Thin Sliceのサイズに関する統計的評価

秦野 智臣[1]　鹿島 悠[1]　石尾 隆[1]　井上 克郎[1]

概要：プログラム理解の時間を減らすための技術として，プログラムスライシングが提案されている．プログラムスライシングは，プログラム内のある文を基準として，その文に影響を与える可能性のあるすべての文をプログラムスライスとして抽出する技術である．しかし，大規模プログラムの場合は，プログラムスライス自体が非常に大きくなってしまい，プログラム理解への利用は難しい．Thin Slicing は，開発者の選んだ文が使用するデータを生成した文のみを抽出することで，プログラムスライスのサイズを減らす技術である．しかし，一般に Thin Slicing が，どの程度の大きさのプログラムスライスを抽出するのかは示されていない．本研究では，7 個の Java プログラムのすべてのデータフローを対象に Thin Slice を計算し，そのサイズに関する統計的評価を行った．その結果，Thin Slice のサイズは平均でプログラムの 2.2%であり，60 から 80%のスライスでは 0.1%以下と十分小さくなることを確認した．

A Statistical Evaluation of Thin Slice Size

TOMOMI HATANO[1]　YU KASHIMA[1]　TAKASHI ISHIO[1]　KATSURO INOUE[1]

Abstract: Program slicing is a technique which supports program comprehension. Program slicing extracts all statements — called a program slice — that may affect a certain statement. However, program slicing is not useful if an analysis target program is too large since program slices of such a program are also often too large. Thin slicing is a technique reducing the size of program slice by extracting only statements producing data which is used by a selected statement. However, the size of Thin Slice in general has not been revealed. In this paper, we computed Thin Slices for every data-flow path in 7 Java programs, and then performed statistical evaluation. As a result, the average size of Thin Slice is 2.2% of a program. Furthermore, 60 to 80% of Thin Slices are 0.1% or less.

1. まえがき

開発者はプログラムの保守作業に多くの時間を費やしており，保守作業の中でも，多くの時間をプログラム理解に費やしている [1], [2]．さらに，プログラム理解を行うためには静的なデータ依存関係やメソッドの呼び出し関係を調査する作業が必要であり，開発者は，この作業に時間をかけていることが指摘されている [3], [4].

プログラム理解の作業時間を減らすための技術として，プログラムスライシングがある [5]．プログラムスライシングは，プログラム内のある文を基準として，その文に影響を与える可能性のあるすべての文をプログラムスライス（以下，単にスライスという）として抽出する技術である．プログラムスライシングによって，開発者がプログラム理解のために読むコードが少なくなり，理解にかかる時間を減らすことができる．しかし，スライスのサイズの平均値は，プログラム全体の約 30%であり [6]，大規模プログラムの保守作業におけるプログラム理解の際に，プログラムスライシングを利用することは困難である．

スライスのサイズを減らす手法として，Thin Slicing が提案されている [7]． Thin Slicing は，基準となる文が使用するデータを生成した文のみを抽出するスライシング手

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

法である．本稿では，Thin Slicing により抽出された文のことを Thin Slice と呼ぶが，Thin Slice は通常のプログラムスライスに比べサイズが非常に小さく，プログラム理解に必要な作業の中でも，複数のメソッドを経由するデータの依存関係を調査する時間を減らす効果が期待できる．[7] では，Thin Slicing によりプログラム理解の時間が減少した例が 22 個紹介されている．

Thin Slicing は有望な手法であるが，一般的な状況において常に有効であるかどうかは明らかではない．従来のスライシング手法については，Binkley らによってスライスサイズの平均値が計算されている [6]．また，Jász ら [8] は，メソッド呼び出しと制御依存関係のみを用いたスライスの高速な近似計算を提案しており，その計算結果の平均値はスライスサイズの平均値より 4.27%大きくなることを示している．Binkley らや Jász らの研究のように，Thin Slicing においても Thin Slice のサイズの平均値を計算し，その有効性を評価する必要がある．

本研究では，Java 言語で記述されたプログラムを対象とした Thin Slicing を実装し，Thin Slice のサイズに関する統計的評価を行う．[7] では，22 個の例で有効であることが確認されているだけなのに対し，本研究では Thin Slice のサイズの平均値を計算し，多くの場合で有効であるかどうかを確認する．スライスサイズが十分に小さければ，開発者が閲覧しなければならない文が少なくなる．また，データの生成元や使用先を Thin Slicing によって特定できれば，開発者がデータ依存関係を調査する作業 [3], [4] の時間が削減できる．しかし，メソッド呼び出しをたどらなくともデータを生成している文が容易に見つかる場合は，Thin Slicing の効果は低いと考えられる．そのため本研究の実験では，プログラム中の全データフローから Thin Slicing の効果が期待できるデータフローの割合を調査するために，スライスサイズに加え，スライスがまたがるメソッド数や，使用しているデータの生成元となっている文の数といった指標を計測した．

2. 背景

2.1 プログラムスライシング

プログラムスライシングは，プログラム内のある文に影響を与える可能性のあるすべての文を抽出する技術である [5]．スライスの計算には，プログラム依存グラフ（Program Dependence Graph，以下 PDG）を拡張したシステム依存グラフ [9]（System Dependence Graph，以下 SDG）を用いる．PDG は，プログラム内の 1 つの手続きの各文を頂点とし，それらの頂点間の依存関係を有向辺で表したグラフである．依存関係はデータ依存関係と制御依存関係があり，それぞれ次のように定義される．

データ依存関係　文 s_1 と s_2 の間に以下の条件がすべて成り立ったとき，s_2 は s_1 にデータ依存するといい，s_1 の頂点から s_2 の頂点にデータ依存辺を引く．

(1) s_1 が変数 x の値を定義する．
(2) s_2 が変数 x の値を使用する．
(3) s_1 から s_2 に，x の値を書き換えない実行経路が少なくとも 1 つ存在する．

制御依存関係　文 s_1 が制御文であり，s_1 の結果によって s_2 が実行されるかどうかが決定される場合，s_2 は s_1 に制御依存するといい，s_1 の頂点から s_2 の頂点に制御依存辺を引く．また，手続きの入口を表すエントリ頂点という特別な頂点を設け，手続き内で他の文からの制御依存辺を持たない文に対して，エントリ頂点から制御依存辺を引く．

SDG は，各 PDG を手続き間の関係で接続したグラフである．手続き間の関係は，実パラメータから仮パラメータへのデータ依存辺，返り値から呼び出し元へのデータ依存辺と手続き呼び出し文からエントリ頂点への制御依存辺で表される．

スライスは，スライスを計算するための基点となる文と変数を定め（これをスライシング基準という），その基点に相当する SDG 上の頂点から依存辺に沿った探索を行い，基点となった頂点自身を含む到達範囲の頂点集合として計算される．スライスの計算方法は，依存辺を逆方向にたどる後ろ向きスライシングと順方向にたどる前向きスライシングに分けられる．

Binkley ら [6] は，合計 136,000 行に及ぶ 43 のプログラムについて，すべての文に対して前向きスライスと後ろ向きスライスを計算した実験を行っている．この実験では，スライスに含まれる文は，平均でプログラム内のすべての文の約 30% であることが述べられている．数百万行を超える大規模プログラムでは，プログラムスライシングを利用しても調査対象となるスライスのサイズが大きくなり，開発者がスライスに含まれるすべての文を閲覧することは困難である．

2.2 Thin Slicing

Thin Slicing[7] は，重要なデータ依存関係のみを取り扱うことでスライスサイズを減らし，プログラム理解やデバッグ作業における開発者の負担を軽減するスライシング手法である．Thin Slicing には，Context-Insensitive Thin Slicing と Context-Sensitive Thin Slicing の 2 種類の計算方法があるが，本研究では Context-Insensitive Thin Slicing のみを扱う．これは，[7] において，Context-Sensitive Thin Slicing の計算時間が膨大であり，実用的なプログラムで用いることが現実的ではないことが示されているためである．

スライシング基準 s に対する Thin Slice とは，s がデータ依存する文の集合である．ただし，Thin Slicing では以下のようにデータ依存関係を定義する．

手続き内のデータ依存関係はプログラムスライシングと

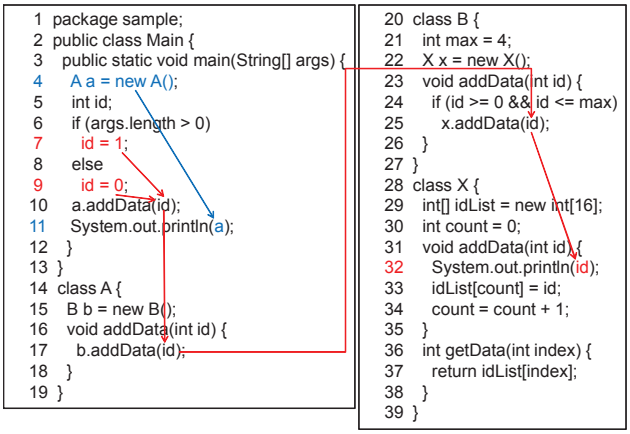

図 1 Thin Slicing の例
Fig. 1 Examples of Thin Slicing

同様に定義する．ただし，ポインタ変数を介してヒープ領域にアクセスしている文については，ヒープ領域についてのみデータ依存関係を考える．たとえば p.f というフィールドを参照する文は，f に関するデータ依存関係のみを考え，p に関するデータ依存関係は考えない．

手続き間のデータ依存関係は手続きの実引数に呼び出し先の仮引数がデータ依存すると定義する．また，呼び出し先の返り値に呼び出し元がデータ依存すると定義する．

ヒープ領域のデータ依存関係は，同一のフィールドまたは配列の内容を代入し参照する可能性のある文の組について，所属する手続きに関係なく，参照する文が代入する文にデータ依存していると定義する．

Thin Slice は，上記の定義に対応するデータ依存辺を有向辺とするグラフで探索を行い計算する．**図 1** に Thin Slicing の例を 2 つ示す．1 つ目は，32 行目の変数 `id` に対する Thin Slicing である．この変数 `id` の値は 25 行目のメソッド呼び出し文から到達し，25 行目では変数 `x` と `id` を使用している．Thin Slicing は変数 `x` の依存関係は考えず，変数 `id` の依存関係のみを探索するので，17 行目と 10 行目のメソッド呼び出し文と 7 行目と 9 行目の代入文に到達する．結果として，32 行目の変数 `id` に対する Thin Slice は，{7, 9, 10, 17, 25, 32} 行目となり，7 行目か 9 行目で生成された値が到達していることが明らかになる．この例では，Thin Slice が 4 つのクラスの 4 つのメソッドにまたがっているので，開発者がメソッドの呼び出し元をたどってデータ依存関係を追跡する作業に有効であると考えられる．2 つ目の例は，11 行目の変数 `a` に対する Thin Slicing である．この例では，Thin Slice が 1 つのメソッド内で完結しており，ソースコードを見れば 4 行目で生成された値が到達していることがすぐに分かるため，Thin Slicing は有効でないと考えられる．一般のプログラムにおいて，どちらのケースに該当するスライシング基準が多いのかは [7] では明らかにされていない．

3. 調査目的

本研究では，Thin Slicing が多くの状況で有効に働くかどうかを評価するために，Thin Slice のサイズに関する統計的評価を行う．具体的なリサーチクエスチョンは，以下の 2 つである．

RQ1 Thin Slice のサイズは，平均的に十分小さいものであるか．

RQ2 データ依存関係の調査において有効であると考えられる Thin Slice はどの程度存在するか．

4. Thin Slicing の実装

本研究では，実験を行うために Java プログラムのバイトコード（**表 1**）を対象とした Thin Slicing を実装した．本実装では，バイトコードの 1 命令を頂点とし，各命令間のデータ依存関係を辺としたグラフを作成する．以降では，バイトコード上のデータ依存関係について説明する．

4.1 ローカル変数とオペランド・スタック間のデータ依存関係

Java 仮想マシンは，オペランド・スタックと呼ばれるスタックに，演算で使用する値やその結果を保持している．そのため，ローカル変数とオペランド・スタック間で値を移動させる命令が存在し，オペランド・スタック上の演算におけるデータ依存関係だけでなく，ローカル変数とオペランド・スタック間のデータ依存関係が存在する．

4.2 フィールド変数と配列変数のデータ依存関係

フィールド変数の依存関係は次の 2 つの方法で決定する．1 つ目は，フィールド変数に値を書き込む命令である PUTSTATIC とフィールド変数の値を読み込む命令である GETSTATIC が，同じクラスの同じフィールド名に対して行われている場合に PUTSTATIC から GETSTATIC に依存辺を引く．2 つ目は，PUTFIELD と GETFIELD が，エイリアスされている可能性のある変数の同じフィールド名に対して行われている場合に，PUTFIELD から GETFIELD に依存辺を引く．エイリアス解析には，Andersen のポインタ解析 [10] に基づいた Object-sensitive Pointer Analysis[11] を用いた．配列変数はフィールド変数と同様に，エイリアスされている可能性のある配列に値を書き込む命令から読み込む命令に依存辺を引く．

4.3 メソッド間のデータ依存関係

あるメソッド呼び出し文から，その呼び出し文で呼び出される可能性のあるすべてのメソッドに対して，実パラメータから仮パラメータへの依存辺と返り値から呼び出し元への依存辺を引く．メソッドの動的束縛の解決には，Variable-Type Analysis[12] を用いた．

表 1　バイトコード命令の一覧と分類
Table 1　Bytecode instructions

命令の種類	分類	該当する命令								
ローカル変数の読み込み	transfer	ILOAD	LLOAD	FLOAD	DLOAD	ALOAD				
ローカル変数の書き込み	transfer	ISTORE	LSOTRE	FSTORE	DSTORE	ASTORE				
フィールド変数の読み込み	transfer	GETFIELD		GETSTATIC						
フィールド変数の書き込み	transfer	PUTFIELD		PUTSTATIC						
配列変数の読み込み	transfer	IALOAD	LALOAD	FALOAD	DALOAD	AALOAD	BALOAD	CALOAD	SALOAD	
配列変数の書き込み	transfer	IASTORE	LASTORE	FASTORE	DASTORE	AASTORE	BASTORE	CASTORE	SASTORE	
オペランド・スタック管理	なし	POP	POP2	DUP	DUP2	DUP_X1	DUP2_X1	DUP_X2	DUP2_X2	SWAP
オブジェクトの生成	source	NEW	NEWARRAY		ANEWARRAY		MULTIANEWARRAY			
値を生成する演算	source かつ sink	IADD LMUL FREM LSHR IINC F2D FCMPG	LADD FMUL DREM IUSHR I2L D2I DCMPL	FADD DMUL INEG LUSHR I2F D2L DCMPG	DADD IDIV LNEG IAND I2D D2F	ISUB LDIV FNEG LAND L2I I2B	LSUB FDIV DNEG IOR L2F I2C	FSUB DDIV ISHL LOR L2D I2S	DSUB IREM LSHL IXOR F2I LCMP	IMUL LREM ISHR LXOR F2L FCMPL
定数の生成	source	ICONST_M1 ICONST_3 LCONST_1 DCONST_0		ICONST_0 ICONST_4 FCONST_0 DCONST_1		ICONST_1 ICONST_5 FCONST_1 ACONST_NULL		ICONST_2 LCONST_0 FCONST_2		LDC BIPUSH SIPUSH
比較演算	sink	IFEQ IF_ICMPNE IF_ICMPLE TABLESWITCH	IFNE	IFLT IF_ICMPLT IF_ACMPEQ LOOKUPSWITCH	IFGE	IFGT IF_ICMPGE IF_ACMPNE	IFLE	IFNULL IF_ICMPGT IFNONNULL	IF_ICMPEQ	
メソッドの呼び出し	特殊	INVOKEVIRTUAL		INVOKEINTERFACE		INVOKESPECIAL		INVOKESTATIC		
メソッドからの戻り	transfer	IRETURN	LRETURN	FRETURN	DRETURN	ARETURN	RETURN			

5. 評価実験

本実験では，Java プログラムのクラスファイルを解析し，すべてのデータを基点とした Thin Slice を計測する．Binkley らの実験と同様に，後ろ向き Thin Slice と前向き Thin Slice を計算し，各 RQ に対応した指標を計測する．

5.1 計測方法

本研究では，バイトコードの命令を，計算に使われるデータを提供する source，データを使用する sink，データを伝播する transfer に分類する．2 項演算である IADD のような演算は，演算に使用する 2 つの数値に関するデータフローの sink であり，計算結果を次の命令に提供する source でもある．また，メソッド呼び出しは，実引数がそれぞれ sink であり，戻り値が次の計算のための source となる．ローカル変数の代入，参照のように，データ自体を変更しない命令は transfer に分類した．具体的な分類は表 1 に示している．

本研究では，sink に対して後ろ向き Thin Slice を，source に対して前向き Thin Slice を計算していく．これは様々な演算やメソッド呼び出し文，条件式に登場する変数やフィールドの値を調査する後ろ向きスライスと，演算の結果や変数の値がどこで使われているかを調査する前向きスライスに相当する．transfer 命令はいずれもデータの受け渡しのみに対応しており，それらを基点にしてスライスを計算しても，それぞれデータフローで接続された sink や source からのスライスと同一の結果を返すことになるため，計測対象からは除外した．

本研究で用いる Thin Slice に関する表記を以下に定義する．また，以下の各集合 S の要素数を $|S|$ と表記する．

***Backward*(v)**　ある sink v をスライシング基準とした後ろ向き Thin Slice に含まれる頂点の集合．

***Source*(v)**　$Backward(v)$ に含まれる source 頂点の集合．

***Forward*(w)**　ある source w をスライシング基準とした前向き Thin Slice に含まれる頂点の集合．

***Sink*(w)**　$Forward(w)$ に含まれる sink 頂点の集合．

***Method*(S_{ts})**　Thin Slice S_{ts} の頂点が所属するメソッドの集合．

***Class*(S_{ts})**　Thin Slice S_{ts} の頂点が所属するクラスの集合．

RQ1 について調査するために，すべての sink v について $|Backward(v)|$ を，すべての source w について $|Forward(w)|$ を計測し，これらの平均値を評価する．

RQ2 について調査するために，すべての sink v について次の指標を計測する．

- $|Method(Backward(v))|$
- $|Class(Backward(v))|$
- $|Source(v)|$

また，すべての source w について次の指標を計測する．

- $|Method(Forward(w))|$
- $|Class(Forward(w))|$
- $|Sink(w)|$

$|Source(v)|$ の値が小さい場合は Thin Slicing を利用することで，使用しているデータの生成元を少数に特定できる．同様に，$|Sink(w)|$ の値が小さい場合は生成したデータの使用先を少数に特定できる．

$|Method(Backward(v))|$，$|Class(Backward(v))|$，$|Method(Forward(w))|$，$|Class(Forward(w))|$ については，これらの値が大きい場合，様々なメソッドやクラスにまたがった Thin Slice であることを意味するため，Thin Slice によってメソッド呼び出しをたどってデータの生成元や使用先を特定する作業の時間を減らすことが期待できる．

5.2 実験対象

実験対象のプログラムは DaCapo benchmark（9.12）[*1]

[*1] http://dacapobench.org/

表 2　実験対象のプログラム
Table 2　Subject programs

プログラム名	クラス数	メソッド数	グラフの頂点数
tomcat	261	2,389	54,468
luindex	560	4,180	123,191
sunflow	657	4,609	190,526
avrora	1,838	9,304	211,343
pmd	2,369	16,439	448,722
xalan	2,805	22,377	815,861
batik	4,417	28,818	968,470

とした．DaCapo benchmark には，Java 言語で書かれた多数のアプリケーションが含まれている．本実験で対象としたプログラムの規模は表 2 の通りである．

5.3　実験結果と考察

本実験で計測した各指標の最大値と平均値を**表 3** に示す．後ろ向き Thin Slice と前向き Thin Slice の平均値は，7 つのプログラムで平均して約 2.2%である．Binkley らの実験 [6] では，従来のスライスの平均値が約 30% であったことから，Thin Slice の平均値は十分小さいことが分かる．**図 2** は，横軸に sink v の $|Backward(v)|$ がプログラム全体に占める割合，縦軸に横軸の各値以下であるスライスの割合を示した累積度数分布図である．この図から，全スライス中の約 60 から 80%のスライスは，そのサイズがプログラム全体の 0.1%以下である一方で，残りの約 20 から 40%のスライスは各プログラムのスライスの最大値付近に分布している．Thin Slicing が特に有効である場合として，後ろ向き Thin Slice が複数のメソッドにまたがり，かつデータの生成元が少ないスライスの数を**表 4** に示す．これらの 10%程度のスライスについては，データフローを複数のメソッドに渡って追跡していく作業を Thin Slicing によって置き換えられるため，特に Thin Slicing の効果が高いと考えられる．

以上の結果から RQ1 について，Thin Slice のサイズは平均的に十分小さいと言える．ただし，スライスの分布は非常に小さいものと大きいものの 2 つに分かれており，すべてのスライスのサイズが小さいわけではない．また，RQ2 について，表 4 に示したような約 10%のスライスが表すデータフローは，データ依存関係の調査において特に有効であると考えられる．

6.　関連研究

スライスが開発者に与える影響について多くの実験が行われている [7], [13], [14]．Ishio ら [13] は，データフローを可視化することによって，開発者がプログラムの調査を早く行えるようになるということを示している．Kusumoto ら [14] は，スライスを利用する場合の方が利用しない場合よりデバッグ作業の時間が早くなるということを示している．Sridharan ら [7] は，Thin Slicing を利用するとデバッグ作業やプログラム理解にかかる時間が減少することを示している．これらの実験から，開発者がデバッグ作業やプログラム理解を行う際に，データフロー情報を提示することが有効であると考えられる．

また，スライスに関する数値指標について様々な調査が行われている [6], [8], [15]．Binkley らは，スライスサイズの分布を調査しており [6]，スライスサイズの分布を利用して，プログラムの構造の複雑さを評価する手法を提案している [15]．Jász ら [8] は，スライスの高速な近似計算の結果がスライスサイズに対してどの程度大きいかということを調査している．

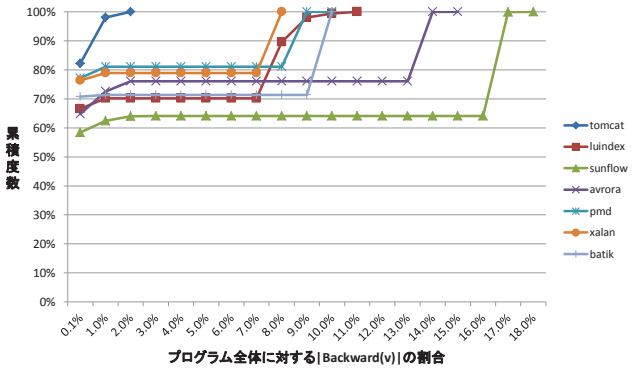

図 2　$|Backward(v)|$ の累積度数分布
Fig. 2　Cumulative frequency distribution for $|Backward(v)|$

表 4　$|Method(Backward(v))| \geq 2$ であり，$|Source(v)|$ が 3 以下であるスライスの割合（%）
Table 4　Ratio of slices whose $|Method(Backward(v))| \geq 2$ and $|Source(v)| \leq 3$

$\|Source(v)\|$	1	2	3	合計
tomcat	2.9	3.6	3.9	10.3
luindex	3.0	4.4	3.5	10.9
sunflow	1.7	2.3	2.3	6.3
avrora	3.5	3.7	2.5	9.7
pmd	2.8	3.5	4.1	10.5
xalan	2.7	3.4	3.3	9.4
batik	2.5	3.5	2.7	8.7

7.　まとめと今後の課題

本研究では，Thin Slice のサイズに関する統計的評価を行った．評価実験では，Thin Slice のサイズ，データの生成元の数，Thin Slice がまたがるメソッド数といった指標を計測した．その結果，Thin Slice のサイズの平均値は十分小さいことを確認した．Ishio らは，単純なデータフローを可視化するだけでも開発者のプログラム理解が速くなることを示している [13]．Thin Slicing は限定したデータ依存関係をたどって探索を行うため，その結果を開発者に提示することは有効であると考えられる．

表 3 各指標の統計量
Table 3 Summary of metrics

		tomcat	luindex	sunflow	avrora	pmd	xalan	batik
グラフの全頂点数		54,468	123,191	190,526	211,343	448,722	815,861	968,470
$\|Backward(v)\|$	最大値	922	12,820	34,512	30,458	40,784	62,957	93,810
	最大値の割合（%）	1.7	10.4	18.1	14.4	9.1	7.7	9.7
	平均値	55.2	3012.8	11264.9	6919.3	7108.6	12482.7	26159.3
	平均値の割合（%）	0.10	2.4	5.9	3.3	1.6	1.5	2.7
$\|Forward(w)\|$	最大値	3,673	23,633	43,041	27,592	52,339	105,749	140,965
	最大値の割合（%）	6.7	19.2	22.6	13.1	11.7	13.0	14.7
	平均値	57.2	1434.0	7834.8	2386.0	6226.3	18481.9	28719.3
	平均値の割合（%）	0.11	1.2	4.1	1.1	1.4	2.3	3.0
$\|Method(Backward(v))\|$	最大値	193	1,342	2,010	3,780	4,277	6,849	7,326
	平均値	8.0	155.6	155.7	128.7	126.5	385.5	576.9
$\|Method(Forward(w))\|$	最大値	368	1,852	2,449	3,702	3,132	7,175	8,011
	平均値	6.4	116.1	230.2	331.0	369.6	1267.1	1632.0
$\|Class(Backward(v))\|$	最大値	41	288	338	1,284	685	1,352	1,792
	平均値	2.9	68.4	112.8	278.0	111.3	264.9	489.7
$\|Class(Forward(w))\|$	最大値	68	375	348	1,147	507	1,306	1,750
	平均値	2.3	35.9	46.2	98.9	67.0	243.4	361.6
$\|Source(v)\|$	最大値	382	5,494	15,451	14,964	18,900	27,441	40,687
	平均値	25.3	1371.4	5218.1	3470.6	3076.6	5215.6	11183.8
$\|Sink(w)\|$	最大値	1,610	9,216	17,369	11,689	19,425	41,423	52,334
	平均値	25.4	537.6	3031.9	1021.4	2264.2	7238.5	10662.3

今後の課題として，Thin Slice の内容について詳細な調査を行うことが考えられる．本実験で約 20 から 40%のスライスについて，そのサイズが大きくなってしまうことが分かった．しかし，そのようなスライスがどのようなデータフローであるかは分かっていないため，スライスサイズが大きくなる原因を調査することが必要である．また，[7] において Thin Slicing の定性的な有効性は調査されているが，定量的な評価も必要である．さらに，鹿島らのツール [16] と組み合わせることで，メソッドの引数に与えられるデータの生成元を可視化し，データ依存関係の調査を支援することが考えられる．

謝辞 本研究は，日本学術振興会科学研究費補助金基盤研究（S）（課題番号:25220003），若手研究（A）（課題番号:23680001）の助成を得た．

参考文献

[1] Corbi, T. A.: Program understanding: challenge for the 1990's, *IBM Systems Journal*, Vol. 28, No. 2, pp. 294–306 (1989).

[2] LaToza, T. D., Venolia, G. and DeLine, R.: Maintaining mental models: a study of developer work habits, *Proceedings of the 28th ACM/IEEE International Conference on Software Engineering*, pp. 492–501 (2006).

[3] LaToza, T. D. and Myers, B. A.: Developers ask reachability questions, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pp. 185–194 (2010).

[4] Wang, J., Peng, X., Xing, Z. and Zhao, W.: An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions, *Proceedings of the 27th IEEE International Conference on Software Maintenance*, pp. 213–222 (2011).

[5] Weiser, M.: Program slicing, *Proceedings of the 5th IEEE International Conference on Software Engineering*, pp. 439–449 (1981).

[6] Binkley, D., Gold, N. and Harman, M.: An empirical study of static program slice size, *ACM Transactions on Software Engineering and Methodology*, Vol. 16, No. 2, pp. 1–32 (2007).

[7] Sridharan, M., Fink, S. J. and Bodik, R.: Thin slicing, *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 112–122 (2007).

[8] Jász, J., Árpád Beszédes, Gyimóthy, T. and Rajlich, V.: Static Execute After/Before as a Replacement of Traditional Software Dependencies, *Proceedings of the 24th IEEE International Conference on Software Maintenance*, pp. 137–146 (2008).

[9] Horwitz, S., Reps, T. and Binkley, D.: Interprocedural slicing using dependence graphs, *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 35–46 (1988).

[10] Andersen, L. O.: Program Analysis and Specialization for the C Programming Language, PhD Thesis, University of Copenhagen (1994).

[11] Milanova, A., Rountev, A. and Ryder, B. G.: Parameterized object sensitivity for points-to analysis for Java, *ACM Transactions on Software Engineering and Methodology*, Vol. 14, No. 1, pp. 1–41 (2005).

[12] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E. and Godin, C.: Practical virtual method call resolution for Java, *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, systems, languages, and applications*, pp. 264–280 (2000).

[13] Ishio, T., Etsuda, S. and Inoue, K.: A lightweight visualization of interprocedural data-flow paths for source code reading., *Proceedings of the 20th IEEE International Conference on Program Comprehension*, pp. 37–46 (2012).

[14] Kusumoto, S., Nishimatsu, A., Nishie, K. and Inoue, K.: Experimental Evaluation of Program Slicing for Fault Localization, *Empirical Software Engineering*, Vol. 7, No. 1, pp. 49–76 (2002).

[15] Binkley, D. and Harman, M.: Locating Dependence Clusters and Dependence Pollution, *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pp. 177–186 (2005).

[16] 鹿島　悠，石尾　隆，井上克郎：エイリアス解析を用いたメソッドの入力データの利用法可視化ツール，ソフトウェアエンジニアリングシンポジウム 2012 論文集, pp1-8 (2012).