Githubのコミット履歴にもとづくCOPY処理の分割によるDockerイメージBuild時間の短縮

遠藤 睦実[1]　高橋 風太[2]　串田 高幸[1]

概要：Docker Build におけるキャッシュの仕様では，前回の Dockerfile と今回の Dockerfile で変更点が無いか確認し，変更点がある直前まで前回の Build 結果をそのまま今回の Build 結果として使用する．一部ファイルの編集により該当の COPY を行うレイヤーがキャッシュできなくなった場合，他の編集されていないファイルのキャッシュが使用できなくなることでイメージの Build 時間が増加する課題がある．この課題を解決するため，Dockerfile の一度に複数ファイルをコピーする記述を，ファイルのコミット回数で 4 分割してコピーするように変更する．4 分割することで，放置されたファイルをキャッシュしながら更新頻度が低いファイルのキャッシュを試みると共に，更新頻度が高くキャッシュが破棄されやすいファイルを後からコピーすることができる．評価として，提案を適用する前後の Dockerfile でキャッシュを利用した Build を行い，Build に要した時間の差を比較する．実験は doge-unblocker というリポジトリを対象に，提案を適用する前後の Dockerfile で各 5 回 Build を行った．適用前は平均 Build 時間が約 3.7 秒となり，提案を適用すると平均 Build 時間は約 3.7 秒となった．一部のファイルがキャッシュを利用できるようになったことでコピー自体は早くなったが，引数が多くなりコピー命令の処理が遅くなる結果となった．

1. はじめに

背景

Dockerfile を記述する際には，イメージのサイズ増加やセキュリティホールの発生を招かないような書き方を心掛ける必要がある [1, 2]．そのために Dockerfile を記述する際に行うべき事柄は，ベストプラクティスという形で研究機関やエンジニアが共有を行っている [3]．ベストプラクティスの 1 つに，前回の Build と比べて処理内容が変化する可能性が高いコマンドを他のコマンドよりも後に配置するように，イメージのレイヤー順を変更するものがある．これは，Docker で Build する際に Docker 独自の仕様を持つキャッシュシステムを有効に活用するためである．Dockerfile を利用した Build におけるキャッシュは，Build したイメージのレイヤー単位で管理されている [4, 5]．前回の Dockerfile と今回の Dockerfile が同一であるか確認し，もし変更がある場合は，前回使用した Dockrfile と同じ処理を行う部分までは前回の Build 結果をそのまま利用し，変更点から新しく Build を行う仕様である[*1]．このとき，変更点以降で前回の処理内容と一致するキャッシュが存在しても，キャッシュが使用されることはない．前のレイヤーが変化したことで後のレイヤーを Build した際に結果が変化する可能性があるためである．そのため Docker の Build において COPY 以外のコマンドは，前回と同一のコマンドである限り，前回実行したコマンドの処理結果は次の Build 時にキャッシュとして使用される．COPY コマンド処理のキャッシュのみは例外である．COPY でコピーされるファイルが，前回の Build 時にコピーされたファイルとチェックサムが同一という条件を満たさなければキャッシュを使用することができない．そのため，Docker の Build において，COPY コマンドのキャッシュは他のコマンドと比較して非常によくキャッシュが破棄されるという特性を持つ．この特性によりキャッシュを用いた Build の時間が遅延しないように，ファイルのコピーは可能な限り Build の後半で行われる必要がある．COPY 以外のコマンドを実行する前にファイルをコピーする必要がある場合は，その地点で最低限必要なファイルのみをコピーし，一連の処理を終えた後に残りのファイルをコピーするように記述する [6, 7]．

課題

Docker の Build における COPY 処理にて，本来使用

[1] 東京工科大学コンピュータサイエンス学部
〒 192–0982 東京都八王子市片倉町 1404-1
[2] 東京工科大学大学院バイオ・情報メディア研究科コンピュータサイエンス専攻
〒 192–0982 東京都八王子市片倉町 1404-1

[*1] Dockerfile リファレンス 24.0(Docker 公式)
https://docs.docker.jp/engine/reference/builder.html

可能であるファイルキャッシュが使用できないために，キャッシュを使用する Build が遅くなるという課題がある．”COPY ./ ./”に代表されるような複数ファイルを 1 コマンドでコピーする処理を行う場合，その中の 1 つでもチェックサムが異なるファイルが存在すると，その処理でコピーするファイル全てのキャッシュが使用不可になるためである．

図 1 は，package.json を使用して yarn でインストールを行う Node コンテナを例とした，js ファイルの編集によって他ファイルのキャッシュが破棄される条件を表したものである．背景が暗くなっている部分が Build 時にキャッシュが破棄されるレイヤーである．図 1 左のように，3 つのファイルを同時にコピーした場合，そのレイヤーのチェックサムが合わないことで全てのファイルキャッシュが破棄される．そのため，単体ではチェックサムが合致しキャッシュが使用可能である他 2 つのファイルもキャッシュが破棄され，再度コピーされる．図 1 右のように，編集されたファイルを編集の無いファイルの次にコピーした場合，先にコピーする編集の無い 2 ファイルのチェックサムが合致するため，キャッシュを使用することができる．

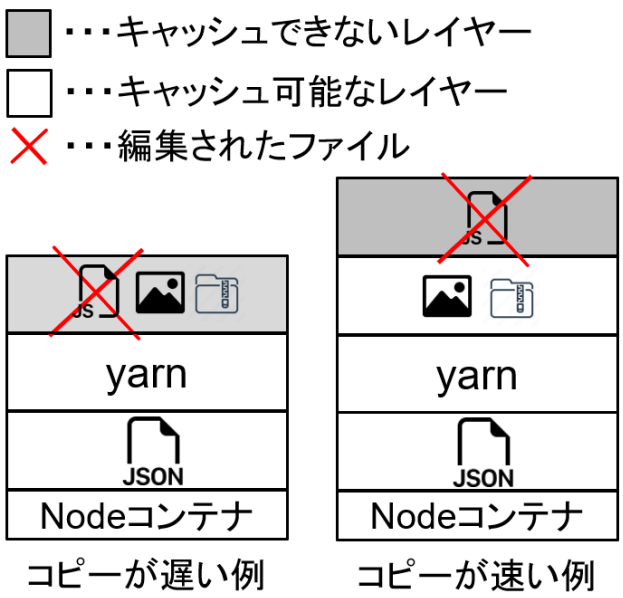

図 1 Build 時におけるファイルキャッシュの問題点

約 10MB のリポジトリを対象に，編集のあったファイルを他のファイルと同じレイヤーでコピーする場合と他のファイルよりも遅いタイミングでコピーした場合で，コピー時間及び Build 時間に差が出るのかを検証する実験を行った[*2]．結果は，他のファイルと同じレイヤーでコピーを行うと Build 時間は 4.9 秒，他のファイルよりも後にコピーを行うと Build 時間は 3.5 秒となった．コピー時間のみで見れば 1.6 秒から 0.2 秒に短縮された．編集のあったファイルのみを後からコピーすることで，実際にファイルをキャッシュできた分だけ Build 時間が短縮できることを確認できた．

[*2] https://drive.google.com/file/d/1lLXKurZXG36jVmk3Ig4D$_i$ceAhGi3dnV/view

各章の概要

本テクニカルレポートは以下のように構成される．第 2 章では，本稿の関連研究について述べる．第 3 章では，本稿で挙げた課題を解決するための提案について述べる．第 4 章では，提案した手法の実装について述べる．第 5 章では，提案手法に対しての実験内容と，その評価について述べる．第 6 章では，提案手法の議論を述べる．第 7 章は，本稿のまとめである．

2. 関連研究

Shipwright という，主に外部環境の変化により同一の dockerfile を使用したにもかかわらず，環境の更新を境に build 時の挙動が変化し，その結果 build 時に失敗する状態になった Dockerfile を，キーワードと解決方法をクラスタリングすることで問題を修正するソフトウェアを提案した研究がある [8]．この研究は，Dockerfile の中身をより処理内容の変化が起きづらい書き方に置き換えることで，Build に失敗する Dockerfile を修復し，可搬性と不変性を確保する事が目的である．しかしこの研究では Build 失敗の原因になるコードのみを対象にするため，Build 自体には成功するが不適切なコーディングがされている Dockefile の最適化は行わない．

RUDSEA という，プロジェクトのソースコードから環境に依存するコードのみを抽出し，それを元に現在のプロジェクトに合った依存関係を求めることで，次の正しい更新内容を生成して Dockerfile に反映するソフトウェアを提案した研究がある [9]．この研究は，プロジェクトの進行中に外部環境の更新もしくはプロジェクトの依存関係変更が発生した際に，同時に Dockefile も依存関係更新後の環境を Build できるように変更を加えることが目的である．外部内部問わず Dockefile で Build する環境を自動で更新し続けることで，ユーザの負担と更新忘れによる Build 失敗を減らすことができる．しかしこの研究は Dockefile の更新が目的であるため，Dockerfile の最適化までは行っていない．

インタープリタ言語を扱うイメージにおいて，変更の無いレイヤーのリ Build を回避し，キャッシュを使用できるようにする手法を提案する研究がある [10]．この研究は，手前のレイヤーキャッシュが破棄されると，それ以後のレイヤーキャッシュが単体で見れば使用できる場合でも，前レイヤー変更によって処理内容が変化する可能性があるために使用できない問題を回避することが目的である．インタープリタ言語に限れば，手前のレイヤーキャッシュが破

棄されると以後のレイヤーキャッシュが使用できない問題を回避して，キャッシュを使用する Build では Build 時間を短縮することができる．しかしこの研究ではインタープリタ言語の特性を利用しているため，コンパイル言語などのインタープリタ言語以外を扱うユースケースは対象にできない．

DockerMock という，Dockerfile のコマンドラインまたはシェルコマンドの一部をダミーに置き換えることで，Dockerfile の Build 障害をコンテキストベースで検出するソフトウェアを提案した研究がある [11]．この研究は検出精度と再現率を高く保ちながら Build 障害を素早く検知することが目的である．既存研究と比較してより少ない Build 回数で Build 障害となった原因を特定することが可能になり，Build に失敗する際の修復までの速度が速くなる．しかしこの研究では Build 処理の完遂を妨げるコードの特定が目的であるため，Build 時に失敗にはならないが Dockerfile の品質を下げるようなコードの特定は行わない．

3. 提案

提案方式

本稿では課題に対し，1 つのコマンドでコピーする処理を，複数コマンドにわたりコピーするように分割した Dockerfile を新しく出力するソフトウェアを提案する．対象はユーザの書いた Dockerfile の内部に存在するディレクトリを COPY する処理とし，判断基準は Github のコミット履歴とする．本提案方式では，前回の Build で作成されたファイルのキャッシュを次回の Build でも使えるようにするため，Dockerfile 内部にて更新頻度の高いファイルを Build 後半へ並び替える．

図 2 は提案方式の図である．ファイル D の更新によってファイル A〜D のキャッシュが破棄されている状態である．1 度に全ファイルがコピーされている状態から，ファイルの更新頻度に基づき段階に分けてコピーするように変更する．そうすることにより，ファイル D の更新によってファイル A〜C のキャッシュが破棄されなくなる．

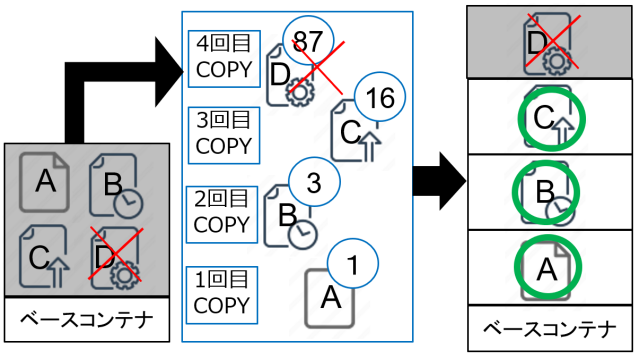

図 2 ファイルのグループ分け

ファイル更新頻度の指標として Github に保存されているコミット履歴を用いる．ファイルの更新頻度はある期間のうちに何回ファイルの更新が行われたかで求めることができる．そのため，いつファイルの更新が行われたかのデータがあれば，ある期間にマッチする更新データをカウントすることでファイルの更新頻度を求めることができる．しかし，ローカル環境にてユーザが何回ファイルを保存したかを指標として用いると，ユーザがファイル保存を行う際に何回か連続で保存する場合がある．この場合，実際にファイルを保存したいとユーザが考えた回数よりも多い数値がデータとして取得される．また，人によってファイル保存間隔は異なるため，複数人が関わる場合は保存間隔にばらつきが発生する．したがって，この方法では正確な頻度を求めることができない．Github に保存されているコミット履歴を指標として用いれば，Github にファイルを Push するタイミングはそのファイルが進化したとユーザ判断してから 1 度だけ行われる．そのため，保存間隔の違いや連続保存によりファイル更新の頻度が実際よりも多い値を取得する可能性を下げることができる．また，ローカルへの保存回数を用いる場合ではローカル環境に置かれたプロジェクトを常に監視する必要があるが，GIthub にコミットした回数を用いる場合であれば Git を利用するプロジェクトであれば追加のアクションを起こさずに更新情報が Github にコミット履歴として残るため，監視を行う必要が無いというメリットがある．

次に示す図 3 は，Github 上で 5 件のリポジトリに対して各ファイルのコミット回数を取得し，割合の平均を表したものである．全体のコミットは約 100〜約 500 回である．

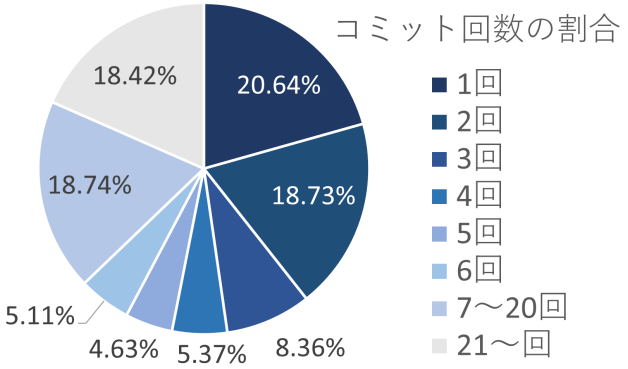

図 3 Github 上での各ファイルのコミット回数割合

調査の結果，コミット回数が 1 回のみのファイルだけでリポジトリの約 20%を占めていた，コミット回数 3 回以下のファイルはリポジトリの約 48%，コミット回数 4 回以下のファイルはリポジトリの約 53%であった．コミット回数が 2〜3 回以下のファイルはリポジトリの約 27%，コミット回数 2〜4 回以下のファイルはリポジトリの約 32%であった．コミット回数が 20 を超えるファイルはリポジトリの約 18%を占めている．コミット回数が 5〜20 の範囲

であるファイルは約 28%であり，7〜20 の範囲では 18%であった．

図 3 の調査結果を元に，図 2 に示す通り，ファイルは 4 グループに分割する．下から順に，コミット回数が 1 以下のグループ，コミット回数が 4 以下のグループ，他 3 つのグループに分類する条件にどれも該当しないグループ，コミット回数上位全体の 20%を占めるグループである．

キャッシュに失敗すると以降のファイルキャッシュが全て使用不可になるため，先にコピーするファイルほどキャッシュできる確率が高くなるように配置する必要がある．1 回目コピーの条件であるコミット回数を 1 回に設定した理由は，1 から増えないということはユーザに放置されている可能性が高いからである．2 回目コピーの条件であるコミット回数を 2〜4 回に設定した理由は，全体のコミットが約 100〜約 500 回ほどある内の 2〜4 回しかないファイルが約 32%存在しており，これらのファイルをキャッシュできた場合は Build の高速化が見込めるからである．4 回目コピーの条件であるコミット回数を上位 20%に設定した理由は，コミット回数が 21 回を超えるファイルが約 18%存在しているからである．コミット回数が 21 回を超えるグループのコミット回数平均は約 34，中央値は 30 であり，全体の約 3 分の 1 のコミットにてコミットされていたファイルである．そのため，この条件に該当するファイルはキャッシュできない可能性が高いため，最後にコピーを行う．

図 2 の例では，提案適用前はコミット回数が 1 回という条件を満たすファイル A は 1 番最初にコピーする．コミット回数が 2〜4 回という条件を満たすファイル B は 2 番目にコピーする．コミット回数が全体の約 80%を占めるファイル D を 4 番目にコピーする．どの条件にも当てはまらないファイル C を 3 番目にコピーする．

ユースケース・シナリオ

本提案によって，Build を行う際に放置されたサイズの重いファイルが先にコピーされることで，Build 時間を短縮することができる．そのため本提案が使用されるケースとして，イメージの Build 時に数 MB 以上のファイルをローカルからコンテナへコピーしたい場合かつ，ファイルの更新を終えた際に毎回 Build を行いコンテナ内でテストを行う場合を想定している．

ユースケースとして，Python で実装した Web アプリケーションを Docker コンテナ内で動かし，プロジェクトは Github で管理し，ファイルの更新を終える度にテストを行う場合を想定する．図 4 にユースケース図を示す．

プログラムの内容は Python で開発しコンテナを使用する Web アプリケーションとし，Web アプリケーションの画面を表示するのにコンテナ内に用意した画像も用いるものとした．ユーザはローカルで開発・用意したファイルを

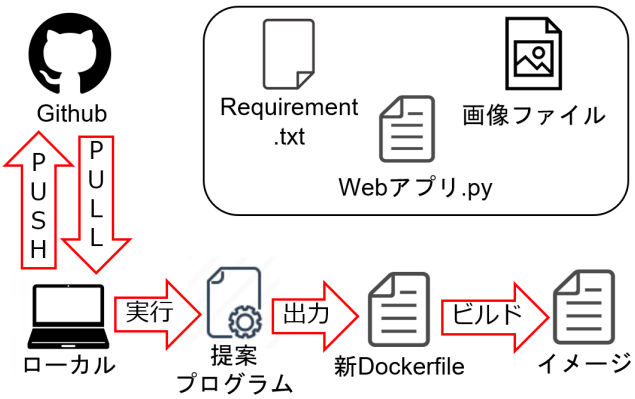

図 4 GitAPI へのアクセスにおける流れ

Github でバージョン管理を行う．編集内容を確定したファイルはリポジトリにコミットされ，リポジトリとローカルの間で全ファイルの内容に齟齬が無い状態を前提とする．Build はプログラム 1 に示す Dockerfile を利用して，ローカルで Build を行う．プログラムに使用するライブラリのインストール方法は，Requirements.txt と pip を用いるものとした．Requirements.txt にライブラリ名とバージョンを記述し，pip が一括でインストールを行う．Build の内容は，まず先に Requirements.txt に記述したライブラリ群を pip を用いてインストールを行い，その後ソースコードと画像ファイルをイメージへコピーするものである．

プログラム 1 requirements.txt を用いて Python 環境を構築する例

```
1 FROM python:3.11.2-slim-bullseye
2 COPY requirements.txt /app/requirements.txt
3 RUN pip install -r /app/requirements.txt
4 COPY . /app
5 WORKDIR /app
6 CMD ["python", "-u", "main.py"]
```

ユースケースに画像ファイルのイメージへのコピーを含めた理由は，画像は一度用意した後に更新されることは稀であり，かつファイルサイズが大きくなりやすいという性質を持つからである．Github 上で画像を扱うリポジトリ 3 件を調査した所，画像ファイルはプッシュされた後に編集されるケースが少なく，かつファイルサイズも合計で数 MB から数十 MB であった．

4. 実装

本提案を実現するために，以下の実装を行った．実装方法については，Python を用いてプログラムを作成した．

Dockerfile の COPY 処理分割

以下がプログラムの流れである．

① 対象にする Dockerfile のパスをユーザから受け取る
② Dockerfile から COPY 行を抽出
③ ファイル単体 COPY のみの処理であれば中断

④ 抽出した COPY 行と Dockerfile の置かれたディレクトリから，実際にコピーするファイルのリストを作成
⑤ ファイルのコミット日時とコミット回数を記した CSV ファイルをロード
⑥ ファイルのコミット日時とコミット回数に基づきファイルをグループ分け
⑦ 旧 Dockerfile をバックアップ
⑧ ファイルグループごとに COPY 処理を生成
⑨ 生成した COPY 処理を Dockerfile に追記

また，図 5 に本実装で行われる入力及び出力を表す．

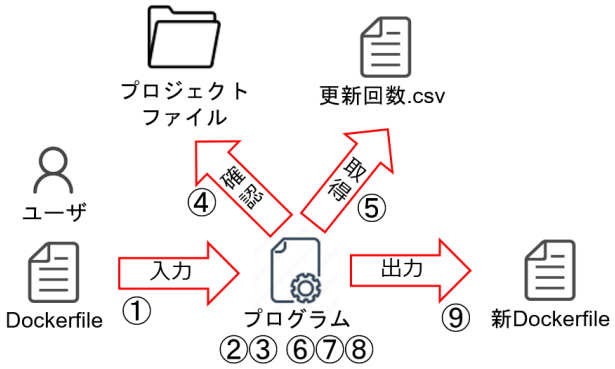

図 5 実装におけるアクセスにおける流れ

ユーザにはソフトウェアへ，提案適用対象にする Dockerfile のパスを入力する．Dockerfile が置いてあるディレクトリに置いてあるファイルとディレクトリのパスを全て取得し，Build 時にコピーするファイルのリストを作成する．パスではなく名前のみを記録すると，別ディレクトリの同名ファイルが存在した場合は競合する．そのため，ファイル名ではなく Dockerfile が配置してあるディレクトリをルートディレクトリとしたファイルパスを記録している．各ファイルのコミット回数を記録した CSV ファイルをロードし，プログラムが Dockerfile のあるディレクトリから検知したファイル全てと照らし合わせ，各ファイルのコミット回数を確認する．ファイルリストかコミット回数が 1 回のファイルを抜き出したリストを，前回の Build までのリストと今回の Build を含むリストの 2 つ作成する．前回の時点ではコミット回数が 1 回であったが，今回の Build ではコミット回数が 2 回以上であるファイルが存在するかどうかを確認する．条件を満たす場合は，提案で作成したファイルコピーコマンド全てを”COPY ./ ./”直前に追記する．条件を満たさない場合は提案の適用を行わない．Dockerfile 自身が更新されることでもキャッシュが破棄されるためである．コピーするファイルのリストを作成する際に，.dockerignore に記述されているファイルは除外する．

ユーザが意図的にあるファイルを他のファイルとは分けており，”COPY ./ ./”よりも前にコピーを行うように Dockerfile を記述している場合がある．ディレクトリではなくファイルをコピーしている行を変更または移動させると Build に失敗する可能性があるため，この条件に該当するファイルは本提案によってコピータイミングを変えないようにする．しかしこの条件を満たすファイルであっても，ユーザではなく本プログラムによって追加された COPY 行によるものであれば，元はユーザが後で一括でコピーしても問題無いと判断したファイルである．そのため，本プログラムが追加した COPY 行は”COPY ./ ./”とその前に登場する COPY 以外のコマンドとの間であれば，COPY の変更，移動，削除を行うことができる．この 2 つのファイル個別コピーを区別するために，本プログラムが何を個別にコピーしたのかを記録する．COPY 行からファイルリストを作成する際に，本ソフトウェアが変更もしくは追加したコマンドが何であるかを認識するために記録からファイルリストを読み出し，該当しないファイルは本提案の対象から除外する．

GitAPI を用いたコミット回数 CSV ファイルの作成

以下がプログラムの流れである．

(I) GitAPI でコミット履歴を取得
(II) コミット履歴からコミット詳細を 1 つ取得
(III) コミット詳細から日付とファイル名を取得
(IV) 日付・ファイル名共に初出ならば，それぞれリストに追加
(V) その日付でファイル名が登場した回数をカウント
(VI) 2〜5 の手順をコミット履歴の数だけ繰り返す
(VII) 最後に日付リストを見出し行，ファイル名リストを見出し列にし，その日のファイルコミット回数を記入した CSV ファイルを出力

また，図 6 にプログラムで行われる入力及び出力を表す．

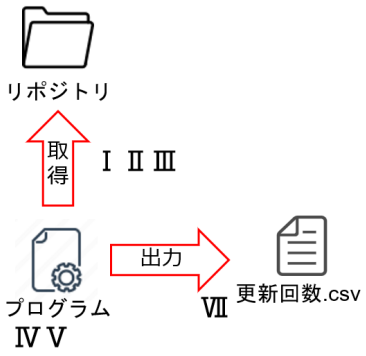

図 6 GitAPI へのアクセスにおける流れ

5. 実験

実験として，提案適用後の Docker を用いて Build を行い，どれだけ Build 時間が削減されたかを測定する．リポジトリに対して古いコミットから順に”git reset –hard”を

行うことで実際のプロジェクト進行をコミット単位の粒度で再現し，各コミット時で Build を行う．Build 時間の測定は以下の手順で行う．

① ”Git log”でコミット履歴を確認
② ”script”コマンドを用いて Docker Build のログを保存
③ コミット履歴の数だけ，次の 4〜6 の手順を繰り返す
④ ”git reset –hard コミット ID”を実行し，
当時コミットされた環境を再現
⑤ 提案プログラムの実行
⑥ Build を実行
⑦ ”exit”で”script”処理を停止
⑧ ログから”FINISHED”と書かれた行のみを抽出

実験環境

実験対象とするリポジトリとして，github から doge-unblocker[*3]を git から clone して用意する．このリポジトリはコンテナへ複数の画像をコピーしている．そのため，Build 時にローカルからイメージにコピーするファイルの合計サイズが約 10MB と，他の Dockrfile を使用する Github のリポジトリが約 1MB であることと比較するとサイズの重いリポジトリになる．ローカルからイメージへコピーするファイルの合計サイズが多ければ多いほど，ファイルをキャッシュできた時の短縮幅が大きくなることと，今回のユースケースに近い条件であるため，今回の実験で使用した．

実験結果と分析

図 7 が実験結果である．

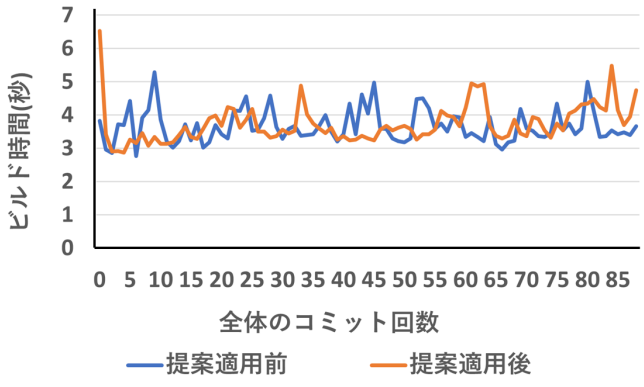

図 7 実験結果

適用前は平均 Build 時間が約 3.7 秒であり，提案を適用すると平均 Build 時間は約 3.7 秒となったため，全体で見た場合では Build 時間に差が出ない結果となった．キャッシュを利用できた Build では，提案を適用する前の Build と比べて提案を適用した後の Build の方が Build 時間が平均で約 13.6%早くなった．しかしキャッシュを破棄した Build では，提案を適用する前の Build と比べて提案を適用した後の Build では平均で約 16%遅くなる結果となった．そのため，一度用意されたファイルが放置され続けるケースでは，本提案を適用すると Build 時間を短縮できる．

キャッシュが使用できない場合では提案を適用した後の Dockerfile の方が Build 時間が遅くなる原因として，COPY に渡す引数が提案前よりも増えたことによる，COPY で行う前処理の時間増加がある．例として，同じ 1 つのディレクトリ内にある 4 つのファイルを 1 回でコピーを行う COPY 処理の場合，”COPY DIR ./”と書く方が”COPY DIR/fileA DIR/fileB DIR/fileC DIR/fileD ./”と書くよりも高速で COPY 処理が行われる．これによる Build 時間の増加が，キャッシュを使用可能になったことによるコピー時間の短縮と同じだったため，図 7 のような結果になった．また，最初の Build においては，提案適用後の方が Build に時間を要する結果となった．これは，この地点では全ファイルのコミット回数が 1 回であることが原因である．この場合は，提案によって最初にコピーするグループに全ファイルが分類されている．そのため，提案前と同じとなる全てのファイルを一度にコピーする処理でありながら COPY の前処理に要する時間のみが増加したため，Build 時間が増大した．

6. 議論

Dockerfile 内部の COPY 処理に渡す引数が多ければ多いほど COPY に要する時間が長くなる．本提案のアルゴリズムをより COPY に渡す引数が少ないものにすることで，キャッシュが使用できなかった Build における Build 時間増加を抑えることができる．本提案において引数を少なくするには，中に入っているファイル全てが前回の Build 時のものと一致するディレクトリをコピーする必要がある．ディレクトリをコピーする場合に，ディレクトリ内部にある一部ファイルはコピーしないという処理は Docker では不可能である．また，先に”dir/hoge.txt”ファイルをコピーしてから”dir”ディレクトリをコピーするような，先にディレクトリ内の一部ファイルをコピーしてからディレクトリをコピーする場合，先にコピーした”dir/hoge.txt”ファイルが 2 回コピーされて Build 時間が増加するという問題もある．そのため，中に入っているファイル全てが前回の Build 時のものと一致する場合は引数を節約することはできるが，そうでは無い場合は引数を節約することができない．そこで，ファイルの分類を Dockerfile 内のみで行うのではなく，ローカル上でもファイルを分類する方法がある．ローカルリポジトリのクローンを作成し，クローンの方で 1，2，3，4 と提案によるコピー順を命名したディレクトリを作成し，その中に該当する順番でコピーするファイルを移動させる．COPY の引数には”ディレクトリ名/*”と渡すことで，オリジナルと同じファイル配置でイ

*3 https://github.com/dogenetwork/doge-unblocker

メージへコピーできる．そうすることで，本提案で追加する COPY 処理の引数にはディレクトリ 1 つのみを渡せばよくなるため，引数が増大することによる COPY 処理に必要な前処理に発生するオーバーヘッドを，提案適用前と同じ程度まで抑えることができる．

今回プログラムはユーザが手動で実行することになっている．そのため，ユーザがプログラムの実行を忘れる可能性が存在する．本提案は，早い段階でコピーするファイル群に編集が行われた場合，再度ファイルの分類を行い更新頻度が増加したファイルを切り離す必要がある．そのため，本提案で先にコピーを行うようにしたファイルが編集されていた場合には，提案が確実に実行され，COPY レイヤーの内容を更新する必要がある．そのため，提案の実行はユーザ任せにはせず，ファイルを監視し更新をトリガーとしてプログラムが実行する方が，ファイル更新頻度の特性が変化した際に確実に対応できる．そのための方法として，Python の watchdog モジュールを使用して，対象となる Dockerfile が配置されているローカルリポジトリ内のファイル更新を監視する方法がある．ファイル更新を検知した際は，更新回数が記録された CSV ファイルから同名のファイルが何回更新されているかを取得し，それが提案で最初にコピーを行うグループのファイルだった場合に提案を実行することで，提案の適用忘れを防ぐことができる．

7. おわりに

課題は，ファイルをまとめてコピーする場合に，コピーするファイルを 1 つでも編集すると全体のキャッシュが破棄され，Build に要する時間が長くなることである．そこでファイルをコミット回数の特徴に基づきグループ分けし，コピー処理を複数回に分けることを試みた．評価実験として，提案適用前後の Dockerfile でキャッシュを利用した Build を行い，キャッシュできたファイルの割合と Build に要した時間の差を比較した．適用前は平均 Build 時間が約 3.7 秒となり，提案を適用すると平均 Build 時間は約 3.7 秒となった．

参考文献

[1] Cito, J., Schermann, G., Wittern, J. E., Leitner, P., Zumberi, S. and Gall, H. C.: An Empirical Analysis of the Docker Container Ecosystem on GitHub, pp. 323–333 (2017).
[2] Wu, Y., Zhang, Y., Wang, T. and Wang, H.: Characterizing the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study, *IEEE Access*, Vol. 8, pp. 34127–34139 (2020).
[3] Nüst, D., Sochat, V., Marwick, B., Eglen, S. J., Head, T., Hirst, T. and Evans, B. D.: Ten simple rules for writing Dockerfiles for reproducible data science, *PLOS Computational Biology*, Vol. 16, No. 11, pp. 1–24 (online), DOI: 10.1371/journal.pcbi.1008316 (2020).
[4] Lu, Z., Xu, J., Wu, Y., Wang, T. and Huang, T.: An Empirical Case Study on the Temporary File Smell in Dockerfiles, *IEEE Access*, Vol. 7, pp. 63650–63659 (online), DOI: 10.1109/ACCESS.2019.2905424 (2019).
[5] Xu, J., Wu, Y., Lu, Z. and Wang, T.: Dockerfile TF Smell Detection Based on Dynamic and Static Analysis Methods, *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1, pp. 185–190 (online), DOI: 10.1109/COMPSAC.2019.00033 (2019).
[6] Wu, Y., Zhang, Y., Wang, T. and Wang, H.: Dockerfile Changes in Practice: A Large-Scale Empirical Study of 4,110 Projects on GitHub, *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 247–256 (online), DOI: 10.1109/APSEC51365.2020.00033 (2020).
[7] Rosa, G., Scalabrino, S. and Oliveto, R.: Assessing and Improving the Quality of Docker Artifacts, *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 592–596 (online), DOI: 10.1109/ICSME55016.2022.00081 (2022).
[8] Henkel, J., Silva, D., Teixeira, L., Marcelod' Amorim, Reps, T.: Shipwright: A Human-in-the-Loop System for Dockerfile Repair, pp. 1148–1160 (2021).
[9] Hassan, F., Rodriguez, R. and Wang, X.: RUDSEA: Recommending Updates of Dockerfiles via Software Environment Analysis, p. 796–801 (2018).
[10] Wang, Y. and Bao, Q.: A Code Injection Method for Rapid Docker Image Building, (online), DOI: 10.48550/ARXIV.1911.07444 (2019).
[11] Li, M., Bai, X., Ma, M. and Pei, D.: DockerMock: Pre-Build Detection of Dockerfile Faults through Mocking Instruction Execution (2021).