Vol.2015-OS-135 No.16
Vol.2015-EMB-39 No.16
2015/11/24

ReBucket を用いた Linux クラッシュログの分類と分析

佐久間 亮[1,a)]　吉村剛[1]　河野健二[1,]

概要：オペレーティングシステムカーネルには多くのバグが存在している．Windows や Linux にも多くのバグが存在しており，Microsoft では Windows Error Reporting (WER) と呼ばれるシステムを用いてクライアントからバグによって障害が起きた際の情報を集めている．集めたクラッシュログを WER システムを用いてバグごとに分類し，多くのユーザで発現するバグからデバッグを行っている．バグの分類にはコールスタックに注目して類似度を計算する事によって分類を行う ReBucket が知られている．しかし，Linux では ReBucket のようなバグの分類をおこなっていないため，適用できれば，開発者がクラッシュレポートを分類する負担を軽減することができる．しかし，Linux ではフレームポインタの最適化によりコールスタックが正確ではない場合があり，同じバグでも異なる情報を出力することがある．本研究は事前調査として ReBucket を Linux に適用した場合の効果について，定量的に評価をおこなった．調査の結果，f-measure が 0.75 と Microsoft 製品に比べ，低いことが分かった．そのため，我々は分類精度を向上するため，Linux カーネルのコールグラフを生成し，正確なコールトレースを取得した．正確なコールトレースを用いて分類した場合， f-measure が 0.75 と以前と変わらないことがわかった．そのため，我々はその結果を分析しリアルバグのコールトレースの性質についての考察をおこなった．

キーワード：エラーレポートの分類，デバッグ，

1. はじめに

近年オペレーティングシステム (OS) のカーネルは多機能，複雑化しており，ソースコードの量も膨大になっている．たとえば，Linux カーネルのバージョン 3.15.2 のコード量を SLOCCount (2.26) で調べたところ，約 1200 万行にもなっている [1]．それとともに，開発期間も決められているため，リリース前にソフトウェアのデバッグを完璧に行うことは難しくなっており，バグをすべて取り除くことは難しい．Palix らの調査でも Linux カーネルに多くのバグがあることが示されている [2]．

そのため，ソフトウェアのリリース後にもバグ修正のために，Windows Error Report(WER)[3]，Mozilla の Mozilla crash report[4]，Apple の Apple crash report[5] のようなエラーレポーティングシステムや，Bugzilla[6] や JIRA[7] のようなバグリポジトリを用いてユーザから不具合の情報を集める必要がある．このようなエラーレポーティングシステムやバグリポジトリには，そのソフトウェア利用者数に比例して多量のクラッシュレポートが送付される．たとえば，OS の Windows やそれに付属する製品は利用者が多く，Microsoft 社には 10 年で数十億のエラーレポートが送られており， 同じように Linux も利用者が多く，Red Hat 社はひと月に数千件のエラーレポートを受信している [8]．

このようなエラーレポートの中には同じ原因（バグ）によってクラッシュした際のエラーレポートが重複しており，デバッグの優先度付けのために，エラーレポートを同じ原因ごとに分類する必要がある．なぜなら，ユーザ側でクラッシュの頻度が高いバグ，すなわち送付されるエラーレポートの多いバグから優先的に対応することが重要だからである．しかし，何千，何万件のエラーレポートを手作業で解釈し，分類することは現実的ではない．そのため，エラーレポートをクラッシュの原因ごとに自動で分類する方法が研究されている．たとえば，Microsoft 社の提案する Bucketing[3] や ReBucket[9] やバグリポジトリの報告を分類する方法 [10] が提案されている．これらの手法を用いることで，エラーレポートを分類することができるため，クラッシュ頻度の高いバグからデバッグを行うことが可能になっている．特に ReBucket では Microsoft 製品のエラーレポートに対して高い精度で分類できることが示されている．しかし， Linux のエラーレポートに対してはこのような研究はされていない．そのため，ReBucket のような分類手法が有効なら，より効率的にデバッグをおこなえる．

[1] 慶應義塾大学
Keio Uniersity
[a)] sakuma@sslab.ics.keio.ac.jp

本稿は，事前調査として Linux に ReBucket を適用してエラーレポートの分類をした場合の精度について調査をおこなった．その結果，F-measure で 0.75 と Microsoft 製品に適用した場合の平均の 0.88 と比べ精度の低下が見られた．この原因のひとつとして，Linux ではコールトレースの出力方法が通常と異なり，コールトレースが不正確なものとなっている可能性があることが考えられる．そのため，本稿では Linux カーネルのコールグラフを作成し，正確なコールトレースを取得し分類を行い，結果の分析をおこなった．

実際に正確なコールトレースを取得し，分類すると正確なコールトレースを取得する前と比べて大きな変化は見られなかった．そのため，我々はその結果を分析しリアルバグのコールトレースの性質についての考察をおこなった．

本論文の構成を以下に示す．2 章では WER と ReBucket について，また Linux におけるエラーレポートを説明する．3 章では事前調査として Linux における ReBucket の有効性を示す．4 章では事前調査の分析と，Linux で ReBucket を用いた際の問題点を示す．5 章では Linux カーネルのコールグラフから正しいコールトレースの取得についてと，実際に取得したコールトレースと生データの違いについてを示す．6 章では 5 章で取得したコールトレースを用いた ReBucket の分類結果について示し， 7 章では関連研究， 8 章では結論と今後の課題に付いて示す．

2. 背景

2.1 Windows Error Report (WER)

Microsoft 社では Windows などの Microsoft 製品で起こった不具合の情報を Microsoft 社のサーバへ集めてデバッグをおこなっている．集められたエラーレポートはクラッシュの原因（バグ）ごとにグループ (Bucket) に分類される．同じバグに対応するエラーレポートは同じグループに，ひとつのグループにはひとつのバグに対応するエラーレポートのみが分類されることが理想の分類となる．このようにエラーレポートを同じバグごとに分類する方法として， Bucketing と ReBucket があげられる．Bucketing は Microsoft 製品特有のヒューリスティックスによって分類をおこなっており，Microsoft 製品以外に適用することは難しい．ReBucket はエラーレポートのコールトレース部分のみを用いて分類するため，Linux にも適用することができると考えられる．

2.1.1 ReBucket

ReBucket はエラーレポート間のコールトレースの類似度を計算し分類をおこなっている．ReBucket は 1) エラーレポート間のコールスタックの類似度の計算，2) 類似度を基にクラスタリング，というふたつの段階に分けることができる．

コールトレースの類似度は，ふたつのコールトレースの最大共通部分列長とふたつの尺度を基にした重み付けによって計算する．この尺度は一致している関数とクラッシュポイントの位置の差 (Distance to the Top Frame) と，一致している関数同士の位置の差 (Alighnment Offset) によって重み付けをおこなう．図 1 に具体例を示す．たとえば，f_1 では Distance to the Top Frame は 1 となり，Alignment Offset は共通する関数同士の位置の差なので，f_5 と f_7' では 2 となる．計算方方法を式 (1) - (3) に示す．最大共通部分列問題を解く方法として知られる動的計画法を用いてコールスタック C_1 ，C_2 の最大共通部分列を式 (1) のように解く．通常の最大共通部分列問題とは異なり，式 (2) で表す $cost(i,j)$ によって重み付けをおこなう．式 (2) で用いる $min(i,j)$ と $abs(i-j)$ はそれぞれ共通する関数の Distance to the Top Frame の最小値と Alignment Offset にあたる．また，c ，o は Distance to the Top Frame と Alignment Offset に対するパラメータで**??**で説明するようにその製品に対して最適なパラメータを用いることができる．最終的なコールスタック間の類似度 $sim(C_1,C_2)$ は式 (3) で求めることができる．

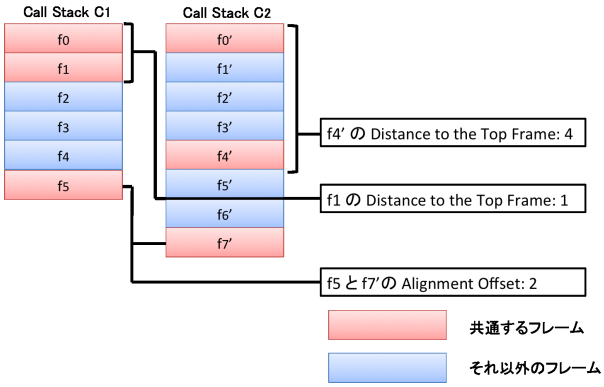

図 1 Distance to the Top Frame と Alignment Offset

$$M_{i,j} = max \begin{cases} M_{i-1,j-1} + cost(i,j) \\ M_{i-1,j} \\ M_{i,j-1} \end{cases} \quad (1)$$

$$cost(i,j) = \begin{cases} e^{-c*min(i,j)} e^{-o*abs(i-j)} & if\ f_i\ of\ C_1 = f_j\ of\ C_2 \\ 0 & otherwise \end{cases} \quad (2)$$

$$sim(C_1,C_2) = \frac{M_{m,n}}{\sum_{j=0}^{l} e^{-cj}} \quad (3)$$

式 (3) で計算されたコールスタック間の類似度をもとにクラスタリングし，クラッシュレポートの分類を行う．ReBucket では凝集型階層的クラスタリングによって，最も類似度が高いエラーレポートから階層的に併合していく．最終的に各クラスタ間の最大距離が閾値に達するまで併合をおこない，その際にグループになっているものが同じバグに対応するエラーレポートとなっている．

```
1 BUG: unable to handle kernel NULL pointer dereference at....
2 ......
3 Pid: 1902, comm: mount Not tainted 2.6.32.60 #1 PowerEdge T410
4 RSP: 0018:ffff8801005839f8 EFLAGS: 00010206
5 RAX: 0000000000000113 RBX: ffff880129962400 RCX: ......
6 ......
7 Call Trace:
8 [<ffffffffa020bc80>] ext3_get_blocks_handle+0x3b6/0x91d [ext3]
9 [<ffffffffa0216f01>] ? __ext3_journal_dirty_metadata+0x25/0x4f [ext3]
10 [<ffffffffa020d16f>] ext3_getblk+0x77/0x196 [ext3]
11 [<ffffffffa020d2a7>] ext3_bread+0x19/0x62 [ext3]
12 [<ffffffffa020fd3a>] ext3_mkdir+0x104/0x2db [ext3]
13 [<ffffffff81121ccb>] vfs_mkdir+0x73/0xcd
14 [<ffffffff81123d9c>] sys_mkdirat+0x99/0xed
15 [<ffffffff81120e54>] ? generic_permission+0x1c/0xa0
16 [<ffffffff81123e08>] sys_mkdir+0x18/0x1a
17 [<ffffffff81011cf2>] system_call_fastpath+0x16/0x1b
18 RIP [<ffffa020bc42>] ext3_try_to_allocate_with_rsv+0xc1/0x815 [ext3]
```

図 2 oops メッセージのサンプル

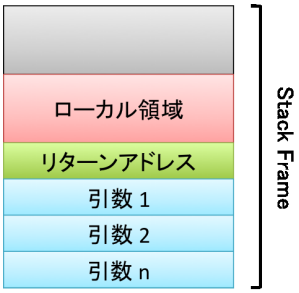

図 3 stack frame

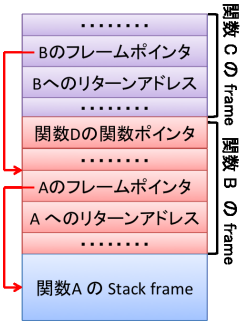

図 4 frame pointer

2.2 Linux のエラーレポート

Linux カーネルはクラッシュした際に，クラッシュログにあたる oops メッセージを出力する．これには，カーネルがクラッシュした際の情報が含まれており， ReBucket に必要なコールトレースもここに含まれている．oops メッセージの例を図 2 に示す．

oops メッセージの 1 行目には，クラッシュの原因が示される．例の場合は NULL ポインタを参照したことによってクラッシュしている．3 行目はクラッシュした際のプロセス情報，4 行目以降はレジスタの情報となっている．8 行目以降はコールトレースが出力されており，上に行くほどクラッシュポイントに近く，実際にクラッシュした関数は 18 行目に示されている．関数名の “+” 以降はバイナリでのオフセットを表しており，8 行目の最後にある “[ext3]” はモジュール名を表している．

2.2.1 コールトレースの “?” について

oops メッセージのコールトレースに注目すると，関数名に “?” がついている関数があることが分かる．図 2 の 9 行目の __ext3_journal_dirty_metadata() という関数にも “?” がついている．“?” がついた関数は Linux 特有のコールトレースの生成方法によって出力されたもので，実際に呼び出された保証がないものである．本節では Linux のコールトレース出力の仕組みを説明する．

コールトレースは各関数ごとのスタックフレームから構成され，関数呼び出しの際にフレームをスタックにプッシュし，処理が終了すると積んだフレームをポップすることによって，関数を呼び出す前の情報を取り出す．スタックフレームは図 3 に示すように関数への引数，リターンアドレス，フレームポインタ，ローカル領域などによって構成される．関数の処理が終了するとフレームに格納されたリターンアドレスを参照し，呼び出し元の関数の命令に戻ることができる．

フレームポインタはスタックフレームのベースアドレスを記憶しておくためのもので，ベースアドレスはフレーム内のリターンアドレスとローカル領域の間のアドレスを示している．通常，ベースアドレスは EBP レジスタに保存されており，関数呼び出しの際に呼び出し元の関数のベースアドレスをフレームポインタとしてスタックにプッシュして記憶している．フレームポインタはカーネルをビルドする際にコンパイルオプションで最適化することができるため，用いられない場合も多い．

通常のコールトレースの出力はフレームポインタを辿ることによっておこなわれている (Linux は少々異なる方法で出力されている)．図 4 はフレームポインタを辿る流れを示している．スタックに関数 A，関数 B，関数 C のフレームが積まれている場合，実行が終了した関数 C は呼び出し元の関数 B に戻るために，フレームポインタを参照し，関数 B のフレームのベースアドレスを知ることができる．さらに，関数 B のベースアドレスに隣接する形で関数 A のフレームポインタが格納されているため，このように順番に関数を辿ってコールトレースを出力することができる．

Linux のコールトレースの出力は通常のコールトレースの出力方法と異なる．Linux の場合，フレームポインタがスタックに積まれていないことを前提にしているため，積まれているスタックを上から見ていき，シンボルテーブルの情報からカーネル内の有効なアドレスを指している場合その関数名を “？” 付きで出力する．フレームポインタを有効にしている場合も同様で，その場合はフレームポインタに隣接するアドレスに格納されるリターンアドレスに対応する関数のみを “？” の付かない通常の関数として出力し，それ以外の関数は “？” 付きで出力をおこなう．そのため，フレームポインタを最適化した場合コールトレースで出力される全ての関数が “？” 付きで出力されてしまう．

このように，Linux カーネルの出力するコールトレースは Microsoft 製品の出力するコールトレースよりも正確でない可能性がある．そのため，コールトレースの情報を用いてエラーレポートの分類をおこなう ReBucket は，Linux では有効でない可能性がある．

3. 事前調査

我々は Linux カーネルのクラッシュログを用いて， Re-

Bucket を適用した際の効果について事前調査をおこなった．

3.1 目的

エラーレポートをクラッシュの原因ごとに分類することは，デバッグの優先度付けや効率化に有効である．分類手法の中でも ReBucket は高精度であることが示されており，Linux のエラーレポートでも高精度で分類できるなら，より効率的にデバッグをおこなうことができる．しかし，2.2 で示したように，Linux カーネルの出力するコールトレースは通常と異なるため，ReBucket が有効でない可能性がある．そのため，我々は Linux エラーレポートに ReBucket を適用した際の精度について定量的に評価をおこなう．

3.2 調査方法

ReBucket の精度を評価するために，クラッシュの原因 (バグ) が明らかで，さらに同じバグに対して複数のエラーレポートを持つデータセットが必要となる．しかし，実際の Linux のバグによってクラッシュした際のエラーレポートをデータセットとすることは難しい．なぜなら， Red Hat の提供する [8] のようにユーザから送付されたエラーレポートではクラッシュの原因となるバグが判別できないので，分類の結果が正しいか判断できないためである．また Bugzilla のようなバグリポジトリに報告されたバグを再現して，複数のクラッシュログを集めることも困難で現実的ではない．

そのため, 本研究は Linux カーネルにソフトウェアフォールトインジェクションをおこない Linux カーネルバグを再現した．フォールトインジェクタには，SAFE を [11] を用いた．これは，実在するソフトウェアバグを調査した [12] バグモデルを基に作成されており，リアルバグに則している．

本調査では， SAFE によって生成された 7096 種類のバグをひとつずつカーネルに挿入した．このバグが挿入されたカーネル上でワークロードを実行して，クラッシュした際に出力されるエラーレポートを用いて分類を行った．また，分類を行うためにひとつのバグに対して複数のエラーレポートが必要であるため，5 つの異なるワークロードを実行してエラーレポートを収集した．5 つの異なるワークロードを用いる理由は，同じバグでもユーザによって異なるワークロードによってクラッシュを引き起こすためである．

上記の方法によって得たエラーレポートを用いて ReBucket で分類し，その精度の評価をおこなう．

3.3 調査環境

調査に用いたカーネルは，vanilla カーネルのバージョン 2.6.32.60 で，バグの挿入はファイルシステムの ext3 におこなった．クラッシュログを集めるために使用したワークロードは UnixBench (ub) ver.5.1.3 [13]，FileBench (fb) ver.1.4.9 [14]，DBench (db) ver.4.0 [15]，IOzone (iz) ver.3.420 [16]，Linux Test Project (ltp) [17] の 5 つである．また，同じワークロードでも出力されるコールトレースが異なる可能性があるため，1 種類のバグに対して同じワークロードを 5 回実行させた．上記の方法で収集したエラーレポート全 10105 件，バグの種類 843 をデータセットとして分類をおこなう．

3.4 評価指標

本調査では ReBucket で用いられている評価指標の Purity，InversePurity，F-measure を用いる．この 3 つの指標は Precision と Recall をもとに計算する．C を今回分類された各クラスタ (Bucket) として，L を実際のバグのカテゴリとすると，C_j つまり C の j 番目の Bucket と，L_i つまり L の i 番目のバグの種類の Precision と Recall は以下のように計算できる．

$$Precision(L_i,C_j) = \frac{|L_i \cap C_j|}{|C_j|} \tag{4}$$

$$Recall(L_i,C_j) = \frac{|L_i \cap C_j|}{|L_i|} \tag{5}$$

上記の式をもとに Purity，InversePurity，F-measure を計算する．N は分類した Bucket の総数を表す．

Purity は各 Bucket の 最大 Precision の加重平均によって計算できる．

$$Purity = \sum_j \frac{|C_j|}{N} max_i\{Precision(L_i,C_j)\} \tag{6}$$

Purity は同じグループ（Bucket）に分類されたエラーレポートのうち，実際に同じバグに対応するエラーレポートの割合を示す．

InversePurity は各バグカテゴリの最大の Recall の加重平均によって計算できる．

$$Purity = \sum_i \frac{|L_i|}{N} max_j\{Recall(L_i,C_j)\} \tag{7}$$

InversePurity は同じバグに対応するエラーレポートが，どれだけ同じグループ（Bucket）に集められているかの割合を示す．

F-measure は Purity と InversePurity の調和平均となっている．

$$F - measure = \sum_i \frac{|L_i|}{N} max_j\{F(L_i,C_j)\} \tag{8}$$

$$F(L_i,C_j) = \frac{2 * Precision(L_i,C_j) * Recall(L_i,C_j)}{Precision(L_i,C_j) + Recall(L_i,C_j)} \tag{9}$$

これらの 3 つの値は 0 から 1 の値をとり，1 に近くなるほど分類の精度が高いことを示す．

	Purity	InversePurity	F-measure
Microsoft 製品の平均	0.925	0.907	0.876
“？” ありの場合	0.376	0.882	0.376
“？” なしの場合	0.283	0.924	0.345

表 1 各評価方法の Purity, InversePurity, F-measure

3.5 調査結果

本調査の結果を表 1 に示す．評価は Microsoft 製品の平均と本調査で得たエラーレポートのコールトレースの “？” ありと “？” なしの場合でおこなった．本調査はフレームポインタを有効にしているため，正確である関数と正確でない関数が分かるようになっているため，“？” ありの場合は正確な関数に加え “？” つきの関数を有効にして分類したものである．“？” なしは “？” がついている関数を取り除いている場合である．

結果を見てみると，Microsoft 製品の F-measure に比べ，本調査で得たエラーレポートの分類結果は “？” ありの場合で 0.376 , “？” なしの場合で 0.345 と大幅に精度が低下している．“？” ありと “？” なしの場合では，“？” ありの結果の方が精度が良く，また Purity と InversePurity がトレードオフの関係になっている．

実際に “？” の関数の影響を調査すると “？” つきの関数が類似していることによって本来異なるグループのエラーレポートが同じグループに分類されてしまうケースや，似ているが異なるバグのエラーレポートが “？” つきの関数の違いによって正しく分類されるケースが確認できた．

4. 事前調査結果の分析

本章では，3 章の結果についての分析をおこなう．

4.1 分類精度が低い原因

3 章の結果から，Microsoft 製品に比べ，本調査の結果は精度が低いことが分かった．この理由について分析する．

表 1 の Purity と InversePurity に注目すると，InversePurity に比べて Purity の値が大幅に低いことがわかる．これはつまり，分類精度の低下の原因は同じ Bucket に異なるバグのエラーレポートが多数分類されているためだと考えられる．

異なるバグのエラーレポートが同じグループに分類されてしまうのは，異なるバグのエラーレポートでもコールトレースが似ているからである．実際に同じグループに分類されているエラーレポートを見ると，対応するバグが異なるにもかかわらず全く同じコールトレースを持つエラーレポートを多数見つけることができた．ではなぜ，似ているコールトレースが多いのか分析すると，本調査では全 834 種類のバグを挿入したが，バグが挿入される関数の総数は 113 種類とバグの種類に比べかなり少ないことがわかった．つまり，本調査では同じ関数に多数のバグを挿入している

	Purity	InversePurity	F-measure
“？” ありの場合	0.739	0.900	0.756
“？” なしの場合	0.719	0.933	0.754

表 2 ランダムサンプリングした場合の Purity, InversePurity, F-measure

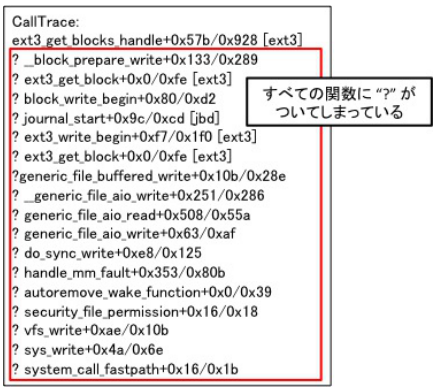

図 5 正確でない oops メッセージ

ことになる．同じ関数に挿入されたバグはクラッシュする際に似ているコールトレースを出力する可能性が高く，それが原因のひとつとなり ReBucket の精度が低下したのではないかと考えられる．

そこで，我々はエラーレポートの数を絞ることによって，どれだけ精度が向上するか調査をおこなった．エラーレポートを絞るために，834 種類のバグから 80 種類のバグのエラーレポートをランダムでサンプリングし分類する．80 種類というのは，ReBucket の評価では平均 75 個のバグの種類を扱っていたため，それに対応した数にするためである．また，偏りがでないためにランダムサンプリングを 10 回おこない，それぞれの分類精度の平均を算出した．

ランダムサンプリングした際の評価を表 2 に示す．F-measure が約 0.75 と，すべてのエラーレポートを対象にした場合の分類よりも大幅に精度が向上していることが分かる．また，Purity の値も約 0.4 向上している．しかし，このランダムサンプリングをしてバグの数を減らしても Microsoft 製品の場合に比べて，まだ少し低いことがわかる．

4.2 その他の原因

ランダムサンプリングして分類をおこなっても，Microsoft 製品に比べるとまだ精度が低いことが分かった．類似しているコールトレースが多いという原因以外の精度低下の原因を考えると，2.2.1 節で示したようにコールトレースが不正確な場合があることが原因のひとつだと考えられる．図 5 に不正確なコールトレースを示す．このコールトレースでは vfs_create() 以降の関数にすべて “？” が付いてしまっている．実際には “？” が付いている vfs_create() 以降関数には実際に呼び出している関数があるはずである．このように，“？” がついている関数が実

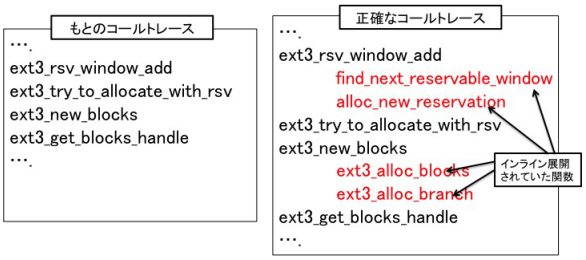

図 6 インライン展開されていた関数

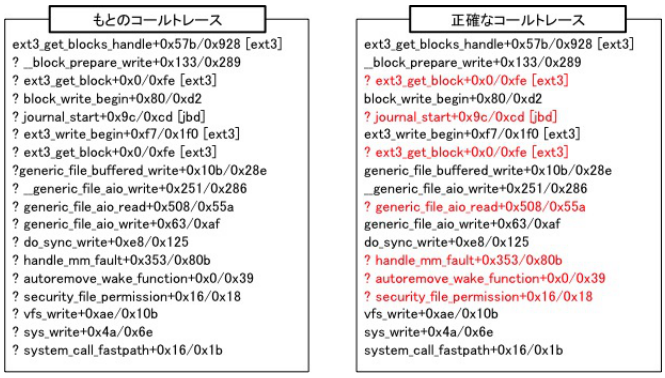

図 7 すべての関数に “？” がついているコールトレース

際は呼び出されている場合や，“？” が付いていない関数でも実際は呼び出されているケースがあると考えられる．また，コンパイルオプションによっては最適化の程度が異なり，インライン展開される関数も異なってしまうため，同じバグでもコールトレースが異なる場合があると考えられる．このように，Linux では不正確なコールトレースになってしまう場合があり，それによって分類の精度低下が起こっているのではないだろうか．

5. 正しいコールトレースの取得

4.2 節で述べたように， Linux では正確でないコールトレースを出力することがあり，それが原因で分類精度が低下しているのではないかと我々は推測した．そのため，我々は分類の精度向上のため Linux の出力するコールトレースから正確なコールトレースを取得し，それをもとに分類をおこなう．

コールグラフの作成：我々は正確なコールトレースを取得するために，Linux カーネルの LLVM-IR を生成し，それを解析することによって Linux カーネルのコールグラフを生成した．コールグラフにより，どの関数がどの関数を呼び出したか判別できるため，正確なコールトレースを取得することができる．今回，できるだけ多くの関数を取得するためコンパイルオプションの最適化では -O0 を指定し，最適化をおこなわなかった．

コールトレースの取得方法：正確なコールトレースの取得は，生データのコールトレースのクラッシュポイント（関数）から逆順に辿っていくことによっておこなう．これはクラッシュポイントは正確だと分かっているからである．具体的な流れはクラッシュを起こした関数をコールグラフから見つけ，その関数を呼び出している関数の一覧を取得する．そして，関数の一覧とクラッシュを起こした前の関数を比較し，一致するものがあればその関数は実際に呼び出された関数である．しかし，コンパイルオプションによっては関数インライン展開されていることもあるため，呼び出し元を 1 段階さかのぼるだけでは不十分なことがある．そのため，我々は 4 段階目までさかのぼる事によって正確なコールトレースを取得する．

	Purity	InversePurity	F-measure
“？” ありの場合	0.743	0.902	0.757
“？” なしの場合	0.723	0.933	0.755

表 3 正確なコールトレースを用いた場合の Purity, InversePurity, F-measure

5.1 取得したコールトレースと生データの違い

実際に正確なコールトレースを取得した場合に，もとのデータと異なるコールトレースを持つものを紹介する．図 6 に例を示す．図 6 の左にあるコールトレースがもとのコールトレースで，右にあるのが左にあるコールトレースから取得した正確なコールトレースである．右のコールトレースの赤字の部分に注目すると，左のコールトレースには無かった関数があることがわかる．これは，左のコールトレースを出力したマシンでは，カーネルをコンパイルする際にインライン展開されていた関数である．しかし，これらの関数は inline 指定されていない関数だった．

また，5 で示したような多くな関数に “？” が付いているコールトレースからも正確なコールトレースを取得することができた．実際に取得したコールトレースを図 7 示す．右にあるのが左にあるコールトレースから取得した正確なコールトレースで，赤字以外の部分は本当に呼び出されていた関数であった．

6. 正しいコールトレースを用いた分類

5 章で述べた方法で正確なコールトレースを取得し，そのデータをもとに ReBucket を用いて分類をおこなう．

6.1 調査結果

分類結果を図 3 に示す．図 2 の結果に比べて分類精度に大きな変化は見られなかった．また，F-measure 以外の値も殆ど変化が見られず，Purity が他の値に比べて Microsoft 製品に対して劣っている．このことから，正確なコールトレースを取得して分類しても，類似している異なるバグのエラーレポートを正しく別々のグループへ分類できていないということが分かる．

6.2 結果の分析

正しいコールトレースを用いた場合，分類の精度が Microsoft 製品の場合に比べて低いことが分かった．この原因として，図 3 に注目すると InversePurity が Microsoft 製品の場合と比べ，ほぼ同じ精度を持っているが， Purity が他の値に比べて低くなっていた．このことから，リアルバグの生成するコールトレースの性質について考察する．

Purity が低いということは，類似しているコールトレースが多いということである．我々の研究では，リアルバグではなく SAFE を用いて擬似的なバグを生成し，そのエラーレポートを分類した．その結果，類似するコールトレースが多くなり，Purity の低下という結果になった．しかし，リアルバグのエラーレポートで分類をしている Microsoft 製品の場合，Purity は 0.9 以上と高い値をとっている．このことから，リアルバグのコールトレースの場合，異なるバグに対応するコールトレースはそれぞれまったくことなるコールトレースを持つのではないか，と考えられる．この推測を検証するため，我々はリアルバグのエラーレポートについてのコールトレースの際について分析する必要がある．

7. 関連研究

Microsoft 社 は Windows Error Reporting を用いてユーザからクラッシュログを集めており，そのクラッシュログを Bucketing[3] によって root cause ごとに分類している．しかし，Bucketing は Windows に固有のさまざまな知識を利用したアドホックな手法となっており，Linux などの他の基盤ソフトウェア上のシステムに適応できるものではない．

また，Bucketing ではうまく分類できなかったクラッシュログを分類するために提案された方法として，Crash Graph [18] と 本研究で用いた ReBucket が知られている．Call Graph は Bucketing の分類で同じ root cause のクラッシュログが複数のグループに分類されてしまう問題に対処している．これは，分類されたグループ内のコールトレースを集約してグラフにすることで，他のグループとの比較を行い同じ root cause に対応するグループを判別するというものである．

一方，本研究で用いた ReBucket は，クラッシュレポートのコールスタックの部分に注目し，類似したコールスタックを持つクラッシュレポートを同グループに分類する．[9] によれば ReBucket は Bucketing に比べ 9 % の精度の向上が示されている．この二つのバグ分類方法は Windows 製品のバグを対象にしているが，本研究では Linux カーネルのバグを対象に分類を行った．

また，エラーレポートではなく，バグリポジトリに報告されたものをバグごとに分類する方法が提案されている [10]．これは，報告者のコメントなどについて機械学習をおこない，類似しているものを同じグループへと分類する．しかし，この方法は ReBucket にくらべて精度が低いことが知られている．本研究では，バグリポジトリのコメントのような自然言語ではなく，コールトレースを用いて分類をおこなっている．

Palix らは Linux 2.6 から 2.6.33 までのカーネルバグの調査をおこなっている [2]．Linux カーネルバグは 10 年間で殆ど変わらない件数を維持しており，これらのバグが原因でユーザから送付されるクラッシュログの分類が ReBucket で可能ならば，開発者の負担を軽減することができる．また，Palix らは Linux カーネルバグはファイルシステムのバグが多いこと示しており，本研究もファイルシステムのバグを対象にバグの分類を行っている．

8. まとめ

本稿では，事前調査として Windows Error Reporting で用いられるバグの分類方法である ReBucket を Linux カーネルバグに適用した場合の効果について調査し，定量的な評価を行った．ソフトウェアフォールトインジェクタの SAFE を用いてカーネルコード中にバグを挿入し，クラッシュした際のクラッシュログを収集した．集まったクラッシュログ全 10294 件，834 種類に ReBucket を適用してバグの分類を行った．分類結果から f-measure が 0.38 と Microsoft 製品にくらべて，大幅に分類精度が低下することが判明した．

分類結果を分析したところ，分類精度が低下した理由のひとつに類似するコールトレースが多く存在するためであるとわかった．これは，SAFE によって 834 種類のバグを 113 種類という少ない関数に挿入してしまったため，同じ関数に複数種類のバグが挿入されてしまい，類似したコールトレースが多くなってしまった．そのため，我々はバグの種類を限定するためにランダムサンプリングをして分類をおこなった．この結果，分類精度の平均は f-measure で 0.75 と大幅に向上した．しかし，この場合でも Microsoft 製品の精度にくらべ低い結果となっている．そのため，我々はもうひとつの精度低下の原因として Linux のコールトレースが不正確である場合があるためだと推測した．

そこで，我々は Linux カーネルのコールグラフを作成し，もとのコールトレースから正確なコールトレースを取得し，それを分類した．その結果，分類精度は f-measure が 0.75 と正確でないコールスタックを用いて分類した場合と比べて大きな変化は見られなかった．

本研究で取得した正確なコールトレースを分類した場合でも，Microsoft 製品の分類結果に比べて精度が低い理由を分析すると，Purity が Microsoft 製品に比べ低いことがわかった．これは，エラーレポートに類似するコールトレースを持ったものが多いことを示している．そのため，我々はリアルバグのコールトレースの特徴として，異なる

バグの場合，コールトレースは全く異なるものになるのではないかと考察した．

8.1 今後の課題

本稿では正確なコールトレースを用いても，Microsoft 製品に比べ精度が低いことが判明した．そこで，我々はリアルバグの場合，異なるバグのコールトレースは全く異なるものになるのではないかと考察した．この考察について分析するために，我々の持っている Red Hat に送付されたエラーレポートを使って ReBucket を適用し，異なる Bucket に分類されたエラーレポートを比べて，実際にどれだけコールトレースがことなるのかを検証する予定である．

参考文献

[1] Wheeler, D.: SLOCCount. http://www.dwheeler.com/.
[2] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J. and Muller, G.: Faults in Linux: Ten Years Later, *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*, pp. 305–318 (2011).
[3] Glerum, K., Kinshumann, K., Greenberg, S., Aul, G., Orgovan, V., Nichols, G., Grant, D., Loihle, G. and Hunt, G.: Debugging in the (Very) Large: Ten Years of Implementation and Experience, *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP '09)*, pp. 103–116 (2009).
[4] : MoZilla: Crash Stats. http://crash-stats.mozilla.com.
[5] : Apple: Technical Note TN2123: CrashReporter. http://developer.apple.com/library/mac/#technotes/.
[6] : Bugzilla. https://www.bugzilla.org/.
[7] : JIRA. https://ja.atlassian.com/software/jira/.
[8] : Linux Kernel Oops. http://www.kerneloops.org.
[9] Dang, Y., Wu, R., Zhang, H., Zhang, D. and Nobel, P.: ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity, *Proceedings of the 2012 International Conference on Software Engineering (ICSE '12)*, pp. 1084–1093 (2012).
[10] Banerjee, S., Cukic, B. and Adjeroh, D. A.: Automated Duplicate Bug Report Classification Using Subsequence Matching, *14th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2012, Omaha, NE, USA, October 25-27, 2012*, pp. 74–81 (2012).
[11] Cinque, M., Cotroneo, D., Natella, R. and Pecchia, A.: Assessing and improving the effectiveness of logs for the analysis of software faults, *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)*, pp. 457–466 (2010).
[12] Durães, J. and Madeira, H.: Emulation of Software Faults: A Field Data Study and a Practical Approach, *IEEE Transactions on Software Engineering (TSE '06)*, Vol. 32, No. 11, pp. 849–867 (2006).
[13] : UnixBench. http://code.google.com/p/byte-unixbench/.
[14] : Filebench. http://sourceforge.jp/projects/sfnet_filebench/.
[15] : DBENCH. http://dbench.samba.org/.
[16] : IOzone. http://www.iozone.org/.
[17] : Linux Test Project. http://ltp.sourceforge.net/.
[18] Kim, S., Zimmermann, T. and Nagappan, N.: Crash graphs: An aggregated view of multiple crashes to improve crash triage, *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11)*, pp. 486–493 (2011).