修士学位論文

題目

限られた保存領域を使用する Java プログラムの実行トレース記録手法の提案と評価

指導教員
井上 克郎 教授

報告者
嶋利 一真

平成 31 年 2 月 6 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 ソフトウェア工学講座

平成 30 年度 修士学位論文

限られた保存領域を使用する Java プログラムの実行トレース記録手法の提案と評価

嶋利 一真

内容梗概

開発者がソフトウェアの実行の様子を観測する手段として，ソフトウェアの処理の進行状況を示すメッセージや重要なデータをプログラムの外部に出力するロギング処理が広く用いられている．しかし，ロギングによって記録されるのは開発時点で選定されたデータのみであり，実際のデバッグにあたって必要なデータがログに含まれていない場合も多い．デバッグの効率的な実施を可能とするためには，プログラムの実行を網羅的に観測，記録する手法が必要である．

デバッグに必要となるデータを自動的に，かつ網羅的に収集する方法の 1 つとして，プログラムの実行中のメモリ状態を時系列で完全に記録する Omniscient Debugging が提案されている．この手法を用いると，任意の時点でのプログラムの内部状態を計算機上で再現し，命令の実行順序や変数の値を観測することが可能である．しかし，デバッグに必要な情報を事前に判断するのは困難であり，事前に必要な情報を網羅しておくには機械的に多くの変数情報を記録する手法が必要となるため，プログラムの命令列である実行トレースのデータ量が膨大となる問題がある．また，記録される実行トレースのデータ量は，解析対象となるソフトウェアの規模や実行する機能の種類によって大きく異なるため把握が困難であり，開発者が現在直面しているバグに対して手法の適用が可能であるのか判断することが難しい．

本研究では，実行トレース記録に使用される保存領域の大きさを開発者が制御可能とし，かつ，デバッグに必要となる命令の実行順序や変数の値を可能な限り網羅的に収集する方法として，限られた保存領域を効率的に使用する実行トレースの記録手法を提案する．提案手法はプログラムの各命令が使用したデータの最新の系列を，命令ごとに一定個数記録するような領域を個別に作成する．命令ごとの記録個数を調整することで，開発者が保有する保存領域に応じた実行トレースの記録が可能となる．また，提案手法の有意性を確かめるために評価実験を行い，単一の大きな記憶領域に記録する場合に比べて，提案手法の実行トレースが同じディスク容量で多くの命令に関する実行時のデータの値およびデータ依存関係を保存できる事を示した．さらにケーススタディとして，ライブラリのバージョンの後方互換性を確認するためのテストに対して提案手法を適用した．提案手法の実行トレースによって得

られたデータ依存関係と完全な実行トレースによって得られたデータ依存関係の比較を，ライブラリのバージョンの組に対して行った結果，ライブラリのバージョン更新前後で同様のデータ依存関係を示すことを確認した．

主な用語

実行トレース
Omniscient Debugging
データ依存関係

目 次

1　はじめに

デバッグとは，外部から観察できるソフトウェア障害の症状から，その原因となるソースコード内の欠陥を突き止める活動である [24]．効果的なデバッグの実施において重要となるのが，ソースコード中に書かれた命令の実行順序や変数の値の観測である [20]．特に，障害が発生する場合としない場合のソフトウェアの実行を観測し，実行中の様々な時点で条件分岐や変数の値にどのような違いがあるかを調査することが有効である [4, 13]．開発者がソフトウェアの実行を観測する手段として，処理の進行状況を示すメッセージや重要なデータをプログラムの外部に出力するロギング処理が広く用いられている [14]．しかし，ロギングによって記録されるのは開発時点で選定されたデータのみであり，実際のデバッグにあたって必要な変数の値がログに含まれていない場合も多い [23]．事前に多くの変数の値を記録するようにロギングを設計することも可能であるが，記録するデータ量が増加すると記憶媒体の管理などの運用面まで気を配る必要がある [10]．デバッグの効率的な実施を可能とするためには，プログラムの実行を網羅的に観測，記録する手法が必要である．

ライブラリのバージョン更新の適用における問題を例に挙げて説明を行う．ライブラリはある特定の機能に特化したソフトウェアであり，現在数多くのライブラリがインターネット上で公開，配布されている．しかし，ライブラリの更新の実施による後方互換性の維持がなされていないという問題が多く報告されている．ここで述べる後方互換性とは，既存のソフトウェアにおいて更新実施後のライブラリが更新実施前のライブラリと同様の挙動を示し，ソフトウェアの振舞いが変化しないことを指す．Mostafa らによる Java プログラムにおけるライブラリの後方互換性の調査 [18] では，ライブラリの新たなバージョンの 76.5%において後方互換性が維持されていなかったことが報告されている．また，同様の調査では連続した API 呼び出しによる後方互換性が維持されていない問題は，ソフトウェアの単体テストを行うだけでは検知できないことも報告されている．このような問題に対して，既存のロギングにおいては障害を避けるためにはライブラリ内にロギング処理を書き入れる必要があるが非常にコストが大きい．また，ライブラリの更新は頻繁に実施されるため，その更新の適用の度にライブラリの変更内容を把握し，適切なロギング処理を加えることは困難である．したがって，ロギング以外でソフトウェアの挙動の変化を把握する手法が必要となる．

デバッグに必要となる変数の値を自動的に，かつ網羅的に収集する方法の 1 つとして，プログラムの実行中のメモリ状態を時系列で完全に記録する Omniscient Debugging[15] が提案されている．この手法は任意の時点でのプログラムの内部状態を計算機上で再現し，命令の実行順序や変数の値を観測することが可能である．しかし，このアプローチは Java プログラムを対象にした場合で，解析対象プログラムの実行 1 秒あたり 10 MB 程度の実行トレースデータの記録を必要とする [19]．また，実行トレース全体の大きさは，解析対象とな

るソフトウェアの規模や実行する機能の種類によって大きく異なる [5]．そのため，開発者が現在直面しているバグに対して手法の適用が可能であるのか判断することが難しい．

本研究は，実行トレース記録に使用される保存領域の大きさを開発者が制御可能とし，かつ，デバッグに必要となる命令の実行順序や変数の値を可能な限り網羅的に収集する方法として，プログラムにおける各命令が使用したデータをそれぞれ指定した回数だけ保存する実行トレース記録手法を提案する．実行トレースを全て記録した場合には，重要な機能を持たないユーティリティ関数の実行の繰り返しなどが含まれる [5]．本研究ではそのような繰り返し実行される命令について記録するデータ量に上限を設けることで，限られた保存領域でプログラム全体の変数情報を記録する．具体的には，観測対象の命令ごとに指定した大きさのバッファを個別に準備し，実行回数が指定した回数より少ない命令に対する観測値はすべて保存し，それを超えて実行された命令に関しては指定した回数分の最新の観測値のみを保持する．観測範囲となるプログラムを構成する命令の数を実行前に数え挙げておくことで，開発者が保有する保存領域に応じて 記録する最大の回数を設定し，実行トレースを記録することが可能となる．

さらに本手法をライブラリの後方互換性テストに用いることで，既存の Omniscient Debugging よりも少ない実行トレースの記録量でソフトウェアの実行の比較による後方互換性テストが行える事例を確認した．

以降，2 章で研究背景について，3 章で提案手法について説明する．4 章で実行トレースが情報を保存する能力を評価し，5 章で提案手法をライブラリ更新における後方互換性テストに適用事例に関して述べ，6 章で関連研究について述べて 7 章でまとめと今後の課題を述べる．

2 研究背景

2.1 ロギング

ロギングとは，開発者がプログラムにあらかじめ特定のデータをプログラムの外部に出力するように命令を記述する方式で，そのデータを分析することで障害の原因を特定することが出来る．ソフトウェアにおける障害の発生を検知し，かつ十分な原因分析を行うためには，ログ出力命令が適切な位置で適切なデータを出力する必要がある．

しかし，オープンソースソフトウェアにおけるロギングの調査において，適切なロギングがなされていないことが報告されている．Yuan らは，OSS の障害におけるロギングに関する調査を行った [22]．調査の結果，5 つの OSS で発生した障害に対してロギングが行われている割合は 43%であることを示した．また，例外処理を行うなどの定型的な障害におけるロギングに関しての調査も同様に行い，ソースコード内部で明らかなロギングの機会があった事例でも 54%においてロギングが行われていなかったことを示した．Yuan らはこれらの障害を捕捉するために，例外処理などの定型的な障害に対してログを自動的に挿入するシステムを作り，障害分析に要する時間を減少させることを確認した．しかし，定型的でない障害に対して適切なログを挿入することは難しく，すべての障害にこのシステムを適用することは困難である．

また，Yuan らは OSS において，デバッグに必要な変数の情報が欠けているログの存在を示し，変数の値の情報を既存のログに付加することで，障害原因の分析を行えることを示している [23]．ただし，この手法で得られるログは障害の発生原因の簡単な分析にしか用いることができないため，デバッグにおいてより詳細な情報，例えば欠陥の感染経路の特定などを行う際にはこの手法では不十分な場合がある．

以上の調査から，既存のロギングではソフトウェアの障害分析を行う際に，適切な位置において適切なデータを出力することが困難であり，十分な情報を収集できない可能性があることが分かる．

2.2 Omniscient Debugging

プログラムの実行中のメモリ状態を時系列で完全に記録する Omniscient Debugging[15] が提案されている．この手法は，網羅的にプログラムによって実行された命令とそれによる値の変化（以下，実行トレースと呼ぶ）を記録してプログラムの状態を再現することで任意の地点のプログラムの状態の閲覧を可能としている．手法を利用することで任意の時点でのプログラムの内部状態を計算機上で再現し，命令の実行順序や変数の値を観測することが可能である．網羅的な記録により過去の実行情報が失われないためログを基としたデバッグが可能であり，それらの実行情報を基に実行を再現できるためブレークポイントベースのデ

バッグやその際のステップ実行なども容易に行うことができる．

しかし，手法の欠点として網羅的に実行トレースを記録する際に，記録量が膨大となることが挙げられる．Pothier らは，スケールした Omniscient Debugging の手法を提案し，実行トレースを全実行トレースの半分程度に削減した．また，この手法を既存の手法と比較した結果，期待した実行トレースの削減に実行に多大なメモリを必要とする手法 [15] に対して，少ないメモリで期待した実行トレース収集できたことを示した [19]．

しかし，この手法は Java プログラムを対象にした場合で，解析対象プログラムの実行 1 秒あたり 10 MB 程度の実行トレースデータの記録を必要とする．実行トレース全体の大きさは，解析対象となるソフトウェアの規模や実行する機能の種類によって大きく異なる [5]．そのため，開発者が現在直面しているバグに対して手法の適用が可能であるのか判断することが難しい．

松村らは， Omniscient Debugging によって収集した実行トレースを用いて，プログラム理解の支援を目的としたメソッドの実行経路可視化ツールを作成した [26] ．この手法においては，同一メソッドの複数の実行経路の違いを可視化することが目的とされている．本研究では松村らの手法で用いられた Omniscient Debugging の実装である SELogger を拡張し，提案手法の実装とする．

2.3 実行トレースの削減手法

プログラムの実行トレースのデータ量がソースコード等と比べて大きくなる要因の 1 つは，プログラムが繰り返し同じ命令を実行することにある．実行トレースを削減するために，繰り返しの実行が互いに類似している，あるいは重要度が低いと仮定した，データ圧縮やサンプリングの適用が提案されている．

Wang ら [21] は，プログラムがアクセスしたメモリアドレスの系列からなる実行トレースを，データ圧縮によって効果的に保存する方式を提案している．プログラムの多くは繰り返し同じ命令を実行するが，それらの命令の多くが配列等の連続的なデータに対する同一の操作になることから，差分エンコーディングが高い圧縮率を達成できることを示した．しかし，この手法で記録可能なのはデータ依存関係と呼ばれるデータの流れのみであり，本研究のような変数の具体的な値の記録にはそのまま適用できない．また，圧縮後のデータの大きさがソフトウェアに依存するという課題も残されている．

Cornelissen ら [5] は，メソッド呼び出しの系列からなる実行トレースに対して，実際に使用できるデータ量の目標値が設定された場合，重要な機能を持たないユーティリティ関数の呼び出しの繰り返しを除去することが，単純なサンプリングよりも情報量を保つ傾向にあることを報告した．一方で，この研究は完全な実行トレースを保存した後，他の解析を行う前にフィルタリングを行うことを前提とした分析を行っていた．本研究では，繰り返しのデー

タの除去を，プログラムの実行中に行うことを提案している．

Hirzel ら [11] は，実行トレースの効果的なサンプリング方法として，データの観測を行う実行区間，行わない実行区間を切り替える Bursty Tracing 手法を提案した．この方式は，観測頻度を均等に低くする通常のサンプリングよりも，プログラムの実行経路について多くの情報を取得することが可能である．しかし，サンプリングで得られるデータはプロファイリング対象に大きく依存しており，デバッグに必要なデータを収集できない可能性もあるほか，収集するデータ量の上限をプログラムの実行前に設定することはできないという制約もある．

2.4 低オーバーヘッドの動的解析

実行トレースの収集には一定のコストがかかるため，特定の解析目的に対して，実行時間あるいは解析に伴うデータ量を最小限にするような解析技術が提案されている．たとえば Liu ら [16] は，バッファオーバーフローやメモリリークなどの欠陥検出を目的として，通常は軽量なメモリ監視を実行し，疑わしい挙動に対して詳細情報の収集を行う解析手法を提案している．Zhang ら [25] は動的にデータ依存関係解析を実行するにあたって，プログラムの実行トレースをそのまま保存するのではなく，実行系列間の依存関係であるプログラムスライスに実行中に変換しメモリに保持することで，記録量を大幅に削減できることを示している．これらの手法は，それぞれ，実行に伴うオーバーヘッドを削減するが，本研究が対象としているような命令の実行順序や具体的な変数の値の取得という目的とは異なる．

3　提案手法

本研究は，Java バイトコードの命令ごとに最新 k 回の実行に関する具体的なデータ値を記録する手法を提案する．最新の実行を残すことで，プログラムのクラッシュなどが生じた場合には，異常な動作の値が記録として残ることが期待される．一方で，プログラムの実行開始時などに一度だけ実行されるような処理も破棄されないため，プログラム実行時の環境設定などの確認も可能である．

3.1　実行トレースのモデル

本研究で取り扱う実行時情報は，Omniscient Debugging の実装の 1 つである松村らの手法 [26] に基づくものである．この手法は，Java プログラムの任意の時点での挙動を確認するために，メソッドの境界を超えて受け渡されたすべてのデータの値を Java バイトコードの命令単位で記録する．具体的には，メソッド呼び出し命令の実行開始および完了，メソッド実行の開始および完了におけるすべての実引数・仮引数・戻り値・例外の値，フィールドおよび配列の読み書き命令の実行における対象オブジェクト，添え字，読み書きされた値を記録する．ここで，オブジェクトについては，参照を区別するための ID 値を保存するものとする．また，引数や戻り値のないメソッドの呼び出しについては，呼び出しの開始や終了の発生を記録するために，それぞれ最低 1 個のデータは記録するとみなす．これらの値には，たとえばフィールドに書き込んだ値とそこから読み出した値の組のように，冗長な情報も含まれるが，複数のスレッドによる同時操作の可能性や，観測範囲外となるネイティブコードやライブラリからのアクセスの可能性も考慮して，それぞれ個別に記録を行う．ローカル変数の値は，メソッドの外部から受け取ったデータから計算によって再現可能であるため，引数等を除くローカル変数の操作は記録しない．

本研究では，このような実行トレースを観測値の系列として表現する．観測値はそれぞれ，観測地点の ID p，命令を実行したスレッドの ID t，観測された値 v の 3 つ組 $\langle p, t, v \rangle$ として考える．観測地点は，データ授受を引き起こした Java バイトコードの命令とデータの種別を識別する．1 つの命令でも，たとえば引数と戻り値という異なる値が観測されるものであれば，それぞれ異なる観測地点とみなす．

3.2　実行トレースの記録方法

提案手法は，観測地点 p が同一であるような観測値 v は最新の k 個だけを残すように実行トレースを記録するというものである．プログラムの実行中に，観測地点 p ごとに大きさ k のバッファを用意して，観測値と，実行全体での観測順序の情報を，メモリ内に蓄積する．プログラムの終了時に，Java 仮想マシンのシャットダウンフック機構を利用して，蓄積

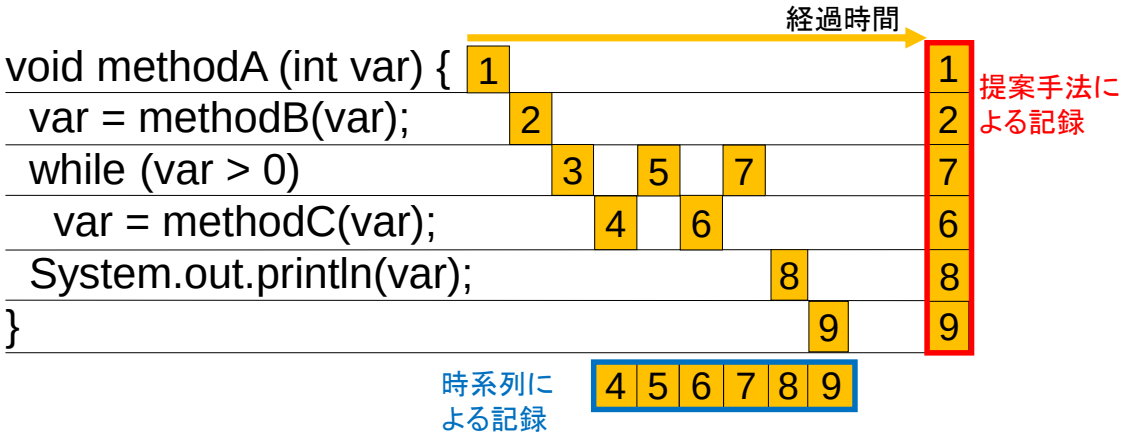

図 1: 提案手法による記録イメージ

したデータをまとめて保存する．なお，従来研究，たとえば松村ら [26] の実行トレースは，すべての観測値を時系列で単純に保存することに相当する．

提案手法の特徴を図 1 の例を用いて説明する．プログラムの各行の右側にある 1 から 9 までの数字が，ある 1 回の実行における命令の実行順序であり，実行トレースとして保存する観測値を表現しているとする．実行トレースの保存領域が限られている場合，一般的なサーバ等での時系列に基づくロギングであれば，最新の観測値から順に保存領域の容量分だけ記録するため，保存容量の大きさが 6 であれば，図 1 の下側にあるように記録が行われる．それに対して，提案手法は同じ容量を，図 1 の右側にあるように，6 個の観測地点それぞれに対し，1 つの観測値を保存するために使用する．これにより，ループで繰り返し実行された命令の観測値は最新の値を除いて破棄するが，プログラムの先頭部分などの実行回数が少ない観測地点における情報を完全に保持する．

プログラムが持つ観測地点の数 m はプログラムの Java バイトコードから静的に数え挙げができるため，実行トレースの保存に必要な保存領域の大きさ $N = k \times m$ を事前に知ることができる．Java では動的にバイトコードを生成する技術も利用されているが，通常の利用であればプログラム中の大きな割合を占めることはなく，m に余裕を持たせた見積もり値の設定が可能である．

提案手法の実装は GitHub に公開されている [1]．この実装では，Java プログラムがクラスをロードする際にバイトコード変換を実行し，対象プログラムに対して観測地点 ID を割り当てながら，観測のための命令を埋め込む．観測地点 ID およびスレッド ID の保存にそれぞれ 4 バイト，値の保存に 8 バイトを使用するため，1 つの観測値あたり 16 バイトの保存領域を必要とする．実際にはプログラムの命令情報など，付加的な情報も出力するが，そ

[1] `https://github.com/takashi-ishio/selogger`, develop branch, commit 2519626.

表 1: ヒープ領域を経由したデータ依存関係

依存元	依存先
PUT_STATIC_FIELD	GET_STATIC_FIELD
PUT_INSTANCE_FIELD	GET_INSTANCE_FIELD
ARRAY_STORE, ARRAY_STORE_INDEX	ARRAY_LOAD, ARRAY_LOAD_INDEX

れらの量はプログラムとしてロードされたクラスの数，バイトコード命令の量にのみ依存して決まる．実行時には順序の保存のために 8 バイトのインデックスを同時に保存するため，記録する観測値 1 つあたり 24 バイトのメモリと，それらを管理する配列等を使用する．

文字列データに関しては，観測値として記録されたオブジェクトについて，一定の長さまでの固定長の記録と内容のハッシュ値の記録によって，一定の範囲で有用な情報を保存することが可能である．ただし，プログラム中で使用される文字列の内容は多岐にわたるため，文字列の適切な保存方法は今後の課題とし，現在の実装では，文字列の内容に関しては可変長データを用いて完全に保存するようになっている．

また，本手法による実行トレースの記録では，オブジェクトに関する代入命令と参照命令においてオブジェクト ID に関しての情報は記録しているが，それ以上の内部情報に関しては記録していない．たとえば，フィールドにおいて List 変数が宣言されている際に，記録が行われるのは List に関する代入や参照が行われたという命令とそのオブジェクト ID のみで，実際に List で読み書きされた値に関しては記録を行っていない．

3.3 実行トレースによって得られる依存関係

実行トレースから得られる情報として，データ依存関係がある．本研究において，データ依存関係はフィールドあるいは配列（ヒープ領域）を経由してデータを受け渡す，代入命令と参照命令の組とする．なぜなら，これらは単純なメソッド呼び出しの系列やメソッド内部の静的解析からは得られず，実行トレースなどの動的解析から得られた情報が必要となるためである．表 1 に本研究で用いるデータ依存関係を示す．本論文では，ある変数に対して，表 1 における依存元の命令が実行された後に依存先の命令が実行され，二命令間で読み書きした値あるいはオブジェクト ID が同一だった場合，二命令間にはデータ依存関係が存在するものと定義する．

提案手法は，観測地点 p が同一であるような観測値は最新の k 個だけを残すように実行トレースを記録するというものである．繰り返し実行されるデータ読み書き命令の一部を記録しないため，データ依存関係の整合性を確保することが難しい．そのため，

- 代入命令と参照命令のデータ依存関係の誤検出

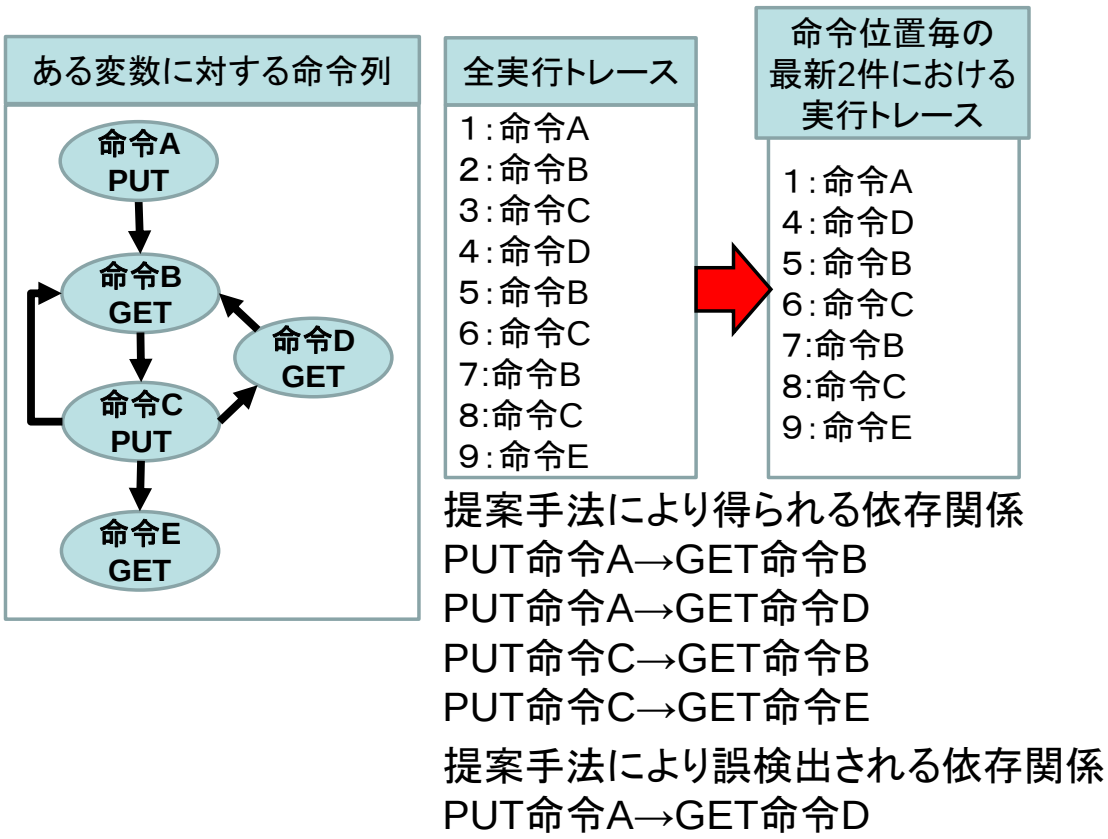

図 2: 代入命令と参照命令のデータ依存関係の誤検出

- 代入命令と参照命令のデータ依存関係の欠落

が生じる恐れがある．特に，オブジェクトの状態遷移を表現するグローバル変数のように，多数の場所で，ある一定の範囲の値の代入，参照が行われるようなフィールドは多数の誤検出と欠落の原因となる．3.3.1 節，3.3.2 節にて具体例を示して，説明を行う．

3.3.1 データ依存関係の誤検出

提案手法によるデータ依存関係の誤検出に関して図 2 を用いて説明を行う．図 2 は，左部がある変数に対する命令列を示し，右部が実行された命令の完全な実行トレースと提案手法の実行トレースが記録されている様子を示している．図 2 の例では，GET 命令 B と PUT 命令 C が繰り返し実行されており，提案手法ではこの二つの命令の古い実行トレースは欠落している．その結果，本来で GET 命令 D に実行されるべき PUT 命令 C が実行されていないかのような実行トレースとなってしまい，PUT 命令 A から GET 命令 D へのデータ依存関係が誤検出されてしまう．依存関係の検出において，読み書きされる値あるいはオブジェク

トIDが異なった際に二命令間の依存関係を記録しないことで，データ依存関係の誤検出の可能性を抑えることは可能ではある．しかし，ある特定のフィールド変数に様々なメソッドから多数のアクセスが行われ，さらに別命令による同一の変数への同一の値の書き込みが行われた場合に実行トレースの欠落が起きた際は，データ依存関係の誤検出は生じてしまう．

3.3.2 データ依存関係の欠落

提案手法によるデータ依存関係の欠落に関して図3を用いて説明を行う．図3は，左部がある変数に対する命令列を示し，右部が実行された命令の完全な実行トレースと提案手法の実行トレースが記録されている様子を示している．図3の例では，PUT命令AとPUT命令Dが繰り返し実行されており，提案手法ではこの二つの命令の古い実行トレースは欠落している．その結果，本来GET命令Cの前に実行されるべきPUT命令Aが実行されていないかのような実行トレースとなってしまい，PUT命令AからGET命令Cへのデータ依存関係がないかのような実行トレースとなる．たとえば，ある特定のフィールド変数に様々なメソッドから多数のアクセスが行われた場合に古いPUT命令は消えてしまい，そのPUT命令によるデータ依存関係の取得が不可能となるため，データ依存関係の欠落は生じてしまう．

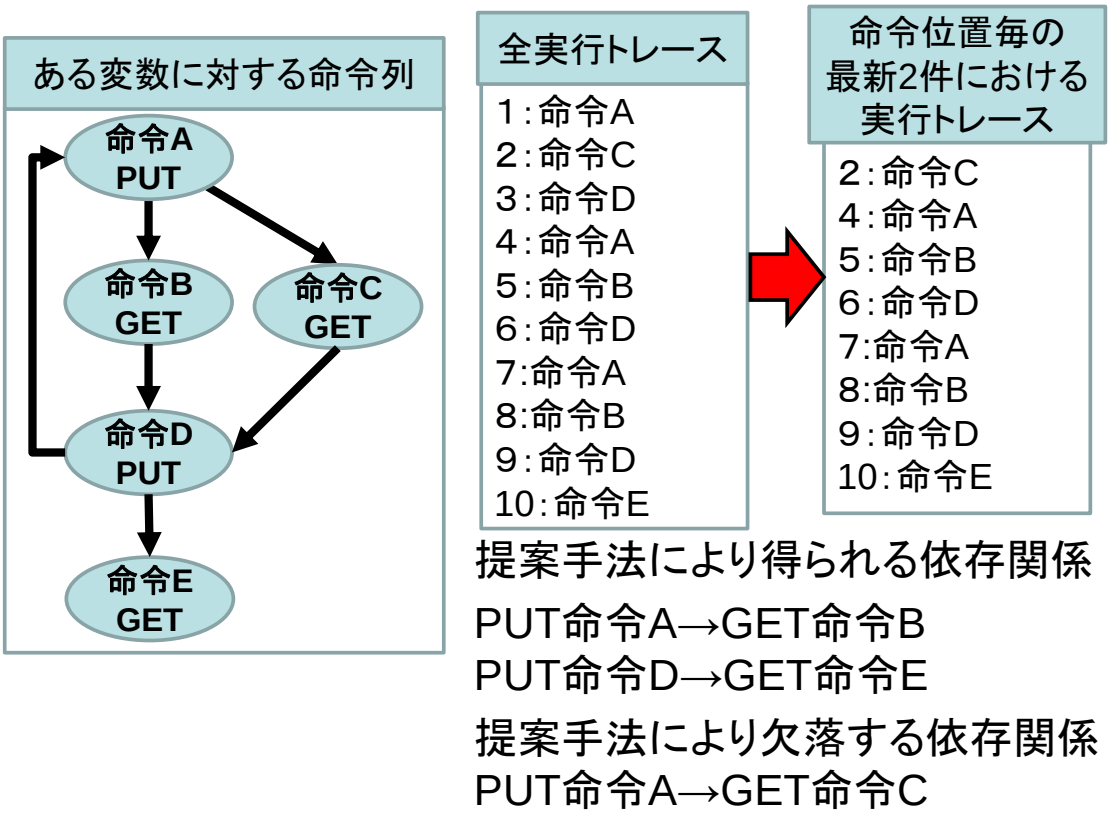

図 3: 代入命令と参照命令のデータ依存関係の欠落

表 2: ベンチマークプログラムおよび実行トレースの規模

ベンチマーク	クラス	メソッド	観測地点数	実行された観測地点数	観測地の総数	データ量
avrora	527	3,456	87,736	38,772	7,880,412,751	117.4 GB
batik	1,122	9,163	218,775	72,630	601,350,099	9.0 GB
fop	1,198	10,401	318,795	127,226	325,806,544	4.9 GB
jython	2,244	23,342	670,054	214,300	5,461,124,807	81.4 GB
luindex	231	2,467	73,094	25,838	1,542,871,620	23.0 GB
lusearch	199	2,140	58,127	16,256	5,503,253,333	82.0 GB

4 評価

提案手法の実行トレースが，限られた保存領域でどれだけの情報を記録可能か，以下の 2 つの観点から評価する．

1. 観測値が完全に保存される観測地点の割合
2. データ依存関係が保存される割合

前者は開発者がランダムに観測地点を選んだ場合にその地点に関するデータがすべて揃っている割合であり，事前にどのデータを必要とするか分からない状況での有用性を評価する．後者は，実行トレースを系列としてみたときに，データの流れに関する順序関係が正しく保存されているかどうかを評価する．

性能計測の環境は，CPU として Intel(R) Xeon(R) W-2123 CPU（3.60GHz），メモリ 32GB，ストレージとして HDD を搭載したワークステーションである．OS として Windows 10 Pro，Java 仮想マシンには Oracle Java SE の build 1.8.0_191-b12, を用いた．

実行トレースの計測には，DaCapo Benchmarks[3] のバージョン 9.12-bach に収録された 14 個のベンチマークのうち，動作が確認できた 6 個を使用した．実行トレースは，ベンチマーク内部での標準ライブラリへの呼び出し等を含んでいるが，Java 標準ライブラリ内部の動作は記録していない．表 2 に，各ベンチマークのプログラム規模と実行トレースの大きさを示す．クラス数は，ベンチマークの実行のためにロードされたクラスから，Java 標準ライブラリ（`java`, `javax`, `sun` などのパッケージ名で識別されるもの）を除いた数である．メソッド数，観測地点数は，それらのクラスに対して数えた値であり，実行されなかったメソッド，命令に対応するものも含んでいる．観測値の総数は，完全な実行トレースを記録し

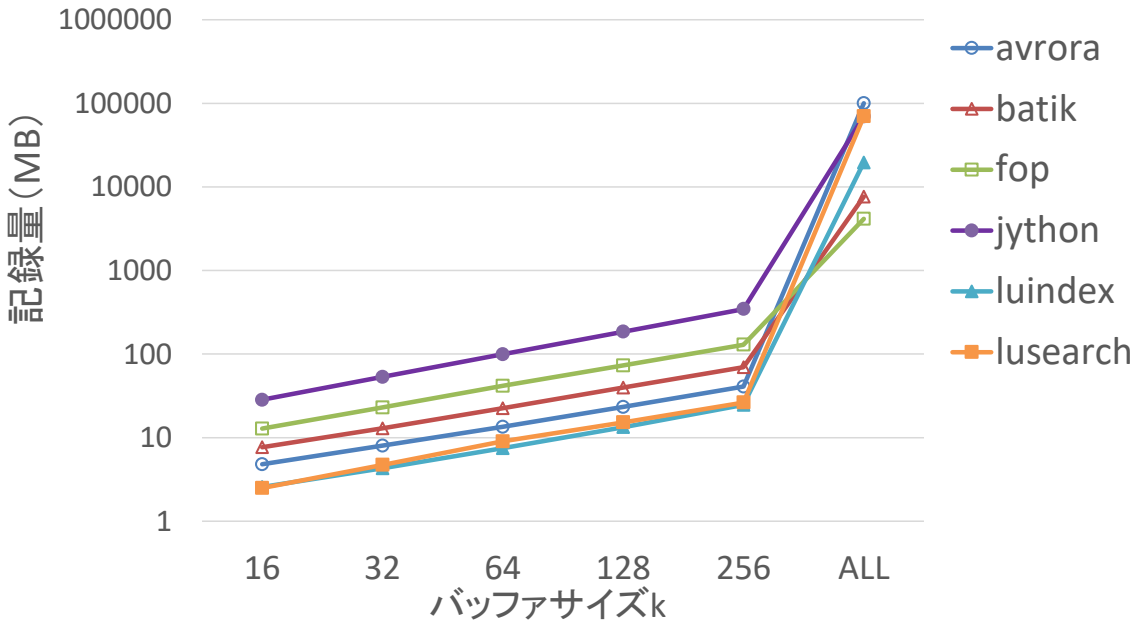

図 4: 提案手法の実行トレースのデータ量

た場合の値であり，データ量は観測値 1 つあたり 16 バイトとして完全な実行トレースを保存した場合の値である．

提案手法のパラメータである k の値として $16, 32, 64, 128, 256$ を使用する．それぞれの k の値に対しての提案手法の記録データ量を図 4 に示す．図中の ALL は，完全な実行トレースを保存した場合のデータ量である．提案手法は観測地点 1 つあたり 16 バイトの観測値を k 個保持することから，観測地点数が最大の jython を $k = 256$ で実行した場合で，$670,054$(観測地点数) $\times$ 256×16 2.7×10^9 となり，最大で約 2.6 GB の実行トレースを収集することになるが，すべての観測地点が実行されるわけではないために，実際の値はそれよりも小さくなっている．提案手法で $k = 256$ とした際の記録データ量は，完全な実行トレースを記録した場合と比較して平均で 1.0 %以下である．

使用できる保存領域が制限された環境での性能のベースラインとして，時系列に基づく方式，すなわち実行トレース全体における最新 N 個の観測値を保存する方法を用いる．以降，提案手法を Latest-per-Location，ベースラインとなるこの方式を Latest-of-All と記載して区別する．

4.1 観測値が完全に保存される観測地点の割合

観測地点ごとに保存する観測値の個数 k を変化させたとき，それぞれのベンチマークで実行された観測地点のうち，完全なデータを保存できる（すなわち実行回数が k 以下であ

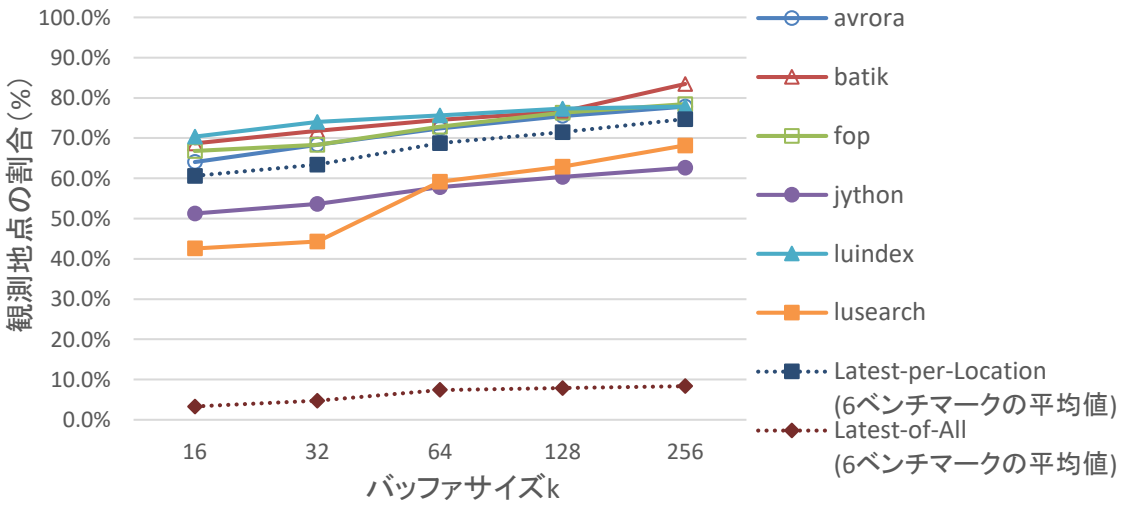

図 5: 観測値が完全に保存される観測地点の割合

る）観測地点の割合の変化を図 5 に示す．また，同図は Latest-per-Location，Latest-of-All の 2 手法における 6 つのベンチマークの値の平均も示している．

提案手法である Latest-per-Location は，$k = 16$ のときで 60%，$k = 256$ のときで 74%の命令に関して完全なデータを保存し，残る命令に関しては最新の k 回のデータを保存する．したがって，従来手法では巨大な実行トレースとなるようなプログラムの実行であっても，1%以下の限られたデータで，多くの観測地点について完全な情報を参照することができる．これに対して Latest-of-All は，約 10%の命令に関してのみデータを保存しており，ほとんどの観測地点に関しては，利用者は完全な観測値を参照できない．

4.2 提案手法の実行時間

提案手法の実行トレースと完全な実行トレースを取得した場合の実行時間を表 3 に，またバッファサイズ k 変更した際の完全な実行トレースの取得時間と提案手法の実行トレースの取得時間の比を図 6 に示す．なお，表 3，図 6 に示した結果は，3 回手法を実行した時間の平均の値としている．

表 3 と図 6 の内容に関して考察を行う．実行トレースの記録量は図 4 で示したように，提案手法において完全な実行トレースに対しておよそ 0.1%から 1%となっており，それに対して実行時間の削減率は，図 6 の通り，ベンチマークによって異なるがおよそ 10%〜20%となっている．この結果から，提案手法の書き込み処理に必要な時間に対して，内部で最新の処理を記録する処理に必要な時間の方がより大きいことが分かる．

また，実行時間はバッファサイズ k によってほとんど変化していない．最も記録量の差が大きかったベンチマーク jython でバッファサイズ k=16 の際に記録量約 40MB，k=256 の

表 3: 実行トレースの取得にかかる実行時間 (ms)

k	16	32	64	128	256	All
avrora	1,514,900	1,615,273	1,596,266	1,638,938	1,572,943	8,182,858
batik	30,424	29,636	30,195	30,539	30,695	349,326
fop	22,827	22,706	24,880	24,676	25,011	195,760
jython	156,858	163,257	169,431	176,465	180,545	2,955,059
luindex	43,574	44,697	46,170	46,626	46,866	866,890
lusearch	1,133,554	1,199,841	1,193,854	1,167,819	1,116,960	5,803,087

表 4: 提案手法によって得られるデータ依存関係の正確さ（全ベンチマークの合計）

	正解の	Latest-per-Location				Latest-of-All			
k	依存関係	依存関係	適合率	再現率	F 値	依存関係	適合率	再現率	F 値
16	52,061	42,603	0.914	0.748	0.823	3,010	1.000	0.058	0.109
32	52,061	45,115	0.903	0.782	0.838	3,144	1.000	0.060	0.114
64	52,061	47,472	0.892	0.813	0.851	3,938	1.000	0.076	0.141
128	52,061	49,826	0.881	0.843	0.861	4,043	1.000	0.078	0.144
256	52,061	51,941	0.872	0.870	0.871	4,054	1.000	0.078	0.144

際に記録量約 450MB で実行時間はおよそ 1.2 倍程度であり，その他のベンチマークに関しては記録量は数 MB から数十 MB 程度しか変化がなく，実行時間に大きな影響を与えていない．

4.3 データ依存関係の正確さ

提案手法がデータの流れに関する順序関係を正しく保存しているかどうかを評価するために，提案手法の実行トレースから算出したデータ依存関係を完全な実行トレースから算出したデータ依存関係と比較する．ここでのデータ依存関係は，フィールドあるいは配列（ヒープ領域）を経由してデータを受け渡す，代入命令と参照命令の組とする．なぜなら，これらは単純なメソッド呼び出しの系列やメソッド内部の静的解析からは得られないためである．

完全な実行トレースから求めたデータ依存関係の集合を正解として，Latest-per-Location と Latest-of-All の 2 手法で得られた実行トレースから得られるデータ依存関係の適合率と再現率，F 値を求めた．表 4 に，すべてのベンチマークから得られたデータ依存関係を用いて算出した結果を示す．

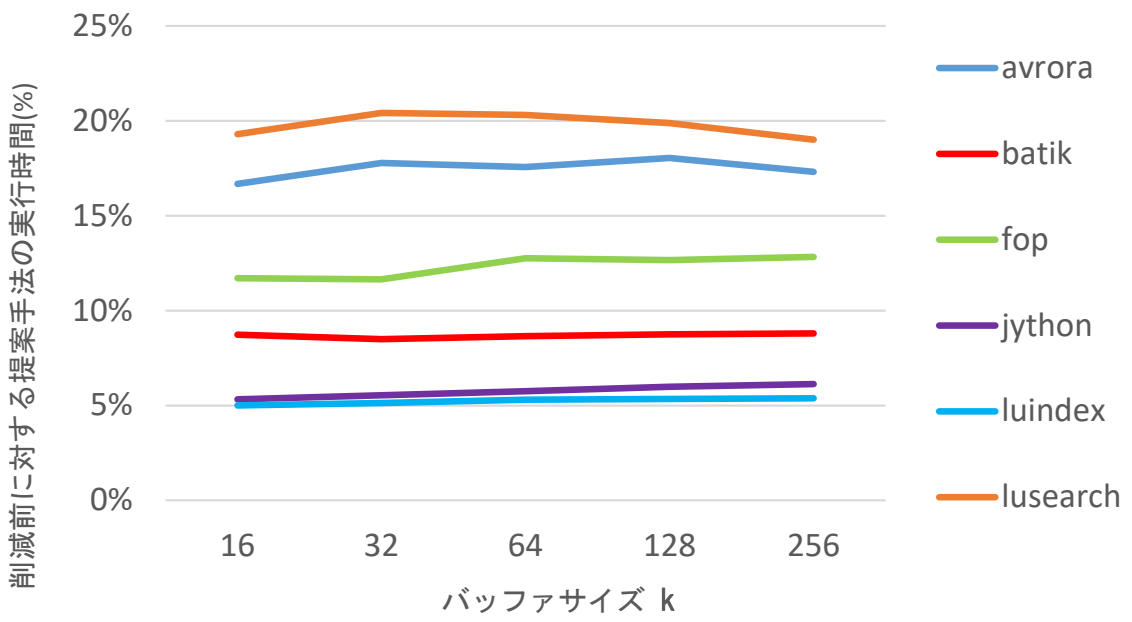

図 6: バッファサイズ k と削減前に対する提案手法の実行時間の割合

提案手法による実行トレースの記録では，オブジェクトに関する代入命令と参照命令においてオブジェクト ID に関しての情報は記録しているが，それ以上の情報に関しては記録していない．たとえば，フィールドにおいて List 変数が宣言されている際に，記録が行われるのは List に関する代入や参照が行われたという命令とそのオブジェクト ID のみで，実際に List で読み書きされた値に関しては記録を行っていない．したがって，Latest-per-Location により途中の命令の欠落が起きた際には，依存関係の整合性を確保することが難しく，誤った代入命令と参照命令の依存関係を記録する可能性がある．Latest-per-Location は，繰り返し実行されるデータ読み書き命令の実行の一部を記録しないため，それによるデータ依存関係の誤検出や欠落が発生する．特に，オブジェクトの状態遷移を表現するグローバル変数のように，多数の場所で，ある一定の範囲の値の代入，参照が行われるようなフィールドが，多数の誤検出と欠落の原因となる．たとえば，ベンチマーク avrora に含まれる avrora.arch.legacy.LegacyInterpreter クラスは，与えられたプログラムを実行するインタプリタの実装であり，100 以上のメソッドがプログラムカウンタに関するフィールド（pc, $nextPC$）を繰り返し操作していた．それぞれの代入，参照命令の実行回数が多いことから，Latest-per-Location では命令の実行順序が十分に保存されず，多数のデータ依存関係が欠落するとともに，同一の値を使用した異なる命令の実行を誤ってデータ依存関係として検出した．ただし，このような誤検出の要因は少数のフィールドに限られており，全体での適合率は 0.9 以上，再現率は 0.8 前後と，保存するデータを 1%に抑えた状態でも高い数値を保

表 5: 提案手法によって得られるデータ依存関係の正確さ（ベンチマーク別, $k = 16$）

ベンチマーク	正解の依存関係	Latest-per-Location				Latest-of-All			
		依存関係	適合率	再現率	F 値	依存関係	適合率	再現率	F 値
avrora	4,325	3,269	0.904	0.683	0.778	289	1.000	0.067	0.125
batik	7,054	6,366	0.951	0.858	0.902	50	1.000	0.007	0.014
fop	13,906	12,143	0.968	0.845	0.902	2,149	1.000	0.155	0.268
jython	20,460	15,342	0.849	0.637	0.728	137	1.000	0.007	0.013
luindex	4,098	3,699	0.952	0.859	0.903	385	1.000	0.094	0.172
lusearch	2,218	1,784	0.915	0.736	0.816	0	N/A	0.000	N/A

表 6: 提案手法によって得られるデータ依存関係の正確さ（ベンチマーク別, $k = 32$）

ベンチマーク	正解の依存関係	Latest-per-Location				Latest-of-All			
		依存関係	適合率	再現率	F 値	依存関係	適合率	再現率	F 値
avrora	4,325	3,431	0.896	0.711	0.793	406	1.000	0.094	0.172
batik	7,054	6,711	0.954	0.890	0.921	51	1.000	0.007	0.014
fop	13,906	12,921	0.966	0.869	0.915	2,157	1.000	0.155	0.269
jython	20,460	18,318	0.825	0.683	0.747	137	1.000	0.007	0.013
luindex	4,098	3,864	0.954	0.881	0.916	393	1.000	0.096	0.175
lusearch	2,218	1,972	0.915	0.771	0.837	0	N/A	0.000	N/A

つことが確認できた．一方で Latest-of-All は最新の実行トレースのみを保持するため，誤ったデータ依存関係が認識されることはないが，プログラム全体でみるとほとんどのデータ依存関係を破棄してしまうため，プログラム全体の調査には不向きである．

4.3.1 バッファサイズによる精度の変化

提案手法の Latest-per-Location におけるバッファサイズ k による適合率，再現率の変化に関して考察を表 4，表 5，表 6，表 7，表 8，表 9 を用いて行う．表 4 は全ベンチマークのデータ依存関係を合計した際の精度，表 5，表 6，表 7，表 8，表 9 は，それぞれ $k = 16, 32, 64, 128, 256$ におけるベンチマークごとにデータ依存関係の精度を求めた詳細な結果である．

適合率は多くのベンチマークでバッファサイズが増えるにつれて減少している．適合率が減少する原因として考えられるのは，バッファサイズの増大に伴う 1 命令当たりの記録量増加によって，記録される PUT 命令の量が増加したうえで，その PUT 命令から正しいデータ依存関係を求められなかったことが原因である．これは，本実験で対象とした DaCapoBenchmark のように繰り返し処理を多く含むものを対象とする場合に，バッファサイズの増加に伴って既存の命令による異なる値の記録が行われたうえで，増加したバッファサイズの分だけデー

表 7: 提案手法によって得られるデータ依存関係の正確さ（ベンチマーク別, $k = 64$）

ベンチマーク	正解の依存関係	Latest-per-Location				Latest-of-All			
		依存関係	適合率	再現率	F 値	依存関係	適合率	再現率	F 値
avrora	4,325	3,686	0.888	0.757	0.818	793	1.000	0.183	0.310
batik	7,054	6,711	0.954	0.907	0.930	55	1.000	0.008	0.015
fop	13,906	12,921	0.964	0.895	0.928	2,490	1.000	0.179	0.304
jython	20,460	18,318	0.804	0.719	0.759	137	1.000	0.007	0.013
luindex	4,098	3,864	0.956	0.901	0.928	463	1.000	0.113	0.203
lusearch	2,218	1,972	0.914	0.813	0.861	0	N/A	0.000	N/A

表 8: 提案手法によって得られるデータ依存関係の正確さ（ベンチマーク別, $k = 128$）

ベンチマーク	正解の依存関係	Latest-per-Location				Latest-of-All			
		依存関係	適合率	再現率	F 値	依存関係	適合率	再現率	F 値
avrora	4,325	3,832	0.876	0.776	0.823	834	1.000	0.193	0.323
batik	7,054	6,932	0.956	0.939	0.947	55	1.000	0.008	0.015
fop	13,906	13,266	0.962	0.918	0.940	2,501	1.000	0.180	0.305
jython	20,460	19,862	0.782	0.759	0.770	190	1.000	0.009	0.018
luindex	4,098	3,924	0.958	0.918	0.937	463	1.000	0.113	0.203
lusearch	2,218	2,010	0.912	0.826	0.867	0	N/A	0.000	N/A

タ依存関係を求める際に誤った依存関係を余分に求めたことが原因であると考える．ただし，バッファサイズが増加することで命令の欠落が起きず，誤った依存関係を取得する可能性が減少することも事実であるため，他の OSS などにおいてバッファサイズの増加によって適合率が減少するかどうかに関しては別途議論が必要であると考える．再現率は全てのベンチマークでバッファサイズが増えるにつれて増加している．この結果は，バッファサイズの増加によって，既存の正しい依存関係は削除されずに，新たな PUT 命令と GET 命令の記録によって正しい依存関係が新たに追加されたことが原因である．F 値に関しては，どのベンチマークでも単純増加していることが分かる．これは，バッファサイズの増加によって適合率は減少して再現率は増加するが，全体としてはバッファサイズが大きい方が精度としては良いことを示している．したがって，単純に実行トレースから求められるデータ依存関係の精度の向上を目的とした場合は，バッファサイズを大きく設定した方が良いと考えられる．

同様の考察を Latest-of-All に関しても行う．適合率に関して，Latest-of-All は連続した実行トレースの記録を行うため，記録量の変化にかかわらず適合率は 1 となる．再現率に関しては，表 4 を参照すると Latest-per-Location に比べて増加幅が小さいことが分かる．また，表 4，表 5，表 6，表 7，表 8，表 9 を見ると，ベンチマークによっては記録量が増加したにも関わらず再現率が全く増加していないものもあることが分かる．F 値に関しては，適合率

表 9: 提案手法によって得られるデータ依存関係の正確さ（ベンチマーク別, $k = 256$）

ベンチマーク	正解の依存関係	Latest-per-Location				Latest-of-All			
		依存関係	適合率	再現率	F 値	依存関係	適合率	再現率	F 値
avrora	4,325	4,082	0.851	0.803	0.826	844	1.000	0.195	0.327
batik	7,054	6,989	0.962	0.953	0.957	55	1.000	0.008	0.015
fop	13,906	13,599	0.963	0.942	0.952	2,501	1.000	0.180	0.305
jython	20,460	21,241	0.767	0.797	0.782	190	1.000	0.009	0.018
luindex	4,098	3,959	0.958	0.926	0.942	464	1.000	0.113	0.203
lusearch	2,218	2,071	0.912	0.852	0.881	0	N/A	0.000	N/A

が一定で再現率が微増であるため，それに伴って微増であることが分かる．これらの結果から Latest-of-All においては，記録量の増加が必ずしも取得可能なデータ依存関係の精度の向上に繋がらない可能性があることが分かる．

これらの結果から，どのベンチマークに関しても提案した実行トレースの削減手法が高い数値を示しており，Latest-per-Location が実行トレースの削減手法として有望であることが分かる．なお，lusearch の適合率・F 値が N/A となったのは，Latest-of-All の手法ではフィールド・配列に関するデータ依存関係が得られなかったためである．

5　ケーススタディ

本実験では，提案手法を用いてソフトウェアの単体テストの実行トレースを取得することで，ソフトウェア単体テストにおける挙動の比較を行い，ライブラリの後方互換性のテストを行う．実験では，プロジェクトにおいてあるライブラリのバージョンの更新を適用した際のプログラム内部の挙動の変化を比較する．内部の挙動の取得方法としては，3 章で述べた提案手法と，松村らの手法で用いられた SELogger を用いる．

5.1　実験内容

5.1.1　実験概要

実験には，表 10, 表 11 にて示したライブラリを用いているプロジェクトにおける単体テストの挙動の比較により行う．比較対象は，あるプロジェクトにおけるあるライブラリの更新の適用前のテスト実行と，更新の適用後のテスト実行である．両者において，完全な実行トレースと提案手法を用いた実行トレースの収集を行い，それぞれデータ依存関係を求める．ライブラリ更新の適用前後において，完全な実行トレースによるデータ依存関係と提案手法の実行トレースによるデータ依存関係を比較することで後方互換性が維持されているかの確認を行う．

5.1.2　実験対象

ライブラリの後方互換性テストを行う対象に関して述べる．使用したライブラリの情報に関して，表 10,11 に示す． 表 10 は順に，maven におけるライブラリの groupID，artifactID を，表 11 は順に比較したライブラリのバージョン組の数，ライブラリの開始バージョンと終了バージョン，ライブラリの開始バージョンの更新時期と終了バージョンの更新時期に関して示している．バージョンに関しては，Mostafa らの調査 [18] で用いられたライブラリから，maven リポジトリ[2]に登録されていることが確認でき，バージョン名に “Alpha” などの完成版でないと考えられる名称が入っていないもの除いたバージョンを対象としている．

実験対象のプロジェクトに関して述べる．今回，調査には TravisTorrent[3]のプロジェクトの中から，pom.xml を含む maven プロジェクトで，表 11 で示されたライブラリとそのバージョンを含み，著者の環境において maven のビルドが成功したものから単体テストが用意されている，maven-plugin-api と joda-time を使用した．maven-plugin-api を用いた実験に関して 5.2 節で，joda-time を用いた実験に関して 5.3 節で説明を行う．

[2]https://mvnrepository.com/
[3]https://travistorrent.testroots.org/

表 10: 使用したライブラリの ID

Mostafa らの定義	groupID	artifactID
log4j	org.apache.logging.log4j	log4j-core
maven	org.apache.maven	maven-plugin-api
beanutils	commons-beanutils	commons-beanutils
codec	commons-codec	commons-codec
fileupload	commons-fileupload	commons-fileupload
commons-io	commons-io	commons-io
ela.Search	org.elasticsearch	elasticsearch
http-core	org.apache.httpcomponents	httpcore
jodatime	joda-time	joda-time
jsoup	org.jsoup	jsoup
neo4j	org.neo4j	neo4j
snakeyaml	org.yaml	snakeyaml

5.2 実験 1：後方互換性が維持されている例

5.2.1 実験概要

maven-plugin-api において後方互換性が維持されていた例を示す．まず，ライブラリ開発者側が利用者に対して後方互換性の有無に関してどの程度の情報を公開しているかをリリースノートを用いて確認する．本実験では，maven-plugin-api のバージョン 3.0.1 から 3.0.5 への更新を対象とする．バージョン 3.0.5 におけるリリースノート[4]の一部を抜粋して以下に示す．

```
Maven 3 aims to ensure backward compatibility with Maven 2,
improve usability, increase performance, allow safe embedding,
and pave the way to implement many highly demanded features.
```

リリースノートによると，後方互換性を担保したうえで機能性や信頼性の向上を行っていることが分かる．しかし，Mostafa らの非互換性の影響を受けた部分の調査 [18] によると，maven-plugin-api における後方非互換性は存在することが分かっている．その影響が実際のプロジェクトに対して，どの程度影響を及ぼしているかを知るために，これらの問

[4]https://maven.apache.org/docs/3.0.5/release-notes.html

表 11: 使用したライブラリに関する情報

Mostafa らの定義	比較	開始	終了	開始時期	終了時期
log4j	2	2.0	2.1	2014-07	2014-10
maven	5	3.0	3.2.5	2010-10	2014-12
beanutils	1	1.9.0	1.9.2	2013-12	2014-05
codec	1	1.6	1.7	2011-12	2012-09
fileupload	3	1.2	1.3.1	2007-02	2014-02
commons-io	5	2.0	2.4	2010-10	2012-06
ela.Search	6	1.0.3	1.3.9	2014-04	2015-02
http-core	6	4.0.1	4.3.3	2009-06	2014-10
jodatime	7	2.0	2.7	2011-07	2015-01
jsoup	11	1.1.1	1.7.3	2010-06	2013-11
neo4j	4	1.8.3	2.0.3	2013-06	2014-04
snakeyaml	7	1.4	1.11	2009-08	2012-09

題に対して提案手法を用いることで，後方非互換性のあるライブラリを使用しているプロジェクトが実際に影響が受けているかどうかの調査を行う．実験対象として，プロジェクトは license-maven-plugin[5]，ライブラリは maven-plugin-api のバージョン 3.0.1 と 3.0.5 を用いた．得られた実行トレースの量を表 12 に示す．

5.2.2 実験結果

実験の結果，完全な実行トレースを用いたデータ依存関係が一致することを確認できた．したがって，プロジェクト license-maven-plugin は後方互換性が維持されている．また，提案手法の実行トレースの比較に関しては，バッファサイズ k=32 においてはデータ依存関係

表 12: license-maven-plugin の実行トレース量

maven-plugin-api のバージョン	完全な実行トレース	提案手法 (k=32)	提案手法 (k=256)
3.0.1	269.9MB	7.8MB	25.8MB
3.0.5	269.9MB	7.8MB	25.8MB

[5] `https://github.com/mycila/license-maven-plugin`,master branch,commit 90936b8

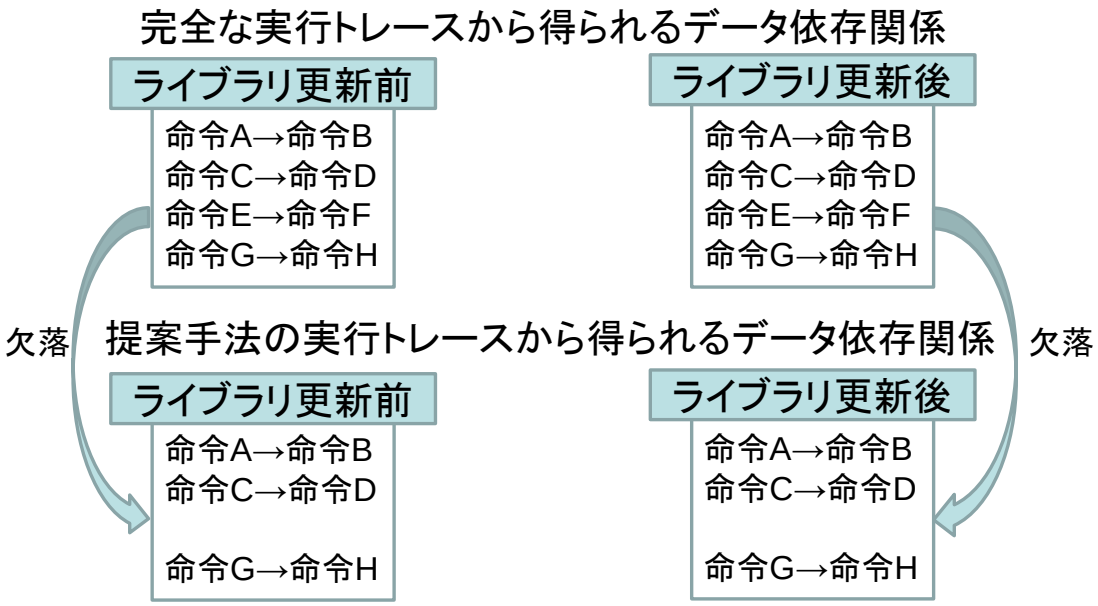

図 7: 完全な実行トレースと提案手法の実行トレースから得られるデータ依存関係の違い

が一致しなかったが，バッファサイズ k=32 に関してデータ依存関係が欠落したのは，3.3.2 節で述べたように特定のフィールド変数に様々なメソッドから多数のアクセスが行われて実行トレースが欠落したことと，それに加えて命令が並行実行されたことが原因である．バッファサイズ k=256 においては実行トレースが欠落せず，並行実行された命令を全て記録できた結果，データ依存関係がライブラリの更新前後で一致したことを確認できた．

ただし，完全な実行トレースから得られたデータ依存関係と提案手法によって得られたデータ依存関係に差分が見られたので，その点に関して図 7 を例に示して説明を行う．図 7 では，提案手法の実行トレースを用いた場合，完全な実行トレースを用いた場合と比較してデータ依存関係が一つ欠落していることが分かる．ただし，ライブラリ更新前後において両手法の依存関係の欠落は全く同様となっている．

提案手法の実装上，完全な実行トレースから得られたデータ依存関係と同様のデータ依存関係を取得することは困難であるため，完全な実行トレースから得られたデータ依存関係と同様のデータ依存関係を得ることは難しい．しかし，同一の挙動を示したソフトウェアに対して提案手法を複数回実行してもそのデータ依存関係が変わることはない．つまり，後方互換性が保たれている状況で，提案手法で後方互換性テストを実行した場合にライブラリの更新前後でデータ依存関係が一致していれば，該当プロジェクトにおいて同様の挙動を示している可能性が高い．本ケーススタディにおいては，完全な実行トレースから得られたデータ依存関係 1489 件に対して，提案手法の実行トレースから得られたデータ依存関係 1475 件（誤検出 2 件，欠落 16 件）という結果となった．

5.3 実験 2：後方互換性が維持されていない例

5.3.1 実験概要

joda-time のバージョン 2.3 と 2.4 において後方互換性が担保されていなかった例を示す．まず，ライブラリ開発者側が利用者に対して後方互換性の有無に関して，どの程度の情報を公開しているかリリースノートを用いて確認を行う．本実験では，joda-time バージョン 2.3 から 2.4 への更新を対象とする．バージョン 2.4 におけるリリースノート[6]から互換性に関する記述部を抜粋して以下に示す．

```
Compatibility with 2.3
----------------------
Build system - Yes
Binary compatible - Yes
Source compatible - Yes
Serialization compatible - Yes

Data compatible - Yes, except
 - DateTimeZone data updated to version 2014e

Semantic compatible - Yes, except
- DateTimeField duration fields have been fixed
  For example, yearOfEra() now has a range of eras() rather than null
  The DurationField instances now compare using equals() correctly
- MutableDateTime.add(DateTimeFieldType,int), addDays(int) and friends [#77]
  Adding zero will no longer change the offset during DST overlap in autumn/fall
```

リリースノートによると，DateTimeField の duration fields に関する修正がなされ，変更に関するいくつかの例が示されていることが分かる．しかし，具体的に修正内容に対してどのような挙動の変化が起こっているかは網羅的に示されておらず，joda-time を使っているユーザがこの後方非互換性の影響を受けるかどうかは判断することは難しい．

また，Mostafa らの非互換性の原因となった API の調査 [18] によると，joda-time の更新後バージョンの 9 つのメソッドにおいて返り値の変更，あるいは正しく処理されない例外が発生することが分かっている．該当の API とその呼出しシナリオと後方非互換性に関して表 13, 表 14 に示す．DateTimeZone や DataTimeField に関する後方非互換性のすべてをライブラリのユーザ側が把握するのは困難であり，非互換性によって影響が生じるかどうかの判断も難しい．

これらの問題に対して提案手法を用いることで，後方非互換性のあるライブラリを使用しているプロジェクトが実際に影響を受けているかどうか判断できるかの調査を行う．実験対

[6]https://www.joda.org/joda-time/upgradeto240.html

表 13: joda-time のバージョン 2.3 と 2.4 で後方非互換性が生じる API

ID	API
BBI1	DateTime.centuryOfEra.getRangeDurationField
BBI2	DateTime.yearOfEra.getRangeDurationField
BBI3	MutableDateTime.centuryOfEra.getRangeDurationField
BBI4	MutableDateTime.yearOfEra.getRangeDurationField
BBI5	DateMidnight.centuryOfEra.getRangeDurationField
BBI6	DateMidnight.yearOfEra.getRangeDurationField
BBI7	DateTimeZone.setProvider
BBI8	DateTimeZone.setNameProvider
BBI9	DateTimeField.get

象として，プロジェクトは seyren[7]，ライブラリは joda-time のバージョン 2.3 と 2.4 を用いた．得られた実行トレース量を表 15 に示す．

5.3.2 実験結果

データ依存関係の比較の結果，joda-time のバージョン 2.4 において，バージョン 2.3 に現れなかったソースファイルである “GJYearOfEraDateTimeField.java” におけるデータ依存関係をバッファサイズ k=32,256 の両方で確認できた．データ依存関係を基に，ソースコードの該当部分を確認すると，下記のように import 文の追加とメソッドのオーバーライドが行われ、返り値の変更が行われていることが分かった．これは，Mostafa らが指摘している後方非互換性に一致する．

```
import org.joda.time.DurationField;

@Override
public DurationField getRangeDurationField() {
    return iChronology.eras();
}
```

実際の後方互換性テストにおいては，該当部分の変数における変更が確認された場合，その変数が影響を及ぼすと考えられる部分の単体テストを追加するなどの手法を取ることで，ライブラリの後方非互換性によって受ける影響を最小化することができる．

[7] https://github.com/scobal/seyren,master branch commit d4cf423

表 14: joda-time2.3 と 2.4 において生じる後方非互換性の内容

ID	呼出しシナリオ	後方非互換性
BBI1	常時	返り値の変更
BBI2	常時	返り値の変更
BBI3	常時	返り値の変更
BBI4	常時	返り値の変更
BBI5	常時	返り値の変更
BBI6	常時	返り値の変更
BBI7	入力が null	正しく処理されない例外
BBI8	入力が null	正しく処理されない例外
BBI9	長すぎるフィールド名	正しく処理されない例外

表 15: seyren における実行トレース量

j joda-time のバージョン	完全な実行トレース	提案手法 (k=32)	提案手法 (k=256)
2.3	1,682.9MB	21.7MB	48.8MB
2.4	1,683.0MB	21.8MB	48.9MB

6　関連研究

6.1　ライブラリ更新の実施

ソフトウェア開発における実装の工程において，外部のプロジェクトからソースファイルを再利用することが一般的に行われている．特に，ほかのソフトウェアで再利用することを目的として開発されたソフトウェアに，ライブラリと呼ばれるものがある．ライブラリはある特定の機能に特化したソフトウェアであり，現在数多くのライブラリがインターネット上で公開，配布されている．ライブラリはオープンソースソフトウェア内で活用されるだけでなく，ソフトウェア企業での製品開発においても活用されることがあり [7][8]，近年のソフトウェア開発においてライブラリは必要不可欠なものとなっている．また，ライブラリの更新は頻繁に実施される．Ihara らは Java ライブラリの更新の実施に関する調査を行った結果，ライブラリは，開発期間が 10 年以内のライブラリであれば 3 か月程度で 1 回，開発期間が 10 年以上のライブラリであってもおよそ 1 年以上の間隔で，定期的に更新が実施されていることが報告している [12].

6.2　ライブラリ更新の適用における問題

しかし，ライブラリの更新による後方互換性の維持がなされていないという問題が多く報告されている．ここで述べる後方互換性とは，既存のソフトウェアにおいて更新後のライブラリが更新前のライブラリと同様の挙動を示し，ソフトウェアの振舞いが変化しないことを指す.

ライブラリの新たなバージョンが後方互換性を持たない場合，ソフトウェアのビルドが通らない，ソフトウェアの障害の原因となるなど，ソフトウェアの保守において非常に大きな問題となる可能性がある．Mostafa らによるライブラリの後方互換性の調査 [18] では，ライブラリの新たなバージョンの 76.5%において後方互換性が維持されていなかったことが報告されている．また，連続した API 呼び出しによって後方互換性が維持されないという問題は，ソフトウェアの単体テストを行うだけでは検知できないことも報告されている.

6.3　クライアント側におけるライブラリ更新の適用の動向

Derr らは Android アプリケーションにおけるライブラリの更新の適用に関して，Google Play のアプリケーション開発者に対してアンケート調査を行った [6]．203 のアプリケーションとその開発者に対する調査の結果，アプリケーションのライブラリの更新を適用する理由として，多く挙げられたものから順に，バグ修正 (96.47%)，セキュリティ (57.65%)，新たな機能の追加 (56.47%) となった．アプリケーションで最新のバージョンのライブラリを使用し

ない理由として，多く挙げられたものから順に，ライブラリがまだ正常に動作していること (57.03%)，非互換性への恐れ (50%)，ライブラリ更新の非認知 (32.81%) となった．85.6%のアプリケーションはソースコードの修正なしでバージョンを一つは更新でき，48.2%のアプリケーションは最新のバージョンまで更新可能であることを報告した．アプリケーション内で実際に使われている脆弱性を含んでいる 16,837 のライブラリのうち，97.8%は単純なパッチ処理により安全なバージョンへと更新可能であることも報告されている．

Backes らも Android のライブラリに関する調査を行っている [2]．調査対象は，2015 年の 9 月から 2016 年の 7 月において Google Play の TOP50 に一度でも入ったことのある 4,666 のアプリケーションである．これらのアプリケーションの中で使用されていたライブラリにおいて最新のバージョンが使われていた割合は，29.60%であった．また，Android の FacebookSDK のバージョン 3.15 において報告された脆弱性の修正に関して，この修正がライブラリを使用しているアプリケーションに実際に適用されるまでの平均所要時間は 188 ± 55 日であることを報告している．この調査から，Android ライブラリにおいて深刻な脆弱性が報告された場合も，脆弱性が修正されたバージョンをエンドユーザが使用可能となるまでに長い期間がかかることが分かる．

Bavota らは，Apache Foundation 内におけるライブラリの更新に関しての調査を行った．調査の結果，クライアント側のソースコードの行数やクラス数は，ライブラリの更新頻度と関連するものはないことを報告した．バージョンの更新可能なライブラリがある場合も，そのうちの 6 割程度を更新し，その他の 4 割は更新しないまま無視しているという事も報告している．構造的な特徴との関連として，API に影響を与えるような大きな変更の際はクライアント側でライブラリを更新する場合が多いが，更新内容がライブラリの機能の削除であった場合はクライアント側のソフトウェアが動作しなくなるため，クライアント側で更新の適用は少ないことが報告されている．また，Apache のメーリングリストの分析を行った結果，ライブラリにおいてバグ修正が行われた際にクライアント側でライブラリの更新をする傾向にあることも報告されている．最後に，ライブラリの更新がクライアントソフトウェアに与える影響に関しても調査を行っている．この調査では，更新されたライブラリを使用しているクラスの全てが影響を受けたと仮定として，更新の影響の見積もりを行っている．調査の結果，平均としてクライアント側のクラスの 5 %，LOC の 6 %にライブラリ更新の影響が及ぶと報告された．また，大きな影響を及ぼす例として，データベースシステムによる分散処理フレームワークの利用が挙げられている．

6.4 ライブラリ開発者側における更新の実施

ライブラリ開発者がライブラリの後方互換性に関してユーザ側に知らせる方法はいくつかある．最も一般的なものとして，リリースノート内で後方互換性の詳細に関して述べるとい

う方法である．joda-time のバージョン 2.4 のリリースノートを抜粋して以下に示す．

```
Data compatible - Yes, except
 - DateTimeZone data updated to version 2014e
```

この例では，データの互換性は DateTimeZone を除いて維持されているということが，リリースノートに示されている．

しかし，現実問題としてライブラリ開発者が，ライブラリの更新における非互換性をすべてライブラリのユーザに伝えることは難しい．Mostafa らはライブラリ開発者が意図的に振舞いの後方非互換性を生成しているかどうか，つまり後方非互換性に関しての情報がドキュメントとしてまとめられているかに関しても調査を行った [18]．その結果，296 の非互換性のうちドキュメントにて記されていたものは 82 であったことを報告している．つまり，ライブラリのユーザがドキュメントからは知ることのできない後方非互換性が，ライブラリ更新において生じる可能性があることが分かっている．また，6.3 節で示したように，ドキュメントが存在しているにもかかわらず，非互換性への恐れなどの様々な理由から更新に積極的ではないライブラリのユーザも多い．

他の手法として，SemVer[1]（セマンティックバージョニング）を用いる手法がある．これは，バージョンナンバーは “メジャー. マイナー. パッチ” の形式として，バージョンを上げる際には

1. API の変更に互換性のない場合はメジャーバージョン
2. 後方互換性があり機能性を追加した場合はマイナーバージョン
3. 後方互換性を伴うバグ修正をした場合はパッチバージョン

を上げるというものである．ライブラリ開発者が SemVer を用いてそのルールを守ることで，ライブラリのユーザは更新内容をバージョン番号から把握することができ，リリースノートから更新内容の詳細について確認する必要はない．

しかし，Foo らの調査によると，調査対象の SemVer を用いている 5106 のライブラリのうち 72%がいくつかのバージョンにおいて SemVer のルールを守っておらず，26%のライブラリにおいては全てのバージョンにおいて SemVer のルールに反していることが分かっている [9]．

ライブラリ開発者が示したライブラリの後方互換性に関する情報だけでは不十分であり，ライブラリのユーザ側でも後方互換性に関して調査可能とする必要があると考えられる．

6.5 ライブラリ更新の適用の支援手法

Foo らは脆弱性のあるライブラリに対して，バージョンの更新で後方互換性が担保されているかを確認する静的解析を用いた手法を提案した [9]．ライブラリのソースコードの制御フロー解析を行うことで，その精度の確認を行った結果，この手法によって提案されたバージョン更新のうち，Java では 19%，Python では 0%，Ruby では 6%が後方互換性を保ったままライブラリの更新ができたことが報告されている．精度が低くなった原因として，そもそもの脆弱性がないバージョンが少なかったことや，過大な見積もりによる偽陽性や Python や Ruby におけるコールグラフ構築が困難であったことを挙げている．

Mirhosseini らは，ライブラリ更新の適用を促す手法である自動プルリクエストとバッジに関してその効果の調査を行った [17]．調査の結果，自動プルリクエストやバッジを用いることで，ライブラリの利用者が最新バージョンのライブラリを利用する割合が大きく，その効果は自動プルリクエストの方が大きかったことを報告している．また，自動プルリクエストはその 32%のみしか適用されず，またビルドが 25%失敗しているため，ライブラリ更新の適用に有効な手法ほど，ライブラリの利用者が更新を適用する際の負担が大きくなっていることを報告している．

7　まとめ

本研究では，ソフトウェアの詳細な内部状態の観測を行う手段として，使用する保存領域を制御可能な実行トレース記録手法を提案した．観測地点ごとに最新 k 個の観測値を保持することで，完全な実行トレースの 1%程度のデータ記録量でも，60%から 74%の観測地点に対して完全な観測値を保持し，また，フィールドや配列に関するデータ依存関係の多くを追跡可能である．また，ケーススタディによって提案手法をライブラリの後方互換性テストに対して適用し，互換性あるいは非互換性をが発見できることを確認した．

本研究の結果をもとに，開発者が多くの環境で常に実行トレースを収集し続けられるようなデバッグ環境を構築することが今後の課題である．また，デバッグに適した適切な文字列データの記録方法の実現や，データ依存関係などをより高い精度で保存できるように静的解析を活用することも今後の課題として挙げられる．

謝辞

本研究，ならびに研究室生活，研究への心構えなど，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井上克郎教授には多岐にわたって常に御指導および御助言を賜りました．指導教員として以上に一研究者として，研究に関して時には厳しい指摘をいただき，自身の研究やその立ち位置に関して多くを考えるきっかけとなりました．また，普段から研究や研究室生活に関して多くの気配りをしていただき，研究を進めるうえで非常に励みとなりました．研究ならびに研究に対する心構えに関してその多くをご指導していただいた井上教授に心より深く感謝いたします．

本研究において，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松下誠准教授には適切な御指導および御助言を賜りました．研究の中間段階において，研究の方向性に悩んでいた際に研究の本質的な部分に関して議論をいただき，研究の方向性を固めるきっかけを得られました．研究の本質を考えることの重要性など研究に関して多くを教えていただいた松下准教授に深く感謝いたします．

本研究において，奈良先端科学技術大学院大学先端科学研究科石尾隆准教授には研究の方針，論文の書き方や細かな技術的な面まで多くのご指導およびご助言を賜りました．研究に用いるツールのデバッグ，進捗状況の管理，研究内容の新規性・有用性に関してその多くを共に考えていただき，無事に研究成果を形にすることができました．また，研究室の後輩のテーマの指導の非常に大きな部分を任せていただきつつ，適切なサポートをしてくださり，研究に向き合ううえで何が必要かの大きな気づきを得ることができました．時間を作っていただき，研究に関する密な指導をいただけた石尾准教授に深く感謝いたします．

本研究にならびに研究室生活において，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神田哲也助教には技術的な面から日常的な面まで様々な御指導および御助言をを頂きました．研究の実装段階において，想定通りの挙動をプログラムが示さなかった際にはデバッグをお手伝いいただき，口頭発表の練習の際には丁寧に発表スライド１枚１枚にご指導いただきました．研究室生活においては，進捗を積極的に気にかけていただき，些細な悩み事に至るまで多くの博士前期課程における研究を大きく支えていただいた神田助教に深く感謝いたします．

本研究において，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春名修介特任教授には適切な御助言を賜りました．研究に関して，企業における実用的な観点からの意見は，普段得られることがない貴重なもので大変参考になりました．研究に多くの貴重なご意見をいただいた春名特任教授に心より感謝いたします．

本研究に限らず，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伊藤薫氏には様々なご指導およびご助言をいただきました．研究において必須の知識から日頃の研究に

対する指導まで，様々なご助力をいただきました．また，研究室の計算機管理者として大きな問題なく計算機を運用できたことも，伊藤氏の多大なご助力のおかげです．研究生活に多大なご支援をいただいた伊藤氏に深く感謝いたします．

研究室生活において，大阪大学大学院大学院情報科学研究科コンピュータサイエンス専攻の事務職員軽部瑞穂氏には多大なご支援をいただきました．研究室生活においては，学生が研究にのみ専念できるように研究室の備品に関して常に気にかけていただいたり，研究がうまくいかない際もコミュニケーションをとっていただき，大変充実した日々を過ごせました．また，学会参加の際は，学生が学会に集中できるように，参加手続きから出張手続きに至るまでご支援いただきました．快適な研究室生活を大きく支えていただき，多くのご支援をいただいた軽部瑞穂氏に深く感謝いたします．

最後に，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井上研究室の皆様にはその他様々な御指導，御助言等を頂きました．3 年間ご迷惑をおかけすることも多かったですが，無事に充実した研究室生活を送れたのはひとえに皆様のおかげです．非常に充実した研究室生活を共に過ごしていただいた井上研究室の皆様に深く感謝いたします．

参考文献

[1] Semantic versioning 2.0.0, https://semver.org/.

[2] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pp. 356–367, 2016.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 169–190, 2006.

[4] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proc. of the 27th International Conference on Software Engineering*, 2005.

[5] B. Cornelissen, L. Moonen, and A. Zaidman. An assessmentmethodology for trace reduction techniques. In *2008 IEEE International Conference on Software Maintenance*, pp. 107–116, 2008.

[6] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pp. 2187–2200, 2017.

[7] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, pp. 25–34, 2013.

[8] C. Ebert. Open source software in industry. *IEEE Software*, Vol. 25, No. 3, pp. 52–53, May 2008.

[9] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. Efficient static checking of library updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pp. 791–796, 2018.

[10] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proc. of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pp. 24–33, 2014.

[11] Martin Hirzel and Trishul Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. 2001.

[12] Akinori Ihara, Daiki Fujibayashi, Hirohiko Suwa, Raula Gaikovina Kula, and Kenichi Matsumoto. Understanding when to adopt a library: A case study on asf projects. In *Open Source Systems: Towards Robust Practices*, pp. 128–138, 2017.

[13] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *Proc. of the 32nd IEEE Symposium on Security and Privacy*, pp. 347–362. Ieee, 2011.

[14] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. Logging library migrations: A case study for the apache software foundation projects. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pp. 154–164, 2016.

[15] Bil Lewis. Debugging backwards in time, CoRR, cs.SE/0310016, 2003.

[16] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pp. 911–922, 2016.

[17] Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 84–94, 2017.

[18] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. Experience paper: a study on behavioral backward incompatibilities of java software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 215–225, 2017.

[19] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In *Proc. of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pp. 535–552, 2007.

[20] Diomidis Spinellis. *Effective Debugging: 66 Specific Ways to Debug Software and Systems*. Addison-Wesley Professional, 2016.

[21] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing java programs. In *Proc. of the 26th International Conference on Software Engineering*, pp. 512–521, 2004.

[22] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pp. 293–306, 2012.

[23] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *SIGARCH Comput. Archit. News*, Vol. 39, No. 1, pp. 3–14, 2011.

[24] Andreas Zeller. *Why programs fail, the 2nd edition*. O 'Reilly Japan, second edition, 2012.

[25] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *Proc. of the 26th International Conference on Software Engineering*, pp. 502–511, 2004.

[26] 松村俊徳, 石尾隆, 鹿島悠, 井上克郎. Remviewer: 複数回実行された java メソッドの実行経路可視化ツール. コンピュータ ソフトウェア, Vol. 32, No. 3, pp. 137–148, 2015.