2008－OS－108 (7)
2008／4／23

# プログラム改変に伴う異常動作検出に関する検討

一柳 淑美† 木原 誠司† 盛合 敏†

† 日本電信電話株式会社 サイバースペース研究所
E-mail: †{ichiyanagi.yoshimi,kihara.seiji,moriai.satoshi}@lab.ntt.co.jp

本稿では，プログラムの開発環境と運用環境の違いによって発生するエラーの原因を解析し終えるまでの時間を短縮する方式について提案する．プログラムとは，サービスを提供するものである．また，サービスとは時間が経つにつれ他者のサービスと比較して相対的に劣化してしまうため，プログラムを改変する必要がある．プログラム改変とは，ソースコードを変更し実行ファイルを更新することである．このソースコードを変更する開発環境は，運用環境と同一の環境ではないため，運用環境において，開発者が想定していない値がプログラムに入力されることや，開発者が想定していない順序でプログラムの処理が実行されることがある．このような事象を契機として，運用環境ではエラーが発生する．このようなエラーの解析期間を短縮するため，自動的にプログラムの動作情報を取得する機能と，取得した動作情報からバグの可能性の高い箇所を抽出するアルゴリズムを検討する．具体的には，プログラムのバイナリコードを変更することによって自動的に動作のログを取得する方式と，サンプルとなる動作情報から生成した実行パターンと監視対象となるプロセスの動作情報を比較する方式について述べる．

# Efficient Detection of Bugs Caused by Software Upgrade

YOSHIMI ICHIYANAGI†, SEIJI KIHARA†, and SATOSHI MORIAI†

† NTT Cyber Space Laboratories
E-mail: †{ichiyanagi.yoshimi,kihara.seiji,moriai.satoshi}@lab.ntt.co.jp

In this paper, we propose a technique to analyse errors which could not be detected in the development environments, however reveal in deployed environments after deployment. Software bugs may remain even after the testing in the development process. The problem is that all software bugs do not show up in the test process, because there are differences between the development environments and the deployed environments, which results in a situation that the software comes to a state test engineers did not expect. To solve this problem, we focused on the difference of call patterns of functions between the developments environments and the deployed environments. In this paper, we describe our method to log all executed functions with low overhead via binary instrumentation and to extract call patterns from the logged data to help debugging after detecting software errors.

## 1. はじめに

プログラムとは，サービスを提供するものである．サービスは，時間が経つにつれ，他者のサービスと比較して相対的に劣化してしまう．サービスを劣化させないためには，ソースコードを改変し実行ファイルを更新することによって，サービスを改善する必要がある．このソースコードを改変し実行ファイルを更新することをプログラム改変と定義する．このプログラム改変を契機とするエラーが発生しないことを保証することはできない．

プログラムを改変した後に運用環境でエラーが発生した場合，ロールバック，プログラムを再起動する，プログラムのソースコードを修正するといった対策が行われる．エラー対策としてロールバックを選択した場合，このエラーのバグを修正する必要はない．しかし，プログラムを改変しないこととなるため，サービスが他者と比較して相対的に劣化してしまう．そこで，プログラムを改変することによって顕在化したバグを修正する必要がある．このとき，顕在化したバグを修正し，エラーが発生してからサービスを提供するまでの時間を短くするほど，改変による利益を増加させることができる．したがって，エラーを発見してから，このエラーのバグを修正し，プログラムを更新するまでの時間，特に，エラーを発見してから，このエラーの原因を特定する時間 (以下，エラー解析時間と記す) を短縮する必要がある．これは，エラー

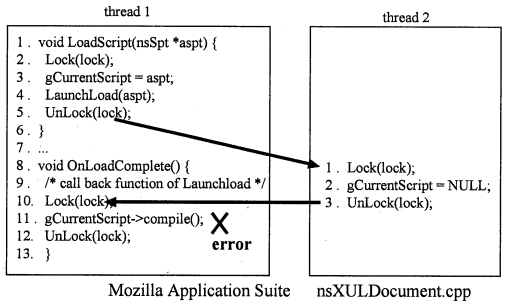


図 1　共有資源を破壊するエラー

解析時間が，エラーを発見してから修正するまでの時間の大部分を占めるためである．

エラー解析時間が長くなる原因の 1 つとして，エラーが発生したときには，このエラーの原因が実行された状況がすでに上書きされ消去される場合もある．エラー解析時間を短縮するためには，エラーの原因が実行されたときのプログラムの動作情報を取得しておく必要がある．しかし，Linux といったオープンソースで構築されたシステムにおいて，エラーを発見するための情報を収集する機能はあるが，エラーを解析するために必要となる情報を収集する機能はない．そこで，エラー解析時間を短くするためには，エラーが発生したときの情報のみではなく，プログラムの動作情報を，自動的に収集し，エラーの原因が実行されたときのプログラムの動作情報 (以下，エラー情報と記す) を開発者に提示する機能が必要となる．

本稿では，プログラムの動作情報から，このエラーの原因が実行されたときのプログラムの動作情報を抽出するために，開発環境と運用環境におけるプログラムの動作の違いに着目する．運用環境で発生するエラーとは，開発環境の検証では発生しなかったエラーである．例えば，運用環境において，開発者が想定していない値がプログラムに入力される場合や，開発者が想定していない順序でプログラムの処理が実行される場合がある．開発環境では発生しなかった事象を契機として，運用環境ではエラーが発生するため，運用環境と開発環境で動作するプログラムの動作の違いに着目する．

以上より，本稿では，プログラムの動作情報を自動的に取得し，この取得した動作情報からエラー情報を選択する手法を確立することを目的とする．

以下，本稿では，2. 節で，本稿で対象とするエラーと，このエラーを既存のデバッグツールで解決する場合の問題点と，この問題点を解決する提案手法の概要について述べる．次に，本稿で対象とするエラーと，プログラムの動作情報の関係について 3. 節で述べる．さらに，動作情報を取得する手法と，この取得した動作情報からエラー情報を取得する手法について 4. 節で述べる．また，これらの手法を実現したプロトタイプを作成し，HTTP サーバを用いた実験について 5. 節で述べる．最後に，本稿のまとめを 6. 節で述べる．

## 2. アプローチ

### 2.1 対象とするエラー

運用環境で発生するエラーは，運用環境と開発環境が異なることによって発生する．運用環境と開発環境の違いとして，以下の項目が挙げられる．

- プログラムの設定や入力値
- プログラムを実行する計算機のハードウェア，ソフトウェア構成
- プログラムの連続稼動時間

以上の違いによって発生するエラーを以下に示す．

**実行条件のミスによるエラー**　開発者の想定外の値がプログラムに入力された場合に発生するエラーである．一般的に，開発されているプログラムは複雑である．このため，開発者が入力されるすべての値を想定できた場合においても，有限期間内ですべての入力値を検証することはできない．そのために発生するエラーである．

**共有資源を破壊するエラー**　共有資源の排他制御の誤りによって発生するエラーである．実行する計算機のハードウェア構成やソフトウェア構成により，エラーが発生するか否かが異なる．そのために発生するエラーである．また，エラーの原因となる処理が実行されるタイミングと，エラーが発生するタイミングが異なるため，エラーの原因を特定することが難しいエラーである．文献 [1] に記載されている例を図 1 に示す．

**計算機資源のリークに関するエラー**　メモリリークのように，計算機資源の処理方法の誤りによって発生するエラーである．このエラーは，連続稼動時間が長くなることにより，性能低下などの問題を引き起こす．

### 2.2 既存のデバッグツール

実行条件のミスによるエラーや共有資源を破壊するエラーを解析するための情報を取得する手法として，以下の既存の手法が存在する．

- `print` 文の挿入

あらかじめ発生するエラーが予測できる場合，そのエラーに特化した `print` 文をソースコードに挿入することにより，エラー解析時間を短くすることができる．しかし，特定のエラーに特化するほど，ソースコードに挿入する `print` 文の位置と取得するデータが，プログラムに依存するという問題点がある．

- ダンプ

ダンプは，エラーが発生したタイミングのプログラムの状態を取得することができるが，エラーの原因が実行されたときのプログラムの動作情報を取得することができない．ダンプで得られるメモリやデバイスの状態といった情報は，データの粒度が細かすぎるため，エラー情報の候補を選択する処理が煩雑になる．具体的には，ダンプで取得したデータが，どのような意味を持つ情報であるのか特定し，データからエラー情報を選別し，選別した情報を分類し，それぞれの情報ごとに，エラー情報の候補を選択する処理が必要となる．

- トレーサ，ロガー

トレーサやロガーは，特定のバグを解析することを目的として挿入された `print` 文の挿入ほど，エラーの原因の特定に

効果的な動作情報を取得することはできない．しかし，プログラムに依存せず，エラー情報を取得することができる．また，トレーサやロガーで取得する動作情報は，ダンプほど粒度が細かくないため，エラー情報の候補を自動的に選択する処理がダンプより簡潔になる．

以上より，トレーサやロガーが，実行条件のミスによるエラーや共有資源を破壊するエラーの情報を取得し選択することに適しているといえる．本稿では，strace，ltrace といった既存のトレーサやロガーより，データを取得するためのオーバヘッドが低く，正確に動作情報となるデータを取得する方法を検討する．

計算機資源のリークに関するエラーを解析するための情報を自動的に取得する手法として，以下の既存の手法が存在する．

- モニタ，プロファイラ

このツールは，プログラムのエラー発生原因を特定することが目的ではなく，プログラムの監視を目的としている．このため，プログラムの計算機資源の使用方法に関したエラーを特定する場合には適している．

本稿では，モニタやプロファイラによって取得できる情報は，計算機資源のリークに関するエラーのバグを特定するための情報として十分であると考える．そのため，本研究では，既存の手法を用いることを検討する．

### 2.3 既存のトレーサやロガー

本研究において，モニタは既存手法を用いる．そのため，本稿では，トレーサやロガーについて特に検討し，既存手法の問題点を以下に整理する．

タイムマシーン [2] は，動作ログとしてメモリ，レジスタ，デバイスの状態を取得することにより，プログラムの動作を再現するツールである．タイムマシーンを使用した場合，エラーを再現することができるため，その発生条件も容易に調べることができる．しかし，ログを取得する頻度が高く，1 回で取得する情報量も多いため，運用環境で用いるには，オーバヘッドが高い．また，タイムマシーンの場合，取得されるプログラムの動作情報が，ダンプと同様に粒度が細かくなるため，エラー情報を選択する処理が複雑になる．

タイムマシーンより粒度の粗い情報を取得しつつ，エラーを再現する機構として Jockey [3] がある．Jockey は，システム依存のシステムコールやアセンブラなどの動作ログを取得し，その動作ログを用いて，プログラムの再現を可能とした機構である．ただし，Jockey は，マルチスレッドの実行順序を再現することはできない．これは，Jockey では，スレッドの実行順を記録していないためである．

strace，ltrace は，それぞれ，システムコール，ライブラリ関数の関数名，引数，返り値を表示するものである．システムコールやライブラリ関数に着目して動作情報を収集することによって，エラーがシステムコール，ライブラリ関数，ユーザプログラムのどこで発生しているのか，問題の切り分けができる．しかし，バグを特定するためには，プログラムの動作情報が不十分である．

上記以外にも，さまざまなトレーサ，ロガーがあるが，既存のトレーサ，ロガーは，動作情報を取得するオーバヘッドが高い，関数の引数の値を正確に取得できないといった問題がある．そこで，以下の機能を有するロガーが必要となる．

- プログラムの動作情報を取得するオーバヘッドが低い
- プログラムの動作情報として正確なデータを取得できる

### 2.4 提案手法のアプローチ

プログラムの動作情報を低いオーバヘッドで取得する場合，動作情報を取得する回数を少なくする，または，1 回で取得する動作情報の量を小さくする必要がある．そこで，取得するプログラムの動作情報の粒度が問題となる．以下に，動作情報の粒度を示す．下記の関数コールブロックとは，プログラムが実現するサービスにおいて，1 つの機能を実現する一連の関数コールを 1 つのブロックとしてまとめたものを指す．

（1） ソースコードのステップ
（2） 条件式
（3） 関数
（4） 関数コールブロック

上記の粒度のうち，下に記述されている粒度ほど粗くなる．粗い粒度ほど動作情報を取得する回数が少なくなるため，オーバヘッドは小さくなる．一方，細かい粒度ほどプログラムの動作情報が詳細に取得でき，プログラムの動作を正確に把握できる情報が取得できるため，バグが存在するソースコードの範囲を絞り込むことができる．本研究では，運用環境において動作情報を取得する手法を提案しているため，オーバヘッドを低くする必要がある．したがって，本稿では，エラー情報の粒度として関数を選択し，提案するロガーが対象とするエラーについて，以下の処理に着目する．

- 実行条件のミスによるエラーについては，頻繁に実行されない処理に着目する．このエラーは，プログラムが，開発者の想定範囲外の状態になった場合に発生するエラーである．頻繁に実行される処理とは，「プログラムが実現しているサービス」を提供している処理であると考えられる．このサービス提供処理は，さまざまな検証が行われていると考えられる．しかし，例外処理といった頻繁に発生しない処理は，検証が不十分になりやすく，エラーを発生しやすい．そのため，頻繁に実行されない処理に着目する．
- 共有資源を破壊するエラーについては，共有資源を対象とした処理に着目する．

## 3. エラー解析に必要となる情報

### 3.1 実行条件のミスによるエラーの解析に必要となる情報

実行条件のミスによるエラーを特定するために必要となるのは，エラーが発生するまでに実行された自プロセスの関数の入出力データや計算結果などの動作情報である．このエラーは，エラーが発生した時と同一の入力値が与えられた場合には，必ずエラーとなる．そこで，「繰り返し実行されない処理」に着目することにより，実行条件のミスによるエラーについての情報が取得できると考えられる．

関数とは，入力値によって出力結果が一意に決定する．そのため，関数の入力値に着目することによって，実行条件のミスによるエラーに対応するバグを推定できると考えられる．

### 3.2 共有資源を破壊するエラーの解析に必要となる情報

共有資源の破壊は，複数のプロセスが，競合状態になることによって発生する．したがって，エラーを推定するために

は，単一のプロセスの動作情報のみではなく，資源を共有する複数のプロセスの動作情報が必要となる．

例えば，図 1 の場合，エラーが発生する理由は，スレッド 1 の処理において想定されていない値が，スレッド 2 によって共有変数に格納されたためである．このとき，エラーはスレッド 1 で発生するが，バグはスレッド 2 が実行したソースコードに存在する．このとき，このスレッド 2 が共有変数を変更したという動作情報が重要になる．

共有メモリにアクセスしたプロセスについての情報を取得する方法として，以下の方法が考えられる．本稿では，実行された関数に着目する方法を用いる．

- 共有資源へのアクセスに着目する方法
- 実行された関数に着目する方法

**3.2.1** 共有資源へのアクセスに着目する方法

共有メモリにアクセスしたプロセスの情報を取得する方法として，Eraser [4]，Valgrind [5] の Helgrind，AVIO [1] といった，メモリといった共有資源のアクセスに着目する方法がある．プログラムの変数，この変数にアクセスするプロセス，そのアクセスの種類，をロギングし解析することにより，競合状態が発生するか否かを調べることができる．

この変数に着目した手法で取得できる情報のみを用いて，共有資源を破壊するエラーを発生させないように，ソースコードを修正することは難しい．例えば，図 1 のエラーについて，この手法を用いた場合，スレッド 1 の 3 行目，スレッド 1 の 11 行目，スレッド 2 の 2 行目によって競合状態が引き起こされる．しかし，図 1 の `LoadScript()` にエラー原因があるのか，`OnLoadComplete()` にエラー原因があるのか，それ以外の関数のコードにエラー原因があるのかを推定することは難しい．つまり，開発者に，共有資源にアクセスすることによって競合状態にさせるソースコードの位置を提示するのみでは，開発者に，ソースコードを修正するために必要十分な情報を提示していない可能性がある．ソースコードを修正するためには，関数の実行パスや，条件文の分岐の仕方といった，より詳細なプログラムの動作情報も必要となる．

さらに，共有資源へのアクセスに着目する方法をソフトウェアで実現する場合，そのオーバヘッドが問題となる．また，ハードウェアで実現する場合は，特殊なハードウェアが必要となるため，汎用性が失われる．以上より，本稿では，この方法を採用しない．

**3.2.2** 実行された関数に着目する方法

共有メモリにアクセスしたプロセスの情報を取得する方法として，実行された関数に着目する方法である．複数のプロセスで共有する変数を更新するプロセスが存在する場合，この共有変数について排他制御処理が必要となる．

本節では，まず，ポータビリティを考慮しているプログラムのエラー原因の推定法を考える．ポータビリティを考慮しているプログラムは，エラーの原因を推定しやすいようにプログラミングされている．そのため，本稿では，エラー情報を選択するという課題を簡単とするため，ポータビリティを考慮したプログラムについて検討する．

Apache HTTP Server(以下，httpd と記す)，PostgreSQL といった大勢の開発者で開発されているサーバソフトウェアは，共有資源にアクセスする関数を限定する傾向にある．そのため，同一の共有資源にアクセスをする処理は同一の関数によって実行される可能性が高い．このとき，複数のプロセスにおいて，同一の関数がどのプロセスから，どのタイミングで呼ばれたか，に着目することによって，共有資源へのアクセス手順を取得することができると考えられる．例えば，図 1 の場合，`Lock()`，`UnLock()` が，共有変数 *lock* を引数として実行されるブロックのなかで実行されている．

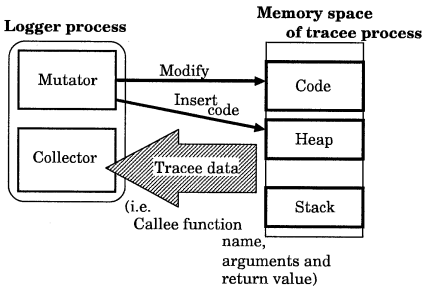


図 2　ロガーの構成図

また，httpd-2.0.48 では，複数のプロセスが同時に，`ap_buffered_log_writer()` を呼んだ場合，共有メモリのデータを破壊するというバグがあった．[1] ここで問題となるのは，引数で指定された共有資源に対するアクセスである．しかし，ポータビリティを考慮したプログラムは，共有資源にアクセスする関数が特定できる傾向にある．これは，ポータビリティを考慮したプログラムは，オブジェクト化や，モジュール化が進んでいるため、グローバル変数でアクセスすることが好まれない傾向があるためである．したがって，ポータビリティを考慮しているプログラムの場合，関数に着目することによって，共有資源を破壊するエラーを推定できると考えられる．

## 4. 提案手法の処理手順

### 4.1 ロガーの構成

提案するロガーの構成を図 2 に示す．Mutator は，トレース対象となるプログラムの動作を変更し動作情報を Collector に出力させる．Collector は，監視対象となるプログラムの動作情報を取得し，それをログとして保存するか決定する．

本稿では，CPU アーキテクチャ，OS，共有ライブラリ，コンパイラとして，i386，Linux，glibc，gcc を用いた場合の設計と実装について示す．以下，本稿では，プログラムが実行する `main()` が定義されているバイナリファイルのことを実行ファイルと定義し，プログラム実行時に実行ファイルにリンクされるバイナリファイルを共有ライブラリと定義する．

### 4.2 プログラムの動作情報の取得方法

ロガーは，プログラムの動作情報として，そのプロセスが実行する関数の情報を取得する．プロセスが実行する関数には，実行ファイル内に定義されている関数と，共有ライブラリに定義されている関数の 2 通りある．実行ファイルや共有ライブラリごとに，そのファイルで定義されている関数の実行時の情報を取得するか否かを選択できる機能をロガーに実装している．これによって，自身が作成したライブラリ内で

定義した関数の実行時の情報を取得できるため，ライブラリについてのエラー解析も可能となる．また，十分にデバッグされ，エラーを発生させる可能性が低いライブラリに関しては，動作情報を取得しないことにより，動作情報を取得するためのオーバヘッドを低くすることができる．ロガーが，動作情報を出力させるために，プロセスのコード領域を変更する処理を以下に示す．

（1） ロガーを起動する．

プログラムが起動している場合には，そのプロセスをシグナルで停止させる．

（2） 共有資源を扱う関数のラッパー関数を作成する．

共有資源を扱う関数の動作情報を取得するために，ラッパー関数を作成する．詳細については，4.2.2 節に記す．

（3） 監視対象となる関数をロガーに登録する．

関数名，この関数のアドレス値をロガーのテーブルに登録する．さらに，実行ファイルに DWARF [6] といったデバッグ情報が付加されている場合，その引数と返り値の型情報も取得し登録する．デバッグ情報とは，関数，引数，返り値，グローバル変数，ローカル変数の名前，型，保存位置などの情報である．このデバッグ情報が取得できない場合，関数の引数，返り値を正確に取得することができない．

（4） プロセスのコード領域を改変し，関数を呼び出す命令をロガーのフック関数に置き換える．

静的リンクされている関数の呼び出しをフックするため，プロセスのコード領域を書き換える．このとき，書き換えたメモリ内のアドレスをオペランドとする命令が存在した場合，このオペランドを適切な値に変更する．さらに，関数ポインタを使用して関数を呼び出す場合，ヒープ領域にコードを作成し，このコードを介して，関数をフックする．具体的な処理は，4.2.1 節に記す．

（5） プログラムを実行する．

プログラムを起動，または，プロセスを再開させる．

**4.2.1** 実行ファイル内に定義された関数のフック方法

動作情報の取得手法として，他のプログラムの機能を使用せず，監視対象となるプロセスが自身で動作情報を出力する手法を採用する．その理由は，他のプログラムによって動作情報を出力させる場合，コンテキストスイッチが発生することにより，動作情報を出力させる処理のオーバヘッドが高くなるためである．本研究では，運用環境におけるエラー解析を提案しているため，低いオーバヘッドで動作情報を取得する必要がある．そのため，オーバヘッドが低くなる，プロセス自身が動作情報を出力する方法を採用する．

さらに，この手法を実現する方式として，JIT(Just In Time) 方式と probe 方式がある．JIT 方式とは，JIT コンパイラを利用し，VM(Virtual Machine) 上でプログラムを実行することにより，プログラムの動作を変更する方式である．この方式を利用しているものとして，Eraser などで用いられている ATOM [7]，AVIO などで用いられている Pin [8]，Valgrind などがある．probe 方式とは，メモリにロードされた実行ファイル内に，トランポリンコードを挿入することにより，プログラムの動作を変更する方式である．この方式を利用しているものとして，Dyninst [9]，Dtrace [10] などがある．JIT 方式は，実行ごとにバイナリコードを変換するため，probe 方式よりオーバヘッドが大きくなる．一方，probe 方式は，実行ファイルを書き換えるため，メモリバリアやセキュリティ上の問題がある．本稿では，オーバヘッドの観点から probe 方式を採用する．

実行ファイルにおいて，関数の呼び出し方は，以下の 2 通り存在する．ただし，Linux を対象としているため，監視対象のプログラムが ELF(Executable and Linkable Format) であることを前提とする．

**同一ファイルに定義されている関数を呼び出す** 呼び出し元は，呼び出す関数のアドレスを指定して `call` する．

**異なるファイルに定義されている関数を呼び出す** 呼び出し元は，ELF の PLT(Procedure Linkage Table) で定義されている関数へのエントリを介して，関数を呼び出す．具体的な処理を図 3 に示す．異なる実行ファイルで定義されている関数を呼び出す場合，PLT に登録されている GOT(Global Offset Table) のエントリを参照する．

関数を呼び出す命令のオペランドとなるアドレスの値により，同一ファイル内に定義されている関数であるか否かを判定することができる．以下に，メモリにロードされた，プログラムの実行ファイルを書き換える方法を示す．

**呼び出し先の関数アドレスを定数値で指定する場合** 呼び出し先の関数のアドレスによって，呼び出し先の関数が監視対象であるか否かを判定する．監視対象となる関数の場合，動作情報を取得する関数を呼び出す命令に，監視対象となる関数を呼び出す命令を書き換える．

**呼び出し先の関数のアドレスをレジスタで指定する場合** この場合，実際にプログラムを動作させ，レジスタの値を取得しない限り，呼び出し先の関数が監視対象であるか否かを判定することはできない．そのため，動作情報を取得する関数を呼び出す命令に，レジスタで指定された関数を呼び出す命令を書き換える．命令を変更することによって，関数をフックし，呼び出し先の関数が動作情報を取得する関数か否かを判定する．

これらの書き換えを行う場合，書き換え先の命令のサイズが，書き換え元の命令のサイズより大きくなる場合がある．その場合は，監視対象を呼び出す命令より後の命令についても書き換える．このとき，プログラムの処理について不整合を発生させないために，実行権限を付加したヒープ領域に，書き換え元の命令をコピーし，実行ファイルを書き換えた後についても実行可能な状態にする．

**4.2.2** 共有資源を扱う関数のフック方法

実行ファイルに定義されていない関数以外に共有資源を扱う関数についても，実行した関数の情報を取得する．共有資源を扱う関数とは，複数のプロセスに資源を共有させる関数と，排他制御を扱う関数である．

複数のプロセスで資源を共有させる関数には，`fork()`，`clone()`，`shmat()`，`shmget()`，`shmdt()`，`mmap()`，`mmap2()`，`munmap()` システムコールがある．共有資源の破壊に関するエラーの原因を特定するためには，これらのシステムコールを呼び出したというプログラムの動作情報を保存する必要がある．一般的に，プログラムは glibc 内のライブラリ関数を介してシステムコールを発行する．よって，これらのシステム

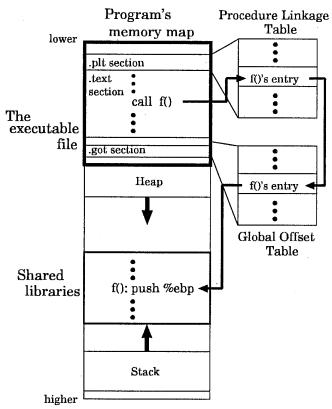


図 3　ELF における共有ライブラリ内の関数の実行手順

コールをフックするため，それぞれのシステムコールに対応した glibc ライブラリ関数をフックする．

排他制御を扱う関数とは，共有資源を競合状態にさせないための関数である．図 1 のように，この排他制御を扱う関数の実行履歴は，共有資源を破壊するエラーの解析に役立つ情報であると考えられる．そこで，排他制御を扱う関数についても，複数のプロセスに資源を共有させる関数と同様に，これらのライブラリ関数をフックし，関数に関する情報を動作情報として取得する．

### 4.3　プログラムの動作情報の選択

エラー情報を選択し保存する処理手順では，ストレージに保存するプログラムの動作情報の量を減らすことが重要となる．エラーの原因が実行されてからエラーが発生するまでの期間を推定することは困難である．そのため，プログラム起動時からの動作情報をストレージに保存する必要がある．よって，長期間の動作情報を保存するためには，保存するデータ量を減らす必要がある．

開発環境で実行したプログラムの動作情報と，運用環境で実行したプログラムの動作情報を比較し，動作情報の概要のみを保存する部分と，詳細な動作情報を保存する部分を選択する手法を検討する．これは，開発環境といった，開発者の想定している環境で実行され，想定している入力値によって実行されたプログラムの動作は，開発者が正常な動作であると想定している動作であると考えられるためである．さらに，運用環境で稼動していた前バージョンが存在する場合，保存するデータ量を減らすため，その前バージョンの動作情報も使用することを検討する．これは，稼動時間が長いプログラムほど，プログラムに対して開発者の想定範囲外の入力が行われる可能性が高まる．このため，より多くの項目について検証され，エラーを発生させる可能性が低いプログラムであると考えられるためである．

ただし，上記以外にも，共有資源を扱う処理については，計算機環境の変化に伴いエラーが発生する危険性があるため，詳細な情報を取得する必要があると考えられる．そこで，共有資源を扱う関数が実行されたという動作情報についても保存する手法も検討する．以下に，保存する動作情報を選択する手法の方針を示す．

- 開発環境で取得した動作情報や，運用環境で取得した前バージョンのプログラムの動作情報と比較し，同等の処理と判定した処理については，その処理の概要のみを保存しする．異なっていると判定した処理については，詳細な処理内容を保存する．
- 上記以外についても，共有資源を扱う処理については詳細な処理内容を保存する．

#### 4.3.1　動作情報の単位を作成する方法

同等の処理か否かを判定するため，動作情報から一連の関数コールを抽出し，1 つの処理を定義する．この 1 つの処理として開発者にとって意味のある処理が抽出することによって，開発者に提示するソースコードの範囲を的確に絞ることができ，エラー解析時間を短縮することができる．そこで，1 つの処理を抽出するために関数のスタックコールの深さに着目した．プログラムの動作において，コールスタックの深さの値が大きくなっていく部分は，実行する処理の前準備，または，処理の実行途中である．コールスタックの深さの値が前後の関数と比較し大きくなる関数は，処理を実行し，終了させる関数である．したがって，関数の呼び出し時のコールスタックの値に着目し，コールスタックの値が小さくなる関数の前で区切り，1 つの処理と定義する．

この 1 つの処理と定義する，関数ブロックを作成する方法を以下に示す．まず，開発環境や運用環境で取得した動作情報について，それぞれの動作情報に対して行う処理を以下に示す．

（1）　動作情報を区切り，コールパターンを作成する．(図 4 参照) このコールパターンに対して，関数名に着目したときに同様のパターンが他に存在するかどうかを調べ，存在する場合は，その関数の引数と返り値も同一かどうかを調べる．

（2）　同様のパターンが存在するコールパターン中の関数名がすべて同一の場合，それは，プログラムが繰り返し実行する処理である．このコールパターンを “繰り返す処理” とラベリングし，Collector に登録する．

（3）　同様のパターンが存在するコールパターン中の関数の引数，返り値の値が，実行された箇所によって異なるか調べる．このとき，引数，返り値の型がポインタの場合，そのポインタが示す値の型で調べる．ただし，このとき，ループを防ぐため，ポインタの階層が 2 段以上となる場合はそのポインタが示す値ではなく，ポインタとして，“頻繁に値が変化する変数” とラベリングし，Collector に登録する．具体的には，ポインタが示す値の型が構造体であり，その構造体のメンバにポインタが含まれていた場合，このポインタのメンバが対象となる．ポインタ以外の場合，同等の値を持つコールパターンが存在するか調べる．同等の値が存在する引数，返り値は，“数種類の値を取る変数” とラベリングし，取得した値とともに，Collector に登録する．すべてのコールパターンで異なる値を取る引数，返り値は，“頻繁に値が変化する変数” とラベリングし，Collector に登録する．すべてのコールパターンで同一の値を取る引数，返り値は，“変化しない変数” とラベリングし，その値とともに，Collector に登録する．

以上の 1 つの動作情報に対する処理が終了した後に，比較

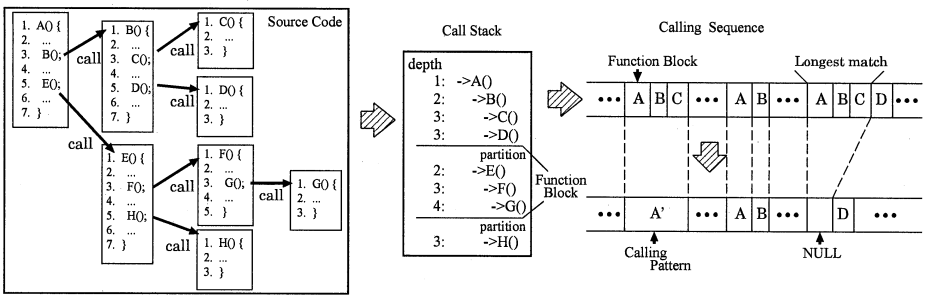


図 4　コールパターン

対象となる複数の動作情報に対して，以下の処理を行う．

（1） 取得した複数の動作情報に対して，他の動作情報で，関数名のみで比較した場合に同様のパターンが存在するか調べる．

（2） 同様のパターンが存在するコールパターンについて，まだ，ラベリングしていないパターンの場合は，“繰り返す処理” とラベリングし，上記の” 繰り返す処理” と同様に処理した後に，Collector に登録する．

（3） 上記以外のコールパターンは，例外処理など，頻繁に発生しない処理である．この処理を行うコードは，繰り返し実行するコードと比較し，プログラミングミス，詳細設計のミスなどを含む可能性が高い．その理由は，開発を行う場合，サービスを提供する処理を実現するために頻繁に実行されるコードを作成することに注力し，例外処理といった，サービスを提供する上で頻繁には発生しないと思われる処理は，その検証が甘くなる傾向があるためである．そのため，頻繁に発生しない処理は “警戒すべき処理” とラベリングし，コールパターンに含まれる関数の関数名，引数，返り値を Collector に登録する．

（4） 作成したコールパターンに ID 番号を付与し，一意に識別できるようにする．

**4.3.2** エラー解析に必要となる情報を選択する方法

監視対象となるプロセスの保存するデータ量を減らすため，4.3.1 節で作成したコールパターンと監視対象となるプロセスの動作情報を比較し，保存するエラー情報を選択する．具体的には，ラベリングされたものと一致したコールパターンは，“変化しない変数”，“数種類の値を取る変数” というラベリングされた引数，返り値について，登録されている値と一致するか否かを調べる．

一致した場合は，一致したコールパターンの ID 番号のみを動作情報として保存する．一致しなかった場合，一致したコールパターンの ID 番号と，その一致しなかった変数の値を動作情報として保存し，その一致しなかった変数に対して付与されているラベルなどを変更する．具体的には，一致しなかった変数に付与されているラベルが，“変化しない変数” の場合は，“数種類の値を取る変数” というラベルに変更し，動作情報のその変数の値を，コールパターン中に追加する．また，“数種類の値を取る変数” の場合，動作情報のその変数の値をコールパターン中に追加する．また，一致しないコールパターンについては，すべての関数名，引数，返り値を動作情報として保存する．さらに，このとき，同期問題を発生しやすい資源，複数のプロセスで共有しているファイルディスクリプタと共有メモリ内のアドレスを引数に持つ関数に関しては，“競合する資源” とラベリングし，その引数の値を動作情報として保存する．

監視対象となるプロセスにおいてエラーが発生した場合，保存したエラー情報から，バグの可能性が高い箇所を選択する処理を行う．このバグの可能性が高い箇所を選択するため，エラーが発生した時点より前に実行されているコールパターンのうち一致しないものについて，1 つの動作情報に対して行う処理と同様に処理する．次に，それぞれのラベルとともに，エラーが発生した動作情報を表示する．このとき，“警戒すべき処理”，“繰り返す処理” ごとの最後に実行された処理，“繰り返す処理” ごとの最後以外に実行された処理という順に，エラーの原因を含む処理である可能性が低くなるという旨も同時に表示する．

## 5. 実 行 例

Mutator のプロトタイプと，コールパターンを生成するプロトタイプを実装した．Mutator のプロトタイプは，gcc-2.95 以上からサポートされている “finstrument-functions” GCC オプションと，libdwarf [11] を利用して作成した．“finstrument-functions” とは，コンパイル対象となるソースファイルで定義されている各関数に，プロファイル関数を挿入するオプションである．これらのプロファイル関数は，ウィークシンボルで作成される．そのため，このプロファイル関数を使用して，呼び出された関数の情報を取得する機能を実装した．

さらに，Mutator で取得した動作情報から，コールパターンを抽出する機能を実装した．コールパターンとして抽出される関数コールシーケンスの理想は，開発者にとって，1 つの意味のなす処理が抽出されることである．例えば，httpd は，クライアントからの GET リクエストを受信した後に，Web コンテンツをクライアントに送信する．この一連の流れについて，GET リクエストを受信する処理，そのリクエストを解析する処理，Web コンテンツを送信する処理が，どのように関数ブロックで表現されているかを調べた．

以下に，httpd-2.2.8 における，PHP で記述されたファイルに対して GET リクエストを行った場合の結果の一部について示す．ただし，以下の結果は，httpd が作成する共有ライブラリである libapr-1.so と libaprutil-1.so についても，動作

情報を取得した．

```
1.  [11585]:[ap_update_vhost_from_headers]
2.  [11586]:[apr_table_get]
3.  [11586]:[apr_parse_addr_port]
4.  [11587]:[apr_palloc]
5.  [11588]:[apr_table_get]
6.  [11588]:[ap_add_input_filter_handle]
7.  [11589]:[add_any_filter_handle]
8.  [11590]:[apr_palloc]
9.  [11589]:[ap_run_post_read_request]
10. [11590]:[match_headers]
11. [11589]:[apr_table_get]
12. [11589]:[ap_update_child_status]
13. [11590]:[ap_update_child_status_from_indexes]
14. [11590]:[ap_process_request]
15. [11591]:[ap_run_quick_handler]
16. [11591]:[ap_process_request_internal]
17. [11592]:[ap_unescape_url]
18. [11592]:[ap_getparents]
19. [11592]:[ap_location_walk]
20. [11592]:[ap_run_translate_name]
21. [11593]:[translate_alias_redir]
22. [11594]:[try_alias_list]
23. [11594]:[try_alias_list]
24. [11595]:[alias_matches]
25. [11593]:[translate_userdir]
26. [11593]:[ap_core_translate]
27. [11594]:[apr_filepath_merge]
28. [11595]:[apr_palloc]
```

上記のログは，動作情報から抽出した関数ブロックであり，関数のコールスタックの深さの値と関数名である．深さの値は，最初に実行されたときの深さを示している．上記の一連の関数コールは，クライアントから PHP で記述されたファイルに対して 30 回 GET リクエストを送信したときの動作情報において，24 回検出された関数ブロックである．この関数ブロックは，GET リクエストの受信処理の関数群が抽出されている．例えば，1 行目の `ap_update_vhost_from_headersq()` から 11 行目の `apr_table_get()` は HTTP リクエストヘッダについての処理である．さらに，12 行目の `ap_update_child_status()` と 13 行目の `ap_update_child_status_from_indexes()` は，子プロセスのステータスを更新する処理である．14 行目の `ap_process_request()` から 16 行目の `ap_process_request_internal()` は，受信したリクエストを解析する処理である．17 行目の `ap_unescape_url()` から 27 行目の `apr_filepath_merge()` は，要求した URL のファイルパスを検索する処理である．28 行目の `apr_palloc()` は，apache のメモリを確保する関数である．

また，上記の処理の前処理として HTTP リクエストを受信する処理と，後処理としてファイルシステムから要求されたファイルを検索する処理が実行される．HTTP リクエストを受信する処理は，呼ばれる関数の出現パターンが一律ではなく，関数の数が 391，出現回数が 15 回となる関数ブロックや，関数数が 24，出現回数が 11 回となる関数ブロックなどがあった．これは，1 秒間で実行されるリクエスト数といった，要求される処理の負荷によって異なると考えられる．ファイルシステムから要求されたファイルを検索する処理は，関数数が 12 から 17 となる関数ブロックが，複数のプロセスで出現した．これは，httpd は，複数のプロセスで HTTP リクエストを処理しているためと考えられる．

以上より，プロトタイプを用いて実験した結果，開発者にとって意味のある一連の処理を抽出できることを確認できた．

## 6. おわりに

本稿では，運用環境でエラーが発生した場合，そのエラーの発生からエラーを特定するまでの時間を短くする手法を提案した．具体的には，エラーの原因を推定するための情報を自動的に取得するロガーと，そのロガーが取得した動作情報からエラーの原因となる可能性が高い動作情報を選択するアルゴリズムを提案した．また，ロガーについては，GCC オプションを用いたプロトタイプを実装し，アルゴリズムについては動作情報から関数の出現パターンを抽出するプロトタイプを実装した．また，このプロトタイプを用いて実験した結果，出現パターンとして，開発者にとって意味のある一連の処理を抽出できることを確認した．

今後，さまざまなプログラムに提案したアルゴリズムを適用し検証することで，このアルゴリズムを改良していきたい．

## 文　献


[1] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, pp. 37–48*, 2006.

[2] Samuel T. King, George W. Dunlap, Peter M. Chen. Debugging operating systems with time-traveling virtual machines. *Proceedings of the 2005 Annual USENIX Technical Conference pp. 1-15*, 2005.

[3] Yasushi Saito. Jockey: a user-space library for record-replay debugging. *Proceedings of the sixth international symposium on Automated analysis-driven debugging, pp. 69–76*, 2005.

[4] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems 15 No. 4, pp. 391–411*, 1997.

[5] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science 89 No. 2*, 2003.

[6] The dwarf debugging standard. http://dwarfstd.org/.

[7] Alan Eustace Amitabh Srivastava. Atom: a system for building customized program analysis tools. *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, pp. 196–205*, 1994.

[8] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 190–200*, 2005.

[9] Jeffrey K. Hollingsworth, Barton P. Miller, Jon Cargille. Dynamic program instrumentation for scalable performance tools. *Proceedings of Scalable High-performance Conputing Conferenceq, pp. 841–850*, 1994.

[10] Bryan M. Cantrill, Michael W. Shapiro and Adam H. LeventhalR. Dynamic instrumentation of production systems. *Proceedings of the 2004 USENIX. Technical Conference, pp. 15–28*, 2004.

[11] libdwarf. http://reality.sgiweb.org/davea/dwarf.html.