アクセスパタンを利用した同一節点へのグラフ走査回数の削減法

Reducing Graph Traversal for Same Node with Access Pattern

楠 和馬 [a]　久米 出 [b]　波多野 賢治 [c]
Kazuma Kusu　Izuru Kume　Kenji Hatano

1. はじめに

デバッグを効率的に行うためには，プログラムの実行時においての状態と，命令や値の間にある依存関係の調査が不可欠とされている [15, 16]．現在普及しているデバッガはブレークポイントで指定された箇所でプログラムを停止させ，その時点の状態を調査することができる．しかし，ブレークポイント以前の実行内容は参照できないため，不正な状態がどのような原因であるか特定する作業を効率的に実施できない [12].

既存のデバッガの問題を解決するために，依存関係を情報として含むトレースを利用した逆回しデバッガ（Back-In-Time デバッガ）と呼ばれる新しい方式のデバッガが開発されてきた [3, 8, 11]．トレースを利用することで，これらのデバッガは変数の値を代入した命令の特定や [8]，命令が実行された（あるいはされなかった）理由の調査 [3]，また，既に呼出し完了したメソッドの実行内容の調査 [11] のように，局所的な視点での依存関係の解析を実現する．

実際，我々の先行研究 [5] では不具合を含む実用的な Java のフレームワークアプリケーションに対してその感染を示唆する兆候を特定する動的依存解析を実現している．

このように我々の先行研究 [5] はトレースに含まれている依存関係に関する動的依存解析の規模の問題解決に明るい見通しを与えるものであるが，一方でその実行効率に関しては大きな課題を残している．実行効率の低下の主な原因はトレースのモデルに豊富なデータが含まれているところに存在する．我々のトレースはバグの原因となるような兆候の特定以外の側面で解析する際の要求を満たすために，Wang 等の研究 [14] のようにトレースのデータ量の抑制を目指す代わりにデータの豊富さを追求している．

既存のデバッガ [3, 8, 11] は特定の命令に関する局所的な動的依存解析を実行しているが，我々の先行研究 [5] ではプログラム実行時の全ての状態変更命令 * を解析の対象としている．そのため，規模の対応は可能であっても実行効率に関しては良い結果が得られていない．実行効率を高めるためには，動的依存解析の基本的な処理である参照関係や依存関係のような依存関係を辿るパフォーマンスを向上させる必要がある．

我々は今まで動的依存解析の処理効率を高めるために，トレースの格納方法や解析方法について考えてきた [6, 7]．これらの研究では処理効率をメモリの使用効率と処理時間で評価を行っており，メモリ使用効率に関しては貢献度が高かった．しかし，処理時間に対する貢献ができなかったため，本研究では，トレースの解析中に発生するグラフ走査の回数を削減することを可能にする方法を提案する．また，本研究で提案した場合，処理時間の削減にどの程度効果があったか，動的依存解析を実行し処理時間を計測することで評価方法について説明する．

2. 関連研究

現在広く使われているデバッガはプログラムコード中のブレークポイントで指定された箇所でプログラム実行を停止させ，作業者が停止時の状態を調査するための機能を提供している．この時点で既に呼出しが完了したメソッドの実行内容はデバッガに記録されていない．プログラムの不具合や感染 † はしばしば既に呼出しが完了したメソッド内に発見される [11]．既存のデバッガでこうしたメソッド呼出しを調査するためにはブレークポイントの設定と実行のやり直しが必要とされ，これがデバッグ作業の効率性を阻害する大きな要因となっている [12].

プログラムの実行履歴を利用することによってこうした既存のデバッガの限界を克服しようとする研究が最近の 10 年間で進められている．これらの研究の発端となった全知デバッガ [8] はある実行時点の変数の値に対してそれを代入した命令文を特定する機能を実装している．また，Ko らによる Whyline [3] はある命令文が実行された，あるいはされなかった過程を対話的に再現することを可能としている．

Lienhard らによる Dynamic Object Flow 解析 [9] はオ

[a] 同志社大学大学院文化情報学研究科 (Graduate School of Culture and Information Science, Doshisha University)
[b] 奈良先端科学技術大学院大学情報科学研究科 (Graduate School of Information Science, Nara Institute of Science and Technology)
[c] 同志社大学文化情報学部, (Faculty of Culture and Information Science, Doshisha University)

* インスタンス変数，クラス変数，配列への代入
† 不具合はプログラムコードの誤りを，感染は不具合箇所の実行に起因する実行時の誤りを意味する [16].

ブジェクトの視点から依存関係を解析，可視化する機能を実現している．Object Flow 解析はオブジェクトに対する参照に焦点を置いておりメソッドの依存関係解析 [10] であると同時にデバッガの利用を念頭に置いたオブジェクトの流れを表現している．

全知デバッガや Whyline による支援はある特定の命令文に対してその関連する依存関係を辿る機能によって実装される．こうした支援を必要とする作業者の関心は特定の命令に限定されており，作業者の関心の範囲が反映される形で制御やデータに関する局所的な依存関係が解析される．一方で制御やデータに関してトレース全体を解析する動的依存解析は我々の過去の研究 [5] 以外のものは我々の知る限りでは存在せず，メソッド呼出しか Object Flow 解析のようなオブジェクトの参照程度しか扱われていない．

局所的な解析に基づく支援は感染が疑われる変数値のようにデバッグの問題解決に直接寄与する状態を発見した場合には有効である．しかし，現実のデバッグ作業の作業者はこうした情報を発見するためにプログラムの実行過程全体を対象に状態を把握し実行の挙動を理解することが求められる [1]．我々の過去の研究 [5] は解析範囲がトレース全体におよぶ制御と値の依存性の解析によってこの種の要求を満たすことが目的である．

文献 [5] では動的依存解析の一つである Outdated-State 解析手法を提案している．この動的依存解析手法は同じオブジェクトの異なる二つ以上の状態に影響を受け実行された命令を検出する．このような実行過程のパタンはオブジェクトのコレクションの状態を参照して繰返し制御を行う時の事例がある．この実行過程のパタンは直接不具合の要因となることや潜在的な不具合の原因となるため，このパタンは検出されれば修正するべき実行パタンの一つとなる．従来の動的依存解析を実行する環境で上記のようなトレースの全体を解析対象とする動的依存解析を行う場合，トレースの節点をメモリ上に読み込んだ上で依存性解析を行うため，トレースのデータ量が大きくなると実行ができない問題があった．また，依存性解析を行うと多数の節点同士の比較や，トレースに記録された依存関係をもとに一つないしは多数の節点の導出が効率的に実行できない問題もある．これら問題を解決するためには，トレースのデータ構造を把握した上でより効率的な解析を支援できる動的依存解析環境を構築する必要がある．

3. 動的依存解析環境の構築

本研究ではデバッグにはさまざまな種類の動的依存解析を実行することが必要であると想定しており，実際に過去の研究でも複数の解析手法を開発している [5]．

さまざまな動的依存解析手法の適用を可能にするために我々のトレースにはあらゆる動的依存解析にも対応することができるデータモデルが採用されている．その代償として実用的なプログラムのトレースは膨大かつ複雑になり易く，それが解析の効率の大きな妨げとなり易い．したがって，効率的な動的依存解析の実現にはトレースに対する動的依存解析を効率的に実行する動的依存解析環境が要求される．

図 1 に Java プログラムの実行から動的依存解析までを実施する動的依存解析環境の概要を示す．図中のトレース生成部は Java Bytecode instrumentation 技法を利用することによってトレースを生成する．トレース処理部は生成されたトレースをグラフデータベース（GDB）に格納し，関係性を用いた解析の効率的な実装を支援する．

3.1 動的依存解析環境に必要な機能

先行研究 [5] では，図 2 のような動的依存解析環境を構築している．図 2 の動的依存解析環境ではデバッグ対象のプログラムを実行し，そのプログラムの実行過程を記録したトレースを生成する．ここで記録されているトレースに含まれている命令や値，依存関係は POJO (Plain Old Java Object)[‡] のオブジェクトとして生成しメモリ上に格納する．また，動的依存解析中において，メモリに格納されたトレースを参照し，命令や値，依存関係を調査することにより動的依存解析を実行している．2. で述べたように，トレースの局所的な範囲を解析する手法とトレース全体を解析範囲とする手法が提案されており，当然ながら後者の手法は解析にメモリを多く消費する．したがって，既存の動的依存解析環境においてトレースのデータ量が大きくなるほど解析に利用できるメモリ領域が不足することになり，膨大なトレースに対して解析範囲がトレース全体に及ぶ動的依存解析手法は実行することができない．

以上より動的依存解析環境に要求されることとしては，あるデータから特定の関係性を辿り別のデータを取得する処理（グラフ走査）を高速に行える必要が第一にある．また，トレースのデータ量の大小にかかわらず，解析範囲がトレース全体に及ぶ動的依存解析手法を実行可能にするためには，トレースの管理や解析処理で消費するメモリを削減する必要がある．さらに，本研究で取り扱う動的依存解析手法では必要ないが，異なるトレース同士を比較することでプログラムの挙動を解析する手法も存在するため，それら手法を将来的に適用することを想定する必要がある．

[‡] フレームワークのような規約に縛られないように設計した Java オブジェクトのこと．

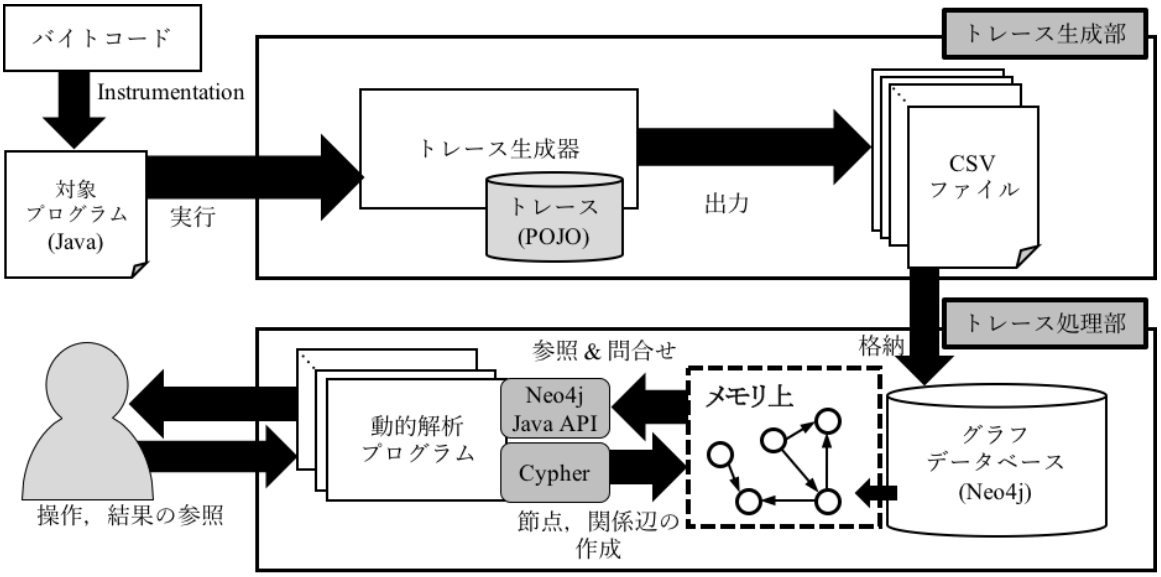


図 1: 本研究で構築する動的依存解析環境

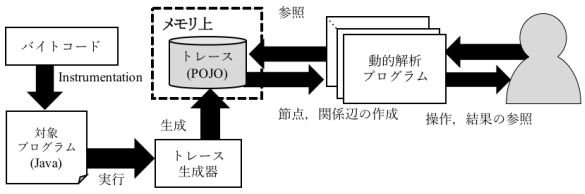
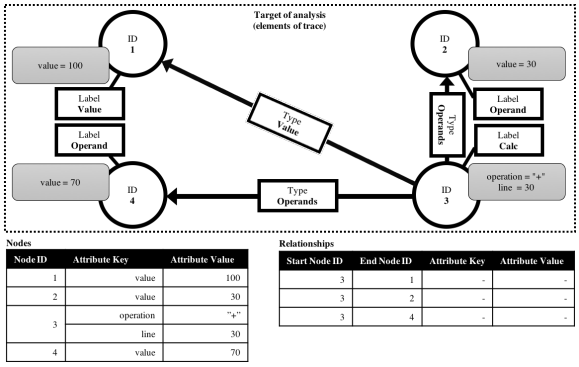


図 2: 先行研究の動的依存解析環境



Nodes

NodeID	Attribute Key	Attribute Value
1	value	100
2	value	30
3	operation	"+"
	line	30
4	value	70

Relationships

Start Node ID	End Node ID	Attribute Key	Attribute Value
3	1	-	-
3	2	-	-
3	4	-	-

図 3: プロパティグラフモデル

3.2 トレースのデータモデル

我々が開発した動的依存解析 [5] は個別のメソッド内部およびメソッド間に跨る依存関係を解析の対象とする．この時，値を生成および参照する命令や，これらの命令が実行されるメソッドに関する情報も解析に利用される．我々は更にある特定の条件を満たすデータに依存する動的依存解析手法も現在開発中である．

こうしたさまざまな解析の要求に答えるために我々のトレースには以下の概念を表現する要素が含まれている．

- メソッド呼出し構造
- メソッドで実行されたバイトコード命令
- バイトコード命令による値の生成参照
- 参照される値

それぞれの要素が表現している概念から要素間にさまざまな関係が導かれる．メソッド呼出しに関しては呼出し側と被呼出し側の関係が導かれる．条件分岐命令やメソッド呼出しのように「制御する」命令とそれらによって実行される命令の間には制御の依存関係（制御依存関係）が形成される．また値の生成と参照の関係を通じて命令同士にデータの依存関係（データ依存関係）が形成される．値とその生成は一対一関係を形成し，値とその参照の間には一対多関係が生成される．トレースはこれらの要素を節点，要素間に導入された関係を辺とする有向グラフとして表現される．

まず，我々のトレース生成手法により生成されるトレースは各構成要素に行番号やスレッド番号のような属性を持つため，図 3 のようなプロパティグラフモデルで表現できる [13]． プロパティグラフモデルは Apache の TinkerPop プロジェクトで定義されているデータモデルである[§]．また，これら属性は動的依存解析の際に参照されるデータや，動的依存解析の結果として理解されやすいように表示の際に参照されるデータが含まれている．

動的依存解析を行う際には，オブジェクトの状態の変遷やプログラムのエラーを追跡など，命令や値の依存関係を辿ることになる．したがって，動的依存解析では制

[§]Apache TinkerPop: https://tinkerpop.apache.org/（閲覧日：2017-07-28）

御依存関係やデータ依存関係などの依存関係を辿る（グラフ走査）処理が頻繁に行われることが想定され，グラフ走査処理の効率化が要求される．

3.3 動的依存解析の処理方法

動的依存解析は解析範囲の広狭に関わらず解析の起点となる命令や値からデータ依存関係や制御依存関係などの依存関係を辿ることによって行うことができる．したがって，グラフ走査の起点となる節点（走査開始節点）を取得後は依存関係を辿りながらトレースを調査していくため，グラフ走査処理の効率化を図ることが動的依存解析の効率化につながる．また従来の場合，解析中は全てのトレースの要素がメモリ上に読み込まれ，その上，トレースの要素の依存関係を解析するため組合せ爆発のような問題が発生し，トレースの全体を解析対象とする動的依存解析に対応できなかった．そのため，膨大なトレースに対する解析を可能にするために，メモリ上に読み込まれるのは解析に必要なデータのみにする必要がある．

そこで，本研究ではグラフ走査処理のパフォーマンス向上やディスク上での管理といった二点の必要性を考慮して，トレース処理部の基盤にはオンディスクデータベースである GDB を用いる．GDB にはさまざまなソフトウェアが存在するが，本研究では Neo4j[¶] を採用することにした．これは，Neo4j がトレースを格納するのに適したプロパティグラフモデルを採用しており，同時に GDB の中でグラフ走査処理の性能が高いことが示されていたからである [2,4]．また，GDB の中でも関係性を利用したグラフ解析に最適化している GDB はグラフ走査の対象となっている節点に直接関係がないデータを読み込まずに解析を行うことができるため，データ量が膨大なトレースに対しても全体的に解析範囲がおよぶ動的依存解析手法の適用を可能にする．

Neo4j のデータベースに格納されたトレースに対する動的依存解析の実装は Java のグラフ走査処理を実装するライブラリである Neo4j Traversal API および，グラフ問合せ用のクエリ言語 Cypher を用いることにより行う．Cypher により解析の起点となるデータを問合せを行い，Neo4j Traversal API を用いることにより命令や値の種類に合わせて条件付けしたグラフ走査を実現することができる．

4. 同一節点に到達するグラフ走査の削減法

トレースをグラフ表現した時には，3.2 節で述べたように，実行時に発生した命令や値が節点になる．また，複数の命令がある特定の値を利用している場合，複数の命令と値を表現している節点の間には，多対一関係の依存関係が発生する．図 4 の命令 1 および命令 2 は同じ値（オブジェクト）を参照，利用しているため，依存関係がオブジェクト a に集まるような形で存在する．本研究で構築した動的依存解析環境は，3.3 節で述べたように，依存関係をグラフ走査で一つひとつ節点を解析していく．これにより，上記のような多対一関係の依存関係をグラフ走査すると，解析の中で同一の節点を複数回読み込むことになる．多対一関係のグラフ走査で，一番初めに読み込む節点を計算機のメモリ上に保持しておくことにより，グラフ走査を行わずに節点を再利用する事ができる．

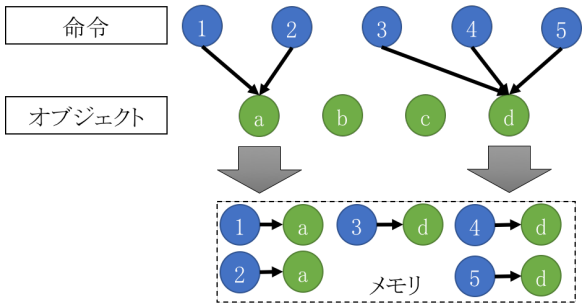


図 4: 同一節点に到達するグラフ走査の削減法

そこで，本研究では，動的依存解析のアルゴリズムで多対一の依存関係が存在する場合，それらの組合せを計算機のメモリに保持することによって，グラフ走査の削減を行う．例えば，図 4 の場合，命令とオブジェクトの対で，{1-a, 2-a, 3-d, 4-d, 5-d} が計算機のメモリ上に読み込まれることになる．ただし，ここで，動的依存解析では発生しない依存関係については，計算機のメモリへ保持することは行わない．これにより，解析速度の向上に貢献する．

5. 評価実験

本節では本提案手法がグラフ走査の処理パフォーマンスにどのように効果をもたらすのか明らかにするため実験を行う．トレースの全域を対象とする動的依存解析は解析時間だけでなく，メモリを効率的に利用できているか評価する必要がある．本研究の提案手法を適用することで解析範囲がトレース全体に及ぶ動的依存解析の処理パフォーマンスが改善されているかどうかについて評価を行う．この実験ではトレースの構造ごとに処理パフォーマンスの比較を行うために，トレース全体を解析対象とする動的依存解析の解析時間，動的依存解析実行中のメモリ消費量を計測する．本実験は OS: Linux[‖]，CPU: 2.26 GHz 4 core[**]，RAM: 64GB の Kernel-based Virtual Machine の環境で行う．

[¶] Neo4j. http://neo4j.com/（閲覧日: 2017-07-28）

[‖] CentOS 7.2-1511

[**] QEMU Virtual CPU ver. 0.9.1

5.1 Outdated-State 解析

本研究ではトレースの全体を解析する必要がある動的依存解析を行うため, 命令で同じオブジェクトの異なる状態を利用している命令を検出することができる Outdated-State 解析を用いる [5]．同じオブジェクトの異なる状態を利用するということは，プログラムアンチパタンとされており，バグや不具合の原因となるため，除去する必要がある．

動的依存解析環境において，以下の手順で Outdated-State 解析を実行する．

1. メソッド呼出しを実行順に一つずつ調査する．
2. 一つのメソッド呼出しに対して，そのメソッドで実行されている複数の命令の種類ごとにオブジェクトの状態との依存関係を調査する．
3. 状態変更命令を解析した際に，その値の変更が何回目か記録した節点を GDB に作成する．
4. 手順 (3) で作成した節点から同じオブジェクトの新しい状態と古い状態の組合せと依存関係を持つ命令が存在するか調査する．

Outdated-State 解析は手順 (1) で示すように，プログラムの実行全体を解析する必要があるため，通常の実装では動的依存解析にメモリ空間が多く必要となる．

5.2 利用するトレースデータ

評価実験で利用するトレースは，さまざまなデータ量のものを用意する．一つは我々の先行研究 [7] で利用している GEFDemo [††] である．これは, Outdated-State 解析により検出されるプログラムアンチパタンが GEFDemo の中に存在しており，また，それが原因で不具合が発生する．また, Outdated-State 解析は正常に動作しているプログラムに対しても，解析を実行することによりプログラムアンチパタンを検出する利用用途がある．したがって，さまざまなデータ量のトレースを用意するために，プログラムに設定するオプションにより負荷を変えることができるベンチマークプログラムを利用する．本研究では，ベンチマークプログラム集 Ashes2 [‡‡] を利用する．本実験で利用するトレースの一覧を表 1 に示す．ラベルの ‘_’ 以降はプログラムのオプション設定である．

5.3 動的依存解析の処理パフォーマンスの評価方法

本研究で提案した手法が動的依存解析の処理パフォーマンスの改善に貢献したか評価するために，評価項目として解析時間および最大メモリ消費量の計測を行う．動的依存解析時間は解析の開始時と終了時の時間を記録して差分を算出する．一方，動的依存解析中の最大メモリ消費量は UNIX コマンドの vmstat で実行中の毎秒のメモリ消費量を計測することで，その最大値と動的依存解析実行前のメモリ消費量の差分を算出する．

動的依存解析を各トレースに対して 100 回試行し，平均解析時間と最大メモリ消費量を算出する． また，トレースの形状に関しては以下の三種類に対して行う．

NON: 変換を行わないトレースの形状

ALL: 属性を全て分割したトレースの形状 [6]

OPT: 分割する属性を最適化したトレースの形状 [7]

つまり，以上の 3 パタンのトレースと，解析に提案手法を適用した場合とそうでない場合の 2 パタンの組合せ 6 パタンに対して評価を行う．

表 1: 実験で利用するトレースの一覧

プログラム	ラベル	GDB データ量 [MB]
GEFDemo	gefdemo	292.63
Ashes2-bisort	bisort_s0025	64.15
Ashes2-bisort	bisort_s0250	164.40
Ashes2-health	health_l01t0500	853.80
Ashes2-health	health_l02t0125	368.82
Ashes2-treeadd	treeadd_l05	60.13
Ashes2-treeadd	treeadd_l10	84.20
Ashes2-treeadd	treeadd_l15	1026.29
Ashes2-perimeter	perimeter_l01	60.13
Ashes2-perimeter	perimeter_l02	60.13
Ashes2-perimeter	perimeter_l04	60.13
Ashes2-perimeter	perimeter_l08	60.13
Ashes2-perimeter	perimeter_l11	60.13
Ashes2-em3d	em3d_n0100d005	244.58
Ashes2-mst	mst_v0064	525.17
Ashes2-voronoi	voronoi_n04000	557.39

6. おわりに

本研究では膨大で複雑なトレースの処理の効率化のために，依存関係を辿ることにより解析する動的依存解析の特徴に着目し，グラフ走査の処理が最適化された GDB を用いて動的依存解析環境を構築した．また，トレースの管理方法について説明したが, 動的依存解析において，複数の命令が同一のオブジェクトを参照，利用しているようなことが多く，同一節点へのグラフ走査が幾度も発生していることが分かっている．この問題に対して，本研究では同一節点へのグラフ走査の削減を実現する手法を提案した．

[††] GEFDemo. http://gefdemo.stage.tigris.org/

[‡‡] Ashes2. http://www.sable.mcgill.ca/~bdufou1/ashes2/（閲覧日: 2017-07-28）

謝辞

本研究は同志社大学ハリス理化学研究所研究助成事業および栢森情報科学振興財団研究研究助成事業，人工知能研究振興財団研究助成事業，日本学術振興会科学研究費助成事業 25240014 および 26280115，15K12009 の助成を受けて遂行された．

参考文献

[1] David J. Agans. *Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems*. Amacom, 2002.

[2] Salim Jouili and Valentin Vansteenberghe. An empirical comparison of graph databases. In *Proceedings of the 2013 International Conference on Social Computing*, SOCIALCOM '13, pages 708–715. IEEE Computer Society, 2013.

[3] Andrew J. Ko and Brad A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 151–158, 2004.

[4] Vojtěch Kolomičenko, Martin Svoboda, and Irena Holubová Mlýnková. Experimental comparison of graph databases. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, IIWAS '13, 2013.

[5] Izuru Kume, Masahide Nakamura, Naoya Nitta, and Etsuya Shibayama. A case study of dynamic analysis to locate unexpected side effects inside of frameworks. *International Journal of Software Innovation*, 3(3), 2015.

[6] Kazuma Kusu, Izuru Kume, and Kenji Hatano. A trace partitioning approach for efficient trace analysis. In *Proceedings of the 4th International Conference on Applied Computing & Information Technology, 2016 4th Intl Conf on Applied Computing and Information Technology / 3rd Intl Conf on Computational Science/Intelligence and Applied Informatics / 1st Intl Conf on Big Data, Cloud Computing, Data Science & Engineering*, pages 133 – 140, 2016.

[7] Kazuma Kusu, Izuru Kume, and Kenji Hatano. A node access frequency based graph partitioning technique for efficient dynamic dependency analysis. In *Proceeding of the Ninth International Conferences on Advances in Multimedia*, pages 73 – 78, 2017.

[8] Bil Lewis. Debugging Backwards in Time. In *Proceedings of the Fifth International Workshop on Automated Debugging*, 2003.

[9] Adrian Lienhard. *Dynamic Object Flow Analysis*. Lulu.com, 2009.

[10] Adrian Lienhard, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. Exposing side effects in execution traces. In *International Workshop on Program Comprehension through Dynamic Analysis*, pages 11–17, 2007.

[11] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. *Practical Object-Oriented Back-in-Time Debugging*. 2008.

[12] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *International Conference on Software Engineering*, pages 485–495, 2012.

[13] Marko A. Rodriguez and Peter Neubauer. Constructions from dots and lines. *Computing Research Repository*, abs/1006.2361, 2010.

[14] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing java programs. In *International Conference on Software Engineering*, pages 512–521, 2004.

[15] Mark Weiser. Program slicing. In *International Conference on Software Engineering*, pages 439–449, 1981.

[16] Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.