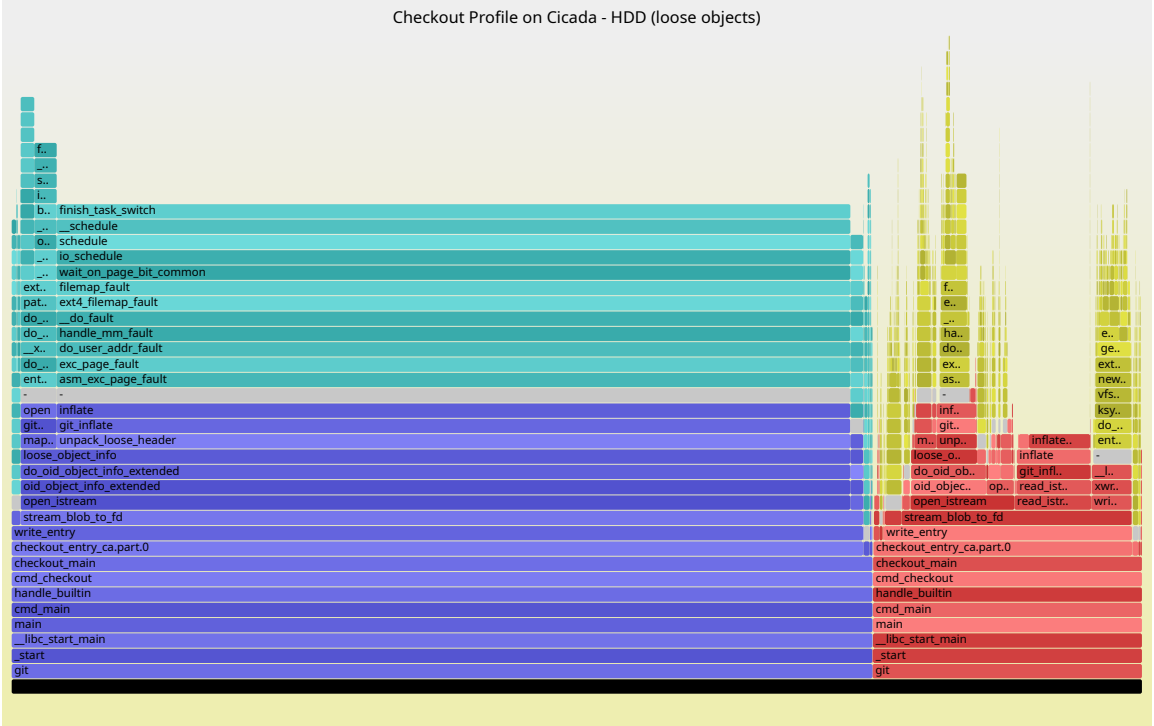


Figure A.2: *Checkout profile flamegraph on machine Cicada - HDD (loose objects case).*

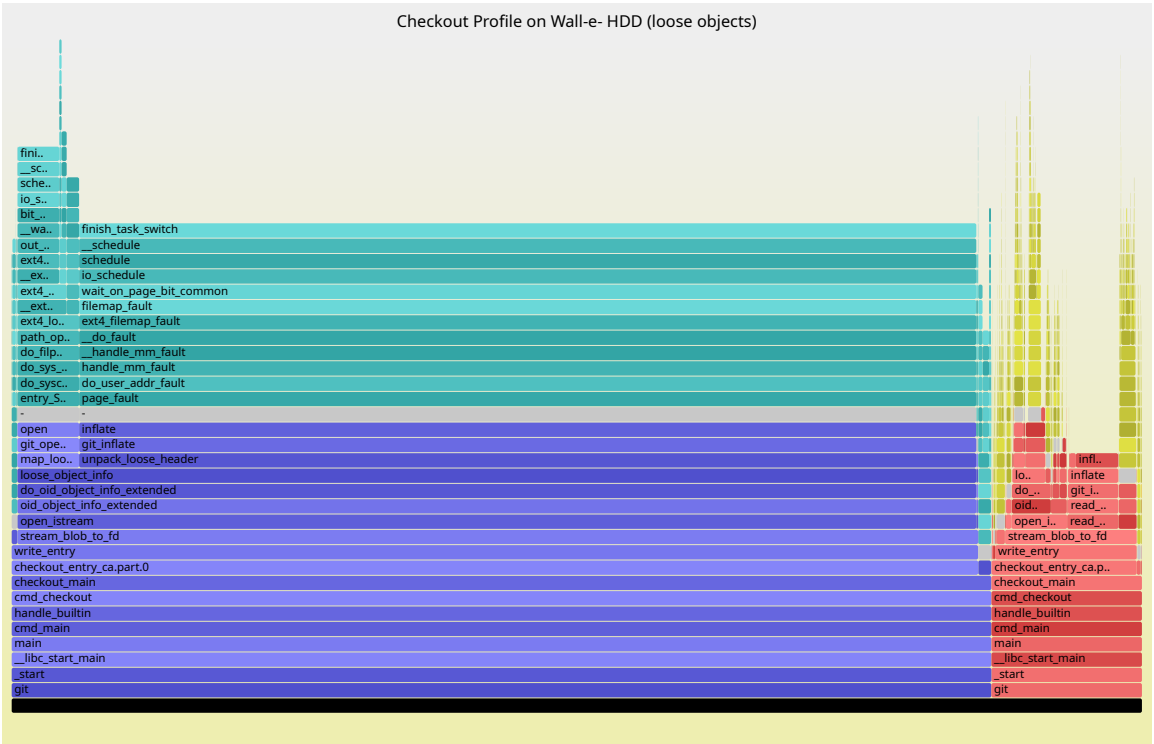


Figure A.3: *Checkout profile flamegraph on machine Wall-e - HDD (loose objects case).*

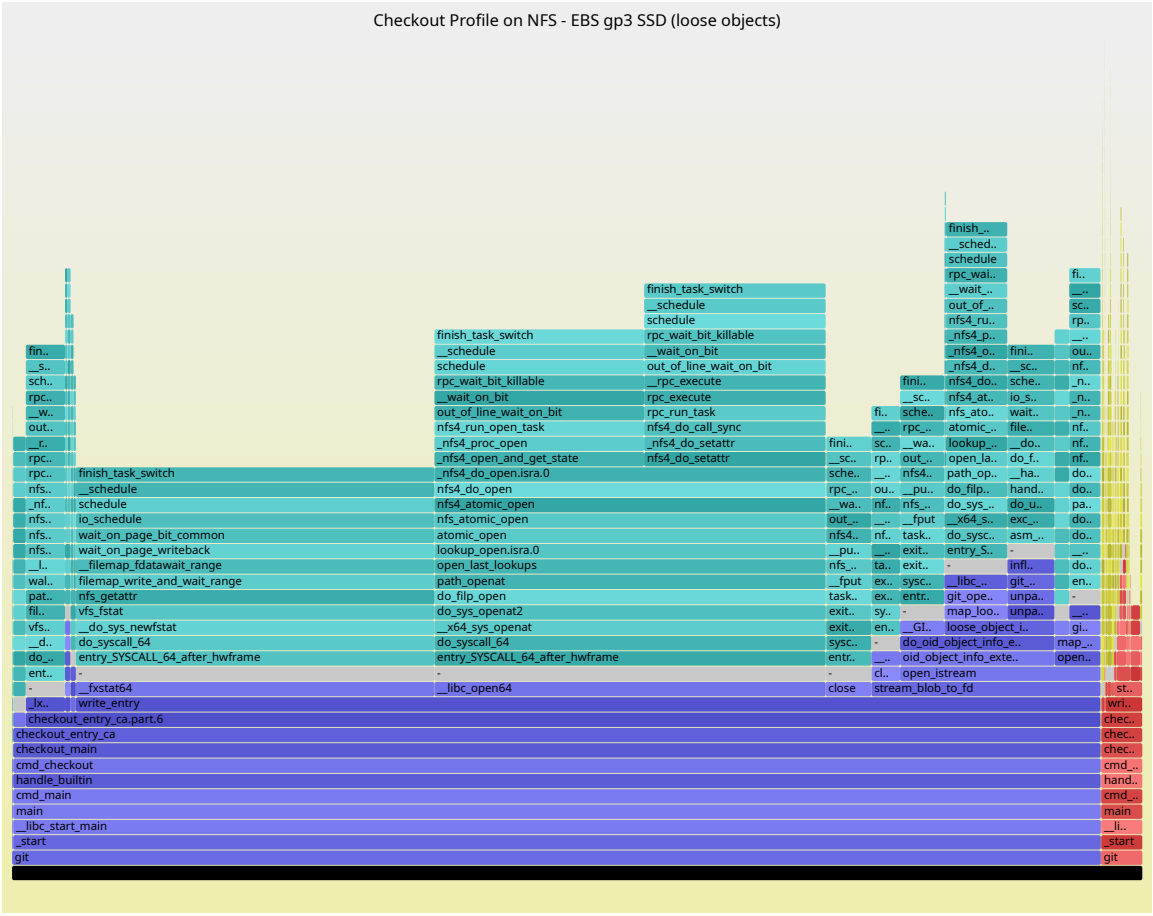


Figure A.4: *Checkout profile flamegraph on NFS - EBS gp3 - SSD (loose objects case).*

Appendix B

The `unpack_trees()` API and Trivial Merges

The `unpack_trees()` machinery only performs "trivial merges", which are tree merges that can be done without looking into the contents of the referenced blobs. For this operation, Git only looks at the hashes of the blobs, the index, and the trees themselves. More complex merging tasks are not handled by the trivial merge routines. For example, the merging of files contents and the resolution of conflicts upon content divergece among multiple trees. Callers of the `unpack_trees()` API can specify which tree merging function they want to use. A few implement their own to achieve specific purposes, but most of them use one of these methods: one-way merge, two-way merge, and three-way merge. There is also the bind merge method, which is used by `git read-tree --prefix` to incorporate a tree into the index at a given prefix.

The one-way merge method is used to replace what is currently in the index with a new tree. This is the case, for example, of `git clone`, `git reset [--hard|--merge]`, and `git checkout -f <branch>`[1]. A one-way merge replaces the index entries while taking care to re-use any valid `stat()` information that was already in the index. Note that one-way merge discards index modifications relative to the current HEAD tree (after all, it is a "one"-way merge, and it only sees the target tree). But it refuses to update any entry whose working tree file is not up to date and entries whose update would cause the loss of untracked data. This can be overruled with the "reset" flag from `struct unpack_trees_options`, which is used e.g. by `git reset --hard` and `git checkout -f <branch>`.

The two-way merge, also referred as "fast-forward", promotes a "switch" from an old tree to a new one. Unlike the one-way method, two-way merges carry forward the changes made in the index relative to the old tree, as long as the entry in the old tree matches the entry in the new one. To exemplify, if the index contains the file "F" with the contents "F data", but "F" is not present in the old tree or the new one, the final index after the merge will still contain "F" with the contents "F data". Furthermore, this will also be

[1] A "`git checkout -f [<branch>] -- pathspecs`" execution does something similar, but it does not use the `unpack_trees()` API.

true if "F" is present in both trees and the contents are not "F data", as long as the entries from the two trees match. An example where this merging routine is used is during branch switching with `git checkout <branch>` (without `-f`). In this case, the two trees that are fed to the two-way merge machinery are the tree at the tip of the previously checked out branch and the one at the target branch's tip. (A `git checkout --orphan=<new-branch-name>` execution, which moves to a new branch with no commits, simply uses an empty tree as target.) Note that clone uses one-way merge because there is no branch previously checked out, and both the index and the working tree should be empty.

Finally, the three-way merge (also called a "true merge") adds an "ancestor" tree to the operation. This merging operation is used by the merge machinery (`git merge`, `git checkout -m`, etc.), and `git read-tree -m` when it receives three or more trees.

Appendix C

Machines Used in Tests

Mango Processor: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Cores: 4
Threads: 8
RAM: 16GiB (2 x 8GiB) SODIMM DDR4 Synchronous 2133 MT/s
SSD: SAMSUNG MZVLB512HAJQ-000L2, PCIe NVMe v1.2

- Linux partition: ext4 (rw,noatime)
- Windows partition: NTFS v3.1

OS: Manjaro Linux, kernel 5.4.123-1-MANJARO and Windows 10.0.19042.1466

Grenoble Processor: Intel(R) Xeon(R) CPU E3-1230 V2 @ 3.30GHz
Cores: 4
Threads: 8
RAM: 32GiB (4 x 8GiB) DIMM DDR3 Synchronous 1333 MT/s
SSD: SanDisk SDSSDA120G, SATA 3.2, ext4 (rw,noatime,nodiratime,discard,errors=remount-ro)
HDD: Seagate Barracuda 7200.14, ST1000DM003-1ER162, 7200 rpm, SATA 3.1, ext4 (rw,relatime)
OS: Debian 10.0, kernel 4.19.0-9-amd64

Songbird Processor: AMD(R) Ryzen(TM) 5 3600 @ 3.60GHz
Cores: 6
Threads: 12
SSD: Corsair Force LE SSD, 163180180001041711 4A, SATA 3.1, ext4 (rw,noatime)
RAM: 16 GiB (2 x 8GiB) DIMM DDR4 Synchronous Unbuffered 2400 MT/s
OS: Arch Linux, kernel 5.13.5-arch1-1

Wall-e Processor: Intel(R) Core(TM) i5-4210U @ 1.70GHz
Cores: 2
Threads: 4
HDD: Seagate Samsung SpinPoint M8, ST1000LM024 HN-M101MBB, 5400 rpm, SATA 3.0, ext4 (rw,noatime)
RAM: 8 GiB (1 x 8GiB) SODIMM DDR3 Synchronous 1600 MT/s

OS: Manjaro Linux, kernel 5.4.123-1-MANJARO

Cicada Processor: Intel(R) Core(TM) i5-3317U @ 1.70GHz
Cores: 2
Threads: 4
HDD: Seagate Laptop HDD, ST500LT012-9WS142, 5400 rpm, SATA 2.6

- Linux partition: ext4 (rw,noatime)
- Windows partition: NTFS v3.1

SSD: Phison SSE020GTTC0-S53 ("Caching SSD" of 20 GiB), KN.0200Q.005m, SATA 2.6, ext4 (rw, noatime)
RAM: 6 GiB (1x 4GiB 1x 2GiB) SODIMM DDR3 Synchronous 1333 MT/s
OS: Manjaro Linux, kernel 5.10.42-1-MANJARO and Windows 10.0.19044.1466

NFS: AWS EBS gp3 NFS v4.1
NFS client on an AWS EC2 c5n.xlarge instance: CPU: Intel(R) Xeon(R) Platinum 8124M @ 3.00GHz (2 cores w/ HT), 10GiB of RAM.
Linux NFS server also on an AWS EC2 instance, using EBS gp3 volume for storage (SDD), XFS file system.

NFS: Cicada NFS v4.1
NFS client on machine "Mango".
Linux NFS server on machine "Cicada", with 32 nfsd threads.
Both machines were connected on LAN through Wi-Fi.

Appendix D

Delayed Checkout / Clone Vulnerability

While working on the parallelization of the checkout machinery, we had to read and modify adjacent code such as the filtering mechanics. In this processes, we found a vulnerability in the delayed checkout feature, which could be used to drive remote code execution (RCE) attacks during a `git clone` invocation. The exploit used path collisions and a misuse of the `lstat()` cache (as described in Section 4.3) to write arbitrary files inside the `hooks` dir, tricking Git to run a malicious script. In this chapter, we will discuss what caused this vulnerability, how we worked with the Git community to fix it, and how this interacted with our roadmap to parallelize checkout.

D.1 Unordered Checkouts and Attack Vector

In Section 4.3, we demonstrated why following the index order during checkout prevents the use of outdated information from the `lstat()` cache. We also explained how dangerous it would be to misuse the cache as the checkout machinery could end up trusting that a path is free of symlinks when it is not, leading to the creation of files at the wrong places. However, as we discovered during this work, the classic checkout machinery already had some problems in this area. Two code paths actually do not follow the index order when checking out entries: the `checkout-index` command, which writes the paths to the working tree at the same order that the user specifies them on the command line[1], and the delayed checkout feature — used by some long-running filter processes like `Git-LFS` — which postpones the checkout of some entries, modifying the checkout order. The second case is the most problematic because it presents a vector for RCE attacks. However, not all Git users are affected by it. The victim must be using a file system which performs some kind of path folding (like case-insensitivity) and supports symbolic links. The user must also have certain smudge filters configured.

Linux typically uses case-sensitive file systems by default, although it is possible

[1] Note, however, that the user must run a forced checkout (with `git checkout-index --force`) in order to see the problematic behavior. Otherwise Git will refuse to replace the colliding files during checkout.

that the user is running Git on external FAT drives or even configured ext4 to be case-insensitive[2]. Mac users are more vulnerable as the default file system installation uses a case-insensitive version of APFS, with support for symlinks. Windows users are also affected, but Git-for-Windows only enables symlink support by default on recent versions of Windows 10 with Developer Mode enabled[3]. If that is not the case, the vulnerability can only be exploited if the user manually turned symlink support on. However, for the cases where it is enabled by default, the chance of a successful attack is further increased as Git-for-Windows already comes with `Git-LFS` configured by default.

The Script D.1 demonstrates how an attacker could use the exploit for an attack. The attacker crafts a malicious repository containing the following files: `.gitattributes`, `A/a`, `A/b`, `A/post-checkout`, and `a`, which is a symbolic link to `.git/hooks`. The files will appear in the given order in the index, but the attacker sets `A/post-checkout` to require filtering by `Git-LFS`. When the victim clones the repository, Git will check out the paths in the index order, starting with the `.gitattributes` and then the `A/*` files. For each path, it first checks whether the leading directory already exists using `has_dirs_only_path()`, and creates the directory if negative. For `A/a`, `has_dirs_only_path()` `lstat()`s `A` and see that it is missing, so Git creates it. For `A/b`, the function `lstat()`s `A` again and this time sees that there is a directory, so this information is cached. Then Git processes the `A/post-checkout` entry, it also checks `A` is a directory, but instead of creating `A/post-checkout` right away, it notices that this entry requires an external smudge filter, so the blob is passed to `Git-LFS`, which replies with a "status=delayed". Git then skips to `a`. It removes the directory `A` and creates the symlink `a`. Now that all entries have been processed, Git iterates through the list of delayed entries, asking the filters for the filtered contents. With the reply from `Git-LFS`, Git finally proceeds to the creation of `A/post-checkout` and, once more, it checks whether `A` is a real directory. As `A` is in the cache, `has_dirs_only_path()` replies that it is indeed a directory (although this is no longer true in the file system), and the post-checkout script is written outside the repository's working tree. After checkout finishes, Git will invoke the post-checkout hook, allowing the attackers code to be executed without the victim's consent.

To fully understand the problem, keep in mind that: 1) the basename of the paths being checked out is **not** saved in the lstat cache, even though the checkout machinery does `lstat()` it outside the caching functions; and 2) the checkout machinery does **not** update the cache when it creates a new directory, it only updates the cache when calling `has_dirs_only_path()`. The second clause is the reason why Script D.1 must create two files inside `A/`: the checkout of the first file will create the missing directory and the second will `lstat()` `A` and save it the cache.

D.2 Considered Alternatives and the Final Patch

When I first encountered this issue with invalid uses of the `lstat()` cache, it was not clear which commands were affected and whether it could really be used to drive an attack. So I emailed the Git security mailing list (a private list), presenting the issue

[2] https://www.collabora.com/news-and-blog/blog/2020/08/27/using-the-linux-kernel-case-insensitive-feature-in-ext4/

[3] https://github.com/git-for-windows/git/wiki/Symbolic-Links

Program D.1 Exploit demonstration for CVE-2021-21300. Adapted from https://www.openwall.com/lists/oss-security/2021/03/09/3

```
#!/bin/sh

# The attacker
git init malicious-repo &&
(
   cd malicious-repo &&

   echo "A/post-checkout filter=lfs" >.gitattributes &&
   mkdir A &&
   touch A/a A/b &&
   printf '#!/bin/sh\n\necho "attack succeeded." >&2\n' >A/post-checkout &&
   chmod +x A/post-checkout &&
   git add . &&
   rm -rf A &&

   # We assume the file system of the attacker supports symlinks, but they
   # could also craft the repository on a fs without symlinks by directly
   # adding `a` to the index with `git hash-object` and `git update-index`
   ln -s .git/hooks a &&
   git add a &&
   git commit -m initial
) &&

# The victim
git clone malicious-repo cloned
```

and asking whether people think it could have security implications. Others developers chimed in and we started to analyze the extent of the problem and how it can affect users. When it became clearer that it could indeed be used to run RCE attacks in some machines, it was suggested to require a CVE® number[4], so this vulnerability was assigned the identification CVE-2021-21300.

The root of the problem was that the `lstat()` cache got outdated during checkout with path collisions, and the unordered writes from the delayed checkout ended up using the outdated information and blindly following symlinks. To fix that, we initially considered the following ideas:

1. **Disable the cache during unordered checkout.** The obvious downside is that this increases the number of `lstat()` calls in `O(number of entries * number of leading directories)`. This could be especially bad for systems where this function is more expensive.

2. **Sort the entries.** This ensures that entries from two different directories (at different levels) do not get written alternately, so we never use any outdated information

[4] As CVE's webpage (https://www.cve.org/About/Overview) describes: "The mission of the CVE® Program is to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities. There is one CVE Record for each vulnerability in the catalog. The vulnerabilities are discovered then assigned and published by organizations from around the world that have partnered with the CVE Program."

from the cache. Some disadvantages of this approach are the extra complexity and the time overhead for sorting the entries. But beyond that, there is no guarantee about the order in which the blobs finish filtering during delayed checkout. So we would have to hold some blobs that were already filtered in memory while waiting for the next in sequence to be ready. Besides the complexity of this change, it could also negatively impact both performance and memory usage.

3. **Detect collisions.** This is certainly a robust approach, but it is also an obviously difficult one. Implementing all the path folding rules for the different file systems would be impracticable to do both correctly and exhaustively. A more plausible way could be to inspect the cached component at the same path level of the last checked out entry (after each entry is written) and make sure it is still a valid directory.

   The idea behind this approach is that only paths with a number of components equal to or smaller than the cached path can cause collisions that might invalidate one of the cached components. Furthermore, only the last component of the checkout path can cause cache invalidations. To visualize that, consider the extreme case where **all** components collide: the cached path can be e.g. `A/B/C/D` and the checkout path `a/b/c`. Although the first two components collide, they will not have their file type changed in the working tree, so only the third and forth components of the cache need to be dropped in this case. More generally, we would `lstat()` the component at the same path level of the written path and, if it is not a directory, erase everything from that point on from the cache.

   One of the advantages of this approach over the two previous ones is that it future-proofs the code against new unordered checkout cases that might appear later. In particular, it would protect the new parallel checkout framework too. On the other hand, this approach produces and additional number of `lstat()` calls proportional to `O(number of entries)`. This might not be that big of a deal in the end, but it is definitely something to keep in mind.

Considering these three options, I sent the first draft of a possible solution to the private mailing list. The patch was composed of 69 line additions and 3 deletions, not including the added regression tests. Its main idea was to implement something along the lines of the third approach described above, but striving to reduce the number of `lstat()` calls on the default ordered checkout case. Since there is no risk of using invalid cache data in this case, the overhead of the additional `lstat()` calls would be completely unnecessary. So the proposed idea was to avoid the calls and unconditionally invalidate the cached components starting from the written path's level downwards. The worst-case scenario would then be an unordered checkout which alternatively writes paths from two vertically distant directories. This could effectively produce the same performance result as not having a cache at all, but its questionable whether this specific case is frequent enough to justify any concerns.

This idea was discussed in the private list and it got some positive feedback regarding correctness, but also some concern as to whether it covered all possible cases. The code to count path components was also not very straightforward. It had to deal with some

considerations like: 1) The path length of two colliding components are not always the same. They are certainly the same for pure case-insensitive collision, but not for other path folding rules, like Unicode normalization. (See for example the two octal representations of "ä": `0303 0244`, and `0141 0314 0210`.) 2) The checkout machinery accepts a prefix string, which it then prepends to each path it is going to check out. This feature is currently only being used by `git checkout-index` and `git difftool`, but in the case of `checkout-index`, the string comes directly from the user without being normalized or converted to an absolute path. So, in this case, the checkout paths may contain artifacts like double slashes.

Johannes Schindelin, the maintainer of Git-for-Windows and one of the Git contributors who also worked on this bug, pointed out that the proposed patch was probably not simple enough for our needs. He explained that, because we where working on this fix under embargo, it was much easier for the review to miss non-obvious issues such as uncovered corner cases or even end up adding new bugs and vulnerabilities. Therefore, we should approach this problem with a simpler and more straightforward fix, seeking the best possible guarantee that it completely eliminates the bug without introducing new ones. So, following these requirements, Johannes sent a draft for an easier-to-review and yet quite robust solution: The vulnerability can only be exploited if a cached component is replaced by another file; and this can only happen after the said component gets removed from the working tree. So we invalidate the cache whenever we remove a file in the checkout code.

Regarding performance, some benchmarks showed that the change significantly increased the number of `lstat()` calls due to some unnecessary cache invalidations. However, we easily mitigated that by only dropping the cache on directory removals, which is the only type of file for which a removal can turn the cache stale on checkout. This solution was already enough to fix the RCE-on-clone attack that we described earlier, and it indeed passed our newly added tests. However, with further inspection, we noticed that some users of the checkout machinery perform their own tasks in between calls to `checkout_entry()`. If some of these tasks included directory removals, the cache could became outdated again and end up producing errors on subsequent `checkout_entry()` calls. So, in the interest of covering the maximum extent possible with our fix and also protecting us against similar-but-not-identical attack vectors, Johannes suggested that we intercept all Git calls to `rmdir()` with a wrapper, which would invalidate the cache at each successful removal. This seemed like the best approach so far in order to keep using the cache in a safe manner.

To mitigate performance impacts, he suggested using `strcasecmp()` to decide when to drop the cache based on whether the removed directory and the cached directory were the same ignoring case differences. However, this unfortunately would not work for more general path folding rules that do not rely exclusively on case sensitivity. We also considered employing the previous idea of comparing the path levels and only dropping the components which could potentially be affected by a collision with the removed directory, but this approach would also have its issues: `rmdir()` may receive relative paths or paths containing double slashes and backslashes. If we were to do any path comparison among the argument of `rmdir()` and the cached path we would have to canonize both of them first. There is no simple and easy way to direct compare them. Therefore, we decided to

stick with the original idea of invalidating the cache on **all** successful directory removals. At this point, we also revisited some of the benchmarks we had run earlier to see whether the current solution would bring any issues performance-wise.

From the checkout operations we had previously tested, a switch from tag v5.8 to v2.6.20 on the Linux repository was one of leaders in terms of the number of `lstat()` calls. So we used it as our benchmark to test the performance of the bug fix. The results, presented in Table D.1, showed that the performance overhead produced by the change was in fact quite small.

	Linux (SSD)	Windows 10 (SSD)	NFS v4.1
Original	5.879 s ± 0.108 s	31.776 s ± 0.531 s	124.673 s ± 3.032 s
Patched	5.911 s ± 0.099 s	32.516 s ± 0.596 s	126.396 s ± 2.896 s

Table D.1: *Time comparison between original checkout and patched version with the RCE-on-clone fix. Mean times and standard deviations for 15 repetitions of a "branch" switch from tag v2.6.20 to tag v5.8 on the Linux repository.*

As the patch was getting into a mature state, we continued inspecting the code and looking for corner cases or additional issues we had not seen yet. This lead us to another finding: the checkout machinery might spawn child processes to check out submodules, and those process can also remove directories. Since the `rmdir()` wrapper will only clean the lstat cache from the calling process, this can make the main process' lstat cache go stale. With further inspection, we ended up founding a case of path collision between a submodule and a regular file which could indeed lead to an invalid cache state and a subsequent file creation at a wrong place[5]. However, this problem was quite straightforward to solve: for each submodule that has to be checked out, the main Git process waits for its spawned child to finish before it continues to the next index entry. Furthermore, the child processes start with a clean lstat cache, so they cannot carry any possibly invalid state from the main process. Therefore, to fix the issue, we just had to invalidate the lstat cache whenever a subprocess finishes. Note that, unlike the `rmdir()` wrapper which only drops the cache when the operation was successful, this time we invalidate the cache regardless of the exit status from the child process, as it can still have removed directories even if it later failed and returned an error code.

We also had to consider parallel checkout itself. However, as we discussed in Section 6.2.2, our parallel workers could not make the main process `lstat()` cache stale as neither they nor the main process remove any directory for the whole duration of the parallel execution. The workers' caches also cannot be find at a stale state for the same reason. Besides, the workers start with a clean cache (just like the child processes created to check out submodules), so no previous collision during the sequential phase could affect their caches. The only case we had to be more cautious about was the unordered checkout of entries by the main process itself. That is, either when it decides to fallback to sequential checkout and has to sequentially write the entries that were enqueued for parallel checkout, or when it has to process entries which collided during parallel checkout. In these

[5] See this test case: https://github.com/git/git/blob/0d58fef58a6f382ba1d35f47a01cb55d8976335f/t/t0021-conversion.sh#L865

cases, the main process could break the ordered checkout premise while having a stale `lstat()` cache. However, the general solution proposed here (i.e. resetting the cache on `rmdir()` and after a child process finishes) already cover these two cases as well.

Finally, as Git commands may perform their own tasks before using the checkout machinery, we decided to invalidate the cache in `unpack_trees()`, as well, before it starts to check out the index entries. This ensure that the respective code starts with a clean (and, thus, valid) cache. Putting it all together, the final version of the vulnerability fix was composed of three patches:

- **checkout: fix bug that makes checkout follow symlinks in leading path**
  https://github.com/git/git/commit/684dd4c2b414bcf648505e74498a608f28de4592
- **run-command: invalidate lstat cache after a command finished**
  https://github.com/git/git/commit/0d58fef58a6f382ba1d35f47a01cb55d8976335f
- **unpack_trees(): start with a fresh lstat cache**
  https://github.com/git/git/commit/22539ec3b5e678c054ab361a37a7cdcc64ca1228

On March 9th, the Git project released[6] the maintenance version v2.30.2 containing these three patches, together with releases for older maintenance tracks from v2.17.6 to v2.29.3. A security advisory was also published on Git's fork at GitHub[7] and GitHub itself published a post[8] in its own blog announcing the release and further instructions on how to reduce the risk of an attack on machines that could not be upgraded immediately.

D.3 Checkout Following Symlinks on File Removal

While working on the delayed checkout vulnerability, I found another bug related to an erroneous use of the lstat cache on checkout. This did not seem as serious as the first one regarding security, but I sent it to the private mailing list first, just to be sure. Johannes Schindelin, who also worked on the fix for the delayed checkout vulnerability, replied confirming that he did not think the bug was exploitable for an attack, so we could hold off the patch until the security fix was released and then send it to the public mailing list for review.

The problem existed since 2011, when the returning code for one of the functions accessing the `lstat()` cache changed but one of its callers in the checkout code was not adjusted to handle this change. The checkout function is `unlink_entry()`, responsible for removing tracked files that were part of the index but should no longer be. As described in Section 2.5.1, this function first checks whether the leading directories of the file are real directories, then remove the file. The first part is done using `check_leading_path()`, which uses the `lstat()` cache to store three types of data: symlinks, directories, and "no

[6] https://lore.kernel.org/git/xmqqim6019yd.fsf@gitster.c.googlers.com/

[7] https://github.com/git/git/security/advisories/GHSA-8prw-h3cq-mghm

[8] https://github.blog/2021-03-09-git-clone-vulnerability-announced/

entry". Regarding the latter, if the cached path at a given moment is `A/B/C`, and the tracking flags say this path is a "no entry", then we know `A` and `A/B` are both real directories, but `A/B/C` is missing on the file system.

Until 2011, `check_leading_path()` used to return: 0 if the given path had a symlink or missing component at its dirname; -1 if all leading components were real existing directories; or the length of a component if it was found to exist but neither as a directory nor a symlink. However, the patch `1d718a5108 (``do not overwrite untracked symlinks'', 2011-02-20)` adjusted this function to only return 0 for "missing entries", so the symlink case was combined with the third case, i.e. the return code being a path length. Since `unlink_entry()` was not adjusted for this change, it started to follow symlinks when removing the entries instead of aborting. This only happens, of course, for symlinks that are in place of the leading directory of a tracked path, and only during forced checkouts. Script D.2 showcases how this bug could be triggered (before Git v2.32.0, when it was fixed).

The fix for this bug was in fact quite simple: we only had to adjust `unlink_entry()` for the new return code of its changed callee while taking care to properly display any necessary warning messages for the users. The patch was sent in March 2021, after the delayed checkout fix was out, and it was merged [9] and released as part of Git v2.32.0, together with another patch [10] to update outdated comments in the same part of the code.

Program D.2 Bug demonstration: checkout following symlinks when removing tracked entries (fixed in Git v2.32.0).

```
1   #!/bin/sh
2
3   git init test-repo &&
4   (
5      cd test-repo &&
6
7      mkdir dir &&
8      touch dir/file &&
9      git add dir/file &&
10     git commit -m "add dir/file" &&
11     mv dir untracked-dir &&
12     ln -s untracked-dir dir &&
13     git checkout -f HEAD~
14     # Note that untracked-dir/file got remove
15  )
```

[9] https://github.com/git/git/commit/fab78a0c3defddff87ea5aa7dd32c5e444c43f1f

[10] https://github.com/git/git/commit/462b4e8dfd688b8964da77daf17b64da5bdc54ad

Appendix E

Additional Benchmarks

At Chapter 7, we saw how parallel checkout performs on a full checkout of the Linux repository (for local file systems on SSD and HDD), and the Git repository (for NFS). As mentioned earlier, this is an interesting operation for us because it involves many file creations, which is the main focus of the parallel checkout feature at the moment. However, it is more common for day-to-day checkout operations to also require file removals and other non-parallelized tasks such as tree merging and index manipulations. So, in this chapter, we will take a look at parallel checkout performance numbers for a few other operations that use some of these mechanics.

E.1 Local File System

For the local HDD and SSD tests, we executed five checkout-related operations in the Linux repository, with different workloads. Because we needed different Linux tags, we only used the repository with the packed objects on these tests. The selected operations were:

- **Checkout I:** a switch from tag `v4.0` to `v5.12`.
- **Checkout II:** a switch from tag `v5.12` to `v4.0`.
- **Checkout III:** a switch from tag `v5.12` to `v5.11`.
- **Checkout IV:** a switch from tag `v5.12-rc7` to `v5.12-rc8`.
- **Stash:** a `git stash push --include-untracked` execution, after moving Linux "`Documentation`" directory to "`docs`".

To better explain the last operation, a `git stash push` stashes away the changes from a dirty working tree (without affecting the current HEAD) and then resets the files to their original contents. By default the operation is limited to tracked files, but with the `--include-untracked` option, we instruct Git to store (and clean) the untracked files as well. For the purposes of this work, we can think of this operation as three different phases: we start by creating the objects for the dirty files and updating the stash reference; then `git clean` is called to remove the untracked files; and finally, `git reset` is called to

restore the tracked files that were dirty. Only this last part is actually able to use parallel checkout at the moment. Furthermore, to be consistent, we removed the objects created by `git stash` after each execution, so that the next one would not be able to reuse those objects and skip the creation.

Table E.1 shows what is the difference in the working tree state before and after each one of the benchmarked operations, regarding the number of created, removed, and modified files. These numbers were collected with `git diff-tree --name-status -r <from> <to>`, except for the stash command, which we used `git ls-files Documentation | wc -l`. Note that these numbers only include regular files and symlinks, not the leading directories that Git had to create/remove as well. We also do not include additional files handled during the operations, like the object files created on stash. Finally, remember that the checkout machinery handles "modified files" in two steps: first it removes the old file, then it checks out the new one. So we could further summarize this table by removing the "modified" row and summing it onto the other two.

	Checkout I	Checkout II	Checkout III	Checkout IV	Stash
C	36524 (43.68%)	13980 (16.72%)	721 (5.96%)	0 (00.0%)	7631 (50.00%)
M	33119 (39.61%)	33119 (39.61%)	10438 (86.24%)	159 (100.00%)	0 (00.0%)
R	13980 (16.72%)	36524 (43.68%)	945 (7.81%)	0 (00.0%)	7631 (50.00%)
Sum	83623	83623	12104	159	15262

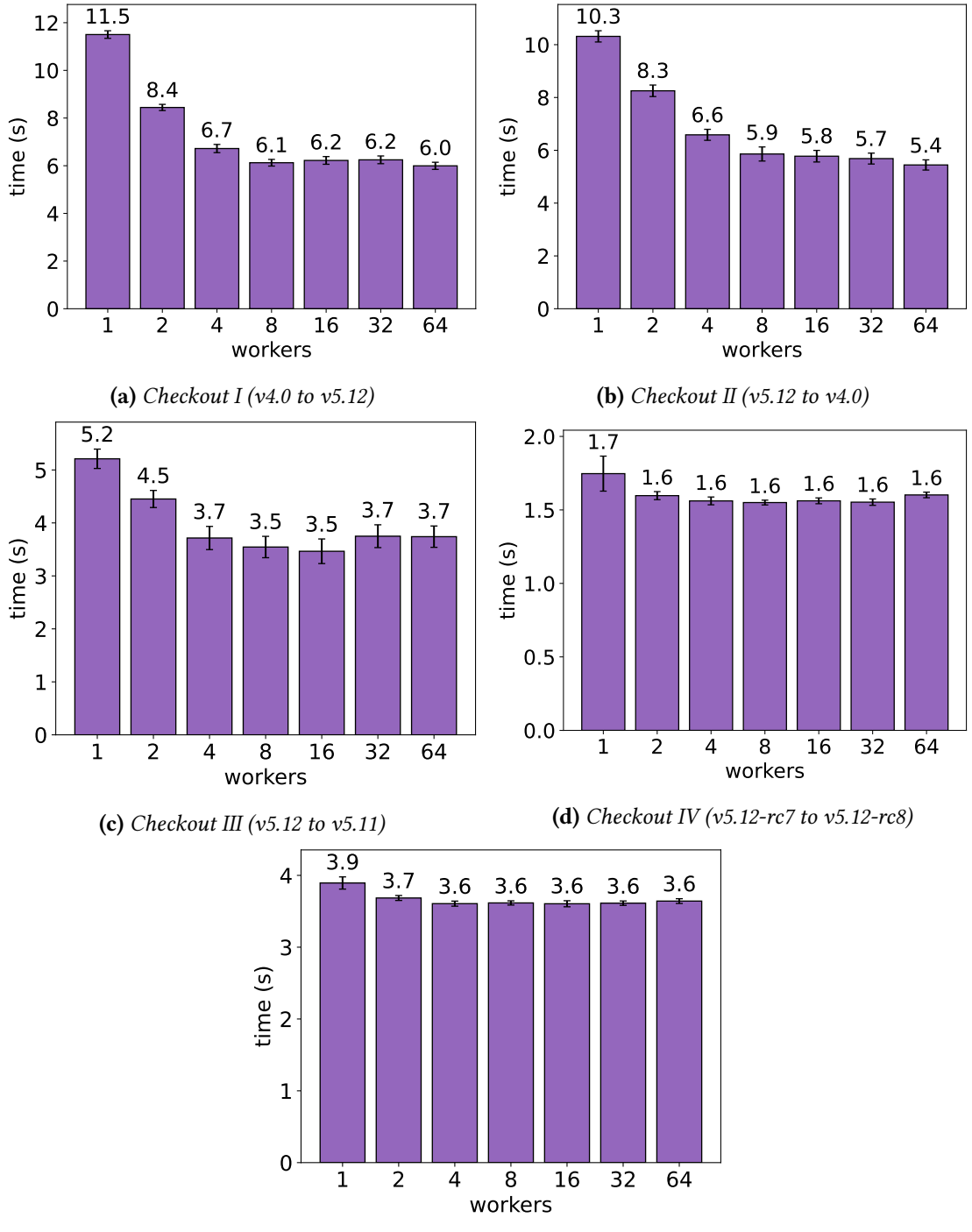
Table E.1: *Differences produced in the Linux working tree by each of the benchmarked operations (in number of files). C, M, and R, respectively correspond to "created files", "modified files", and "removed files".*

Figure E.1, shows the results on machine Mango (SSD) and Figure E.2 shows the results on machine Cicada (HDD), both running Linux. All values correspond to the average run time and confidence interval for a set of 15 samples using a confidence level of 95%. On the SSD, parallel checkout was able to reduce the run time for all benchmarked operations (Figure E.1). However, the most significant speedups only appear on the operations with higher file creation workloads, which is expected. The best results came from Checkout I (Figure E.1a), with a speedup just above 1.9x. On the HDD, parallel checkout only performed marginally better than the sequential mode on Checkout III (Figure E.2c) and Checkout IV (Figure E.2d). But its overall performance was slower than sequential checkout on spinning disks.

E.2 Network File System

For the NFS tests, we once again used the Git repository as testing data because the previous benchmarks using Linux were taking too long to execute. So we selected five new operations to benchmark, trying to match the workloads from the ones we used on the local file system tests:

- **Checkout I-git:** a switch from tag `v1.9.0` to `v2.32.0`.
- **Checkout II-git:** a switch from tag `v2.32.0` to `v1.9.0`.



(a) *Checkout I (v4.0 to v5.12)*

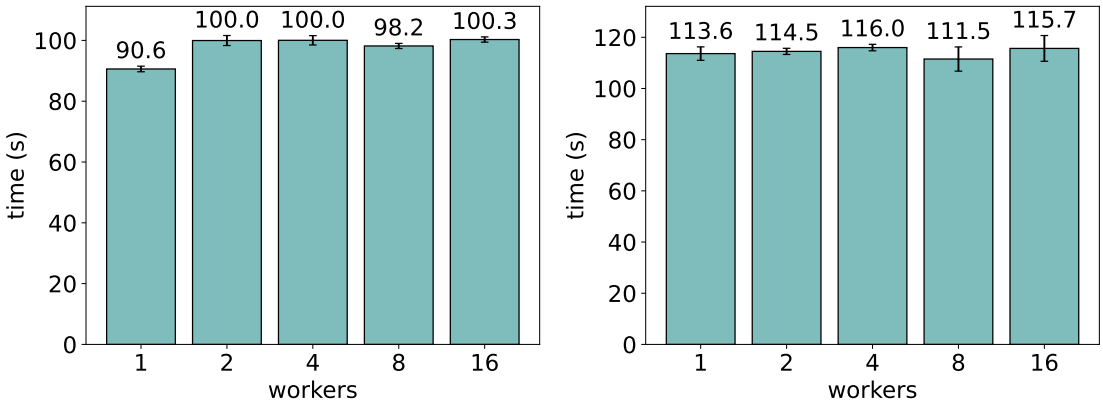
(b) *Checkout II (v5.12 to v4.0)*

(c) *Checkout III (v5.12 to v5.11)*

(d) *Checkout IV (v5.12-rc7 to v5.12-rc8)*

(e) *Stash*

Figure E.1: *Additional benchmarks on machine Mango - SSD.*



(a) *Checkout I (v4.0 to v5.12)*

(b) *Checkout II (v5.12 to v4.0)*

(c) *Checkout III (v5.12 to v5.11)*

(d) *Checkout IV (v5.12-rc7 to v5.12-rc8)*

(e) *Stash*

Figure E.2: *Additional benchmarks on machine Cicada - HDD.*

- **Checkout III-git:** a switch from tag `v2.32.0` to `v2.31.0`.
- **Checkout IV-git:** a switch from tag `v2.32.0-rc0` to `v2.32.0-rc3`.
- **Stash-git:** a `git stash push --include-untracked` execution, after moving Git "`Documentation/RelNotes`" directory to "`Documentation/release-notes`".

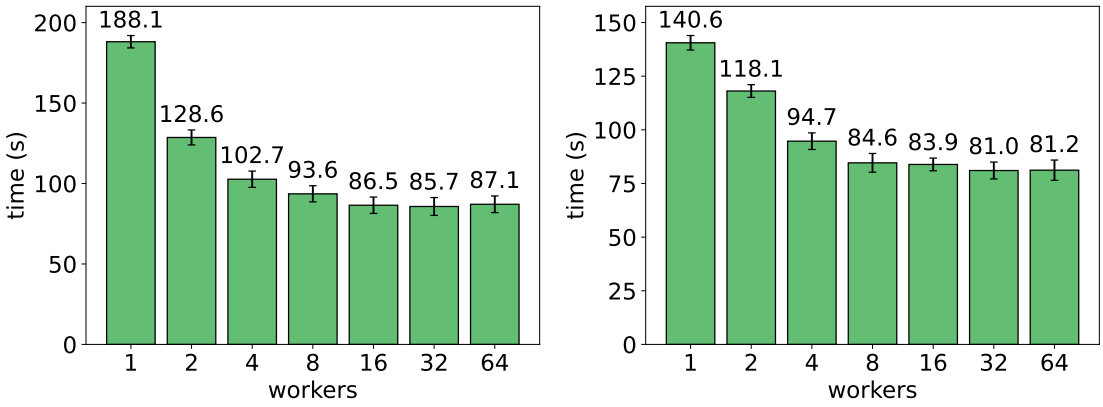
Table E.2 shows the differences produced in the Git working tree for each of these operations. Since not all of these operations meet the minimum number of files to trigger parallel checkout (i.e. 100), we ran these tests with `checkout.thresholdForParalellism` set to zero. (We set the threshold to zero on the local benchmarks as well, although the workloads there were already above the default threshold.)

The results on the NFS Cicada setup, using both the SSD and the HDD, are respectively show at Figures E.3 and E.4. Once again, these values show the average run time and confidence interval for 15 samples, using a confidence level of 95%. Like we observed on the local SSD tests, from Figure E.1, parallel checkout was able to reduce the run time for all benchmarked operations on the NFS. However, the more significant improvements are seen at the first tree operations, which have higher file creation workloads. On the SSD-based NFS setup, the best result comes from Checkout I (Figure E.3a), with a speedup of 2.19x. On the HDD-based NFS setup, the best result comes from checkout III (Figure E.4c), with a speedup of 1.99x — just below checkout I (Figure E.4a), with its 1.98x speedup.

Based on some quick experiments and prototypes, we believe that both NFS setups could see even higher speedups on these operations if we were to perform file removals and directory creations in parallel as well. However, this task involves extra complexity and it comes with its risks. We would have to be extra careful in coordinating the removals and creations (i.e. possibly performing them in two separate phases), to avoid race conditions in case of path collisions. Furthermore, files can be removed in different places inside the checkout machinery, so to support them all, some amount of refactoring and/or a work queue implementation would be required. Finally, the directory creation step uses the `lstat()` cache. Any changes in it should be done with plenty of care to avoid re-introducing the vulnerability we had found during this work, or other similar bugs. For these reasons, we left these two additional optimizations out of the main parallel checkout series.

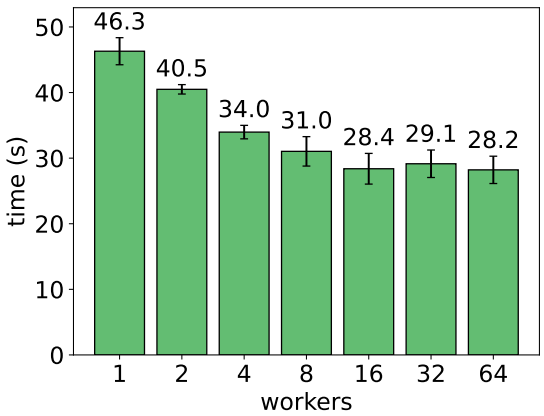
	Checkout I-git	Checkout II-git	Checkout III-git	Checkout IV-git	Stash-git
C	1609 (47.59%)	283 (8.37%)	447 (86.29%)	0 (0.00%)	410 (50.00%)
M	1489 (44.04%)	1489 (44.04%)	64 (12.36%)	57 (100.00%)	0 (0.00%)
R	283 (8.37%)	1609 (47.59%)	7 (1.35%)	0 (0.00%)	410 (50.00%)
Sum	3381	3381	518	57	820

Table E.2: *Differences produced in the Linux working tree by each of the benchmarked operations (in number of files). C, M, and R, respectively correspond to "created files", "modified files", and "removed files".*

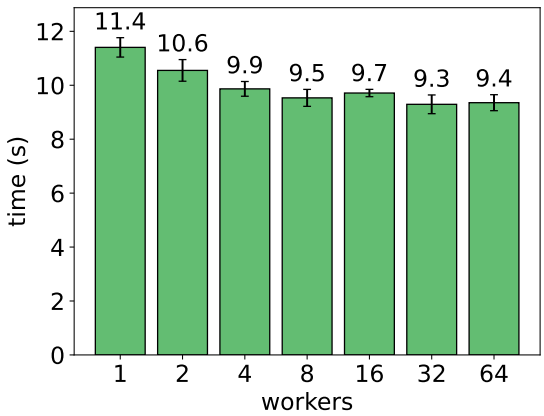


(a) *Checkout I-git (v1.9.0 to v2.32.0)*

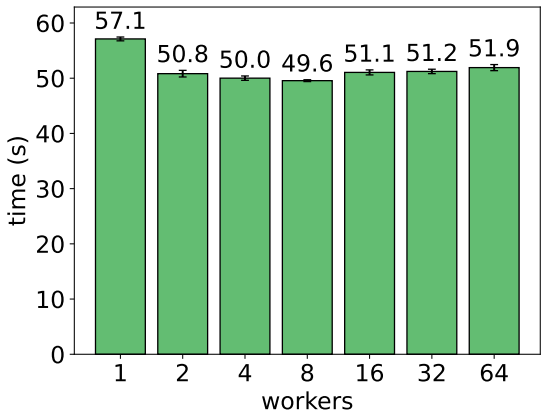
(b) *Checkout II-git (v2.32.0 to v1.9.0)*



(c) *Checkout III-git (v2.32.0 to v2.31.0)*

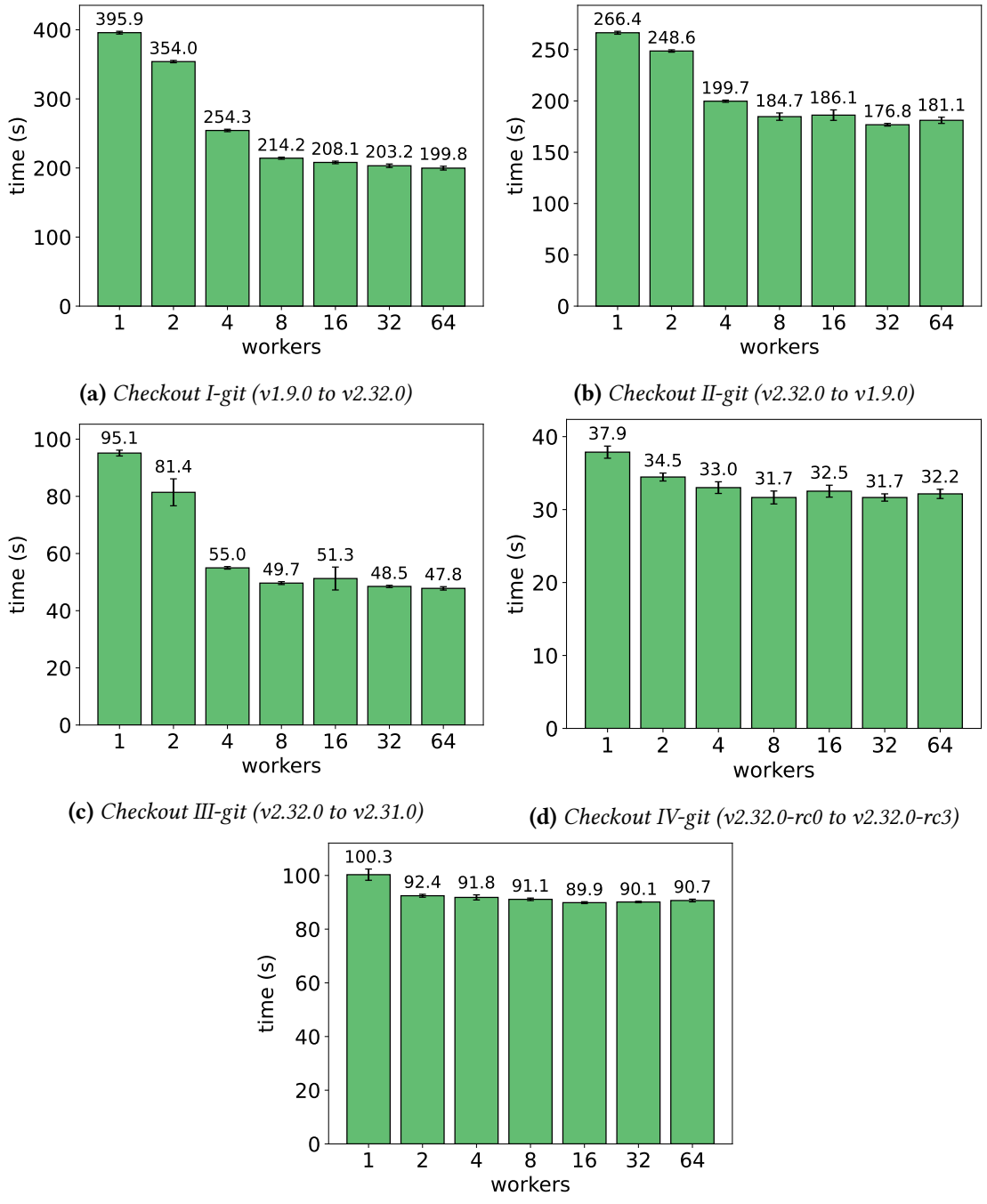


(d) *Checkout IV-git (v2.32.0-rc0 to v2.32.0-rc3)*



(e) *Stash (git repo)*

Figure E.3: *Additional benchmarks on NFS Cicada - SSD.*



(a) *Checkout I-git (v1.9.0 to v2.32.0)*

(b) *Checkout II-git (v2.32.0 to v1.9.0)*

(c) *Checkout III-git (v2.32.0 to v2.31.0)*

(d) *Checkout IV-git (v2.32.0-rc0 to v2.32.0-rc3)*

(e) *Stash (git repo)*

Figure E.4: *Additional benchmarks on NFS Cicada - HDD.*

E.3 Addendum for Cicada's Caching SSD

At Chapter 7, we mentioned that machine Cicada has a caching SSD which we took advantage of for the NFS tests, despite the fact that is was not designed for general storage. The more curious reader might also be interested in seeing how parallel checkout behaves in this SSD when it is used as a local storage. So we repeated the `git checkout .` benchmark on machine Cicada, using the caching SSD as a local ext4 file system. Results are shown in Figure E.5. This time we took 15 samples for the packed objects checkout, but 45 samples for the loose objects checkout (in the hopes that it would reduce the variance observed in this case).

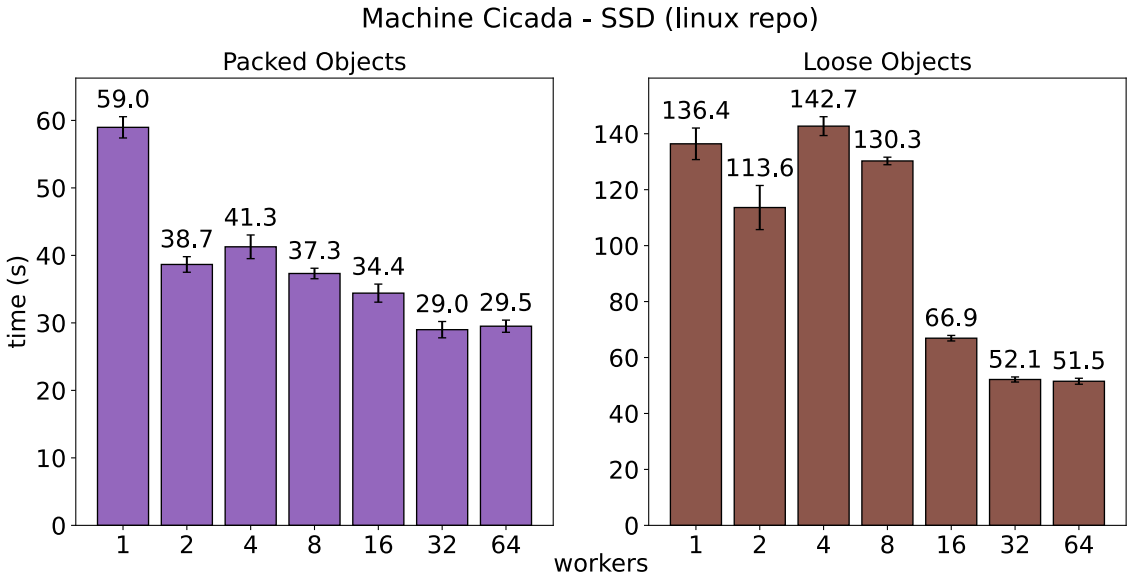


Figure E.5: *Checkout benchmark on machine Cicada - "Caching" SSD*

As the figure shows, parallel checkout was still able to reduce the run time in this caching SSD, but the overall behavior was quite different from what we saw in the other SSDs at Chapter 7. Besides the higher variance, we see a weird elevation around 4~8 workers. Nevertheless, we must highlight once again, that we are stretching the functionality of this SSD by using it in a way it was not designed for. Thus, this results are shown merely for the sake of curiosity and completeness.

References

[Apache 2022] Apache. *Apache Subversion*. URL: https://subversion.apache.org/ (visited on 03/10/2022) (cit. on p. 2).

[Beller 2016] Stefan Beller. *Email "Re: Parallel checkout (Was Re: 0 bot for Git)"*. 2016. URL: https://lore.kernel.org/git/CAGZ79kZP_TUUk3vWHe=c301n66FtQpnwPfPmJ6oD8n-Zz-SVyg@mail.gmail.com/ (visited on 03/09/2022) (cit. on p. 39).

[Bernardino *et al.* 2020] Matheus Tavares Bernardino, Giuliano Belinassi, Paulo Meirelles, Eduardo Guerra, and Alfredo Goldman. "Improving parallelism in git and gcc: strategies, difficulties, and lessons learned". In: *IEEE Software* (2020). DOI: 10.1109/MS.2020.3020932 (cit. on p. 32).

[Boito *et al.* 2018] Francieli Zanon Boito *et al.* "A checkpoint of research on parallel i/o for high-performance computing". In: *ACM Comput. Surv.* 51.2 (Mar. 2018). ISSN: 0360-0300. DOI: 10.1145/3152891. URL: https://doi.org/10.1145/3152891 (cit. on p. 26).

[Chacon and Straub 2014] Scott Chacon and Ben Straub. *Pro Git*. Apress, Nov. 2014 (cit. on pp. 1, 2).

[Chen, Hou, *et al.* 2016] Feng Chen, Binbing Hou, and Rubao Lee. "Internal parallelism of flash memory-based solid-state drives". In: *ACM Trans. Storage* 12.3 (May 2016). ISSN: 1553-3077. DOI: 10.1145/2818376. URL: https://doi.org/10.1145/2818376 (cit. on pp. 24, 38, 39).

[Chen, Lee, *et al.* 2011] Feng Chen, Rubao Lee, and Xiaodong Zhang. "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing". In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 2011, pp. 266–277. DOI: 10.1109/HPCA.2011.5749735 (cit. on p. 24).

[Corbet 2022] Jonathan Corbet. *ELC: How much memory are applications really using?* URL: https://lwn.net/Articles/230975/ (visited on 03/10/2022) (cit. on p. 81).

[CVS 2022] CVS. *CVS - Open Source Version Control*. URL: https://www.nongnu.org/cvs (visited on 03/10/2022) (cit. on p. 2).

[Duy 2016a] Nguyễn Thái Ngọc Duy. *Email "Parallel checkout (Was Re: 0 bot for Git)"*. 2016. URL: https://lore.kernel.org/git/20160415095139.GA3985@lanh/ (visited on 03/09/2022) (cit. on p. 42).

[Duy 2016b] Nguyễn Thái Ngọc Duy. *Email "Re: Parallel checkout (Was Re: 0 bot for Git)"*. 2016. URL: https://lore.kernel.org/git/CACsJy8BruDfmGvA=q+BW61ZKKsTjrF96VzpujEJidm=OtC0_Rg@mail.gmail.com/ (visited on 03/09/2022) (cit. on p. 40).

[Duy 2016c] Nguyễn Thái Ngọc Duy. *Email "Re: Parallel checkout (Was Re: 0 bot for Git)"*. 2016. URL: https://lore.kernel.org/git/CACsJy8Ab=q0mbdcXn9O7=dKHaOuhUCNk4g6BU5kZHdPM+z7yng@mail.gmail.com/ (visited on 03/09/2022) (cit. on p. 44).

[Duy 2016d] Nguyễn Thái Ngọc Duy. *Parallel checkout patches at Duy's fork of git on GitLab*. 2016. URL: https://gitlab.com/pclouds/git/-/commits/parallel-checkout (visited on 03/09/2022) (cit. on p. 42).

[Ghodsnia *et al.* 2014] Pedram Ghodsnia, Ivan T. Bowman, and Anisoara Nica. "Parallel i/o aware query optimization". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 349–360. ISBN: 9781450323765. DOI: 10.1145/2588555.2595635. URL: https://doi.org/10.1145/2588555.2595635 (cit. on p. 38).

[Git Development Community 2021a] Git Development Community. *Git index format*. 2021. URL: https://github.com/git/git/blob/v2.32.0/Documentation/technical/index-format.txt (cit. on p. 35).

[Git Development Community 2021b] Git Development Community. *Git parallel-checkout technical document*. 2021. URL: https://github.com/git/git/blob/v2.32.0/Documentation/technical/parallel-checkout.txt (cit. on pp. 62–64).

[Git Development Community 2021c] Git Development Community. *Git protocol common document: pkt-line specifications*. 2021. URL: https://github.com/git/git/blob/v2.32.0/Documentation/technical/protocol-common.txt#L51 (cit. on p. 43).

[Git Development Community 2021d] Git Development Community. *Trivial merge rules*. 2021. URL: https://github.com/git/git/blob/v2.32.0/Documentation/technical/trivial-merge.txt (cit. on p. 13).

[Git Development Community 2022] Git Development Community. *Git SCM*. URL: https://git-scm.com/ (visited on 03/10/2022) (cit. on p. 2).

[Hamano 2006] Junio C. Hamano. "Git - a stupid content tracker". In: *Proceedings of the Linux Symposium - Volume One*. 2006. URL: https://www.kernel.org/doc/mirror/ols2006v1.pdf (cit. on p. 2).

[Harry 2017] Brian Harry. *The largest Git repo on the planet*. 2017. URL: https://devblogs.microsoft.com/bharry/the-largest-git-repo-on-the-planet/ (visited on 01/24/2022) (cit. on pp. 3, 84).

[Hostetler 2020] Jeff Hostetler. *Pull request "[RFC] Parallel checkout"*. 2020. URL: https://github.com/gitgitgadget/git/pull/628 (visited on 03/09/2022) (cit. on p. 44).

[King 2018] Jeff King. *Email "Re: [PATCH v1 0/3] [RFC] Speeding up checkout (and merge, rebase, etc)"*. 2018. URL: https://lore.kernel.org/git/20180718213420.GA17291@sigill.intra.peff.net/ (visited on 03/09/2022) (cit. on p. 55).

[Linux Development Community 2021] Linux Development Community. *CPU hotplug in the Kernel*. 2021. URL: https://www.kernel.org/doc/html/latest/core-api/cpu_hotplug.html (visited on 05/03/2022) (cit. on p. 77).

[Mercurial 2022] Mercurial. *Mercurial SCM*. URL: https://www.mercurial-scm.org/ (visited on 03/10/2022) (cit. on p. 2).

[Microsoft 2021] Microsoft. *[MS-FSCC]: Microsoft File System Control Codes*. June 2021. Chap. 2.1.5.2.1. URL: https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-fscc/18e63b13-ba43-4f5f-a5b7-11e871b71f14 (cit. on p. 33).

[Nurlisa *et al.* 2018] Nazatul Nurlisa, Zolkifli, Amir Ngah, and Aziz Deraman. "Version control system: a review". In: *Procedia Computer Science* (2018). DOI: 10.1016/j.procs.2018.08.191 (cit. on p. 1).

[Otte 2009] Stefan Otte. "Version control systems". In: 2009. URL: http://www.mi.fu-berlin.de/inf/groups/ag-tech/teaching/2008-09_WS/S_19565_Proseminar_Technische_Informatik/otte09version.pdf (cit. on p. 1).

[Peart 2018] Ben Peart. *Email "[PATCH v1 0/3] [RFC] Speeding up checkout (and merge, rebase, etc)"*. 2018. URL: https://lore.kernel.org/git/20180718204458.20936-1-benpeart@microsoft.com/ (visited on 03/09/2022) (cit. on p. 55).

[Perforce 2022] Perforce. *Perforce HelixCore Version Control*. URL: https://www.perforce.com/products/helix-core (visited on 03/10/2022) (cit. on p. 2).

[Pickens 2008] James Pickens. *Email thread "[RFC PATCH 0/2] Add support for multi threaded checkout"*. 2008. URL: https://lore.kernel.org/git/3BA20DF9B35F384F8B7395B001EC3FB3265B2A01@azsmsx507.amr.corp.intel.com/t/#u (visited on 03/09/2022) (cit. on p. 41).

[Spinellis 2005] D. Spinellis. "Version control systems". In: *IEEE Software* (2005). DOI: 10.1109/MS.2005.140 (cit. on p. 1).

[Stack Exchange, Inc. 2018a] Stack Exchange, Inc. *Stack Overflow Developer Survey Results 2018*. 2018. URL: https://insights.stackoverflow.com/survey/2018#work-_-version-control (visited on 10/29/2021) (cit. on p. 2).

[Stack Exchange, Inc. 2018b] Stack Exchange, Inc. *Stack Overflow Developer Survey Results 2021*. 2018. URL: https://insights.stackoverflow.com/survey/2021#most-popular-technologies-tools-tech (visited on 10/29/2021) (cit. on p. 2).

[The Bazaar Team 2022] The Bazaar Team. *Bazaar*. URL: https://bazaar.canonical.com/en/ (visited on 03/10/2022) (cit. on p. 2).

[The Linux Foundation and Torvalds 2014] The Linux Foundation and Linus Torvalds. *10 Years of Git: An Interview with Git Creator Linus Torvalds*. 2014. URL: https://www.linuxfoundation.org/blog/10-years-of-git-an-interview-with-git-creator-linus-torvalds/ (visited on 11/09/2021) (cit. on p. 2).

[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).