情報処理学会論文誌 Vol.55 No.2 971–980 (Feb. 2014)

Thin Sliceのサイズに関する統計的評価

秦野 智臣[1,a)] 鹿島 悠[1,b)] 石尾 隆[1,c)] 井上 克郎[1,d)]

受付日 2013年5月16日, 採録日 2013年11月1日

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 2.1%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．

キーワード：プログラムスライシング，データフロー解析，静的解析，Java

A Statistical Evaluation of Thin Slice Size

Tomomi Hatano[1,a)] Yu Kashima[1,b)] Takashi Ishio[1,c)] Katsuro Inoue[1,d)]

Received: May 16, 2013, Accepted: November 1, 2013

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 2.1% of a program. Furthermore, 60 to 80% of Thin Slices are 0.1% or less.

Keywords: program slicing, data-flow analysis, static analysis, Java

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といわれている [1], [2]．プログラム理解において重要となるのが，変更したい機能が実装されているソースコードの位置を知ることである．機能の実装位置を知っている開発者は，そうでない開発者よりも効率的にソースコードの変更作業を行うことができる [3]．開発者がプログラムの機能とソースコードとを対応づけるための手がかりの 1 つが，変数の代入と参照の関係を表すデータ依存関係である．引数やフィールドを介して同一のデータを使用する複数のメソッドを特定することが，機能の実装位置を正しく認識するために有効であると報告されている [4]．一方で，静的な依存関係を探索するために開発者が多くの時間を必要とすることも報告されている [5]．

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 [6]．プログラムスライシ

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565–0871, Japan
[a)] t-hatano@ist.osaka-u.ac.jp
[b)] y-kasima@ist.osaka-u.ac.jp
[c)] ishio@ist.osaka-u.ac.jp
[d)] inoue@ist.osaka-u.ac.jp

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（以降，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 [7]，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 [8]．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なる．また，同一のデータが使用される場所を特定する場合は，データフローだけに着目する Thin Slicing のほうがプログラムスライシングより適しており，プログラムの機能の実装位置を認識する作業に有用であると考えられる．文献 [8] 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つね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 [7]．また，Jász ら [9] 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．文献 [8] 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て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. 関連研究

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 [6]．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 [10]（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．SDG は，手続きごとの PDG を呼び出し関係によって接続したグラフである．

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 上の頂点から有向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，後ろ向きスライシングと前向きスライシングに分けられる．

後ろ向きスライシングは，スライシング基準の実行に影響を与える可能性のあるすべての文の集合を計算する．**図 1** の 10 行目の文 c = add(a, b) と変数 c をスライシング基準として後ろ向きスライシングを実行すると，10 行目の文が使用する変数 a と変数 b の値を定義している 6 行目と 7 行目の文，さらに手続き呼び出し先の add で 2 つの値の和を計算している 19 行目が抽出され，スライスは {6, 7, 10, 19} 行目となる．前向きスライシングは，スライシング基準の実行に影響を受ける可能性のあるすべての文の集合を計算する．図 1 の 8 行目の文 x = new A() と変数 x をスライシング基準として前向きスライシングを実行すると，変数 x の値を使用する文とその値によって実行を制御される文が抽出され，スライスは {8, 9, 11, 12, 13, 14, 15} 行目となる．

2.2 プログラムスライシングの利用方法

プログラムスライシングは，プログラム理解やデバッグ

```
 1  package slice;
 2  public class Main {
 3    public static void
        main(String[] args) {
 4      A x, z, w;
 5      int a, b, c;
 6      a = 1;
 7      b = 2;
 8      x = new A();
 9      z = x;
10      c = add(a, b);
11      w = x;
12      w.f = c;
13      if (w == z) {
14        int v = z.f;
15        System.out.println(v);
16      }
17    }
18    private static int
        add(int x, int y) {
19      return x + y;
20    }
21  }
22  classA {
23    int f;
24  }
```

図 1 サンプルプログラム
Fig. 1 Example program.

に利用されている．プログラム理解においては，開発者が注目しているソースコード断片の内容を理解するにあたって，そこに登場する変数の値がどのように計算されているかを調べる必要があるためである [11]．デバッグにおいても，プログラムの誤った出力がどのように計算されたかを知るために，プログラムスライシングが用いられている [12]．Kusumoto らの研究 [13] では，開発者にスライスを提供することでデバッグが効率的になることが示されているほか，コードインスペクション環境におけるスライシングの利用が報告されている [14]．

2.3 スライスサイズを削減する技術

2.2 節で示した用途では，プログラムスライシングの結果から得られるプログラムの一部を開発者が直接閲覧するため，スライスサイズが小さいほど開発者の作業時間を短縮する効果がある．そのため，スライスサイズを減らす様々な手法が提案されており，一般的な場合で利用できる手法と，ある特定の条件で利用できる手法に分けられる．

一般的な場合で利用できる手法として，Chopping と呼ばれる手法 [15] は，データを生成している頂点と使用している頂点について，前向きスライシングと後ろ向きスライシングを計算し，その積集合を計算する．Reps ら [16] は，1 つの手続き内で制限されていた Chopping を手続き間で用いることができるように拡張している．Distance-Limited Slicing [17] は，PDG における頂点間の距離を定義し，その距離内の範囲で到達できる頂点のみを探索する手法である．

ある特定の条件で利用できる手法として，Chen ら [18] は，プログラムのある機能について，その機能の実装方法とプログラム中の位置を開発者が理解することを目的としたグラフの探索方法を提案している．この方法では，開発者が指定したスライシング基準とその頂点から有向辺が引かれている頂点のみを最初に表示し，開発者は探索したい頂点を選択する．同様に，選択された頂点から有向辺が引かれている頂点をさらに表示し，開発者は探索したい頂点を選択する．このように，開発者の要求に応じてスライスを表示することで，開発者が閲覧するスライスサイズを減らしている．Callstack-Sensitive Slicing [19] は，デバッグにおいて，バグ発生時のスタックトレースを利用することでスライスサイズを減らし，バグの発生位置と原因を特定する手法である．Ceccato ら [20] は，Barrier Slicing [21] と呼ばれる手法を用いて，ある特定の処理を実装している部分を探している．Barrier Slicing は，グラフ探索を打ち切る頂点を事前に指定してから探索を行い，スライスサイズを減らす手法である．Wang ら [22] は，ある関数を変更した際の影響を調査したい場合に，ソースコード中の文字や依存関係を SDG に対して入力すると，その入力条件に合った SDG の部分グラフを抽出する手法を提案している．

2.4 Thin Slicing

Thin Slicing [8] は，スライシング基準で使用しているデータを生成している文のみを抽出する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，文献 [8] 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．Thin Slicing においてデータ依存関係は，手続き内，手続き間，ヒープ領域の 3 種類を用いる．

手続き内のデータ依存関係は次のように定義する．文 s_2 が s_1 にデータ依存するとは，以下の条件がすべて成り立つことをいう．

(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．

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無視する．

手続き間のデータ依存関係は次のように定義する．

- 手続きの実パラメータに対応する頂点に，呼び出し先の仮パラメータに対応する頂点がデータ依存する．
- 呼び出し先の返り値に対応する頂点に，呼び出し元の返り値を受け取る頂点がデータ依存する．

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 2** は図 1 のプログラムに対するグラフであり，手続き内の代入文と手続き呼び出し文を丸角矩形の頂点で表し，手続き名と制御文をひし形の頂点で表している．実パラメータと仮パラメータに対応する頂点は p(x) = a，x = p(x) のようにパラメータ名 v に対して p(v) と表記し，返り値を受け取る頂点は ret = x + y，c = ret のように ret という仮想の変数を用いて表記し，いずれも長方形の頂点で表している．これらの頂点が所属する手続きを点線の枠で表し，頂点間のデータ依存関係を矢印で表している．たとえば，図 1 の 15 行目の文と変数 v をスライシング基準とした後ろ向き Thin

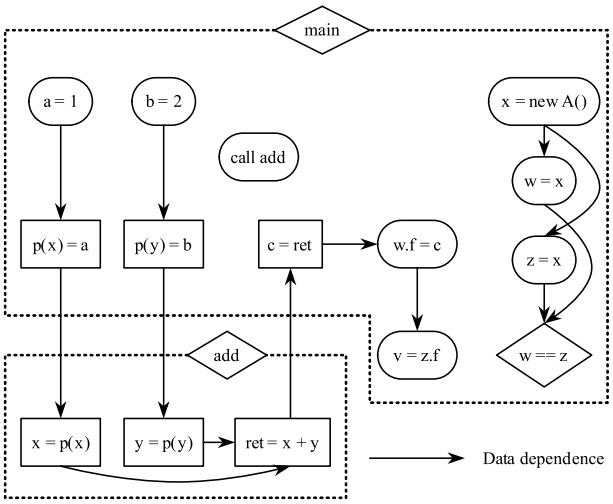

図 2 図 1 のプログラムに対する Thin Slicing 計算用のグラフ
Fig. 2 A graph to compute Thin Slicing for the program in Fig. 1.

```
 1 package sample;
 2 public class Main {
 3   public static void main(String[] args) {
 4     A a = new A();
 5     int id;
 6     if (args.length > 0)
 7       id = 1;
 8     else
 9       id = 0;
10     a.addData(id);
11     System.out.println(a);
12   }
13 }
14 class A {
15   B b = new B();
16   void addData(int id) {
17     b.addData(id);
18   }
19 }
```

```
20 class B {
21   int max = 4;
22   X x = new X();
23   void addData(int id) {
24     if (id >= 0 && id <= max)
25       x.addData(id);
26   }
27 }
28 class X {
29   int[] idList = new int[16];
30   int count = 0;
31   void addData(int id) {
32     System.out.println(id);
33     idList[count] = id;
34     count = count + 1;
35   }
36   int getData(int index) {
37     return idList[index];
38   }
39 }
```

図 3 Thin Slicing の例
Fig. 3 Examples of Thin Slicing.

Slice は，{6，7，10，12，14，15，19} 行目となる．一方，従来のスライスは実行可能なすべての文となる．

このように Thin Slicing は従来のスライシングよりスライスサイズが小さくなり，使用している変数の値を生成した文がより明確になる．プログラム理解において開発者がデータ依存関係を調査する際に，この特徴を利用することによって調査にかかる時間を減らすことが期待できる．

特に，ソースコード中のある文で使用しているデータについて開発者が調べたい際に，そのデータが様々なメソッドを経由して到達している場合や多くの文をたどって到達している場合は Thin Slicing の効果が高い．**図 3** の例では，Thin Slicing で計算するデータフローを矢印で示している．たとえば，32 行目で出力している変数 id の値について調べたい場合は，クラス X の addData メソッドの呼び出し元をたどっていかないと変数 id の値を生成した文は分からないが，Thin Slicing を利用すると Thin Slice として {7，9，10，17，25，32} 行目が抽出され，7 行目と 9 行目で生成された値を使用していることが特定できる．文献 [8] では，Thin Slicing は，13 個のデバッグの例，22 個のプログラム理解の例で有効であることが示されている．

一方で，調べたいデータに関する依存関係が 1 つのメソッド内で完結している場合は，そのメソッドの内容が複雑でない限り，Thin Slicing の効果は低い．たとえば，図 3 において，11 行目で出力している変数 a の値について調べたい場合は，その周辺のコードを読めば Thin Slicing を利用しなくても，4 行目で生成された値を使用していることが特定できる．

2.5 スライスサイズに関する調査

プログラムスライシングの各手法が，一般にどの程度有効であるかという点については，既存研究で，平均的なスライスサイズを求める形で評価が行われている．Binkley ら [7] は，スライスサイズは平均でプログラム全体の約 30%になることを示している．スライスの高速な近似計算を提案した Jász ら [9] は，その計算結果がスライスサイズより平均で 4.27%大きくなることを示している．Horwitz ら [19] は，デバッグ作業において，調査したいスタックトレースを限定した場合のスライスサイズが通常の約 3 分の 1 になることを示している．本研究は Thin Slicing に対して，このような実験によるスライスサイズの評価を行ったものである．

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を対象とした Thin Slicing を実装した．バイトコードはソースコードに比べ解析が容易であるため，本研究の実装ではバイトコードを用いる．バイトコードは**表 1** のような命令からなる．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．本研究で扱うバイトコードの例を**図 4** に示す．このバイトコードは，図 1 の main メソッドおよび add メソッドに対応しており，そのメソッド内でのインデクスと命令を表している．ただし，FRAME-OP や括弧付きで表現されている（L1591151994）と（line=6）などの行は，バイトコードの属性情報であり，命令ではない．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．

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	LADD	FADD	DADD	ISUB	LSUB	FSUB	DSUB	IMUL
		LMUL	FMUL	DMUL	IDIV	LDIV	FDIV	DDIV	IREM	LREM
		FREM	DREM	INEG	LNEG	FNEG	DNEG	ISHL	LSHL	ISHR
		LSHR	IUSHR	LUSHR	IAND	LAND	IOR	LOR	IXOR	LXOR
		IINC	I2L	I2F	I2D	L2I	L2F	L2D	F2I	F2L
		F2D	D2I	D2L	D2F	I2B	I2C	I2S	LCMP	FCMPL
		FCMPG	DCMPL	DCMPG						
定数の生成	source	ICONST_M1		ICONST_0		ICONST_1		ICONST_2		LDC
		ICONST_3		ICONST_4		ICONST_5		LCONST_0		BIPUSH
		LCONST_1		FCONST_0		FCONST_1		FCONST_2		SIPUSH
		DCONST_0		DCONST_1		ACONST_NULL				
比較演算	sink	IFEQ	IFNE	IFLT	IFGE	IFGT	IFLE	IFNULL	IF_ICMPEQ	
		IF_ICMPNE		IF_ICMPLT		IF_ICMPGE		IF_ICMPGT		
		IF_ICMPLE		IF_ACMPEQ		IF_ACMPNE		IFNONNULL		
		TABLESWITCH		LOOKUPSWITCH						
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			

```
slice/Main#main#
([Ljava/lang/String;)V
 0: (L1591151994)
 1: (line=6)
 2: ICONST_1
 3: ISTORE 4 (a)
 4: (L2025190714)
 5: (line=7)
 6: ICONST_2
 7: ISTORE 5 (b)
 8: (L1912008895)
 9: (line=8)
 10: NEW
 11: DUP
 12: INVOKESPECIAL
 slice/A#<init>()V
 13: ASTORE 1 (x)
 14: (L188204557)
 15: (line=9)
 16: ALOAD 1 (x)
 17: ASTORE 2 (z)
 18: (L2058061115)
 19: (line=10)
 20: ILOAD 4 (a)
 21: ILOAD 5 (b)
 22: INVOKESTATIC
 slice/Main#add(II)I
 23: ISTORE 6 (c)
 24: (L314057576)
 25: (line=11)
 26: ALOAD 1 (x)
 27: ASTORE 3 (w)
 28: (L1998359153)
 29: (line=12)
 30: ALOAD 3 (w)
 31: ILOAD 6 (c)
 32: PUTFIELD slice/A#f:int
 33: (L1925529038)
 34: (line=13)
 35: ALOAD 3 (w)
 36: ALOAD 2 (z)
 37: IF_ACMPNE L1935465023
 38: (L1425840452)
 39: (line=14)
 40: ALOAD 2 (z)
 41: GETFIELD slice/A#f:int
 42: ISTORE 7 (v)
 43: (L330459891)
 44: (line=15)
 45: GETSTATIC
 java/lang/System#out:
 java/io/PrintStream
 46: ILOAD 7 (v)
 47: INVOKEVIRTUAL
 java/io/PrintStream#
 println(I)V
 48: (L1935465023)
 49: (line=17)
 50: FRAME-OP(0)
 51: RETURN
 52: (L646676895)
slice/Main#add#(II)I
 0: (L283836798)
 1: (line=18)
 2: ILOAD 0 (x)
 3: ILOAD 1 (y)
 4: IADD
 5: IRETURN
 6: (L883883999)
```

図 4 図 1 に対応するバイトコード
Fig. 4 A list of Java bytecode for the program in Fig. 1.

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 [23] に基づいた Object-sensitive Pointer Analysis [24] 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 [25] を用いた．

4.4 オペランド・スタック管理命令

オペランド・スタックを直接操作するための命令については，本研究の実装では，命令の影響をデータ依存関係の計算に直接反映する．たとえばメソッド呼び出し命令の戻り値が使用されない場合，バイトコード上ではその値を破棄する POP 命令が使用されるが，メソッド呼び出し命令から POP 命令へとデータ依存辺を引く代わりに，メソッ

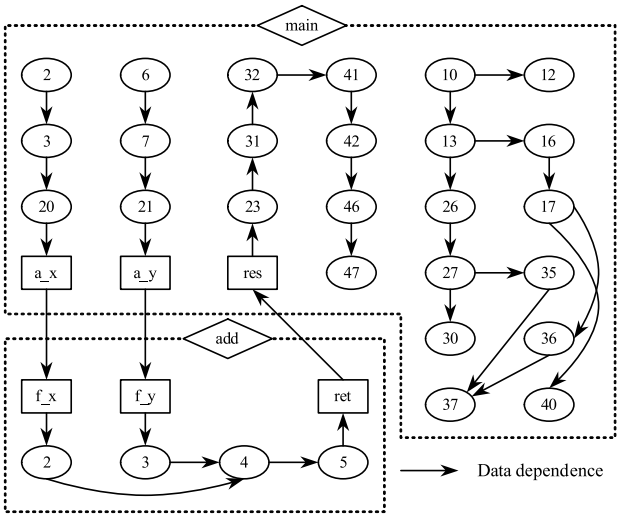

図 5 図 1 のバイトコードのグラフ
Fig. 5 A graph of bytecode for the program in Fig. 1.

ド呼び出し命令から戻り値に関するデータ依存辺が存在しないものとして扱う．そのため，オペランド・スタック管理命令から依存辺が引かれたり，これらの命令に依存辺が引かれたりすることはない．

4.5 バイトコードのグラフ

図 5 は，図 1 のプログラムに対応するバイトコードのグラフである．楕円内の数字は図 4 のバイトコードのインデクスに対応している．

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，実パラメータがそれぞれ 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．

***LOC*(S_{ts})** Thin Slice S_{ts} のソースコード上での行番号の集合．

$LOC(S_{ts})$ は，クラスファイルの行番号情報を利用して計算する．仮パラメータ頂点はメソッドの宣言部（図 1 の 3，18 行目）に対応し，ソースコード 1 行分として計算する．返り値頂点はメソッドの終了部（図 1 の 17，20 行目）に対応し，ソースコード 1 行分として計算する．実パラメータ頂点と返り値を受け取る頂点は対応するバイトコードの命令の行番号と同じであるとする．

例として図 5 において，main メソッドのインデクス 47 の頂点をスライシング基準とした後ろ向き Thin Slice は $Backward(47_{main}) = \{(2,\ 3,\ 6,\ 7,\ 20,\ 21,\ 23,\ 31,\ 32,\ 41,\ 42,\ 46,\ 47,\ \text{a\_x},\ \text{a\_y},\ \text{res})_{main},\ (2,\ 3,\ 4,\ 5,\ \text{f\_x},\ \text{f\_y},\ \text{ret})_{add}\}$ で，$|Backward(47_{main})| = 23$ となる．また，$Source(47_{main})$ は $Backward(47_{main})$ に含まれる source である命令の集合なので，$Source(47_{main}) = \{2_{main},\ 6_{main},\ 4_{add}\}$ で，source 命令の数は $|Source(47_{main})| = 3$ となる．スライスに含まれるメソッドの集合は $Method(Backward(47_{main})) = \{main,\ add\}$ で，メソッド数は $|Method(Backward(47_{main}))| = 2$ となる．スライスに含まれるクラスの集合は $Class(Backward(47_{main})) = \{Main\}$ で，$|Class(Backward(47_{main}))| = 1$ となる．$LOC(Backward(47_{main})) = \{6,\ 7,\ 10,\ 12,\ 14,\ 15,\ 18,\ 19,\ 20\}$ で，$|LOC(Backward(47_{main}))| = 9$ となる．

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．また，$|LOC(Backward(v))|$ の平均値を計算し，バイトコードとソースコードのサイズを比較する．

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．

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

プログラム名		tomcat	luindex	sunflow	avrora	pmd	xalan	batik
グラフの頂点数		54,468	123,191	190,526	211,343	448,722	815,861	968,470
ソースコードの行数		-	36,060	43,974	68,936	-	-	-
$\|Backward(v)\|$	最大値	922	12,820	34,512	30,458	40,784	62,957	93,810
	最大値の割合（%）	1.7	10.5	15.3	14.4	9.1	7.7	9.7
	平均値	55.2	3012.8	7739.2	6919.3	7108.6	12482.7	26159.3
	平均値の割合（%）	0.10	2.5	4.2	3.3	1.6	1.5	2.7
$\|Forward(w)\|$	最大値	3,673	14,326	22,737	27,592	52,339	105,749	140,965
	最大値の割合（%）	6.7	11.7	12.4	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	2.1	3.3	1.1	1.4	2.3	3.0
$\|LOC(Backward(v))\|$	最大値	-	4678	7144	10602	-	-	-
	最大値の割合（%）	-	13.0	16.2	15.4	-	-	-
	平均値	-	1102.6	1933.2	2425.6	-	-	-
	平均値の割合（%）	-	3.1	4.4	3.5	-	-	-
$\|Method(Backward(v))\|$	最大値	193	1,342	1727	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	1,482	870	3,702	3,132	7,175	8,011
	平均値	6.4	116.1	213.4	331.0	369.6	1267.1	1632.0
$\|Class(Backward(v))\|$	最大値	41	288	300	1,284	685	1,352	1,792
	平均値	2.9	68.6	83.6	278.0	111.3	264.9	489.7
$\|Class(Forward(w))\|$	最大値	68	329	170	1,147	507	1,306	1,750
	平均値	2.3	63.9	43.1	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	1375.8	3575.9	3470.6	3076.6	5215.6	11183.8
$\|Sink(w)\|$	最大値	1,610	444,1	8,108	11,689	19,425	41,423	52,334
	平均値	25.4	819.8	2202.1	1021.4	2264.2	7238.5	10662.3

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

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] を

[*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	行数
tomcat	261	2,389	54,468	-
luindex	560	4,180	123,191	36,060
sunflow	650	4,507	183,185	43,974
avrora	1,838	9,304	211,343	68,936
pmd	2,369	16,439	448,722	-
xalan	2,805	22,377	815,861	-
batik	4,417	28,818	968,470	-

Windows 7（64 bit）上で JDK 1.6.0_45 を用いてビルドしたものである．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** のとおりである．ただし，tomcat，pmd，xalan，batik については，一部のライブラリのソースコードが取得できなかったため，行数に関する調査は行っていない．これらについては，ダウンロードした jar ファイルを実験対象としている．また，表中の行数はコメント行，空行を除いたものである．

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 2.1%である．Binkley らの実験 [7] 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．luindex，sunflow，avrora について，$|LOC(Backward(v))|$ は $|Backward(v)|$ より平均して約 0.3 ポイント大きくなっている．また，これら 2 つの値の相関係数は 0.9996 となっており，ソースコードにおける Thin Slice のサイズも同様に十分小さいことが予想される．**図 6** 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わた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いえる．ただし，スライス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．

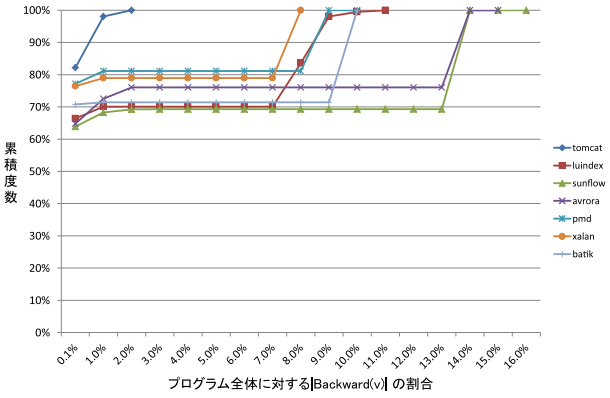

図 6 $|Backward(v)|$ の累積度数分布

Fig. 6 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	4.5	3.6	11.0
sunflow	1.8	2.5	2.4	6.7
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

6. 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として，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おり [26]，Thin Slicing の結果を開発者に提示することも有効であると期待できる．

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，文献 [8] 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 [27] 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, *Proc. 28th ACM/IEEE International Conference on Software Engineering*, pp.492–501 (2006).

[3] Mäder, P. and Egyed, A.: Assessing the Effect of Requirements Traceability for Software Maintenance, *Proc. 28th IEEE International Conference on Software Maintenance*, pp.171–180 (2012).

[4] Kuang, H., Mader, P., Hu, H., Ghabi, A., Huang, L., Jian, L. and Egyed, A.: Do Data Dependencies in Source Code Complement Call Dependencies for Understanding Requirements Traceability?, *Proc. 28th IEEE International Conference on Software Maintenance*, pp.181–190 (2012).

[5] Wang, J., Peng, X., Xing, Z. and Zhao, W.: An Exploratory Study of Feature Location Process: Distinct Phases, Recurring Patterns, and Elementary Actions, *Proc. 27th IEEE International Conference on Software Maintenance*, pp.213–222 (2011).

[6] Weiser, M.: Program Slicing, *Proc. 5th International Conference on Software Engineering*, pp.439–449 (1981).
[7] Binkley, D., Gold, N. and Harman, M.: An Empirical Study of Static Program Slice Size, *ACM Trans. Software Engineering and Methodology*, Vol.16, No.2, pp.1–32 (2007).
[8] Sridharan, M., Fink, S.J. and Bodik, R.: Thin Slicing, *Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.112–122 (2007).
[9] Jász, J., Árpád Beszédes, Gyimóthy, T. and Rajlich, V.: Static Execute After/Before as a Replacement of Traditional Software Dependencies, *Proc. 24th IEEE International Conference on Software Maintenance*, pp.137–146 (2008).
[10] Horwitz, S., Reps, T. and Binkley, D.: Interprocedural Slicing Using Dependence Graphs, *Proc. ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp.35–46 (1988).
[11] LaToza, T.D. and Myers, B.A.: Developers Ask Reachability Questions, *Proc. 32nd IEEE/ACM International Conference on Software Engineering - Volume 1*, pp.185–194 (2010).
[12] Zeller, A.: *Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufmann Publishers Inc. (2005).
[13] 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).
[14] Anderson, P., Reps, T. and Teitelbaum, T.: Design and Implementation of a Fine-Grained Software Inspection Tool, *IEEE Trans. Softw. Eng.*, Vol.29, No.8, pp.721–733 (2003).
[15] Jackson, D. and Rollins, E.J.: A New Model of Program Dependences for Reverse Engineering, *Proc. 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp.2–10 (1994).
[16] Reps, T. and Rosay, G.: Precise Interprocedural Chopping, *Proc. 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp.41–52 (1995).
[17] Krinke, J.: Visualization of Program Dependence and Slices, *Proc. 20th IEEE International Conference on Software Maintenance*, pp.168–177 (2004).
[18] Chen, K. and Rajlich, V.: Case Study of Feature Location Using Dependence Graph, *Proc. 8th IEEE International Workshop on Program Comprehension*, pp.241–247 (2000).
[19] Horwitz, S., Liblit, B. and Polishchuk, M.: Better Debugging via Output Tracing and Callstack-Sensitive Slicing, *IEEE Trans. Softw. Eng.*, Vol.36, No.1, pp.7–19 (2010).
[20] Ceccato, M., Preda, M.D., Nagra, J., Collberg, C. and Tonella, P.: Barrier Slicing for Remote Software Trusting, *Proc. 7th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp.27–36 (2007).
[21] Krinke, J.: Slicing, Chopping, and Path Conditions with Barriers, *Software Quality Journal*, Vol.12, No.4, pp.339–360 (2004).
[22] Wang, X., Lo, D., Cheng, J., Zhang, L., Mei, H. and Yu, J.X.: Matching Dependence-related Queries in the System Dependence Graph, *Proc. 25th IEEE/ACM International Conference on Automated Software Engineering*, pp.457–466 (2010).
[23] Andersen, L.O.: Program Analysis and Specialization for the C Programming Language, Ph.D. Thesis, University of Copenhagen (1994).
[24] Milanova, A., Rountev, A. and Ryder, B.G.: Parameterized Object Sensitivity for Points-to Analysis for Java, *ACM Trans. Software Engineering and Methodology*, Vol.14, No.1, pp.1–41 (2005).
[25] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E. and Godin, C.: Practical Virtual Method Call Resolution for Java, *Proc. 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp.264–280 (2000).
[26] Ishio, T., Etsuda, S. and Inoue, K.: A Lightweight Visualization of Interprocedural Data-flow Paths for Source Code Reading, *Proc. 20th IEEE International Conference on Program Comprehension*, pp.37–46 (2012).
[27] 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，pp.1–8 (2012).

秦野 智臣

平成 25 年大阪大学基礎工学部情報科学科卒業．現在，同大学大学院情報科学研究科博士前期課程 1 年．プログラム解析に関する研究に従事．

鹿島 悠

平成 22 年大阪大学基礎工学部情報科学科卒業．平成 24 年同大学大学院情報科学研究科博士前期課程修了．同年同大学院博士後期課程入学．プログラム解析に関する研究に従事．

石尾 隆 （正会員）

平成 15 年大阪大学大学院基礎工学研究科博士前期課程修了．平成 18 年同大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員（PD）．同年ブリティッシュコロンビア大学ポストドクトラルフェロー．平成 19 年大阪大学大学院情報科学研究科コンピュータサイエンス専攻助教．博士（情報科学）．データフロー解析，アスペクト指向プログラミングに関する研究に従事．日本ソフトウェア科学会，ACM，IEEE 各会員．

井上 克郎 （フェロー）

昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年同大学大学院博士課程修了．同年同大学基礎工学部情報工学科助手．昭和 59～61 年ハワイ大学マノア校情報工学科助教授．平成元年大阪大学基礎工学部情報工学科講師．平成 3 年同学科助教授．平成 7 年同学科教授．平成 14 年大阪大学情報科学研究科コンピュータサイエンス専攻教授．平成 20 年国立情報学研究所客員教授．同年情報処理学会フェロー．同年電子情報通信学会フェロー．工学博士．ソフトウェア工学の研究に従事．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．