Vol.2019-SE-201 No.6
2019/3/7

変数の変動履歴からバグ原因の変数を予測する試み

上村和貴[†1]　尾花将輝[†1]　深海悟[†1]

概要：近年，ソフトウェアは年々多機能化し，ソースコードは肥大化しており，些細なバグを修正するだけでも，多大なコストが掛かる．特に，バグ原因となる箇所を特定する事は容易ではない．そこで，正常動作時と障害発生時ではソフトウェアの変数の値が異なる点に着目し，変数の変動履歴からバグ原因となる変数を予測する手法を提案する．全ての変数に代入される値をログとして保存し，障害が発生したログと正常動作したログで異なる値を出力している変数を求めることでバグ原因の変数であるかを予測する．本提案を 1 つのオープンソースプロジェクトに適用した結果，代入命令 32861 箇所中，バグ原因を含んだ代入命令を 270 個まで絞り込む事ができ，バグ原因となる代入命令及び変数を予測できる可能性を示唆できた．

Case study: Detecting doubtful program statements with variable fluctuations in execution log

KAZUKI UEMURA[†1]　MASAKI OBANA[†1]　SATORU FUKAMI[†1]

Abstract: Recently, software becomes having many functions and enlarging source code continuously. It needs much cost even if developers correct minor software bugs. Especially, correct detection of causes for bugs is difficult in such large-scale software. Therefore, we propose a method that selects doubtful variables. The doubtful variables may be related to the bugs in execution log with dynamic analysis. The method is focuses on different values of a variable between normal log and failure log. Using the different values, the method can predict doubtful program code with the variable. The proposed method has been applied to an open source project. As a result, we were able to narrow the doubtful 270 variables from all 32861 variables. The doubtful 270 variables are useful to detect doubtful program statements that may have some defects.

1. はじめに

近年，オープンソースソフトウェアを用いたシステム開発やサービスの提供が多く提供されている．例えば WordPress[1]等のコンテンツマネージメントシステムを用いて大学のホームページを運用する等が挙げられる．このようにオープンソースソフトウェアの需要は増加の傾向にある．しかしその一方では，ソフトウェアのソースコードの規模は年々肥大化している．例えば，2018 年 12 月において最新版である WordPress ver.5.0.2 では，ファイル数は 1710 ファイル，ソースコードの行数の合計は空行を除き 560811 行と肥大化の一途を辿る．

肥大化するソースコードにはバグが混入する可能性が多くなる他にもソースコードの詳細を把握することが困難となる．オープンソースソフトウェアの場合，開発に関わっていない人も利用することができるが，ソースコードの全容を把握することは難しい．結果，何らかの障害（バグ）が発生しても運用者では原因追及する事すら困難であるため，運用時の応急処置も困難である．

そこで，本研究では実行時の変数の代入命令に着目した障害発生箇所を予測する試みを行う．これにより，開発者の支援する他にも，運用者でもどの機能がどのような動作をした時に障害が発生したかを発見し，対応を行う事ができる．

具体的な手法としては，変数の値を全てログとして保存し，実際の運用等でログを蓄積する．その後，蓄積したログから変数ごとの値の出現頻度を計算する．出現頻度を求める理由は，障害発生時の変数の値は正常時とは異なる可能性が高いと考えられるためである．つまり，障害発生時の変数の値は正常動作時の変数の値には無い傾向にあるという考えから出現頻度を求める．これにより，膨大な変数からバグ原因となる変数を絞り込むことで，開発者や運用者を支援することを目的とする．2 章では関連研究を述べ，3 章で提案手法，4 章で提案手法をオープンソースに適用した結果を述べ，5 章で考察を述べる．

2. 関連研究

本研究では静的なソースコードに対して変数の値をログとして保存する処理を埋め込み，実行結果のログからバグ原因となる変数を特定する．過去にも実行結果をログに保存する研究は多数存在する．松村らは Java の実行トレースを可視化する REMViewer [2]や，Joshi らは各モジュールのイベントを詳細に記録し，再生する SCARPE を提案している[3]．これらは，選択した特定のモジュールやクラスを対象に動的に実行される制御フローとデータフローを記録し，可視化ツールでその状態を再生する事ができる．また，

†1 大阪工業大学 情報科学部

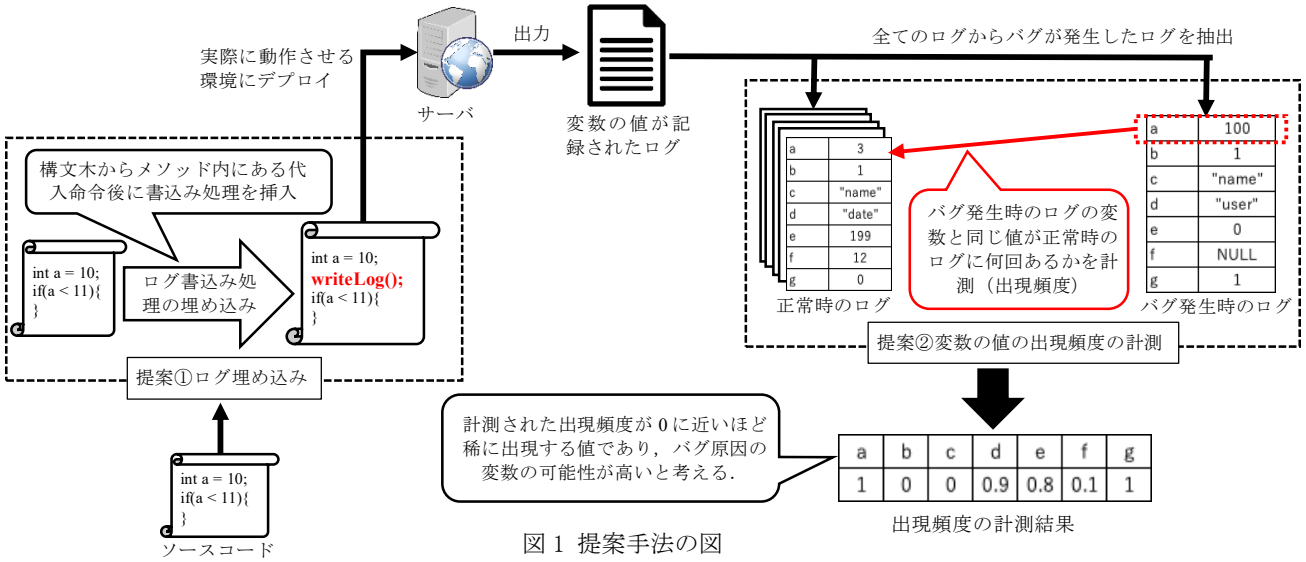

図 1 提案手法の図

全ての命令をログに保存し，再生する手法も提案されている．Bil はすべての命令と実行順序を記録し，記録したログからプログラムの状態を再生できる Omniscient Debugging を提案した[4]．これらは，実行履歴をログに保存するアプローチは本研究と同じであるが，プログラムを再生することでデバッグを支援する目的にしている点が異なる．

実行履歴をログにする研究もなされている．Yanyan らは実行履歴をログに保存する際にキャッシュを用いてログの量を削減した CARM を提案している[5]．また，例外処理発生時に変数等の実行命令をログ保存する事で計算機への負担を減らす提案がされている[6]．櫻井らは実行トレースを用いたデバッグを効率化するための Traceglassed を提案した[7]．トレースの形式を木構造で持ち，実行履歴を局所的，大域的でトレース形式を統一することでデバッグ支援ができる事である．他にも採取したログから動作情報を可視化する事でデバッグを支援する手法等が提案されている[8,9]．これらの研究も実行履歴が保存されるログの軽量化や，効率の良いデバッグ支援方法の提案が主であり，本研究のようにログからバグの原因箇所の特定には至っていない．

3. 提案手法

3.1 本提案のアプローチ

変数にはプログラムのロジックによって処理された値が保持される．ここで述べるプログラムのロジックとは if 文や for 文等によるものであり，ロジックに誤りがあることで誤った値が変数に保存され，その変数が次のロジックに利用されるため障害が発生する．また，バグの存在箇所のうち 41.5%が分岐文, 22.4%が代入文で発生しているという報告がある[10]．つまり，変数に代入される値を精査することにより，バグは未然に防げるのだが，精査するテスト工程で開発者が値を全て網羅することは困難である．

想定されやすいテストケースでのバグは判明するが，想定外の値を考慮したバグはテストケースでは判明しない場合がある．つまり，開発者が想定しない稀に発生する値を検出することでバグ原因を特定できるのではないかと考えられる．こうしたバグはリリース後にも出現せず，稀に発生すると考えられる．

そこで，開発者が予期しない値を保持する変数を検出することによりバグの原因箇所を特定する．具体的には，変数の値の変更を全てログに記録し，稀に値が変更される値を検出することにより，バグの原因になりうる変数を予測する試みを行う．

本研究の全体像を図 1 に示す．提案①として，ソフトウェアのソースコードに対してログ出力を行う処理を埋め込む．埋め込む際にはソースコードを構文木にし，メソッド内に代入命令後の変数の値を出力するコードを埋め込む．埋め込み後はサーバにデプロイし実際に稼働させることでログデータを出力する．この時にログにはどのアクセスがどの代入命令に対応するかを記録する．本論文ではセッション ID を用いて管理している．

続いて提案②として，出力されるログを正常に動作したログとバグが発生したログに切り分ける．バグが発生したログにある変数が，正常に動作したログの中にはどれだけ同じ値があるかを計測する事で，稀に値が変わる変数を検出する．

3.2 提案手法

3.2.1 ログ出力用のプログラムを埋め込み方法

本研究では変数の値の変更を記録するための手法として，対象のソースコードにログ出力処理の埋め込みを行い，変数の代入命令が実行された後に代入 ID と，変数名とその値，クラス・メソッド名をログに記録する．速度とのトレードオフを考慮しミドルウェアから値を取得する等の工夫が必要であるが，本研究では言語に依存せず，確実に値を取得するために本手法を採用した．つまり C や Java 等の

コンパイルが必要な言語から PHP 等のスクリプト言語まで幅広く本提案を適用できることを目指した．

3.2.2 稀に値が変化する変数の検出方法

本研究で対象とする稀に変化する値について述べる．一般的に変数の値はソフトウェア実行時によって変動するが，この値は変数により様々なパターンがあると考えられる．本研究ではそのうちのほとんどが同じ値であるが，バグが発生した際には値が異なる変数を検出対象とする．

具体的な例を表 1 に示す．表 1 では変数が 6 個あり，それぞれの実行時の代入命令が exe1，exe2，exe3 と書かれており， exe4(err)はエラー発生時の代入命令の値である．この時，F.x，G.y は毎回異なる値を保持しており，F.y，F.str は毎回同じ値を保持している．G.x の exe1，exe2，exe3 は常に同じ値であったが，exe4(err)の時だけは異なる値になる．つまり，この代入命令にバグがあると考えられ，開発者はその周辺の処理をデバッグすると考えられる．

このように，エラー発生時にだけ出現する値を計測するためには単純に過去の実行履歴にエラー発生時と同じ値が出現した回数を計算すれば良い．例えば，表 1 では出現割合という過去に同じ値が何回出現するかを実行回数で割った値が記載している．しかし，この計算例では G.x の値と F.x の値が同じ値になってしまう問題がある．そこで本研究では以下のような出現頻度を示すF_{val_i}を提案する．

$$F_{val_i} = \frac{S_{a_i} \cdot (T_i - 1)}{N_i - 1}$$

S_{a_i}: *障害発生時の代入命令の値と同じ値の出現回数*

T_i: *値の種類数*

N_i: *代入命令の実行回数*

出現頻度F_{val_i}の計算には，変数への代入命令iの時，過去の値の種類数T_i，障害発生時の代入命令iと同じ値の出現回数S_{a_i}，iの実行回数N_iが使われる．

出現頻度F_{val_i}の値が 0 のとき，その代入命令で出現する値の種類は 1 種類のみ，つまり過去に代入された値とバグ発生時の値が全く同じ値であることを示す．また，値が大きくなれば値が毎回異なる事を示すことになる（表 1 の提案手法参照）．つまり，出現頻度F_{val_i}が 0 もしくは 1 に近づくほどバグ原因の代入命令の可能性が低いと考える．

表 1　稀に変動する変数の具体例

変数	exe1	exe2	exe3	**exe4(err)**	出現割合	提案手法
F.x	20	172	13	12	1/4 = 0.25	(1*(4-1))/(4-1)=1.00
F.y	30	30	30	30	4/4 = 1.00	(4*(1-1))/(4-1)=0.00
F.str	hoge	hoge	hoge	hoge	4/4 = 1.00	(4*(1-1))/(4-1)=0.00
G.x	8	8	8	**0**	1/4 = 0.25	(1*(2-1))/(4-1)=0.33
G.y	78	1876	12	345	1/4 = 0.25	(1*(4-1))/(4-1)=1.00
G.str	foo	bar	foo	bar	2/4 = 0.50	(2*(2-1))/(4-1)=0.67

本提案では出現頻度F_{val_i}が 0 に限りなく近いものほど稀に出現する値として考え，バグ原因の代入命令である可能性が高いと考える．

また，複数回同じ箇所の代入命令が行われることもあるが，本研究ではバグ原因の変数を絞り込む事が目的であるため，その場合は対象とする代入命令の出現頻度F_{val_i}の最小値を利用する．よって，出現頻度F_{val_i}は代入命令

3.2.3 出現頻度の粒度について

本研究ではバグ原因を特定する手法として稀に値が変わるものを対象としている．しかし，代入命令ごとに計算した出現頻度は代入命令という粒度が細かく，また，規模が大きくなるほど代入命令の数も多いため，代入命令を対象とするだけではバグ原因の箇所を特定するのは困難である．そこで，計算された出現頻度F_{val_i}を関数もしくはメソッド単位等で計測する評価値MF_{val_j}を以下に示す．

$$MF_{val_j} = \min(F_{val})$$

F_{val}: *メソッドiに属する代入命令の集合*

評価値MF_{val_j}は，その関数もしくはメソッドに属する代入命令の出現頻度の最小値とする．実際の利用方法としては，評価値が限りなく 0 に近い値から調査し，その範囲の中で更に出現頻度が 0 に近い変数を重点的にデバッグする事により，ソフトウェア全体からバグ原因となった箇所を徐々に予測する．

4. 適用事例

4.1 実験概要

提案手法をオープンソースプロジェクトである OpenPNE[11]に適用し，バグの原因となる箇所が特定できるかの調査を行った．OpenPNE は 2004 年から開発が開始され，2019 年 1 月現在の最新バージョンは 3.8.30 であり，Git でのコミット数は 3494 件である．今回はバグが含まれたバージョンでの動作を行うためバージョン 3.8.14 を利用

表 2　実験環境の詳細

	ホストマシン	ゲストマシン
OS	Windows2008Server	CentOS7
CPU	Xeon E5506 × 2	2コア
メモリ	48GB	4GB
HDD	HDD 2TB (Raid10)	500GB
ミドルウェア	Hyper-V	Apache2, PHP5.4, MariaDB

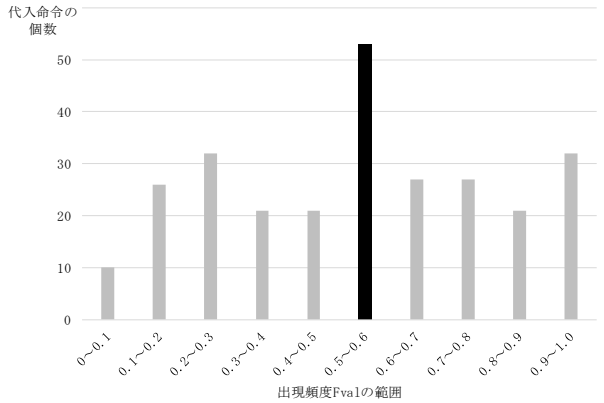

図 2 代入命令ごとの出現頻度

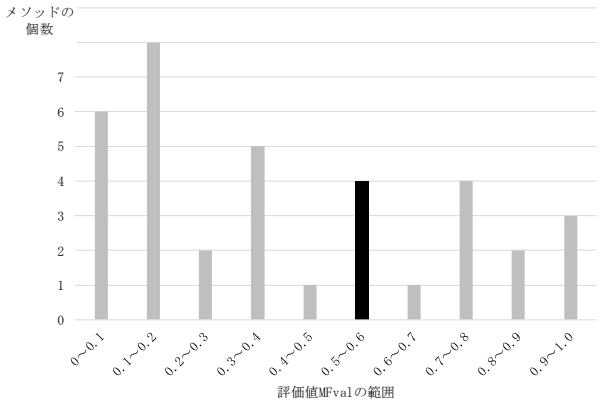

図 3 メソッドごとの評価値

し，サーバ環境は CentOS7 を仮想的に構築した（表 2 参照）．今回の検証方法を以下に示す．

Step(1). OpenPNE プロジェクトを管理する Redmine からサーバサイドに関するバグを無作為に取得する．

Step(2). Step(1)のバージョンに変数の値を出力するプログラムを埋め込み，サーバにデプロイする．

Step(3). 一般ユーザに試験的に利用してもらいログを蓄積する．ただし，バグに関する情報はユーザに通知しない．

Step(4). 1 度だけバグが起こる操作を行い，ログデータを保存する．

以降，Step(3)で取得したログを正常ログと呼び，Step(4)で取得したログをエラーログと呼ぶ．

4.2 実験対象とするバグについて

本実験で対象とするバグ内容は,「プロフィール入力値が空の場合の公開設定がプロフィール編集時のデフォルト値に反映されるものとされないものがある」である[1].
プロフィールを編集した時に，ある項目，例えば性別の値が設定されずに公開設定をデフォルト値から異なるものに変更させても，デフォルトの公開設定になるというものである．値が設定された場合は公開設定の変更が反映される．この原因は，値が正常かのチェックをし，値が正常でなければ性別等に関するデータをデータベースから削除することである．これにより，値と一緒にデータベースに登録されている公開設定も削除されてしまうため，デフォルト値になる．

4.3 計測結果

今回の実験では利用者数 10 人，運用期間は 7 日間，収集した正常ログのデータ量は 2.25GByte のデータであり，正常ログには 7401 箇所，エラーログには 1092 箇所の代入命令があった．エラーログにある 1092 の代入命令に対し，出現頻度を計測した結果，検証対象としたバグの原因となる代入命令の出現頻度は 0.511 であった．またエラーログにある 1092 の代入命令のうち，出現頻度が 0 と 1 以上の値を取り除いた個数は 270 個であった．

続いて，代入命令の出現頻度を計測した結果を図 2 に示す．図 2 の結果はエラーログに存在した値を 0.1 刻みでカウントしており，出現頻度が 0 と 1 以上の値は除外した結果を示している．

次に，メソッドごとの評価値を計測した結果を図 3 に示す．図 3 の結果も図 2 と同様に，0.1 刻みでカウントし，評価値が 0 と 1 以上の値を除外した結果を示している．

結果，バグ原因となった変数の代入命令は全体の中央に近い値になった．つまり，稀に出現する値というよりは正常ログの中に半分程度に障害発生時と同じ値が代入されているという結果になった．また，エラーログにある代入命令の内，出現頻度の値でも中央値と同じ程度であった．そこで，計測結果の平均，中央値，偏差，分散をまとめたものを表 3 に示す．

表 3 より，出現頻度が 0 と 1 以上の値を含んだ場合の平均は約 0.4，中央値は 0 であった．中央値が 0 なのは出現頻度が 0，つまり値が常に同じ代入命令が 1092 個中，600 個

表 3 出現頻度の平均・中央値・偏差・分散

全体	
平均	0.409
中央値	0.000
偏差	0.951
分散	0.905

表 4 出現頻度から 0 と 1 以上の値を除いた出現頻度の平均・中央値・偏差・分散

評価値0, 1以上を除いたもの	
平均	0.523
中央値	0.500
偏差	0.269
分散	0.072

1 http://redmine.openpne.jp/issues/3602

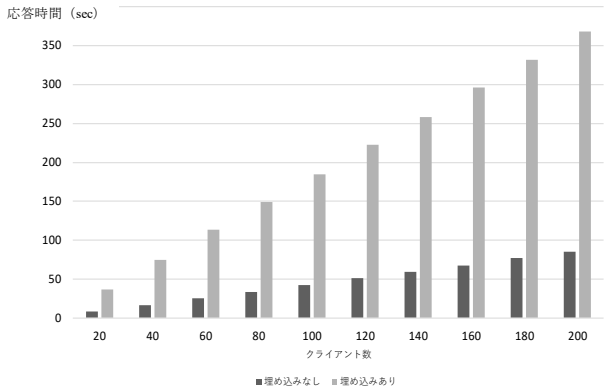

図 4　性能実験の結果

以上あったためである．そこで，出現頻度 0 と 1 以上の値を除外した結果を表 4 に示す．結果，バグ原因となった代入命令の出現頻度の値が平均を下回る結果となった．

次に，評価値の分析結果を表 5 に示す．今回，評価値の粒度をメソッド単位とし，エラーログで実行されたメソッド数は 130 メソッドであった．また，130 メソッドの内，評価値の値が 0 と 1 以上の値を除いたメソッド数は 36 個となり，バグ原因となったメソッドは 22 位という結果となった．つまり，全てのメソッド 15117 メソッドの中で 22 番目に属しており，更に動作しているメソッド等を調査することでバグ原因箇所を早急に発見する事ができる可能性が示唆できた．

4.4 性能実験

本提案では全ての代入命令があった場合にテキストデータに変数の値を記録する．そのため，著しく性能が低下する事が容易に予想できる．そこで，性能がどの程度低下するかの調査を行った．

調査方法は検証でも利用した OpenPNE の同じバージョンにログ機能を埋め込んだものと，埋め込まなかったソフトウェアを同一のサーバで動作させ，Apache JMeter[12]を用いて性能テストを行った．性能テストはそれぞれのソフトウェアのトップページに対して同時に 20～200 アクセスさせ，全てのリクエストが完了するレスポンス時間を計測した．計測結果を図 4 に示す．

表 5 代入命令，メソッドに関する分析

代入命令に関する分析	
全ての代入命令数・・・(1)	32861
バグ発生時に呼び出した代入命令数・・・(2)	1092
(2)から出現頻度0，1以上を除いたもの・・・(3)	270

メソッドに関する分析	
全てのメソッド数・・・(4)	15117
バグ発生時に呼び出したメソッド数・・・(5)	130
(5)から評価値0，1以上を除いたもの・・・(6)	36

計測結果より，リクエスト数に関係なく，約 4 倍以上の時間が掛かる結果となった．特にファイル書き込み量が膨大であるため，IO によるオーバーヘッドが非常に大きい事がわかった．今後，実際の運用でも利用できるためにも改善が必要である．

5. 考察

5.1 計測結果について

計測実験では，バグ原因となった代入箇所の出現頻度が 0.511 であった．出現頻度は 0 に限りなく近くなれば稀に出現する値となるが，0.511 という値の場合は過去の実行履歴にも約半分ほど同じ値が書き込まれていたという事になる．この点について考察する．

出現頻度はある代入命令に対し，代入された値が過去に何回現れているかを全実行回数で割ることにより算出される．つまり，対象となる代入命令自体の実行回数が少なければ出現頻度の値は高い傾向になる．これにより，実行回数が少ない代入命令では出現頻度の精度が低くなる傾向にある．

今回対象としたバグの発生に関する操作は，ユーザプロフィールの更新というものであり，一般ユーザが 1 度設定すれば以降操作しない機能である．更に，今回の被験者には何も言わずにシステムを利用して貰ったため，ユーザプロフィールの機能があまり使われず，バグがあった機能の実行回数が著しく低下した事により，あまり良い値にならなかった．

本提案では実行回数に依存するため，提案手法の精度をより良くするためには，より多くのログを収集する必要があることが分かる．本実験は利用者数 10 人で運用期間 7 日間という設定であったが，利用者数を増やし，運用期間を延ばすことにより，精度を高くすることが可能であると考えられる．

5.2 有用性について

本提案では代入命令ごとにログにその値を記録するため著しくサーバ性能を低下させる．性能実験の結果より，提案手法を適用したものは約 4 倍以上の処理時間が長くなることが分かる．本提案を実際の運用で利用することはパフォーマンスの問題上，非常に難しい事が分かる．本提案を利用するにあたりサーバの負担をどれだけ軽減させるかが重要である　そこで，提案する出現頻度を用いたサーバの負担軽減方法について考察する．

本提案では稀に変数の値が変わる事によって発生するバグのみを対象としている．つまり，出現頻度の値が 1 以上，または 1 に近い値は本提案では対象外のアプローチである．そこで，ある一定回数実行するごとに全ての出現頻度を求め，その段階で 1 以上，もしくは 1 に限りなく近い（閾値）

の場合はログに書かない事により，無駄な IO を減らす．

今回の性能実験で分かったことはファイル書き込みが莫大となり，IO がオーバーヘッドになっている点が大きな性能低下に繋がっていることである．そこで，IO への負担を軽減することによってサーバ性能を向上させることができると考えられる．つまり，データベースを用いることや，for 文や while 文において，同一箇所の代入命令が複数回行われる場合に，ログ出力を毎回行わず，繰り返し処理が終了した時点でまとめて書き込むことにより IO への負担を軽減する．これにより，改善の余地があると考えられる．

5.3 結果の利用方法

本提案の利用方法について述べる．従来，ソフトウェアに障害が発生した際には大きく 2 つの手間が発生する．1 つ目は同様の現象を再現する事，2 つ目が現象の原因となるメソッドや箇所を特定する事である．

1 つ目の問題点は，現象を再現するためにはある程度のコードの知識が必要となる事である．例えば，特定データの際にのみ起こりうる現象の場合，その現象が起きるデータを作成し，更にその時の変数の実行状態を記録する必要がある．単純に現象を再現するためにもソースコードを熟知していなければ再現できないケースも想定できる．本提案では，ログに全ての実行された変数の状態が保存されており，疑わしい値を持つ変数を検出するため，現象の再現をするために必要なデータを作成する手間が省ける．

2 つ目は，障害の起こったソフトウェアを熟知した開発者ならばこの点については問題ないが，熟知しない開発者の場合，ある機能が実行された時に，どのメソッドが実行されるかの調査から始めなければならないことである．規模が小さければ容易であるが，規模が大きくなるにつれ調査するコストが多大になる．本提案は，ソースコードのロジックや，設計等に縛られず，実行履歴から疑わしい値を検出する事ができるため，熟知しない開発者であっても疑わしい値の周辺のロジックを確認することでバグの修正が可能となる．

このように，提案する出現頻度からバグの原因となる箇所の目安を付ける事の支援ができる．具体的には，メソッドの評価値が低い順にソフトウェアプログラムの調査を行い，メソッド内部の代入命令で出現頻度の低い値が代入されたときに, どのような処理が行われるか調査することで，そのメソッド及び代入命令がバグ原因であるかの検証を行うことができる．

6. おわりに

本研究では実行時の変数の代入命令に着目したバグの原因となる変数の予測をする試みを行った．バグの原因となる変数は稀にしか出現しない値であるというアプローチを基に値の出現頻度を計測する手法を提案した．出現頻度は過去の実行履歴からバグが発生した実行履歴とで比較され，同じ変数の代入命令によって計測され，過去に同じ値が代入されたことがあるかによって計測される．

提案手法をオープンソースプロジェクトである OpenPNE に適用し，提案手法の有用性を調査した．結果，無作為に選んだ障害に対し，その原因となった変数を調査した．結果，提案する出現頻度が利用できる可能性を示唆する事ができた．今後の課題としては，対象とするバグを増やし，更なる調査を増やす必要がある．

参考文献

[1]. WordPress, https://github.com/WordPress/WordPress
[2]. 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，日本ソフトウェア科学会 コンピュータ ソフトウェア , Vol.32 , No.3 , 2015.
[3]. Shrinivas Joshi, Alessandro Orso , SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions , The International Conference on Software Maintenance, 2007.
[4]. Bil Lewis, Debugging backwards in time. International Workshop on Automated Debugging, 2003.
[5]. Yanyan Jiang, Tianxiao Gu, Chang Xu , Xiaoxing Ma, CARE: cache guided deterministic replay for concurrent Java programs , The 36th International Conference on Software Engineering, 2014.
[6]. 宗像 聡, 梅川 竜一, 上原 忠弘, 再生可能スタックトレースによる Java Web アプリケーションの例外発生過程の調査, ソフトウェア科学会 ソフトウェア工学の基礎ワークショップ(FOSE),pp.23 － 32, 2017.
[7]. 櫻井 孝平, 増原 英彦, 古宮 誠一 , Traceglasses：欠陥の効率良い発見手法を実現するトレースに基づくデバッガ，情報処理学会論文誌 プログラミング , Vol.3 , No.3 , 2010.
[8]. J, Quante. and R, Koschke.: Dynamic object process graphs, Journal of Systems and Software, Vol.81, pp.481501, (2008).
[9]. 中原 進, 紫合 治，オブジェクト指向プログラムの動作の可視化，情報処理学会研究報告, 2010.
[10]. 下村 隆夫, Case ツールの開発におけるソフトウェアバグの分析, 情報処理学会論文誌, Vol.35, No.7, 1994.
[11]. OpenPNE, https://www.openpne.jp/about/
[12]. Apache JMeter, https://jmeter.apache.org/