一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

OverlayGit: OverlayFS を用いた高速な Git ファイルシステムの試作

三原　公平†　　柗本　真佑†　　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{k-mihara,shinsuke,kusumoto}@ist.osaka-u.ac.jp

あらまし　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．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．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）という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．提案ツールと Git の比較実験の結果，提案ツールが Git より高速にチェックアウトを実行できることを確認した．

キーワード　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

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．ソフトウェア開発者の 93% 以上が Git を使用しているという調査結果もある(注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 [1]．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 [2] [3] [4].

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，開発履歴や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．

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 [5] [6]．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 [7] [8]．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(注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を可能にすることで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 [9]．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場合には効果的である．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場合には適さず，高速化が難しい．チェックアウト処理の計算コストを抑える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` コマンド等を利用して，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，空間及び時間計算量の双方について効率的なチェックアウトを実現する OverlayGit を提案する．OverlayGit はまず前処理として，対象プロジェクト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だけを統合することで，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．

(注1)：`https://survey.stackoverflow.co/2022`

(注2)：`https://github.com/microsoft/scalar`

まず時間計算量について，OverlayGit は前処理を行うことで，`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，展開したコミットの統合に OverlayFS を用いることで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統合して 1 つのディレクトリに見せることができるファイルシステムである．OverlayFS を用いることで，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，平均では OverlayGit が Git の 1.5 倍高速にチェックアウトを実行できることを確認した．

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 [1].

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 [2] [3] [4]．MSR 研究の分野では，ソースコード自体を対象にした研究 [10] 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 [11] [12] 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 [5] [13] や MSR をサポートするツール [14] [15] 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 [9] [16] [17] 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）という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．

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 [5]．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を可能にすることで，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ような操作は 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 [9]．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．

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として行っておくことで，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

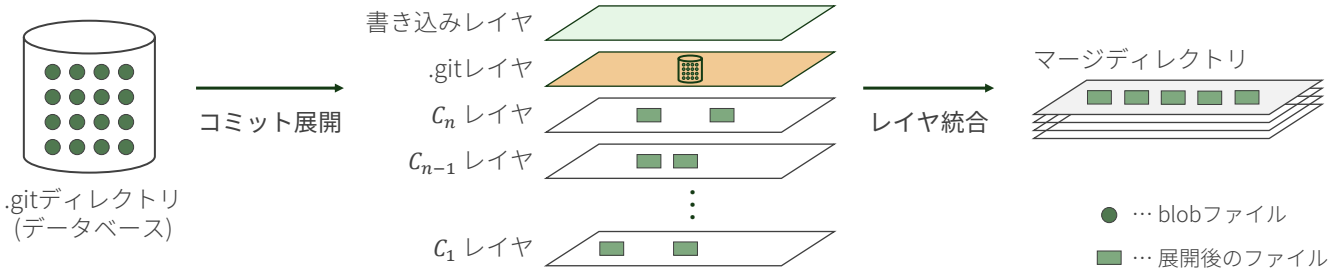


図 1: OverlayGit

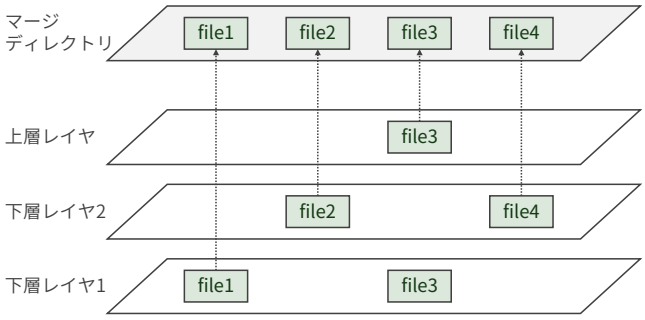


図 2: OverlayFS によるレイヤの統合

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 のリポジトリはなるべく小さくすることが推奨されており(注3)，最大級のリポジトリである Linux カーネルプロジェクトでも圧縮状態で 7GB 程度である(注4)．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ことができ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．

3.2 OverlayFS

OverlayFS は複数のディレクトリを論理的に統合して 1 つのディレクトリに見せることが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．最初，file1 〜4 は 3 つのレイヤに分散して存在している．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ユーザ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場合，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．

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，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．その時，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とが分かる．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．

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 というディレクトリがすべてのデータを保存するデータベースとなっている．コミット展開処理では，このデータベース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(注5)．

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 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 C_n から C_1 に対して OverlayFS をマウントすることを考える．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．

(注3)：https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-large-files-on-github
(注4)：https://github.com/torvalds/linux
(注5)：https://libgit2.org

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することで，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残すことができる．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，チェックアウトしたリビジョンにキャッシュを残すことができれば，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することで，提案手法での Git サブコマンドの速度を高めている．

4. 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

本調査のため，単純なディレクトリ構造を用意して実験を行った．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．

4.1 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 3 に結果を示す．統合するレイヤ数に対してマウント実行時間は線形に増加しているが，レイヤ数が最大の場合においても約 0.04 秒程度と高速である．提案手法ではマウントするレイヤ数とリポジトリのコミット数が対応するため，この結果より 1 万コミット以下のリポジトリで高速にチェックアウトできる可能性が示された．

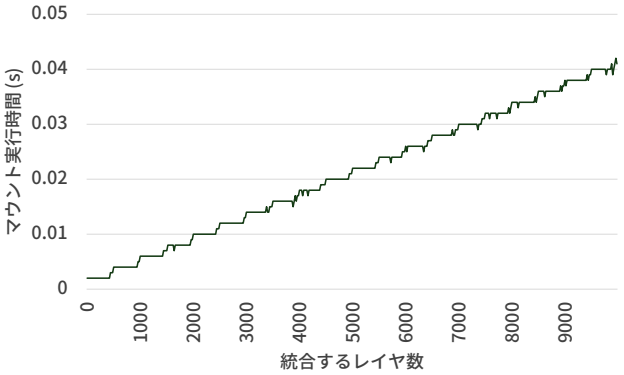


図 3: 統合するレイヤ数と OverlayFS マウント実行時間

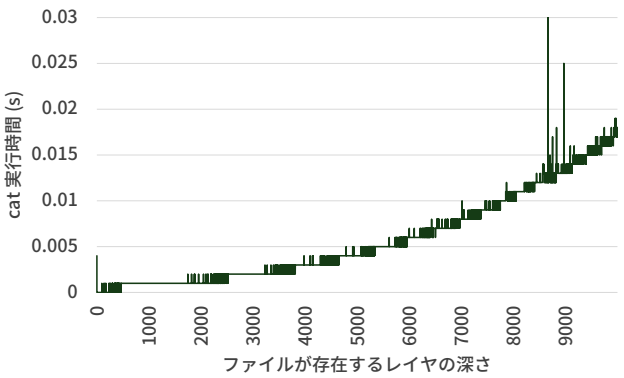


図 4: ファイルが存在するレイヤの深さと読み込み時間

4.2 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 4 に示す．グラフの形から，ファイルの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することが読み取れる．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 4 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，提案手法は十分高速にファイル読み込み可能である．

4.3 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 5 に示す．図より，`find` コマンドの実行時間は線形増加するが，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

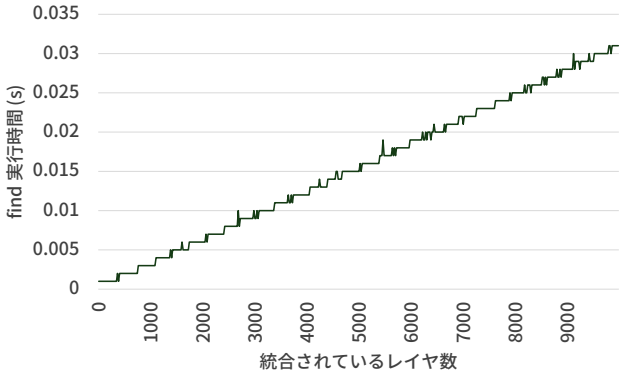


図 5: 統合されているレイヤ数とディレクトリ読み込み時間

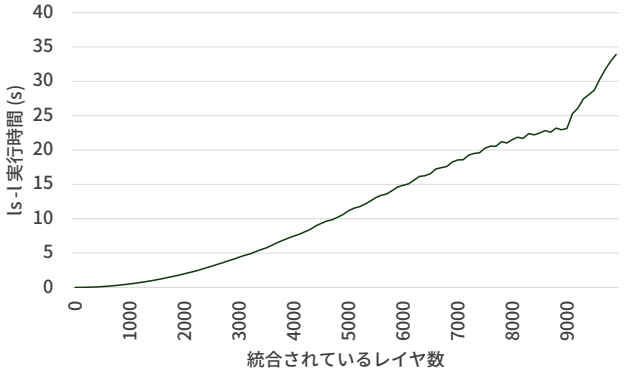


図 6: 統合されているレイヤ数とメタデータ読み込み時間

名を用いる操作が高速に実行できることが分かる．

4.4 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 6 に結果を示す．図より，ファイルの詳細なメタデータを取得する `ls -l` コマンドは非常に大きい時間的コストを要することが分かる．これは，`ls -l` コマンドが全レイヤのファイルを 1 つ 1 つ確認する処理を行う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(注6)．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，提案手法は全ファイルのメタデータを取得するよう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を確認することは少ない．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．

5. 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．提案手法の評価では，Apache Commons Compress(注7) を対象に実験を行った．Apache の OSS プロジェクトは多くの MSR 研究で対象になっており，Commons Compress も Defects4J というバグデータセット構築の MSR 研究で対象にされている [5]．Commons Compress はファイル圧縮ライブラリを提供するプロジェクトであり，本稿執筆時点でコミット総数は 5,487，スター数は 341 である．

5.1 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，元の Git リポジトリが 175MB に対し，前処理実行後のリポジトリは 733MB，約 4 倍となった．

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比較的小さいと考えられる．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(注8)．したがって，提案手法によってリポジトリサイズが 10 倍程度に増えた場合においても，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ことが多い．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．

5.2 時間計算量

提案手法の時間計算量の評価として，前処理及びチェックアウト処理の実行時間を計測した．まず前処理について結果及び考察を述べる．Commons Compress の Git リポジトリに対して前処理を実行したところ，10.4 秒で実行が終了した．前処理は 1 回しか実行しないことを考慮すると，10.4 秒という実行時間は十分に高速であると考えられる．

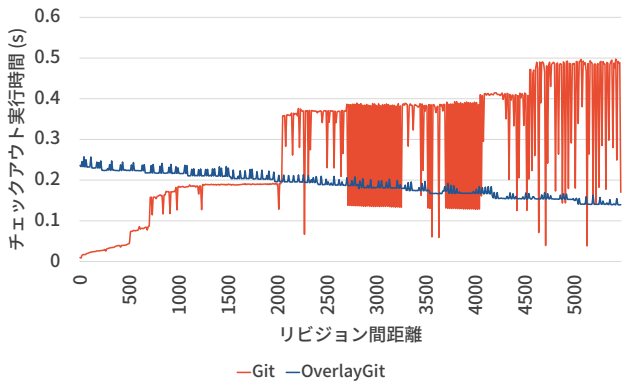
次にチェックアウト処理について計測結果及び考察を述べる．チェックアウトの計測では，最新から過去のリビジョンへのチェックアウトと，過去から最新リビジョンへのチェックアウトの 2 種類を計測した．結果を 図 7 に示す．Git に着目すると， 図 7(a) と 図 7(b) ともにチェックアウトするリビジョン間距離が大きくなるほど実行時間が増加している．リビジョン間距離が大きくなるほど実行時間が大きくなる原因は，リビジョン間距離に応じてワーキングツリーの差分が大きくなるためである．

Git に対して，提案手法の実行時間はほぼ一定である．提案手法でリビジョン間距離が小さいほど実行時間が長い原因は，提案手法では最新リビジョン付近でのマウントが初期リビジョン付近でのマウントより計算コストが大きいからである．提案手法では 1 コミットにつき 1 レイヤを用意するため，最新付近のリビジョンではレイヤ数が多く，OverlayFS のマウント及びアンマウント実行時間が長い．このため，チェックアウト先またはチェックアウト元のリビジョンが最新に近づくほど，チェックアウトの実行時間が長くなる．

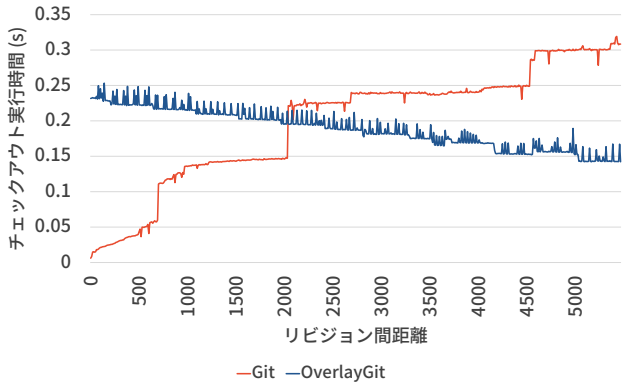
(注6)：https://docs.kernel.org/filesystems/overlayfs.html

(注7)：https://github.com/apache/commons-compress

(注8)：https://support.atlassian.com/bitbucket-cloud/docs/reduce-repository-size



(a) 最新リビジョンから過去のリビジョンへのチェックアウト



(b) 過去のリビジョンから最新リビジョンへのチェックアウト

図 7: リビジョン間距離とチェックアウト実行時間

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，最新から過去リビジョンのチェックアウトは 0.27 秒，過去から最新へのチェックアウトは 0.29 秒であった．これに対し，OverlayGit は最新から過去と過去から最新の両方で平均が 0.19 秒であり，Git より約 1.5 倍高速である．よって，提案手法は Git より高速なチェックアウトを実現している．

6. 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 を用いてコミットレイヤを統合することで 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．

今後の課題として，より実践的な状況での評価が考えられる．提案手法によるチェックアウト高速化は特に MSR において価値が大きいが，本研究では MSR に与える影響を十分に評価できていない．したがって，提案手法を用いて MSR を実行した場合にどのような影響が生じるか確認する必要がある．

さらに，提案手法の改善策をいくつか考案している．ここではより高速なチェックアウトの実現策を紹介する．提案手法では 1 コミットに 1 レイヤを用意しているが，複数コミットをまとめたレイヤをいくつか設置することで，チェックアウトをより高速に実行できる可能性がある．現在の手法では，チェックアウトするリビジョンより過去のコミットすべてを統合している．まとめレイヤを用意すれば，チェックアウトするリビジョンからまとめレイヤまでのコミットだけを統合すればよいため，チェックアウトがより高速になると考える．

謝辞 本研究の一部は，JSPS 科研費（JP24H00692, JP21H04877, JP21K18302）による助成を受けた．

文　献

[1] O. Dabic, E. Aghajani, and G. Bavota, “Sampling projects in github for msr studies,” International Conference on Mining Software Repositories, pp.560–564, 2021.

[2] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, “DeFlaker: Automatically detecting flaky tests,” International Conference on Software Engineering, pp.433–444, 2018.

[3] G. Gousios and D. Spinellis, “GHTorrent: Github’s data from a firehose,” Working Conference on Mining Software Repositories, pp.12–21, 2012.

[4] N. Tsantalis, A. Ketkar, and D. Dig, “RefactoringMiner 2.0,” Transactions on Software Engineering, vol.48, no.3, pp.930–950, 2022.

[5] R. Just, D. Jalali, and M.D. Ernst, “Defects4J: a database of existing faults to enable controlled testing studies for java programs,” International Symposium on Software Testing and Analysis, pp.437–440, 2014.

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

[7] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests,” International Symposium on Foundations of Software Engineering, pp.643–653, 2014.

[8] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, “A study on the lifecycle of flaky tests,” International Conference on Software Engineering, pp.1471–1482, 2020.

[9] S. Vitalis and S. Diomidis, “RepoFS: File system view of git repositories,” Journal on SoftwareX, vol.9, pp.288–292, 2019.

[10] T. Nakamaru, T. Matsunaga, T. Yamazaki, S. Akiyama, and S. Chiba, “An empirical study of method chaining in java,” International Conference on Mining Software Repositories, pp.93–102, 2020.

[11] H.Y. Lin, P. Thongtanunam, C. Treude, and W. Charoenwet, “Improving automated code reviews: Learning from experience,” International Conference on Mining Software Repositories, pp.278–283, 2024.

[12] J. Wu, L. Bao, X. Yang, X. Xia, and X. Hu, “A large-scale empirical study of open source license usage: Practices and challenges,” International Conference on Mining Software Repositories, pp.595–606, 2024.

[13] P. Oliver, J. Dietrich, C. Anslow, and M. Homer, “CrashJS: A nodejs benchmark for automated crash reproduction,” International Conference on Mining Software Repositories, pp.75–87, 2024.

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

[15] Y. Ma, C. Bogart, S. Amreen, R. Zaretzki, and A. Mockus, “World of Code: An infrastructure for mining the universe of open source vcs data,” International Conference on Mining Software Repositories, pp.143–154, 2019.

[16] S. Perez De Rosso and D. Jackson, “What’s wrong with git? a conceptual design analysis,” International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, pp.37–52, 2013.

[17] S.P. De Rosso and D. Jackson, “Purposes, concepts, misfits, and a redesign of git,” Journal on SIGPLAN Not., vol.51, no.10, pp.292–310, 2016.