修士学位論文

題目

OverlayFS を用いた効率的な Git チェックアウトの提案
— リポジトリマイニングの高速化を目的として —

指導教員

楠本 真二 教授

報告者

三原 公平

令和 7 年 1 月 28 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

令和 6 年度 修士学位論文

OverlayFS を用いた効率的な Git チェックアウトの提案
— リポジトリマイニングの高速化を目的として —

三原 公平

内容梗概

Git はバージョン管理システムのデファクトスタンダードである．現在では開発者の 9 割以上がバージョン管理システムとして Git を使用しており，Git リポジトリのホスティングサービスも盛んに使用されている．データの豊富さから，ソフトウェアリポジトリを対象としたマイニング（Mining Software Repositories，MSR）が多数行われている．MSR における Git の課題の 1 つとして，チェックアウト処理の速度が遅い点が挙げられる．MSR ではプロジェクトの多数のリビジョンを参照するため，チェックアウト処理が頻繁に行われる．しかし，MSR で対象にされる大規模なリポジトリでは Git のチェックアウト処理は遅く，MSR の作業効率を低下させている．チェックアウト処理の時間的コストが大きい原因は Git 独自のファイルシステムにある．Git ではすべてのデータを blob（binary large object）と呼ばれる形式に変換して保存しており，データアクセス時には blob ファイルを作業ディレクトリに展開する処理が行われる．この展開処理に大きなコストがかかっている．本研究では，OverlayFS を用いた Git リポジトリ用ファイルシステムを提案する．提案ツールは時間的コストの大きい展開処理を不要にし，Git のチェックアウト処理を高速化する．具体的には，前処理として blob ファイルの展開を行っておき，チェックアウト時には展開したファイル群の必要部分を統合するだけで 1 つのリビジョンを表現する．本研究では提案手法の実行時間について評価実験を行った．実験の結果，提案手法は平均で Git の 3〜5 倍高速なチェックアウトを実現できていた．提案手法のより実践的な評価として，マイニング作業を模したスクリプトを実行する実験を行った．この実験では，提案手法が Git より 1.03 倍程度長い実行時間を要する結果となった．

主な用語

Git，バージョン管理システム，`git-checkout`，マイニング，高速化，OverlayFS，ファイルシステム

目次

1 はじめに 1

2 準備 3
- 2.1 Git とマイニング . . . 3
- 2.2 Git の課題 . . . 3
- 2.3 既存の Git 高速化手法 . . . 4

3 提案手法 5
- 3.1 概要 . . . 5
- 3.2 OverlayFS . . . 6
- 3.3 コミットの展開 . . . 7
- 3.4 OverlayFS によるコミットレイヤの統合 . . . 7
- 3.5 書き込みレイヤ . . . 8
- 3.6 .git レイヤ . . . 8

4 実装 9
- 4.1 前処理の流れ . . . 9
- 4.2 チェックアウトの実現 . . . 10
- 4.3 OverlayFS のレイヤ数制限 . . . 11
- 4.4 ファイル削除コミットの実現 . . . 13

5 OverlayFS の性能調査 15
- 5.1 実験 1：OverlayFS のマウント速度評価 . . . 15
- 5.2 実験 2：ファイル読み込み速度評価 . . . 16
- 5.3 実験 3：ディレクトリ読み込み速度評価 . . . 16
- 5.4 実験 4：メタデータ読み込み速度評価 . . . 17

6 評価 19
- 6.1 空間計算量 . . . 20
- 6.2 時間計算量 . . . 21
- 6.3 実践的な評価 . . . 23

7 議論 25

7.1 提案手法の実用性 25
7.2 ベースレイヤ実装とネスト実装の比較 25

8 **妥当性への脅威** 27

9 **展望** 28

10 **おわりに** 29

謝辞 30

参考文献 31

図目次

1 OverlayGit . . . 5
2 OverlayFS によるレイヤの統合 . . . 6
3 OverlayFS と Git の構造的類似 . . . 7
4 前処理の流れ . . . 9
5 統合するレイヤの順序 . . . 10
6 グラフ分割 . . . 11
7 レイヤ数制限解決策 1：ベースレイヤ . . . 12
8 レイヤ数制限解決策 2：OverlayFS のネスト . . . 12
9 OverlayFS におけるファイル隠蔽 . . . 14
10 統合するレイヤ数と OverlayFS マウント実行時間 . . . 16
11 ファイルが存在するレイヤの深さと読み込み時間 . . . 17
12 統合されているレイヤ数とディレクトリ読み込み時間 . . . 18
13 統合されているレイヤ数とメタデータ読み込み時間 . . . 18
14 Commons Compress のチェックアウト実行時間 . . . 22
15 Commons Math のチェックアウト実行時間 . . . 22
16 Closure Compiler のチェックアウト実行時間 . . . 23
17 ネスト実装のチェックアウト実行時間 . . . 26

表目次

1 実験対象プロジェクト 19
2 前処理前後の空間計算量 20
3 前処理実行時間 21
4 チェックアウト実行時間の平均 23
5 メトリクス計測実行時間 24

1 はじめに

Git はソフトウェア開発におけるバージョン管理システムのデファクトスタンダードである．Stack Overflow の調査によれば，ソフトウェア開発者の 93% 以上が Git を使用している [1]．Git リポジトリのホスティングサービスも盛んに使用されており，GitHub には 2021 年時点で 8,000 万件以上のパブリックリポジトリが存在している [2]．データの豊富さから，ソフトウェアリポジトリを対象としたマイニング研究（Mining Software Repositories，MSR）が多数行われている [3][4][5]．

Git の課題として，過去の情報へのアクセス，特に `git-checkout` コマンドが遅い点が挙げられる．過去情報へのアクセスの時間的コストが大きい原因は，Git 独自のファイルシステムにある．Git のファイルシステムでは，過去のファイルデータは全て blob（binary large object）と呼ばれるバイナリ形式のデータに変換される．`git-checkout` コマンドは，実行の度に blob から元データ形式への展開とワーキングディレクトリの書き換えを行う．この展開と書き換え処理がチェックアウトの計算コストを増大させている．特に，ファイル数が多い大規模リポジトリではチェックアウト処理の計算コストが大きくなる．筆者が実際に Linux カーネルプロジェクト[*1]において `git-checkout` コマンドを実行したところ，初期のコミットから最新のコミットへのチェックアウトに約 10 秒を要した．

チェックアウト処理の計算コスト増大は通常のソフトウェア開発にも影響するが，特に MSR において時間的コスト増大を招く．MSR では 1 つのプロジェクトに対して多数のチェックアウト処理が行われる場合がある．例えばバグデータセットの構築においては，バグ混入リビジョンの候補を多数チェックアウトし，バグが混入したリビジョンを特定する [6][7]．また，Flaky Test の分析においては，実行結果が変化するテストを検出するため，複数のリビジョンをチェックアウトしテストを繰り返す [8][9]．さらに，このような MSR ではスクリプト作成中にも多数のチェックアウトが実行される．このように MSR では多数のチェックアウト処理が実行されるため，チェックアウト処理の計算コスト増大は MSR の実行時間に大きな影響を与える．

チェックアウト処理を高速化する手法はいくつか提案されている．Scalar[*2]は大規模リポジトリ向けの Git 効率化ツールである．Scalar は部分的クローンや部分的チェックアウトの実現によって処理の高速化を図っている．また，チェックアウト処理の効率化ツールとして RepoFS が提案されている [10]．RepoFS はファイルアクセスの遅延実行の実現によってチェックアウトの高速化を図っている．これらのツールはリポジトリの一部だけを使用する状況では効果的である．しかし，MSR においてリポジトリ内の多数のファイルを扱う状況には適さず，高速化が難しい．また，RepoFS は読み取り専用のツールになっており，書き込みが必要な作業には使用できない．チェックアウト処理の計算コストを抑える

[*1] `https://github.com/torvalds/linux`

[*2] `https://github.com/microsoft/scalar`

シンプルな解決策として，ワークツリー複製も考えられる．ワークツリー複製とは，`git-worktree` コマンド[*3]やシェルの `cp` コマンドを利用して，チェックアウトしたリビジョンをコピーして置いておく手法である．ワークツリー複製はディレクトリを変更するだけでチェックアウトが実現できるため，高速にチェックアウト可能である．しかし，非常に多くの冗長ファイルが発生し，空間計算量が悪くなってしまう．

本研究では，効率的な空間・時間計算量を実現するチェックアウト高速化手法 OverlayGit を提案する．OverlayGit はまず前処理として，対象リポジトリの全コミットの内容を先に展開する．次に展開したコミットの必要部分のみを統合し，リビジョンのチェックアウトを実現する．なお，本稿ではある時点でのリポジトリのスナップショットをリビジョン，リビジョン間の差分をコミットと呼び，厳密に使い分ける．

OverlayGit は時間及び空間計算量について効率的である．OverlayGit は前処理によって `git-checkout` の計算コスト増大の原因である展開や置き換え処理を不要にする．このため，高速なチェックアウトが実現可能である．また，OverlayFS を用いたコミットの統合によって空間計算量を抑えたチェックアウトを実現する．OverlayFS は複数のディレクトリを論理的に統合し，1 つのディレクトリとして構成するファイルシステムである．OverlayFS を用いれば，展開したコミット群の必要部分だけを統合してリビジョンを表現可能である．このため，ワークツリー複製のように冗長なファイルを作成する必要がない．

提案手法の評価のため，OverlayGit を Python を用いて実装し評価実験を行った．実験の結果，OverlayGit のチェックアウト速度は平均で Git の 3〜5 倍であった．提案手法のより実践的な評価として，マイニング作業を模したスクリプトを実行する実験を行った．この実験では，提案手法が最大で Git の 1.03 倍程度の実行時間を要する結果となった．

[*3] `https://git-scm.com/docs/git-worktree`

2　準備

2.1　Git とマイニング

Git はソフトウェア開発におけるバージョン管理システムのデファクトスタンダードである．Git は 2005 年に開発が開始され，現在では Linux カーネルのような大規模かつ多人数が参加するプロジェクトから，個人単位の小規模なプロジェクトまで，規模や分野を問わず様々なプロジェクトが Git を用いてソースコードを管理している．また，GitHub や BitBucket，GitLab など Git リポジトリのホスティングサービスが多数存在し，数多くの公開リポジトリが存在する．例えば GitHub では 2021 年時点で 8,000 万件以上のパブリックリポジトリが存在している [2].

この豊富なデータに基づく研究として，ソフトウェアリポジトリを対象としたマイニング（Mining Software Repository，MSR）が多数行われている [4][5][11][12][13]．MSR 研究の分野では，ソースコード自体を対象にした研究 [14] から，コードレビューやライセンスなど開発管理体制を対象とした研究 [15][16] まで様々な研究が存在する．また，MSR の結果としてのデータセット [6][17] や MSR をサポートするツール [18][19][20][21] も多数提案されている．

2.2　Git の課題

Git の課題を挙げた研究はいくつか存在する [10][22][23][24][25] が，本研究では特に速度に関する問題に着目する．Git では，過去の情報へのアクセス，特にチェックアウト処理に大きな計算コストを要する．これは Git 独自のファイルシステムが原因である．Git ではファイルやコミットログ等すべてのデータを blob（binary large object）形式に変換して保存している．Git はチェックアウト実行の度に blob ファイルを展開し，ワーキングディレクトリを書き換える．この展開・書き換え処理の計算コストが大きく，チェックアウト処理は大きな時間的コストを要する．特に，ファイル数が多いリポジトリやファイルサイズが大きいリポジトリではチェックアウト処理の時間的コストは大きい．筆者が実際に Linux カーネルプロジェクトにおいて `git-checkout` コマンドを実行したところ，初期のコミットから最新のコミットへのチェックアウトに約 10 秒を要した．

チェックアウト処理の時間的コストの大きさは，特に MSR の作業時間に大きな影響を与える．MSR では，調査対象リポジトリの多くのリビジョンを確認するため，通常の開発と比べて遥かに多くのチェックアウト処理が行われる．例えば R. Just らの研究では Java のバグを集めたバグデータセット Defects4J を提案している [6]．Defects4J の構築では，バグの存在するリビジョンを特定するため，複数のリビジョンにおいてビルドとテストを実行している．この際，テスト実行するリビジョンのソースコードを取得するために多数のチェックアウト操作が行われている．このように MSR ではチェックアウト処理が頻繁に行われるため，チェックアウト処理の時間的コストの大きさは MSR の作業効率を大

きく左右する．

2.3　既存の Git 高速化手法

Git の高速化手法はいくつか提案されている．Scalar は大規模リポジトリを対象とした Git 高速化ツールである．リポジトリサイズが数 GB を超えるリポジトリでは，クローンやチェックアウトに非常に長い時間がかかる．Scalar は部分的クローンや部分的チェックアウトの実現によってこれらの問題を解決している．しかし，MSR ではリポジトリの一部だけでなく全体を扱う場合がある．例えばバグデータセット構築におけるビルド実行のように，あるリビジョンの全ファイルを扱う操作は Scalar によって高速化できない．

また，Git を用いた作業の効率化手法として RepoFS がある [10]．RepoFS は MSR におけるチェックアウト作業を効率化する読み取り専用のツールである．RepoFS は遅延実行の実現によってチェックアウト自体の処理時間を短縮している．RepoFS を用いたチェックアウトでは，ワークツリーのファイルリストだけがチェックアウト時に更新され，実際にファイルアクセスが発生したときにファイルの内容を取得・提示する．RepoFS はチェックアウト単体の処理は高速化するが，リポジトリ全体のファイルを操作する状況には不向きである．また，読み取り専用のため，プロジェクトのルートディレクトリにファイルを書き込みむビルドやテストを実行する MSR では扱いにくい．

チェックアウト処理を高速化するシンプルな手法として，ワークツリー複製も考えられる．ワークツリー複製とは，`git-worktree` コマンドやシェルの `cp` コマンドを用いて，Git のワークツリー全体をコピーする手法である．チェックアウトしたリビジョンをコピーして別ディレクトリに保存しておけば，そのディレクトリに移動するだけでチェックアウト処理を実現できる．しかし，ワークツリー複製では極めて多数の冗長ファイルが生成される．MSR のように多数のリビジョンにチェックアウトする状況では空間的コストが非常に大きくなり，採用が難しい手法である．

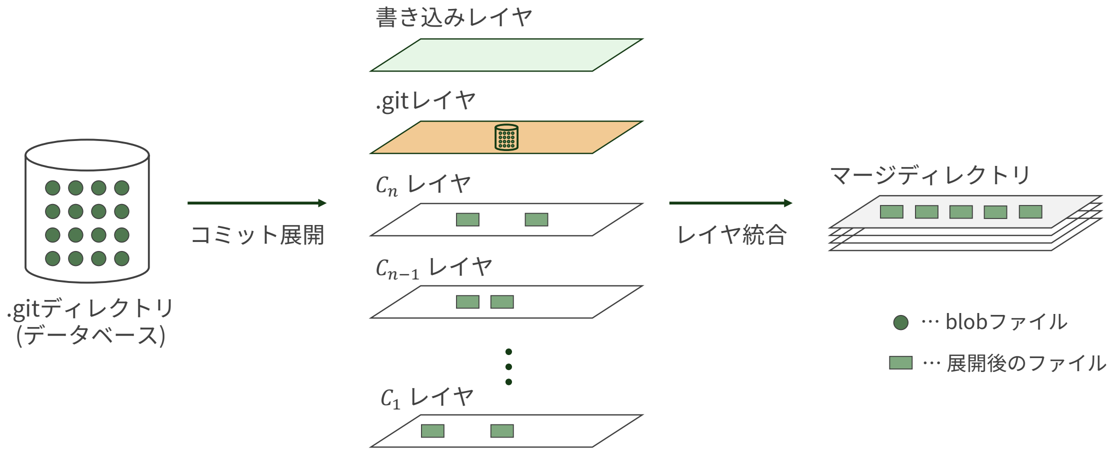

図 1: OverlayGit

3　提案手法

3.1　概要

本研究では，Git のチェックアウト高速化手法 OverlayGit を提案する．OverlayGit のキーアイデアは “空間的資源を活用した時間計算量の削減” である．Git のチェックアウト処理が遅い原因は blob ファイルの展開にある．OverlayGit ではこの展開を前処理で行い，チェックアウトの計算コストを軽減する．Git において圧縮形式で保存されるファイルを OverlayGit では展開するため，Git より多くの空間的資源を必要とする．しかし，Git のリポジトリサイズは 1～2GB が推奨されており [26][27]，最大級のリポジトリである Linux カーネルプロジェクト*4でも 7GB 程度である．そのため，大抵のリポジトリでは全ファイルを展開しても数十 GB 以下となる．

OverlayGit の概観を 図 1 に示す．OverlayGit は 2 段階の処理で高速なチェックアウトを実現する．まず前処理として，Git のデータベースにある blob ファイルをコミットごとに展開する．次に，展開したコミットディレクトリを OverlayFS のマウントによって統合する．OverlayFS は複数のディレクトリを統合して 1 つのディレクトリに見せるファイルシステムである．OverlayFS を用いると，必要なコミットだけを統合して 1 つのリビジョンを表現できる．提案手法は，一度前処理を行えば以降は OverlayFS のマウントを実行するだけでチェックアウトを実現できるため，高速なチェックアウトが実現可能である．OverlayFS と提案手法の詳細について，以降の節で説明する．

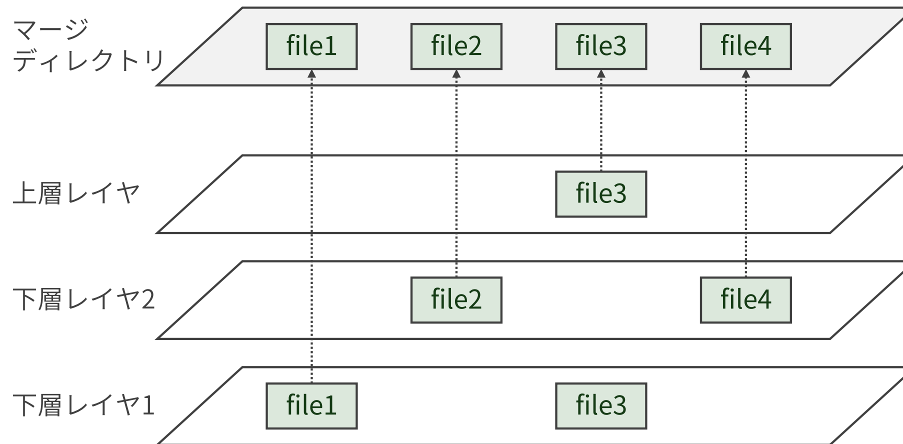

図 2: OverlayFS によるレイヤの統合

3.2 OverlayFS

OverlayFS は複数のディレクトリを論理的に 1 つのディレクトリに統合できるファイルシステムである．OverlayFS では統合するディレクトリそれぞれをレイヤと呼ぶ．図 2 に OverlayFS によるレイヤ統合の様子を示す．file1〜4 は 3 つのレイヤに分散して存在している．このとき，OverlayFS を 3 つのレイヤに対してマウントすると，マージディレクトリに全てのファイルが存在するように見せることができる．OverlayFS の特徴は，レイヤを重ねて上から観測するように統合する点である．具体的には，OverlayFS では統合するレイヤに同名のファイルがある時，上にあるレイヤのファイルがマージディレクトリで採用される．例えば 図 2 のように，上層レイヤと下層レイヤ 1 に file3 が存在する場合，マージディレクトリから見えるファイルは上層レイヤの file3 となる．なお，OverlayFS では統合する最も上のレイヤだけが上層レイヤと呼ばれ，それ以外のレイヤは下層レイヤと呼ばれる．下層レイヤは読み込み専用であり，上層レイヤだけが書き込み可能である．このため，下層レイヤは複数のプロセス間での共有が非同期に行える．Docker は OverlayFS を用いて複数のコンテナ間でファイルを共有し，空間的コストを削減している [28].

OverlayFS の構造は Git と類似している．図 2 において 3 つのレイヤの内容がそのまま Git のコミットであると仮定する．すなわち，図 3 に示すように最初のコミットで file1 と file3 を追加し，次のコミットで file2 と file4 を追加，最新のコミットで file3 を編集したとする．このとき，Git リポジトリのワークツリーはちょうどマージディレクトリの内容と同一になる．このように，レイヤがコミットに，マージディレクトリはあるリビジョンにおけるワークツリーと対応し，OverlayFS と Git は構造的に類似している．本研究では，この類似点に着目し提案手法を考案した．なお，本稿ではある時点でのリポ

[4] `https://github.com/torvalds/linux`

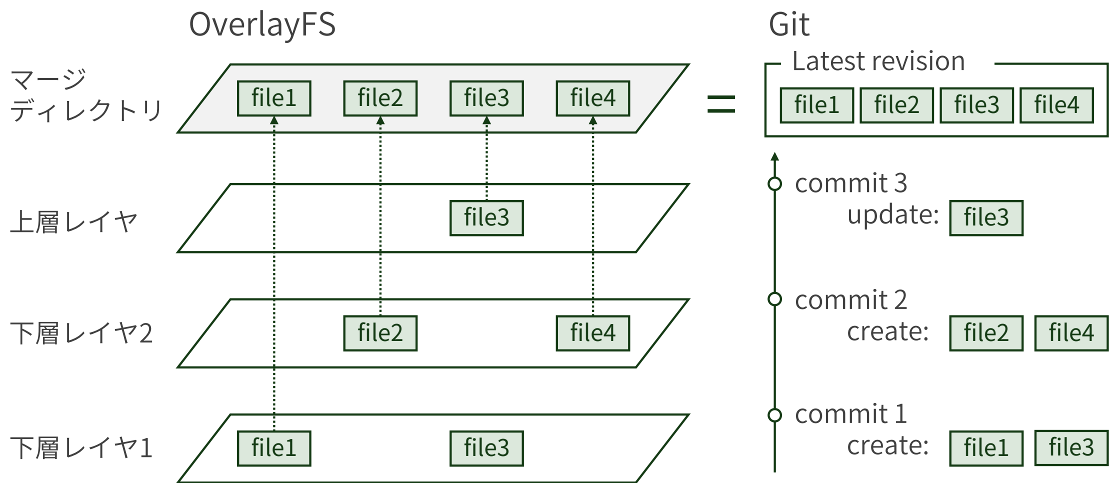

図 3: OverlayFS と Git の構造的類似

ジトリのスナップショットをリビジョン，リビジョン間の差分をコミットと呼び，厳密に使い分ける．

OverlayFS と同種のファイルシステムとして，AUFS[*5]や UnionFS[*6]が存在する．これらのファイルシステムは OverlayFS と同様に，複数のディレクトリを統合して 1 つのディレクトリに見せることができる．OverlayFS と AUFS，UnionFS の動作はほぼ同等であるが，OverlayFS は現在 Linux カーネル内に組み込まれているという違いがある．したがって，他のファイルシステムと比べ安定で高速な動作が期待できる．

3.3 コミットの展開

OverlayGit は前処理としてコミットを展開する．具体的には，図 1 の C_1〜C_n のように 1 つのコミットに対して 1 つのレイヤを用意し，コミットで編集されたファイルだけを対応するレイヤに設置する．Git では`.git` ディレクトリがデータベースとなっており，すべてのデータが`.git` に保存されている．コミット展開処理では`.git` ディレクトリから各コミットで変更されたファイルを取得し，対応するレイヤに展開する．コミットで変更されたファイルリストの取得や blob ファイルの展開は，Git の提供する API によって実現される[*7]．

3.4 OverlayFS によるコミットレイヤの統合

OverlayGit では，OverlayFS によるコミットレイヤの統合によってリビジョンを表現する．前処理によってコミットの内容がそれぞれのレイヤに設置されているとき，コミットレイヤに対して OverlayFS

[*5] `https://aufs.sourceforge.net/`
[*6] `https://unionfs.filesystems.org/`
[*7] `https://libgit2.org`

をマウントするだけで 1 つのリビジョンが表現できる．例として，図 1 においてコミットレイヤ C_n から C_1 への OverlayFS のマウントを考える．前処理で展開したコミットレイヤをコミットの時系列順に並べると，ちょうどコミット履歴のようになる．OverlayFS はレイヤを重ねて上から観測するように統合するので，レイヤ C_n から C_1 を OverlayFS でマウントすると，マージディレクトリは最新リビジョンのワークツリーと同一になる．

提案手法では，OverlayFS でマウントするレイヤを変更するだけでチェックアウトが実現できる．例えば 図 1 においてコミット C_n からコミット C_{n-1} にチェックアウトする場合，OverlayFS によって統合するレイヤを C_{n-1} 以下にするだけでチェックアウトが実現できる．

3.5 書き込みレイヤ

提案手法ではリビジョンごとに書き込みレイヤを用意し，書き込み可能な Git ファイルシステムを実現している．リビジョンごとに異なる書き込みレイヤを使うと，各リビジョンで個別に書き込み内容を保持できる．MSR において書き込み可能な Git ファイルシステムは有用である．例えばチェックアウトしたリビジョンでビルドやテストを行う場合，そのリビジョンにキャッシュがあれば再ビルドの実行時間が短縮できる．

3.6 .git レイヤ

本研究では，Git の外部仕様を変更せずにチェックアウトを高速化することを目指している．そのために，提案手法では.git ディレクトリを設置した.git レイヤを用意している．.git ディレクトリは Git におけるデータがすべて格納されたデータベースであり，Git サブコマンドの入出力の大部分は.git に対して行われる．よって提案手法では.git レイヤを設置しており，Git サブコマンドのほぼすべてを Git と同様に使用できる．

なお，.git レイヤは全てのコミットレイヤの上に設置する．これは，OverlayFS では上にあるレイヤほどファイルアクセス速度が速いためである．Git は.git ディレクトリへ頻繁にアクセスするため，.git レイヤを最も上に設置して Git コマンドの速度を高めている．

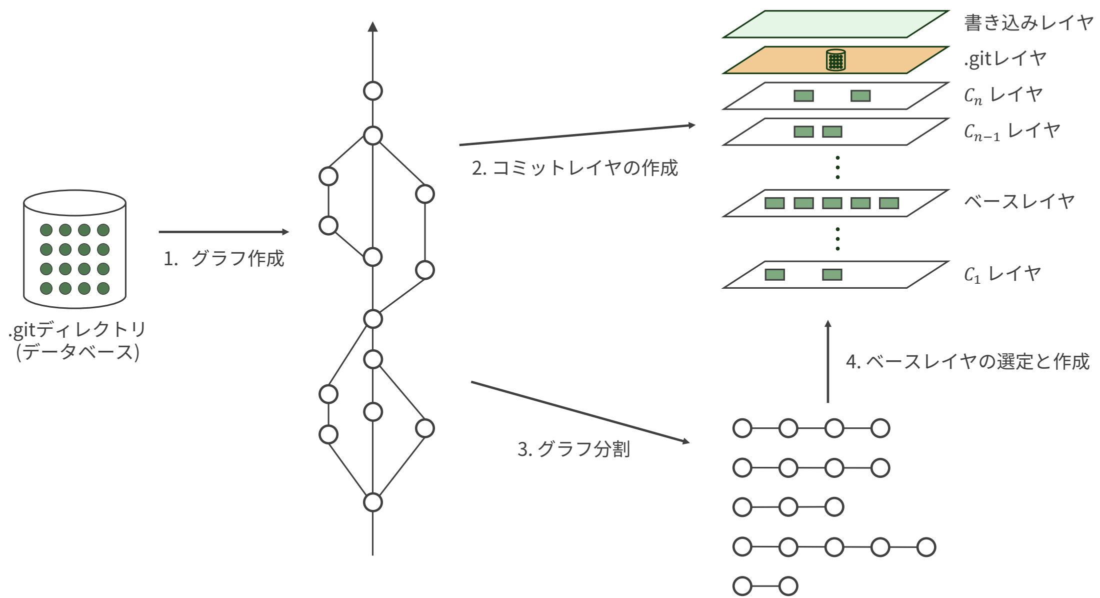

図 4: 前処理の流れ

4　実装

本研究では，OverlayGit を Python を用いて実装した．OverlayGit は前処理及びチェックアウトの 2 つの処理を実現する．前処理では，Git コミットのレイヤへの展開と，コミット履歴の取得及び記録を行う．前処理でコミット履歴の取得を行っている理由は，チェックアウトをより高速化するためである．チェックアウト処理では，チェックアウトするコミットに必要なレイヤを選定して OverlayFS でマウントする必要がある．このレイヤ選定の一部を前処理で行っておけば，チェックアウトをより高速化できる．

4.1 節で前処理，4.2 節でチェックアウト処理の実現についてそれぞれ記述する．4.3 節では OverlayFS の実装上の制約について説明し，本手法でその制約についてどのように対処しているか紹介する．4.4 節では，Git のコミットにおけるファイル削除をどのように実現しているか紹介する．

4.1　前処理の流れ

OverlayGit の前処理の流れを 図 4 に示す．前処理ではまず，Git のデータベースからコミット履歴を取得し，コミットグラフを作成する．コミットグラフとは，コミットを頂点とし，親子関係にあるコミットを辺でつないだグラフである．一度コミットグラフを作成しておけば，以降はグラフを全探索すればすべてのコミットについて処理できる．グラフ作成後はコミットグラフを全探索しながらコミットレイヤを作成していく．コミットレイヤの作成と同時に，`.git` レイヤ及び書き込みレイヤも作成する．

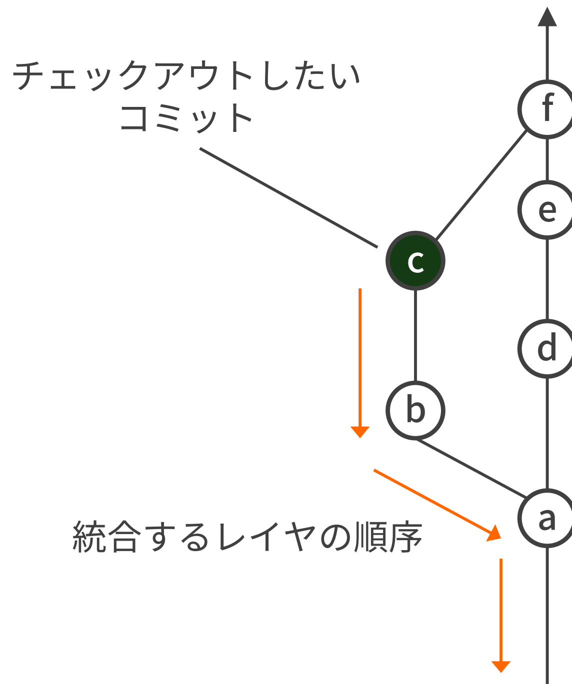

図 5: 統合するレイヤの順序

コミットレイヤの作成後は，コミットグラフの分割及びベースレイヤの選定と作成を行う．コミットグラフの分割はチェックアウト実現のための処理である．詳細については 4.2 節で紹介する．ベースレイヤは，OverlayFS の仕様上の制約に対応するためのレイヤである．詳細については 4.3.1 節で紹介する．

4.2 チェックアウトの実現

3 節で紹介した通り，提案手法のチェックアウトは OverlayFS によるコミットレイヤの統合によって実現される．統合するコミットレイヤは，コミットグラフを参照して選定できる．例えば 図 5 において，c のコミットへのチェックアウトを考える．c のコミット時における Git のワークツリーは，コミット b と a の内容を含み，コミット d や e の内容は含まない．したがって，提案手法のチェックアウトでは c，b，a のコミットレイヤのみを統合する．提案手法では統合するレイヤを時系列順に上から下に並べるため，レイヤの順序は上から順に $c \to b \to a$ となる．

他の一例として，f のコミットにチェックアウトする場合を考える．f のコミット時のワークツリーは，コミット a ～e の内容すべてを含むため，これらすべてのコミットレイヤを統合する．統合するレイヤの順序について，c と b と a，e と d と a については上から順に $c \to b \to a$，$e \to d \to a$ となる必要がある．そのため，提案手法では $f \to e \to d \to c \to b \to a$ の順で統合する．

本手法では，チェックアウトの処理負荷軽減のため，前処理でコミットグラフを作成・記録しておく．

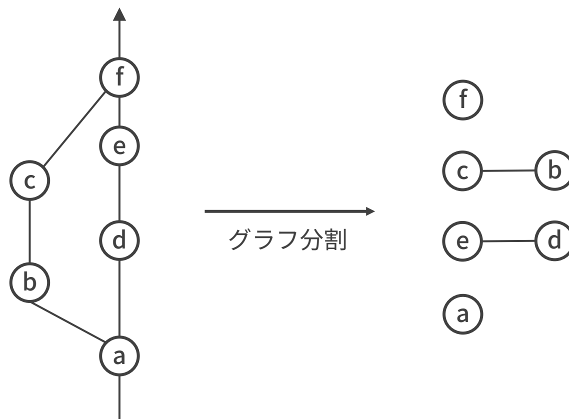

図 6: グラフ分割

グラフの記録においては，図 6 のように分割してから記録する．これは，グラフを分割し単純な枝分かれのないグラフの形で記録しておくことで，レイヤ選定の処理を単純化し高速化するためである．

4.3 OverlayFS のレイヤ数制限

OverlayFS は複数のディレクトリを統合できるファイルシステムであるが，統合するレイヤ数は 500 までに制限されている[*8]．提案手法では対応できるリポジトリのコミット数とレイヤ数が対応するため，通常では 500 コミット以下のリポジトリしか対応できない．本研究ではこの問題に対して 3 つの解決策を考案した．1 つ目はリビジョンの内容をそのまま持つベースレイヤを設置する方法，2 つ目は OverlayFS をネストする方法，3 つ目は OverlayFS 自体のソースコードを書き換える方法である．本研究の実装では，ベースレイヤを設置する方法を採用した．以降の節で，それぞれの解決策の概要と，ベースレイヤ設置を採用した理由を紹介する．

4.3.1 レイヤ数制限解決策 1：ベースレイヤ

OverlayFS のレイヤ数制限に対する解決策として，図 7 に示すベースレイヤを設置する方法がある．以降，この解決策をベースレイヤ実装と呼ぶ．ベースレイヤはリビジョンの内容をそのまま置いたレイヤであり，そのリビジョンより下のコミットレイヤの内容を全て含んでいる．したがって，チェックアウト処理ではベースレイヤを基準にその上のレイヤを統合すれば，任意のリビジョンを表現可能である．このベースレイヤを 500 レイヤごとに設置すれば，チェックアウトに必要なレイヤ数が常に 500 以下に抑えられ，OverlayFS のレイヤ数制限を解決できる．

ベースレイヤ実装ではリビジョンの内容をそのまま持つベースレイヤを作成するため，ディスク容量

[*8] https://github.com/torvalds/linux/blob/master/fs/overlayfs/params.h#L20

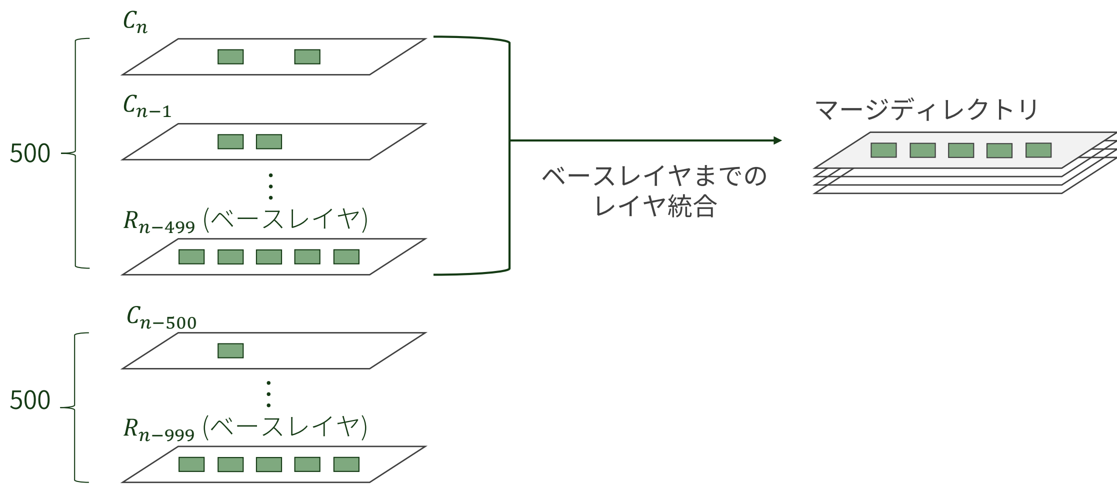

図 7: レイヤ数制限解決策 1：ベースレイヤ

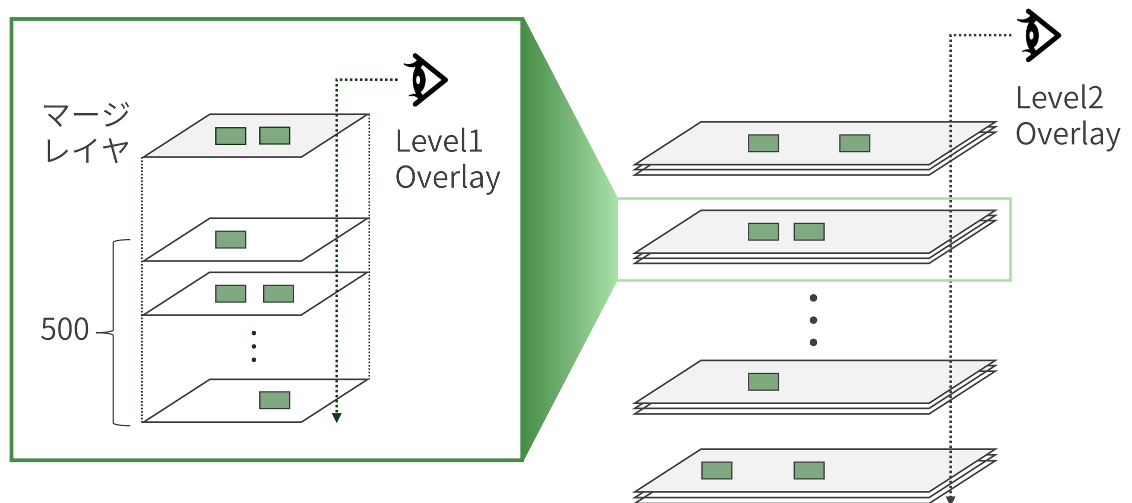

図 8: レイヤ数制限解決策 2：OverlayFS のネスト

及び前処理の時間計算量をある程度必要とする．しかし，後述する解決策 2 と比べてチェックアウトの時間計算量は優れている．また，OverlayFS のレイヤ数制限に沿った解決策になっており，解決策 3 と比べ安全かつ可搬性が高い．このため，本研究ではベースレイヤ実装を採用した．

4.3.2　レイヤ数制限解決策 2：OverlayFS のネスト

解決策 2 は，OverlayFS のネストによってレイヤ数制限を緩和する方法である．以降，この解決策をネスト実装と呼ぶ．図 8 にネスト実装の概要を示す．OverlayFS は 2 段階までのネストが可能である．したがって，500 個以下のレイヤを統合したマージレイヤを複数並べ，さらにそれらを OverlayFS で

マージすれば 500 のレイヤ数制限を緩和できる．1 回のマージで最大 500 レイヤまで統合可能なため，最大で $500^2 = 250,000$ レイヤの実現が可能である．

ネスト実装は OverlayFS によるレイヤ統合を複数回実行する必要がある．そのため，ベースレイヤ実装と比べてチェックアウトの時間計算量が大きくなる手法である．また，5 節で紹介する OverlayFS の予備調査の結果から，OverlayFS で扱うレイヤ数が増えるほど統合後のファイル操作速度は遅くなると考えられる．このため，チェックアウト速度だけでなくファイル操作速度の観点から見ても，ネスト実装はベースレイヤ実装より時間計算量が大きい手法となる．ただし，ベースレイヤ実装と比べ新たにレイヤを作成しないので，空間計算量ではネスト実装が優れている．よって，ネスト実装は空間計算量を抑えたい場合に利用できる．そのため，5 節ではネスト実装も考慮して調査を行っている．また，本研究ではネスト実装の試作を実装しており，7.2 節で試作した結果について述べる．

4.3.3 レイヤ数制限解決策 3：OverlayFS の書き換え

解決策 3 は，OverlayFS 自体のソースコードの書き換えによって 500 のレイヤ数制限を変更する方法である．OverlayFS のレイヤ数制限は，ソースコード自体に定数として記述されている[*9]．この定数値を書き換えてコンパイルするだけで，OverlayFS のレイヤ数制限を変更できる．実際に定数値を 500 から 1,000 に書き換えてコンパイルしたところ，1,000 レイヤの統合に成功した．

解決策 3 は上述した 2 つの解決策と比べ，余分な空間・時間計算量の消費が存在しない．しかし，安全性と可搬性に問題がある方法である．安全性の問題とは，レイヤ数制限を書き換える副作用が不明であることである．OverlayFS がレイヤ数を制限している理由は未だ調査中であり，レイヤ数制限を書き換えると Linux カーネルの動作に影響を与える可能性がある．可搬性の問題とは，提案手法を利用するマシンごとにカーネルの書き換えが必要な点である．OverlayFS は Linux カーネルに組み込まれているため，利用する Linux マシンごとにソースコードの書き換え・コンパイルが必要となる．ソースコードの書き換え箇所自体は数行であり，容易に書き換え可能である．しかし，Linux カーネルのコンパイルには数十分〜数時間がかかり，大きな時間的コストを必要とする．以上の理由から，解決策 3 は安全性と可搬性に課題のある方法である．

4.4 ファイル削除コミットの実現

Git では `git-rm` コマンドでファイルを削除できる．提案手法では，Git のファイル削除コミットに対応するため，OverlayFS のファイル隠蔽の仕組みを利用する．OverlayFS はホワイトアウトファイルと呼ばれる特殊なデバイスファイルを用いてファイルを隠蔽できる．例えば 図 9 のように，下層レイヤ 2 に file2 がある状況を考える．このとき，上層レイヤに file2 というファイル名でホワイトアウト

[*9] `https://github.com/torvalds/linux/blob/master/fs/overlayfs/params.h#L20`

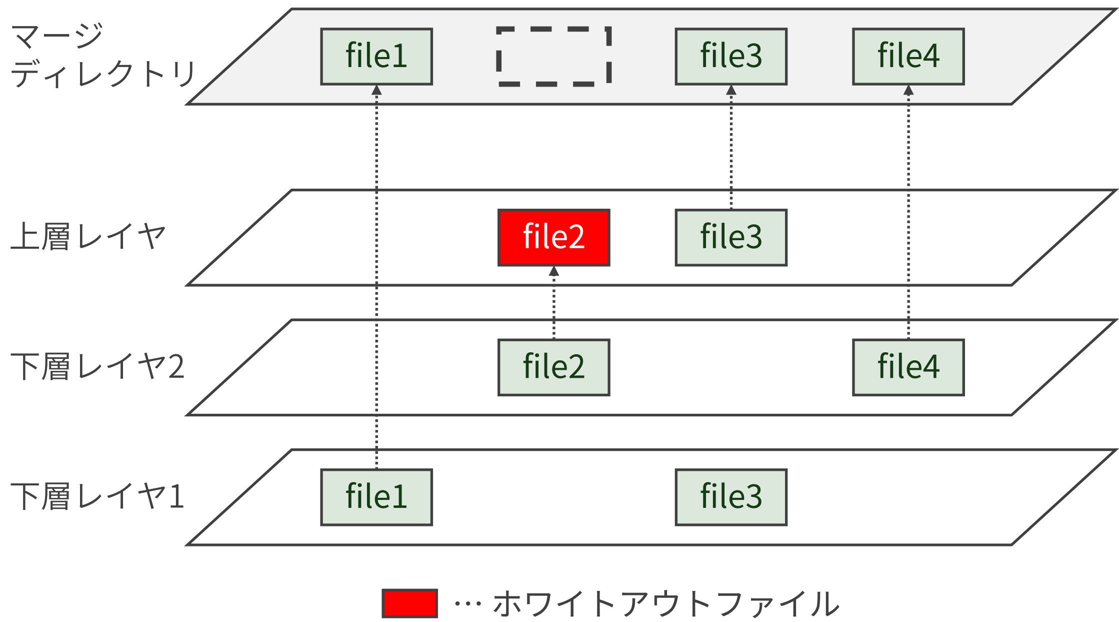

図 9: OverlayFS におけるファイル隠蔽

ファイルを置くと，マージディレクトリでは file2 が隠蔽され，ユーザに見えなくなる．このように，ホワイトアウトファイルを隠蔽したいファイルより上のレイヤに設置すると，ファイルを隠蔽できる．提案手法では，ファイル削除があったコミットのレイヤにホワイトアウトファイルを置き，Git におけるファイル削除を実現している．

5　OverlayFS の性能調査

本研究では提案手法の評価の前段階として，OverlayFS 自体の性能評価を行う．提案手法のチェックアウトは OverlayFS のマウントによって実現される．そのため，提案手法の性能は OverlayFS のマウント速度及びファイルシステム内のファイルアクセス速度に強く依存する．よって本研究では予備調査として以下の 4 つの実験を行い，OverlayFS 自体について評価する．

実験 1：OverlayFS のマウント速度評価

実験 2：ファイル読み込み速度評価

実験 3：ディレクトリ読み込み速度評価

実験 4：メタデータ読み込み速度評価

OverlayFS のマウント速度及びファイル操作速度は，対象とするディレクトリ構造やファイル内容によって変化する．そのため，実際の Git リポジトリではなく理想的で単純なディレクトリ構造を用意して調査を行った．具体的には，ディレクトリを 1 万個用意し，それぞれのディレクトリに 100KB のファイルを 1 つずつ設置した．本調査では，このディレクトリ群に対してマウント及びファイル操作を実行し，速度を計測する．

なお，4.3 節で紹介した通り OverlayFS にはレイヤ数制限が存在する．本研究では 4.3.1 節で紹介したベースレイヤ実装によってレイヤ数制限を解決するため，OverlayFS で扱うレイヤ数は 500 以下となる．しかし，4.3.2 節で紹介したネスト実装も，空間的計算量の点で優れた手法となっている．したがって，本調査では 500 レイヤ以上の統合時における性能調査も実施し，ネスト実装の可能性も同時に確認する．このため，500 以上のレイヤ統合を OverlayFS のネストによって実現し，1 万レイヤまでの OverlayFS の性能を調査する．

5.1　実験 1：OverlayFS のマウント速度評価

OverlayFS によって統合するレイヤ数とマウント実行時間の関係を確認するため，マウントするレイヤ数を変化させながらマウント実行時間を計測した．図 10 に結果を示す．約 500 レイヤごとに階段状にマウント実行時間が増加している．これは，統合するレイヤが 500 増えるごとに OverlayFS のマウント回数が 1 回ずつ増加しているためである．0〜500 レイヤではマウント速度が一定かつ高速であり，レイヤ数が 1 万程度でも約 0.04 秒程度と高速である．提案手法では，ベースレイヤ実装の場合マウントするレイヤ数は 500 以下であり，ネスト実装ではマウントするレイヤ数がリポジトリのコミット数と等しくなる．したがってこの結果より，ベースレイヤ実装及びネスト実装の両方で高速にチェックアウトできる可能性が示された．

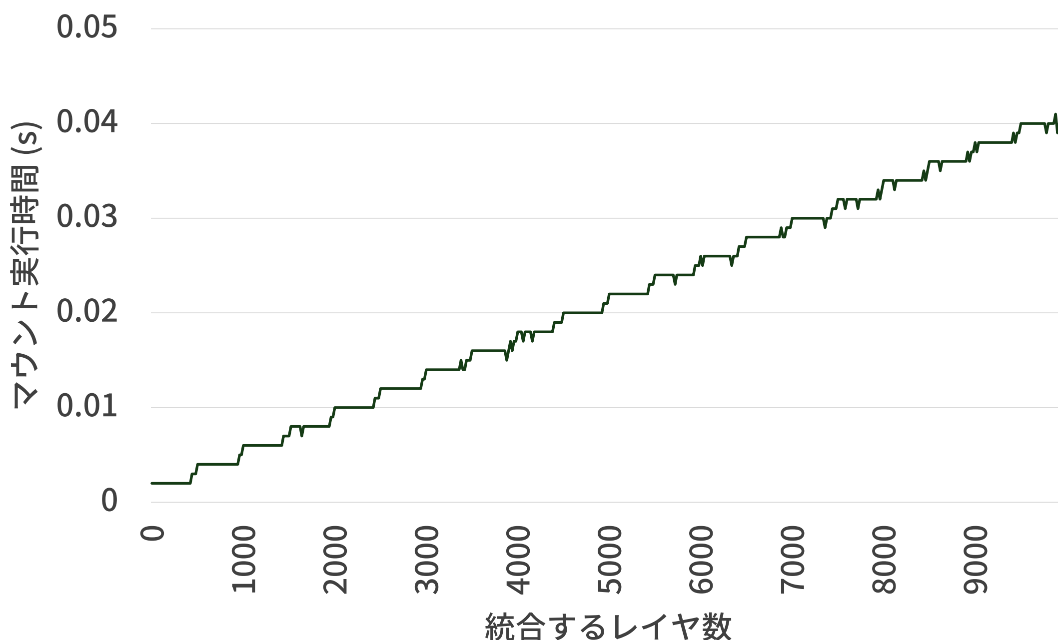

図 10: 統合するレイヤ数と OverlayFS マウント実行時間

5.2 実験 2：ファイル読み込み速度評価

実験 2 では，ファイルが存在するレイヤの深さ（最も上のレイヤからの距離）によってファイル読み込み速度が変化するか調査した．ファイル読み込みは cat コマンドによって実行した．結果を 図 11 に示す．ファイル読み込み時間はそのファイルが存在するレイヤの深さに応じて指数的に増加している．この結果から，提案手法では更新頻度が低いファイルほどアクセス速度が遅くなることが示唆される．提案手法では直近に編集されたファイルほど浅い位置のレイヤにあり，更新が長くされていないファイルほど深いレイヤにある．そのため，更新頻度が少ないファイルは深い位置にあり，アクセス速度が遅くなる可能性がある．しかし，図 11 の通り深さ 1 万のレイヤにあるファイルでも 0.02 秒以下で読み込み可能である．多くのリポジトリが 1 万コミット以下であることを考慮すれば，提案手法はネスト実装においても十分高速にファイル読み込み可能である．

5.3 実験 3：ディレクトリ読み込み速度評価

実験 3 では，統合されているレイヤ数とディレクトリの読み込み速度の関係を調査した．ディレクトリの読み込みは find コマンドによって実行した．結果を 図 12 に示す．図より，find コマンドの実行時間はマウント実行速度と同様に階段状に増加するが，1 万レイヤの統合時でも 0.03 秒程度と高速である．よって，提案手法ではファイル名を用いる操作が高速に実行可能である．

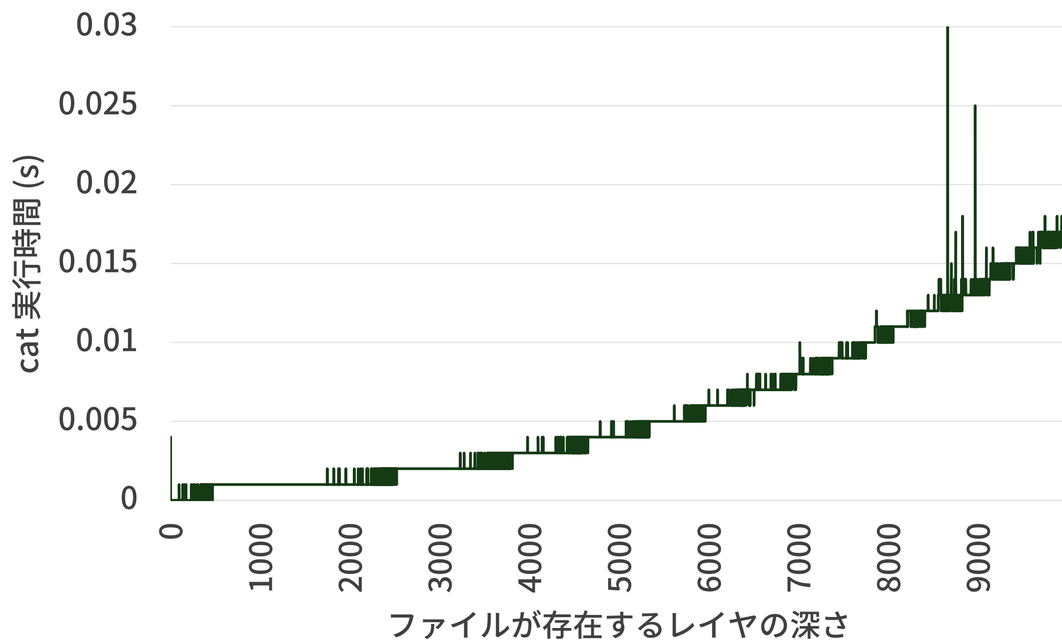

図 11: ファイルが存在するレイヤの深さと読み込み時間

5.4 実験 4：メタデータ読み込み速度評価

実験 4 では，統合されているレイヤ数とメタデータ読み込み速度の関係を調査した．メタデータ読み込みは `ls -l` コマンドによって実行した．図 13 に結果を示す．図より，500 レイヤ以下では `ls -l` コマンドが高速に実行できている．しかし，1,000 コミットを超えると実行時間が増加し，1 万レイヤでのコマンド実行は非常に大きい時間的コストを要する．これは，`ls -l` コマンドでは全レイヤのファイルを 1 つ 1 つ確認していることが原因と考えられる．OverlayFS はマウント時に統合したレイヤのファイル名リストを取得するが，ファイルの内容やメタデータは取得しない[*10]．そのため，全ファイルのメタデータを確認する `ls -l` は，統合したレイヤを 1 つ 1 つ参照する必要があり，長い実行時間を要する．この結果から，提案手法のネスト実装は全ファイルのメタデータ取得が必要な状況には不向きであると考えられる．しかし，MSR においてファイル名以外のメタデータの参照は少ない．そのため，この時間的コストの大きさが提案手法を使った MSR に与える影響は小さい．また，ベースレイヤ実装については常に 500 レイヤ以下に収まるため，`ls -l` コマンドのようなレイヤ 1 つ 1 つを参照する操作であっても高速に実行可能である．

[*10] `https://docs.kernel.org/filesystems/overlayfs.html`

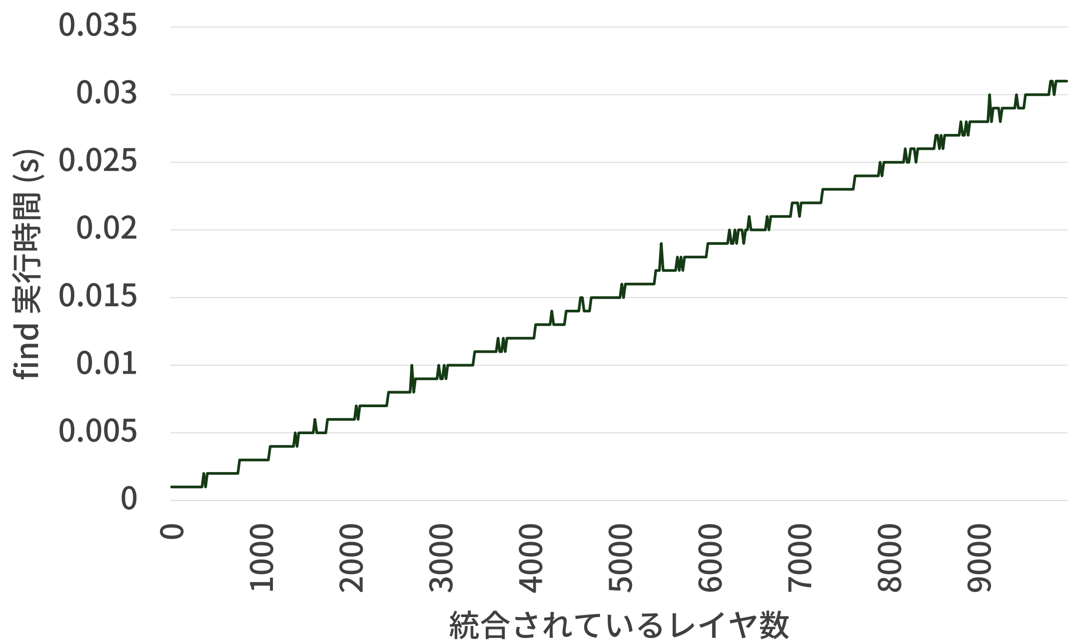

図 12: 統合されているレイヤ数とディレクトリ読み込み時間

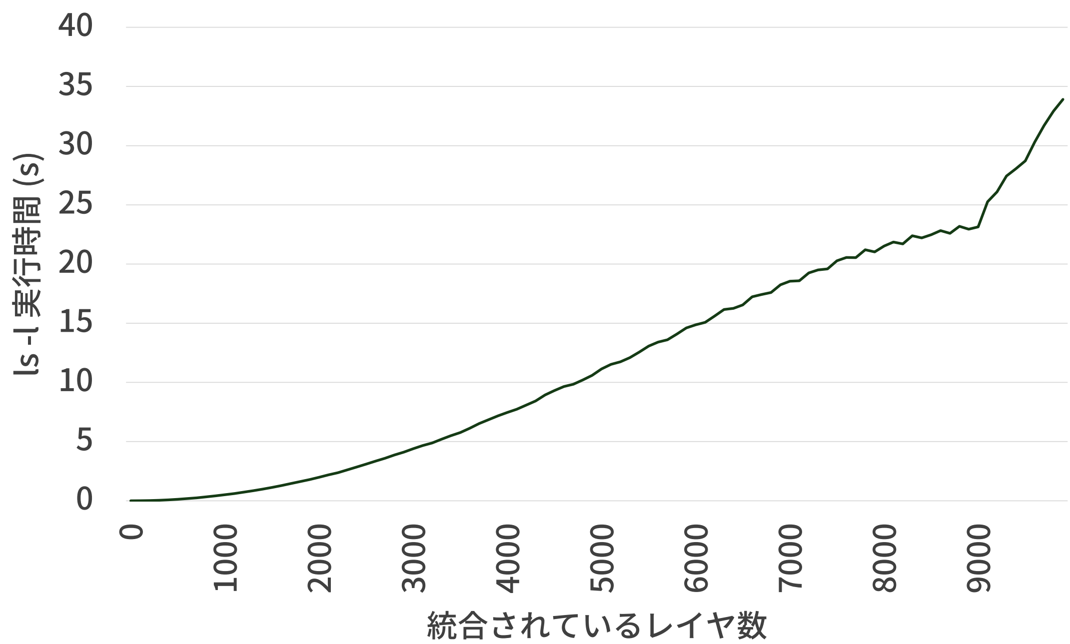

図 13: 統合されているレイヤ数とメタデータ読み込み時間

6 評価

提案手法の性能を評価するため，前処理及びチェックアウト処理の空間・時間計算量について計測した．ここで，空間計算量とは提案手法が必要とするディスク容量を指す．また，より実践的な評価として，マイニング作業を模したスクリプトを実行し，Git と提案手法で実行時間を比較する実験を行った．6.1 節で空間計算量について，6.2 節で時間計算量について実験の結果を紹介し，6.3 節で実践的な実験について詳細と結果を紹介する．なお，本節で紹介する実験結果は 4.3.1 節で紹介したベースレイヤ実装による結果である．

提案手法の評価では，表 1 に示す 10 個のリポジトリを用いた．ただし，時間計算量の実験及び実践的な実験では Commons Compress，Commons Math 及び Closure Compiler を対象に実験を行った．これら 10 個ののリポジトリはマイニング研究でも対象にされている著名な OSS リポジトリである

表 1: 実験対象プロジェクト

プロジェクト名	コミット数	スター数	プロジェクト概要
RStudio*[11]	43,055	4,700	IDE
Spring Framework*[12]	32,058	57,200	開発フレームワーク
Google Closure Compiler*[13]	19,441	7,400	ソースコード最適化ツール
Apache Commons Lang*[14]	8,354	2,800	汎用ライブラリ
Jackson Databind*[15]	8,101	3,500	JSON 変換ライブラリ
Apache Commons Math*[16]	7,199	588	数値計算ライブラリ
Mockito*[17]	6,233	15,000	モックフレームワーク
Apache Commons Compress*[18]	5,644	347	圧縮ライブラリ
Jackson Core*[19]	3,037	2,300	JSON ライブラリ
Apache Commons CSV*[20]	2,711	382	CSV ライブラリ

*[11] https://github.com/rstudio/rstudio
*[12] https://github.com/spring-projects/spring-framework
*[13] https://github.com/google/closure-compiler
*[14] https://github.com/apache/commons-lang
*[15] https://github.com/FasterXML/jackson-databind
*[16] https://github.com/apache/commons-math
*[17] https://github.com/mockito/mockito
*[18] https://github.com/apache/commons-compress
*[19] https://github.com/FasterXML/jackson-core
*[20] https://github.com/apache/commons-csv

[6][13].

6.1 空間計算量

提案手法の空間計算量の評価として，前処理実行前後のリポジトリの容量を計測した．計測にはシェルコマンドの du を用いた．計測の結果，リポジトリのディスク容量は 表 2 のようになった．リポジトリサイズの増加は 5 倍〜39 倍の範囲で収まっている．

この結果から，提案手法において要求される空間的コストは実用の範囲内であると考えられる．Git ではパフォーマンスや保守性の観点から，数 GB 以下のリポジトリサイズが推奨されている．具体的には，GitHub では 1GB，BitBucket では 2GB 以下が推奨されている [26][27]．したがって，提案手法によってリポジトリサイズが数十倍程度に増えたとしても，多くのリポジトリで必要とする空間コストは数十 GB 程度である．大量のリポジトリを対象に扱う MSR においては大容量のストレージを用意する場合が多い．そのため，提案手法の空間計算量は MSR においては十分許容できる範囲であると考えられる．

表 2: 前処理前後の空間計算量

プロジェクト名	コミット数	ディスク容量 (MB)		倍率
		前処理実行前	前処理実行後	
RStudio	43,055	1,346	21,270	15.8
Spring Framework	32,058	330	12,394	37.6
Google Closure Compiler	19,441	199	7,701	38.7
Apache Commons Lang	8,354	40	1,304	32.6
Jackson Databind	8,101	99	3,816	38.5
Apache Commons Math	7,199	51	1,295	25.4
Mockito	6,233	60	671	11.2
Apache Commons Compress	5,644	176	1,271	7.2
Jackson Core	3,037	35	672	19.2
Apache Commons CSV	2,711	83	480	5.8

6.2 時間計算量

6.2.1 前処理の実行時間

前処理について結果及び考察を述べる．表 1 で示した 10 個の Git リポジトリに対して前処理を実行したところ，前処理の実行時間は 表 3 のようになった．コミット数が多いリポジトリほど前処理に時間がかかっているが，数万コミットのリポジトリにおいても数分以内に完了できている．前処理は 1 回しか実行しないことを考慮すると，提案手法の前処理は十分に高速であると考えられる．

6.2.2 チェックアウト処理の実行時間

チェックアウト処理について計測結果及び考察を述べる．チェックアウト実行時間の計測では，最新から過去のリビジョンへチェックアウトする速度を計測し，リビジョン間の距離と実行時間の関係を調査した．実験には 表 1 に示したリポジトリのうち，Apache の Commons Compress と Commons Math 及び Google Closure Compiler を用いた．結果を 図 14，15 及び 16 に示す．Git に着目すると，どのリポジトリに対しても，チェックアウトするリビジョン間距離が大きくなるほど実行時間が増加している．リビジョン間距離が大きくなるほど実行時間が大きくなる原因は，リビジョン間距離に応じてワーキングツリーの差分が大きくなるためである．Git に対して，提案手法の実行時間はほぼ一定であり，リビジョン間距離にかかわらず高速なチェックアウトを実現できている．

本実験における Git のチェックアウト実行時間の平均を算出すると，表 4 のようになった．表より，

表 3: 前処理実行時間

プロジェクト名	コミット数	前処理実行時間 (s)
RStudio	43,055	530.7
Spring Framework	32,058	333.9
Google Closure Compiler	19,441	136.0
Apache Commons Lang	8,354	18.1
Jackson Databind	8,101	79.2
Apache Commons Math	7,199	19.8
Mockito	6,233	11.4
Apache Commons Compress	5,644	18.2
Jackson Core	3,037	10.2
Apache Commons CSV	2,711	6.1

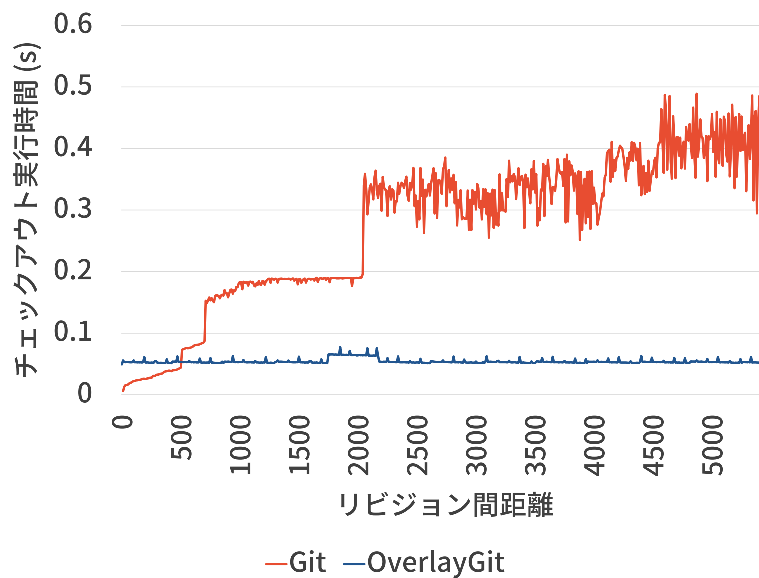

図 14: Commons Compress のチェックアウト実行時間

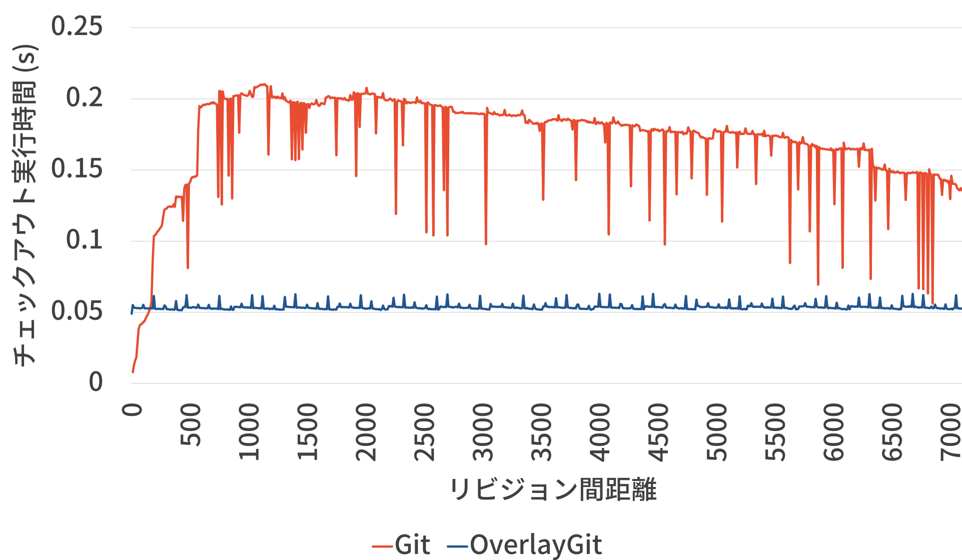

図 15: Commons Math のチェックアウト実行時間

OverlayGit のチェックアウト実行時間の平均は，どのリポジトリにおいても Git より高速である．よって，提案手法は Git より高速なチェックアウトを実現できていると考える．

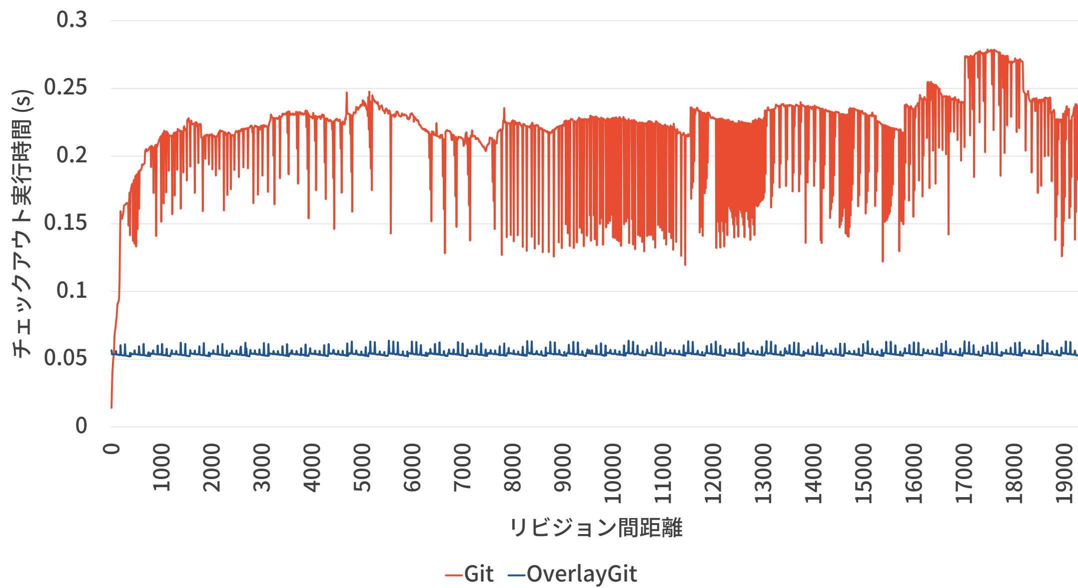

図 16: Closure Compiler のチェックアウト実行時間

6.3 実践的な評価

前節で紹介した実験では，提案手法のチェックアウトが Git より高速であることを確認した．しかし，提案手法が実際の Git を使った作業において役立つか，実践的な評価が不十分である．そこで，本研究では実践的な評価を行うため，マイニング作業を模した実験を行った．MSR では複数のリビジョンにチェックアウトし，ソースコードを静的または動的に解析する操作が行われることがある [29][30][31]．そこで，本実験ではプロジェクトの複数のリビジョンにチェックアウトし，静的解析する実験を行った．具体的には，100 コミットごとにリビジョンをチェックアウトし，各リビジョンにおいてサイクロマティック複雑度を計測した．サイクロマティック複雑度はソースコードの複雑度を表す指標であり，ソースコードの静的解析の代表例であるため，本実験ではこの指標を採用した．実験対象には Apache Commons Compress，Commons Math，Google Closure Compiler を用いた．

表 4: チェックアウト実行時間の平均

プロジェクト名	Git (s)	OverlayGit (s)
Apache Commons Compress	0.27	0.05
Apache Commons Math	0.17	0.05
Google Closure Compiler	0.21	0.05

表 5 に実験結果を示す．“checkout”，“measure”，“total” の列はそれぞれ，「実験全体でかかったチェックアウト実行時間の総和」「実験全体でかかったサイクロマティック複雑度計測時間の総和」「スクリプト全体の実行時間」を表している．表のとおり，Closure Compiler のチェックアウト実行時間は Git より OverlayGit の方が高速である．したがって，1 万コミットを超える大規模なリポジトリでは 100 コミット程度の移動でも Git より高速である．しかし，マイニング作業全体の実行時間を見ると Git の方が高速である．よって，100 コミット程度の移動しか実行しないマイニング作業では OverlayGit は実行時間を削減できないと考えられる．

表 5: メトリクス計測実行時間

プロジェクト名	Git の実行時間 (s)			OverlayGit の実行時間 (s)		
	checkout	measure	total	checkout	measure	total
Apache Commons Compress	2.0	84.7	86.7	3.4	86.2	89.6
Apache Commons Math	4.1	290.3	294.4	4.4	300.6	305.0
Google Closure Compiler	22.5	2,017.5	2,039.8	12.3	2,039.9	2,043.3

7 議論

7.1 提案手法の実用性

6.3 節で示した実験の結果では，提案手法は Git より長い実行時間を要した．しかし，この結果によって提案手法が Git より高速な動作が不可能であるとは考えていない．6.2.2 節で示した通り，チェックアウトするリビジョン間距離が数百以上の場合，OverlayGit が Git より高速にチェックアウト可能である．したがって，数百コミットを移動するチェックアウトが多いマイニング作業では，依然 OverlayGit が Git より作業を効率化できる可能性がある．

また，本研究ではマイニング作業のうち，実際にマイニングスクリプトを実行する部分のみを計測している．しかし，マイニング作業ではスクリプトを実行するまでに様々な試行錯誤がある．スクリプト作成中及び実行中には動作確認が必須であり，スクリプト実行中にミスを発見してやり直す場合がある．またマイニング結果によって調査方法を変更する場合も考えられる．よって，マイニングスクリプトを実行する以前の準備段階を考慮するとチェックアウトの回数はより多くなり，提案手法による高速化が期待できる．さらには，マイニング作業において各リビジョンで書き込みを伴う処理を行う場合，提案手法は Git より大きく高速化できる可能性がある．3.5 節で紹介した書き込みレイヤにより，提案手法ではリビジョンごとにキャッシュを残すことができる．このため，ビルドやテストを実行するマイニングでは，マイニング準備段階及びスクリプト実行段階において，Git より大きく高速化できる可能性がある．

さらに，提案手法は実装言語の変更によってさらなる高速化が期待できる．本研究では OverlayGit を Python を用いて実装している．Python はスクリプト言語であり，Git を実装している C 言語や C++ 言語等のコンパイル言語よりも動作が遅くなる場合が多い [32]．そのため，OverlayGit を C 言語等のコンパイル言語で実装すると，より高速な動作を実現できる可能性がある．

7.2 ベースレイヤ実装とネスト実装の比較

4.3.2 節では，ベースレイヤ実装以外のレイヤ数制限解決策の 1 つとしてネスト実装を紹介した．本研究では，ネスト実装の試作を用意し，簡単に空間計算量・時間計算量について評価実験を行った．以下に結果を紹介する．なお，評価実験では Apache Commons Compress を対象とした．

まず，ネスト実装による提案手法の前処理を行ったところ，リポジトリサイズが 176MB から 733MB，約 4 倍に増加した．ベースレイヤ実装では 7.2 倍の増加であったことを考慮すると，ネスト実装はベースレイヤ実装より空間計算量を抑えられている．また，前処理の実行時間を計測したところ 10.4 秒であり，ベースレイヤ実装の 18.2 秒より約 8 秒高速である．以上の結果から，ネスト実装はベースレイヤ実装より空間計算量及び前処理の時間計算量が小さい手法になっていると考えられる．これは，4.3.2

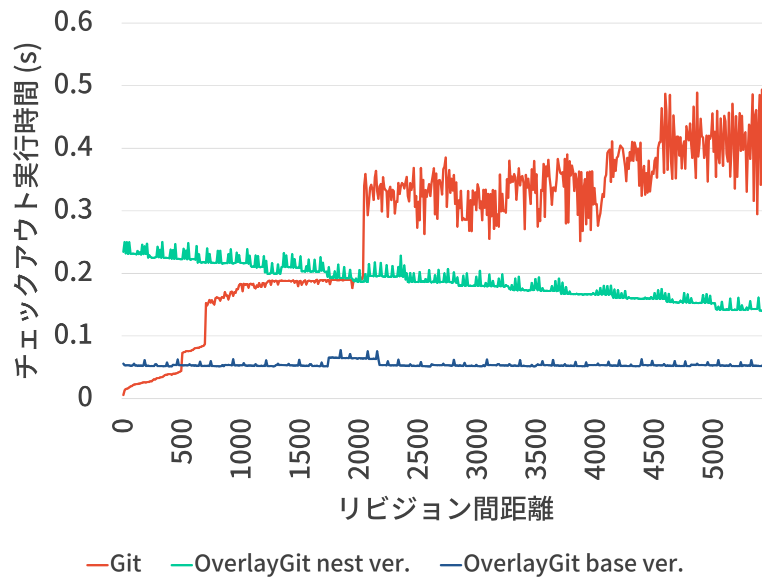

図 17: ネスト実装のチェックアウト実行時間

節で紹介した通り，ネスト実装はベースレイヤ実装に比べて余分なレイヤを作成しないためである．

続いて，チェックアウト実行時間についての結果を 図 17 に紹介する．図の通り，ネスト実装による提案手法でも，リビジョン間距離が一定以上の時に Git より高速なチェックアウトが実現できている．また，ネスト実装はベースレイヤ実装より実行時間が遅くなっている．これは，ベースレイヤ実装より OverlayFS のマウント回数が多いからである．

以上の結果からベースレイヤ実装とネスト実装を比較すると，空間計算量はネスト実装の方がよく，チェックアウトの時間計算量はベースレイヤ実装の方がよい．したがって，使用用途に応じて 2 つの実装を使い分ければ，空間・時間計算量についてさらに効率的なマイニング作業が実現できると考えられる．

8　妥当性への脅威

6 節で紹介した評価実験について妥当性への脅威が存在する．本研究では 10 個の OSS プロジェクトを実験対象としたが，他プロジェクトのリポジトリを用いた場合，異なる結果が得られる可能性がある．提案手法は前処理で圧縮ファイルを展開するが，ファイルの種類によって展開後のファイル容量は変化する．また，リポジトリ内のファイル数によって圧縮ファイルの展開時間は変化する．したがって，提案手法の空間・時間計算量は適用するリポジトリによって大きく変化する可能性がある．

また，本手法では OverlayFS によってコミットレイヤの統合を行ったが，AUFS を利用してレイヤの統合を行う手法も考えられる．AUFS は OverlayFS と同様，複数のディレクトリを重ね合わせて統合できるファイルシステムである．AUFS は OverlayFS と異なり，Linux カーネルには含まれていない．しかし，OverlayFS と同様に Linux カーネルに AUFS を組み込めば，OverlayFS より高速な動作が実現できる可能性がある．

9 展望

提案手法の前処理はより効率化できると考えている．現在の前処理では，対象リポジトリのコミット全てをレイヤに展開している．しかしマイニング作業においても，リポジトリの最新コミットから最初のコミットまで全てを対象にする状況は少ない．したがって，ユーザが使用したい範囲を指定して展開できれば，空間計算量・前処理の時間計算量の双方を削減できる．

さらに，現状ではチェックアウト処理の高速化のみを実現しているが，Git の他サブコマンドも高速化できる可能性がある．例えば `git-status` は，Git のワーキングディレクトリにおいて変更があったファイルをリストアップするコマンドである．Git では，以前のワーキングディレクトリの状態と `git-status` 実行時のワーキングディレクトリの状態を比較し，変更があるファイルを検知する．変更の検知は 1 ファイルごとに行うため，ワーキングディレクトリにファイルが多数あるとき，`git-status` は非常に遅くなる可能性がある．これに対し提案手法では，チェックアウト後のワーキングディレクトリへの書き込みは書き込みレイヤに行われる．したがって，書き込みレイヤにあるファイルを参照するだけで，どのファイルに変更があったか検知可能である．このため，`git-status` コマンドの動作を高速化できる可能性がある．

10 おわりに

本研究では，Git のチェックアウト処理を高速化する OverlayGit を提案した．OverlayGit は前処理としてコミットを展開し，OverlayFS を用いたレイヤ統合によって 1 つのリビジョンを構成する．提案手法の可能性を探るため，OverlayFS 自体の性能について評価を行い，OverlayFS は高速にマウント可能であることを確認した．また，提案手法の性能調査のため，空間・時間計算量について評価を行った．結果，OverlayGit は空間・時間計算量の双方で効率的であることを確認した．また実践的な評価では，大規模なリポジトリにおけるチェックアウト処理のみ，Git より高速に動作することを確認した．

今後の課題として, Git の他コマンドへの影響調査が考えられる. マイニング作業では, `git-checkout` のほかに `git-diff` や `git-blame` コマンドが頻繁に利用される．したがって，マイニング作業に与える影響を調査するためにはこれらのコマンドへの影響調査が必要不可欠である．

また，今後の追加実験としてディスク IO についての評価も考えられる．Git におけるチェックアウト処理はワーキングディレクトリの書き換えを伴い，大量のディスク IO が発生する場合がある．これに対し提案手法では，チェックアウトが OverlayFS のマウントによって実現されるため，ディスク IO がほぼ発生しない．このため，提案手法はディスク IO の観点からも Git のチェックアウト処理を改善できる可能性がある．したがって，ディスク IO の評価実験が必要である．

謝辞

本研究を進めるにあたって，多数の方にご助力いただきました．この場をお借りして，感謝の意を述べさせていただきます．

楠本真二教授には，研究発表の練習を通じて様々な助言をいただきました．いただいた助言によって新たな視点での気づきが生まれ，研究をより深く価値のある内容にできました．また，配慮の行き届いた研究室運営によって，何不自由なく研究室生活を送ることができました．研究が佳境に入った時には励ましの言葉をもらい，辛い時期を乗り切る活力になりました．

柗本真佑准教授には，研究のアイデアから論文の書き方に至るまで，研究のすべての場面において多数のご助力をいただきました．多角的な知識・経験を熱心に教えていただき，研究が大きく進歩しただけでなく，私自身が大きく成長することができました．出張時にはご飯に誘ってくださったり，研究発表前には激励の言葉をもらい，精神的にも心強いサポートをしていただきました．研究者として目指すべき模範を体現してくださり，学部 4 年時から 3 年間，常に私の目標でした．

ソフトウェア工学講座の肥後芳樹教授をはじめ，情報科学研究科の先生方には多数の助言・知識を与えていただきました．所属研究室と異なる研究室で発表した時には，他分野からの新しい視点に驚かされ，所属研究室では得難い気づきが得られました．また，基礎工学部での 4 年間や情報科学研究科で 2 年間学んだ経験は，研究を支える基盤になっていると強く感じています．

楠本研究室事務補佐員の橋本美砂子様には，事務の方面から手厚いサポートをいただきました．出張の手続きから研究室の備品管理まで様々な面できめ細かな気遣いをしていただき，事務処理に煩わされることなく研究の本質に没頭できました．また，事務室では研究と無関係な世間話をして，リラックスできるひと時を作ってくださいました．

研究室の学生の方々は，切磋琢磨し学びあう学友として，また苦楽をともにする仲間として，研究生活に色どりを与えてくださいました．先輩方の教えや研究成果は私の道しるべとなり，進むべき方向に迷った時，いつも先輩方が参考になりました．研究室の後輩の方々はとても優秀で，逆に学ぶことがとても多く，また優秀な方々の先輩として恥じないようにすることで，よい緊張感を持てました．

最後に，研究室の同期の方々には，修士 2 年間を通して多くの気づきとエネルギーをいただきました．研究で悩みがあれば大小関係なく相談しあい，学問的に興味深いアイデアを多数共有できました．研究外の話を語らい，時には食事や旅行に行き，楽しく実りある時間を過ごすことで，また研究に熱中するための英気を養えました．アイデアが浮かばず諦めたくなった時や，成果が思うように出ず苦しかった時も辛抱強く研究を進められたのは，研究室の同期の方々の助力があったからだと感じています．

末筆ながら，本研究を支えてくださったすべての方に心より感謝いたします．

参考文献

[1] Stack Overflow, : Stack Overflow Developer Survey 2022, `https://survey.stackoverflow.co/2022/#version-control-version-control-system` (Accessed at 2024-11-25).

[2] Dabic, O., Aghajani, E. and Bavota, G.: Sampling Projects in GitHub for MSR Studies, in *International Conference on Mining Software Repositories*, pp. 560–564 (2021).

[3] Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T. and Marinov, D.: DeFlaker: Automatically Detecting Flaky Tests, in *International Conference on Software Engineering*, pp. 433–444 (2018).

[4] Hashemi, N., Tahir, A. and Rasheed, S.: An Empirical Study of Flaky Tests in JavaScript, in *International Conference on Software Maintenance and Evolution*, pp. 24–34 (2022).

[5] Zeng, Z., Xiao, T., Lamothe, M., Hata, H. and McIntosh, S.: A Mutation-Guided Assessment of Acceleration Approaches for Continuous Integration: An Empirical Study of YourBase, in *International Conference on Mining Software Repositories*, pp. 556–568 (2024).

[6] Just, R., Jalali, D. and Ernst, M. D.: Defects4J: a database of existing faults to enable controlled testing studies for Java programs, in *International Symposium on Software Testing and Analysis*, pp. 437–440 (2014).

[7] Saha, R. K., Lyu, Y., Lam, W., Yoshida, H. and Prasad, M. R.: Bugs.jar: a large-scale, diverse dataset of real-world Java bugs, in *International Conference on Mining Software Repositories*, pp. 10–13 (2018).

[8] Luo, Q., Hariri, F., Eloussi, L. and Marinov, D.: An empirical analysis of flaky tests, in *International Symposium on Foundations of Software Engineering*, pp. 643–653 (2014).

[9] Lam, W., Muşlu, K., Sajnani, H. and Thummalapenta, S.: A study on the lifecycle of flaky tests, in *International Conference on Software Engineering*, pp. 1471–1482 (2020).

[10] Vitalis, S. and Diomidis, S.: RepoFS: File system view of Git repositories, *Journal on SoftwareX*, Vol. 9, pp. 288–292 (2019).

[11] Gousios, G. and Spinellis, D.: GHTorrent: Github's data from a firehose, in *Working Conference on Mining Software Repositories*, pp. 12–21 (2012).

[12] Allamanis, M. and Sutton, C.: Mining source code repositories at massive scale using language modeling, in *Working Conference on Mining Software Repositories*, pp. 207–216 (2013).

[13] Tsantalis, N., Ketkar, A. and Dig, D.: RefactoringMiner 2.0, *Transactions on Software Engineering*, Vol. 48, No. 3, pp. 930–950 (2022).

[14] Nakamaru, T., Matsunaga, T., Yamazaki, T., Akiyama, S. and Chiba, S.: An Empirical Study of Method Chaining in Java, in *International Conference on Mining Software Repositories*, pp. 93–102 (2020).

[15] Lin, H. Y., Thongtanunam, P., Treude, C. and Charoenwet, W.: Improving Automated Code Reviews: Learning From Experience, in *International Conference on Mining Software Repositories*, pp. 278–283 (2024).

[16] Wu, J., Bao, L., Yang, X., Xia, X. and Hu, X.: A Large-Scale Empirical Study of Open Source License Usage: Practices and Challenges, in *International Conference on Mining Software Repositories*, pp. 595–606 (2024).

[17] Oliver, P., Dietrich, J., Anslow, C. and Homer, M.: CrashJS: A NodeJS Benchmark for Automated Crash Reproduction, in *International Conference on Mining Software Repositories*, pp. 75–87 (2024).

[18] Casalnuovo, C., Suchak, Y., Ray, B. and Rubio-González, C.: GitcProc: a tool for processing and classifying GitHub commits, in *International Symposium on Software Testing and Analysis*, pp. 396–399 (2017).

[19] Spadini, D., Aniche, M. and Bacchelli, A.: PyDriller: Python framework for mining software repositories, in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 908–911 (2018).

[20] Ma, Y., Bogart, C., Amreen, S., Zaretzki, R. and Mockus, A.: World of Code: An Infrastructure for Mining the Universe of Open Source VCS Data, in *International Conference on Mining Software Repositories*, pp. 143–154 (2019).

[21] Ksontini, E., Abid, A., Khalsi, R. and Kessentini, M.: DRMiner: A Tool For Identifying And Analyzing Refactorings In Dockerfile, in *International Conference on Mining Software Repositories*, pp. 584–594 (2024).

[22] Perez De Rosso, S. and Jackson, D.: What's wrong with git? a conceptual design analysis, in *International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pp. 37–52 (2013).

[23] De Rosso, S. P. and Jackson, D.: Purposes, concepts, misfits, and a redesign of git, *Journal on SIGPLAN Not.*, Vol. 51, No. 10, pp. 292–310 (2016).

[24] Just, S., Herzig, K., Czerwonka, J. and Murphy, B.: Switching to Git: The Good, the Bad, and the Ugly, in *International Symposium on Software Reliability Engineering*, pp. 400–411 (2016).

[25] Bulteau, L., David, P.-Y. and Horn, F.: The Problem of Discovery in Version Control Systems, *Journal on Procedia Computer Science*, Vol. 223, pp. 209–216 (2023).

[26] GitHub, : About large files on GitHub, `https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-large-files-on-github#repository-size-limits` (Accessed at 2025-01-16).

[27] Atlassian Support, : Reduce repository size, `https://support.atlassian.com/bitbucket-cloud/docs/reduce-repository-size` (Accessed at 2025-01-16).

[28] Docker Docs, : OverlayFS storage driver, `https://docs.docker.com/engine/storage/drivers/overlayfs-driver` (Accessed at 2025-01-16).

[29] Nagappan, N., Ball, T. and Zeller, A.: Mining metrics to predict component failures, in *International Conference on Software Engineering*, pp. 452–461 (2006).

[30] Paixão, M., Uchôa, A., Bibiano, A. C., Oliveira, D., Garcia, A., Krinke, J. and Arvonio, E.: Behind the Intents: An In-depth Empirical Study on Software Refactoring in Modern Code Review, in *International Conference on Mining Software Repositories*, pp. 125–136 (2020).

[31] Härtel, J. and Lämmel, R.: Operationalizing threats to MSR studies by simulation-based testing, in *International Conference on Mining Software Repositories*, pp. 86–97 (2022).

[32] Bugden, W. and Alahmar, A.: The Safety and Performance of Prominent Programming Languages, *Journal on Software Engineering and Knowledge Engineering*, Vol. 32, No. 05, pp. 713–744 (2022).