修士学位論文

題目

バイトコード間の動的依存情報を抽出する Java バーチャルマシン

指導教官

井上克郎 教授

報告者

誉田 謙二

平成 14 年 2 月 13 日

大阪大学 大学院基礎工学研究科

情報数理系専攻 ソフトウェア科学分野

平成 13 年度 修士学位論文

バイトコード間の動的依存情報を抽出する Java バーチャルマシン

誉田 謙二

内容梗概

プログラムデバッグを効率よく行う手法の一つに，プログラムスライスがある．プログラムスライスとは，直感的には，プログラム中のある文のある変数の値に影響を与え得る文の集合であり，単にスライスとも呼ばれる．一般に，スライスはプログラム文間の依存関係解析により得られる．これまでに多くのスライス計算技法が提案されており，主なものに，静的スライス，動的スライス，DC スライスがある．

また，近年のソフトウェア開発環境におけるオブジェクト指向言語の利用の高まりに伴い，クラスや継承など，オブジェクト指向特有の概念を考慮した静的スライス，動的スライス，DC スライスが提案されてきた．オブジェクト指向プログラムは実行時決定要素を数多く含んでおり，プログラムデバッグの際には，静的スライスよりも，実行時決定要素を扱うことのできる動的スライスが適しているといえる．しかし，動的スライスは，全ての実行系列を保存する必要があるため，多大な解析コストを要する．一方，DC スライスは，いくつかの実行時決定要素のみを扱うため，静的スライスより精度が高く，また動的スライスより解析コストが大幅に小さい．そのため，オブジェクト指向プログラム対するその有効性は非常に高い．

本研究では，Java バーチャルマシンに基づく DC スライス計算手法を提案する．提案手法では，バイトコードに対して DC スライスを適用する．その際，Java コンパイラが生成するバイトコードとソースコードと対応表を利用することで，バイトコード上でのスライス結果をソースコードに対応付ける．提案手法では，バイトコード単位での依存関係解析に基づいていることから，細粒度の DC スライス計算が可能となる．

主な用語

Java
Java Virtual Machine
プログラムスライス
動的解析
静的解析

目 次

1 まえがき **3**

2 プログラムスライス **5**

2.1 プログラムスライス計算法 5

2.2 静的スライス 7

2.3 動的スライス 7

2.4 Dependence Cache (DC) スライス 7

2.5 各スライス手法の比較 15

3 JVM に基づく DC スライスの Java への適用 **18**

3.1 方針 18

3.2 DC スライスのバイトコードへの適用 20

3.2.1 静的制御依存関係解析 21

3.2.2 動的データ依存関係解析 25

3.2.3 PDG の構築 27

3.2.4 スライス計算 27

3.3 バイトコードとソースコードの対応 28

4 考察 **30**

4.1 関連研究 30

4.1.1 バイトコードの埋め込み方式による動的解析 30

4.1.2 ソースコードの埋め込み方式による動的解析 30

4.2 提案手法の有効性 31

5 提案手法の実現 **33**

5.1 システム構成 33

5.2 評価 36

5.2.1 スライスサイズ 37

5.2.2 解析コスト 38

6 むすび **42**

謝辞 **43**

参考文献 44

1 まえがき

現在の一般的なデバッグ作業においてはプログラム全体を扱うのが原則である．しかし，近年プログラムはより大規模で複雑なものになっている．このように大規模化，複雑化したプログラムではデバッグ時にエラー原因の特定に時間がかかり，結果としてプログラム開発の効率が悪くなる．

このような場合に，エラーに関係がある可能性の高い部分だけをプログラムから抽出できれば，開発者はその部分だけに注目することができる．その結果，プログラム中のエラーの位置を発見する負担を軽減することが可能となり，開発効率の向上が期待される．

プログラムスライス($Program\ Slice$，以下，スライス) は Weiser[18] によって提案されたものである．プログラムスライシング ($Program\ Slicing$) とは，プログラム中のある文 s におけるある変数 v (スライス基準 ($Slicing\ Criterion$) $<s, v>$ と呼ぶ) に対して v の値に影響を与えるすべての文をプログラムから抽出する技法で，その結果取り出された文の集合がスライスとなる．一般に，スライスはプログラム文間の依存関係 ($Dependence\ Relation$) の解析により得られる．

プログラムスライスを利用することにより，開発者はデバッグ時にエラーに関係のある部分のみに注目することができる．その結果，プログラム中のエラーの位置を発見する負担を軽減することができ，開発効率の向上が期待される．

当初，Weiser によって考案されたこのスライス計算手法は，ソースコードのみを用いて依存関係を解析していた．それゆえ，この手法によって計算されたスライスは静的スライス ($Static\ Slice$) と呼ばれる．この手法では，起こり得る全ての実行経路を考慮し，依存関係解析を行い，スライスを計算する．このため，実行に関係しない文 (デバッグ時に必要ないと考えられる文) もスライスに含まれるという欠点がある．

これに対して Agrawal[1] は，よりデバッグに有効な手法として動的スライス ($Dynamic\ Slice$) を提案している．動的スライスの計算には，ソースコードの代わりに実際に実行された文，実行系列 ($Execution\ Trace$) が用いられる．この手法では，特定の実行経路のみを考慮することになり，その結果，静的スライスと比較して，スライスサイズの減少 (精度の向上) が期待できる．

デバッグに利用することを考えると，動的スライスは静的スライスと比較してより望ましい結果だということができる．しかし，実行系列はソースコードと比較して膨大な量になることが多く，その保存と解析に多くの空間，時間コストを必要となる．

そこで，静的スライスと動的スライスの両方の利点を生かした **DC** スライス ($DC\ Slice$) [4][7] が提案されている．DC スライス計算法は，配列やポインタを詳しく解析するためにデータ依存関係解析を動的に，必要コスト削減のために制御依存関係は静的に行う手法であ

る．動的にデータ依存関係解析を行うことにより静的スライスより精度の高いスライスを計算できる．さらに，実行系列を保存しないため，動的スライスに比べ非常に小さい解析コストでスライス計算を行うことができる．

また，近年のソフトウェア開発環境において，C などの手続き型言語だけでなく，Java や C++など，いわゆるオブジェクト指向言語の利用が高まっている．オブジェクト指向言語には，従来の手続き型言語とは異なり，クラスや継承など，オブジェクト指向独特の概念が導入されているため，既存のスライス計算手法をそのまま適用することは困難である．そこで，オブジェクト指向言語特有の概念を考慮した静的スライス [10]，動的スライス [19]，DC スライス [13] が提案されてきた．

オブジェクト指向言語には多くの実行時決定要素が含まれており，起こり得るすべての可能性を考慮する静的スライスは精度の点で問題がある．一方，動的スライスは実行時決定要素を容易に扱うことができ，精度の高いスライスを計算できるが，実行系列の保存が必要であるため，多大な時間，空間コストを要する．DC スライスは，動的スライスに比べ格段に小さな解析コストで，静的スライスより高精度なスライスの計算が可能であり，オブジェクト指向プログラムにおけるスライス計算に有効であると考えている．

しかし，[13] での DC スライスの実現はソースコードの埋め込みによる実現を行っているため，Java 言語の構文による埋め込み位置の制約が厳しく，十分な解析の精度が得られない問題がある．またソースコードの変更を前提とするため，ソースコードの存在しないクラスを利用するプログラムに対しスライス計算を行う場合，極端に解析精度が低下する．

そこで，本研究では，オブジェクト指向言語 Java で記述されたプログラムに対する，**Java** バーチャルマシン（*Java Virtual Machine*，以下，JVM）に基づく DC スライス計算手法を提案する．提案手法では，バイトコードに対して DC スライスを計算し，コンパイラによって生成されたバイトコードとソースコードと対応表を利用し，スライス計算結果をソースコードに対応付ける．バイトコード単位での依存関係の解析に基づいていることから，構文による制約を受けることもなく，ソースコードの存在に関わらずに細粒度の DC スライス計算が可能となる．

以降，2. では既存のプログラムスライス計算手法について述べ，3. で JVM に基づく DC スライス計算手法について述べる．4. で提案手法の有効性を関連研究と比較しながら述べ，5. で，提案手法の実装と評価について述べる．最後に 6. で，まとめと今後の課題について述べる．

2　プログラムスライス

プログラムスライシング（*Program Slicing*）技術とは，プログラム中のある文 s におけるある変数 v（スライス基準<s,v>と呼ぶ）に対して v の値に影響を与える全ての文をプログラムから抽出する技術で，その結果取り出された文の集合をプログラムスライスまたは単にスライス（*Slice*）と呼ぶ．ある値 v に影響を与える文を抽出することで，プログラム中に存在するフォールトの位置特定に有効であるだけでなく，プログラム保守，プログラム理解等にも利用される．

2.1　プログラムスライス計算法

スライスの計算にはさまざまな手法が存在するが，本研究ではプログラム依存グラフによるスライス計算手法を用いる [14]．以降，いくつかの諸定義をしたのち，手法 [14] によるスライス計算手順について述べる．

プログラム文間に存在する依存関係

プログラム中の文の間には，以下に定義する制御依存関係，データ依存関係の 2 種類の依存関係が存在する．

制御依存関係

プログラム中の 2 文 s，t に関して，以下の条件を満たすとき，s から t の間に制御依存関係（*Control Dependence*，CD 関係）が存在するという．

1. s は条件文である
2. t が実行されるかどうかは，s の判定結果に依存する

データ依存関係

プログラム中の 2 文 s，t に関して，以下の条件を満たすとき，s から t の間に変数 v に関するデータ依存関係（*Data Dependence*，DD 関係）が存在するという．

1. s で v が定義される
2. t で v が参照される
3. s から t の実行可能な経路で，その経路において s から t 間に変数 v を再定義している文が存在しない，というものが存在する

プログラム依存グラフ

プログラムに対して上述の依存関係の解析を行うことによって得られた依存関係情報は，プログラム依存グラフ (*Program Dependence Graph* . 以降，PDG) として出力される . PDG とは，プログラム内の文間の依存関係を表す有向グラフであり，その節点は，プログラムに含まれる条件判定部分，代入文，入出力文，手続き呼び出し文を表し, その有向辺は 2 つの節点間の制御依存関係およびデータ依存関係を表す (それぞれを制御依存辺，データ依存辺と呼ぶ) . また，関数間にわたるデータ依存関係を表現するために特殊節点及び特殊辺も存在する [17] .

プログラムスライス計算手順

上述に示した 2 つの依存関係とプログラム依存グラフの概念を用い，以下の手順でスライスの計算がなされる .

スライス計算法

Phase 1: 依存関係解析

各プログラム文に対し，以下の依存関係解析を行う .

(a) 制御依存関係解析

(b) データ依存関係解析

Phase 2: プログラム依存グラフ構築

Phase 1 で求めた依存関係を利用し，PDG を構築する .

Phase 3: スライス計算

スライス基準 $<s,\ v>$ に対するスライスを計算する . スライス基準 $<s,\ v>$ に対するスライスは，s に対応した PDG の節点 V_s から，逆方向に制御依存辺およびデータ依存辺を経て推移的に到達可能な節点集合に対応する文の集合を計算することによって計算される .

PDG や計算されるスライスの具体的例は，2.2. , 2.3. および 2.4. で紹介する .

スライスには，依存関係解析手法の違いにより，静的スライスと動的スライスの 2 種類に大別される . 静的スライスは，プログラムに対して，起こり得る全ての可能な入力データを考慮したスライスであり，動的スライスは，ある特定の入力のもとで生成される実行系列のみを解析して求められたスライスである . また、静的スライス・動的スライスを組み合わせたスライスとして，DC スライスがある .

以降，静的スライス，動的スライスおよび DC スライスについて，順に述べる .

2.2 静的スライス

静的スライスは，スライス計算の Phase 1 において (a) 制御依存関係解析，(b) データ依存関係解析をともに静的に行うスライス計算手法である．与えられたソースコードの各文を節点とし，起こり得るすべての実行経路に対して依存関係解析を行う．

静的スライスは，現実には短い時間で計算される．静的スライスは，プログラムに起こり得るすべての実行経路を考慮して PDG を構築するため，プログラムに存在する特定の機能を抽出したい場合には有効である．しかし，プログラム実行においてすべての実行経路が利用されることは少なく，実行時エラーの原因を把握するためのフォールト位置特定に対しては効果的とはいえない．

図 1 の C 言語で記述されたソースコードに対し，静的スライスを計算するときに構築される PDG は図 2 のようになる．さらに，図 1 のソースコードの，スライス基準 $< 37, d >$ に関する静的スライスを図 3 に，比較のため元のソースプログラムと併せて示す．

2.3 動的スライス

動的スライスは，スライス計算の Phase 1 において (a) 制御依存関係解析，(b) データ依存関係解析をともに動的に行うスライス計算手法である．まず，特定の入力を与えてプログラムを実行する．そして，得られた実行系列の各実行時点 (*Execution Point*) を節点とし，特定の実行経路に対する依存関係解析による PDG を構築し，スライスを計算する．

動的スライスでは，解析対象を特定の実行経路に限定し，その実行の際に発生する依存関係に基づくものであるため，一般に計算されるスライスは静的スライスに比べて小さくなる．また，実際に実行された部分の中からのみスライスが計算されるため，フォールト位置特定を効率よく行うことができる．しかし，動的スライスの計算には，実行系列および，実行中に発生するすべての制御依存関係とデータ依存関係を記憶しなければならないため，多大な空間コストと時間コストを要する．とりわけ，実行系列の大きさはプログラム実行文の数に比例することから，入力データによっては非常に大きなものとなり，それに伴いスライス計算に要する時間も増大する．

図 1 の C 言語で記述されたソースコードに対し，入力として 2 を与えた場合の実行系列を図 5 に示す．また，構築される PDG を図 4 に示す．さらに，スライス基準 $< 37, d >$ に関する動的スライスを図 6 に，比較のため元のソースプログラムと併せて示す．

2.4 Dependence Cache (DC) スライス

例えば，配列を含むプログラムに対して静的スライスを計算する場合，配列の添字の値を静的に把握することは難しく，可能性のある添字値をすべて考慮しなければならないため不

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 3:
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 7:
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
13:
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
19:
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
23:
24:      printf("Input: ");
25:      scanf("%d", &c);
26:
27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }
30:
31:      d = cube(b[c]);
32:
33:      if (d < 0) {
34:          d = -1 * d;
35:      }
36:
37:      printf("%d\n", d);
38:  }
```

図 1: サンプルソースコード

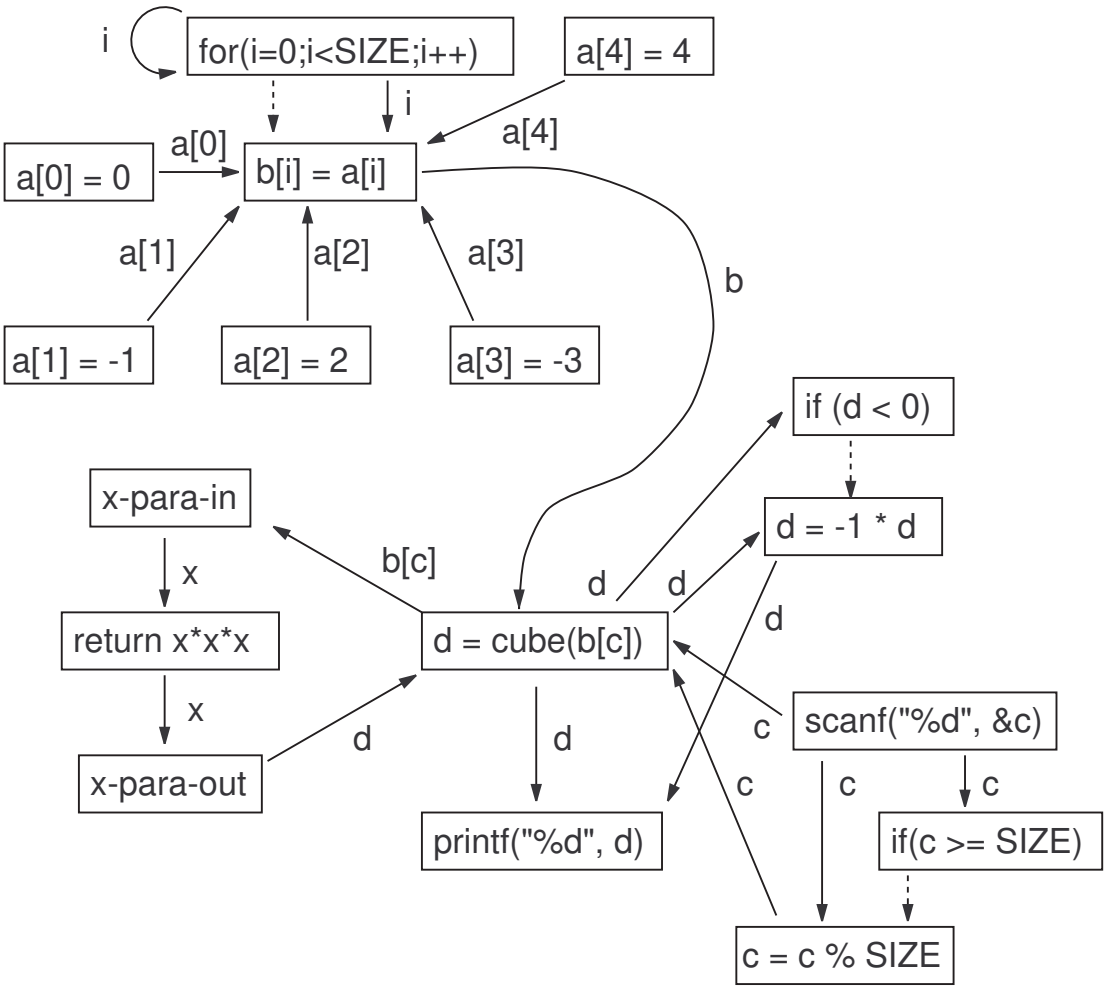


図 2: 図 1 のプログラムに対して静的に構築される PDG

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 3:
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 7:
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
13:
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
19:
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
23:
24:      printf("Input: ");
25:      scanf("%d", &c);
26:
27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }
30:
31:      d = cube(b[c]);
32:
33:      if (d < 0) {
34:          d = -1 * d;
35:      }
36:
37:      printf("%d\n", d);
38:  }
```

サンプルコード

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
25:      scanf("%d", &c);
27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }
31:      d = cube(b[c]);
33:      if (d < 0) {
34:          d = -1 * d;
35:      }
37:      printf("%d\n", d);
38:  }
```

<37，d>に関する静的スライス

図 3: 図 1 のプログラムの $< 37, d >$ に関する静的スライス

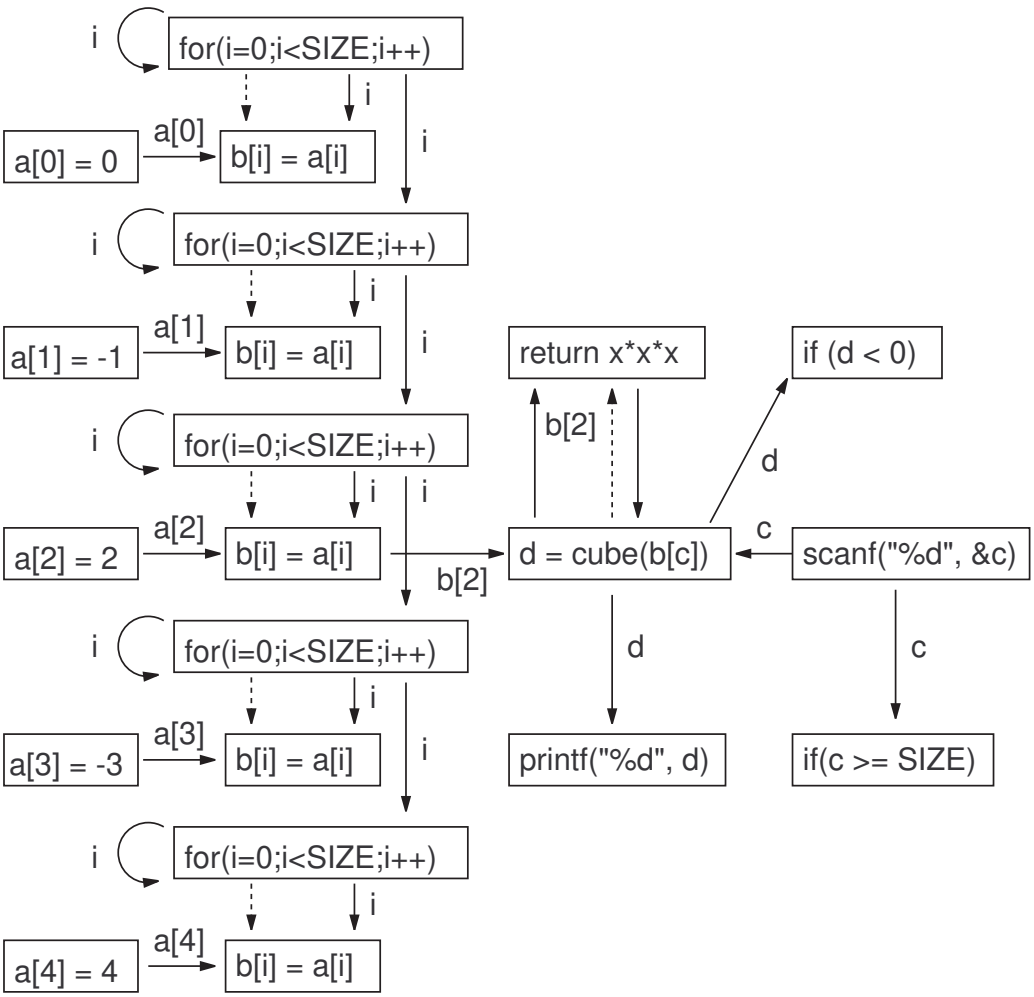


図 4: 図 1 のプログラムに対して動的に構築される PDG

```
 t1:  #include <stdio.h>
 t2:  #define SIZE 5
 t3:  void main(void)
 t4:  int a[SIZE];
 t5:  int b[SIZE];
 t6:  int c, d, i;
 t7:  a[0] = 0;
 t8:  a[1] = -1;
 t9:  a[2] = 2;
t10:  a[3] = -3;
t11:  a[4] = 4;
t12:  for (i=0; i<SIZE; i++)
t13:  b[0] = a[0];
t14:  for (i=0; i<SIZE; i++)
t15:  b[1] = a[1];
t16:  for (i=0; i<SIZE; i++)
t17:  b[2] = a[2];
t18:  for (i=0; i<SIZE; i++)
t19:  b[3] = a[3];
t20:  for (i=0; i<SIZE; i++)
t21:  b[4] = a[4];
t22:  scanf("%d", &c);
t23:  if (c >= SIZE)
t24:  d = cube(b[2]);
t25:  int cube(2)
t26:  return 2*2*2;
t27:  if (d < 0)
t28:  printf("%d\n", d);
```

図 5: 図 1 のプログラムに入力 2 を与えた場合の実行系列

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 3:
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 7:
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
13:
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
19:
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
23:
24:      printf("Input: ");
25:      scanf("%d", &c);
26:
27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }
30:
31:      d = cube(b[c]);
32:
33:      if (d < 0) {
34:          d = -1 * d;
35:      }
36:
37:      printf("%d\n", d);
38:  }
```

サンプルコード

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
16:      a[2] = 2;
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
25:      scanf("%d", &c);
31:      d = cube(b[c]);
37:      printf("%d\n", d);
38:  }
```

<37，d>に関する動的スライス

図 6: 図 1 のプログラムに入力 2 を与えた際の，< 37，$d >$ に関する動的スライス

要に多くのデータ依存関係を生成してしまうことがある．また，ポインタを介したエイリアス（$Alias$）などによる陽に現れないデータ依存関係を考慮しなければならないため，静的なデータ依存関係解析には限界がある．

DC スライスは，スライス計算の Phase 1 における (a) 制御依存関係解析を静的に，(b) データ依存関係解析を動的に行うスライス計算手法である．動的にデータ依存関係解析を行うことにより，配列の添字やポインタの参照先などの実行時決定要素を正確に把握することができる．一方，制御依存関係解析については静的に行うため，実行系列を保存する必要がなく，動的スライスに比べ解析コストを抑えることができる．

動的データ依存関係解析

プログラムの実行時に軽量な手間でデータ依存関係を抽出することを考える．プログラム中のある文 s においてある変数 v が参照されるとき，v を定義した文 t が分かれば，s から t の間に，v に関するデータ依存関係が存在することが把握できる．つまり，各変数 v について，その変数がどこで定義されたかを保存しながらプログラムを実行すれば，動的なデータ依存関係解析を実現することができる．

そこで，DC スライスの計算では，プログラム中で用いられるすべての変数 v に対しキャッシュ（$Cache$）$C(v)$ を用意する．$C(v)$ には変数 v が最後に定義された文番号が格納されており，文 t の実行時に変数 v に対するアクセスがあった場合，次のような処理が行われる．

文 t で v が定義された場合

$C(v)$ の値を t の文番号に更新する．

文 t で v が参照された場合

$C(v)$ に対応する命令と t に対応する命令の間に発生する v に関するデータ依存関係を抽出する．

例として，図 7 のような配列を含むプログラムに対して動的データ依存関係解析を行う場合を考える．入力として変数 c に 0 を与えて実行させたときの各実行時点における各変数 v のキャッシュ$C(v)$ の推移を表 1 に示す．

文 1 から文 6 では，それぞれ変数 $a[0]$，$a[1]$，$a[2]$，$a[3]$，$a[4]$，c が定義されているため，文 6 の実行が終了した時点で $C(a[0]) = 1$，$C(a[1]) = 2$，$C(a[2]) = 3$，$C(a[3]) = 4$，$C(a[4]) = 5$，$C(c) = 6$ となる．文 7 で変数 $a[0]$ が参照されるため，文 7 の実行時に文 $C(a[0])$，つまり文 1 と文 7 の間に $a[0]$ に関するデータ依存関係が発生することになる．

```
1: a[0] = 0;
2: a[1] = 1;
3: a[2] = 2;
4: a[3] = 2;
5: a[4] = 2;
6: read(c);
7: b = a[c] + 5;
```

図 7: 配列を含むプログラム

表 1: 図 7 におけるキャッシュの推移

実行文	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	b	c
1	1	-	-	-	-	-	-
2	1	2	-	-	-	-	-
3	1	2	3	-	-	-	-
4	1	2	3	4	-	-	-
5	1	2	3	4	5	-	-
6	1	2	3	5	4	-	6
7	1	2	3	5	4	7	6

上述のように動的に抽出されるデータ依存関係と，静的に抽出される制御依存関係を用いて PDG が構築される．そして，他の手法と同様に，スライス基準に対応する節点から，グラフを探索し，到達可能な節点集合を求め，それに対応する文を得ることによって DC スライスが計算される．

DC スライスの例として，図 1 の C 言語で記述されたソースコードに対し，動的スライスと同様に入力 2 を与えて実行し，スライス基準 $< 37, d >$ に関する DC スライスを計算した結果を図 8 に，比較のため元のソースプログラムと併せて示す．

2.5 各スライス手法の比較

静的スライス，動的スライス，DC スライスの計算手法の違いを表 2 に示す．

表 2: 各スライス手法の違い

	静的スライス	DC スライス	動的スライス
CD 関係解析	静的	静的	動的
DD 関係解析	静的	動的	動的
PDG 節点	プログラム文	プログラム文	実行時点

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 3:
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 7:
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
13:
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
19:
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
23:
24:      printf("Input: ");
25:      scanf("%d", &c);
26:
27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }
30:
31:      d = cube(b[c]);
32:
33:      if (d < 0) {
34:          d = -1 * d;
35:      }
36:
37:      printf("%d\n", d);
38:  }
```

サンプルコード

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
25:      scanf("%d", &c);
31:      d = cube(b[c]);
37:      printf("%d\n", d);
38:  }
```

<37 , d>に関する DC スライス

図 8: 図 1 のプログラムの $< 37, d >$ に関する DC スライス

また，計算手法の違いにより，解析精度（スライスサイズ），解析コスト（依存関係解析時間）に関し以下のような特性を持ち，これらは手続き型言語 Pascal で記述されたプログラムに対する実験により実証されている [4][16]．

解析精度（スライスサイズ）：静的スライス $\geq$ DC スライス $\geq$ 動的スライス

解析コスト（依存関係解析時間）：動的スライス $\gg$ DC スライス $>$ 静的スライス

これらのことから，DC スライスは，動的スライスより格段に小さい解析コストで，静的スライスより高い精度のスライスを計算できる手法といえる．

3　JVM に基づく DC スライスの Java への適用

オブジェクト指向言語である Java において，データ格納の基本単位であるオブジェクト (または，クラスのインスタンス) は，データおよびそれを扱うメソッドの組み合わせにより構成される．そのため，Java プログラムに対してスライス計算を行う場合，実行メソッドを特定するためにインスタンスの型を把握しておかねばならない．

しかし，静的スライスではプログラム実行を伴わないため，実行時のインスタンスの型を静的に類推する必要がある．既存の手法 [15] を用いることで，いくつかの型に限定することは可能であるが，その型を一意に決定することは難しく，実行メソッドを特定するのは困難である．そのため，静的スライスは精度の面で限界があるといえる．

一方，動的スライスはプログラム実行を伴うため，実行時のインスタンスの型を容易に把握することが可能である．そのため，実行メソッドも特定できることから，精度の高いスライス計算が可能である．

このように，Java に対するスライス計算を考える際には，精度の点で動的スライスは静的スライスより優れているといえる．しかし，動的スライスは実行系列の保存を必要とするため，解析コストが膨大なものとなる．そこで本研究では，Java に対し DC スライスを適用する．DC スライスでは，インスタンスの型などの実行時決定要素を正確に把握でき，高い精度のスライス計算が可能となる．さらに，実行系列の保存を必要とせず，動的スライスに比べ解析コストを大幅に抑えることができる．

3.1　方針

Java における実行時決定要素としては，以下のようなものを挙げることができる．

配列の添字
: 配列の添字は実行時に決定される場合があり，その場合にアクセスされる配列の要素は，実行時に決定される．

オブジェクト参照
: 参照型変数が参照するオブジェクトやその型は実行時に決定される．

動的束縛
: 実行中のオブジェクトのクラスに基づいて適切なオーバーライドメソッドが選択される．

例外処理
: 全ての文の実行で例外の発生する可能性があり，制御はその種類の例外を処理できる最初の catch 節に移る．

並列処理

並列プログラムの制御の流れは一意に決定されない．

DC スライスの計算に必要となる動的データ依存関係解析の実現には，これらを正確に把握するための環境が求められる．そこで，Java ソースコードのコンパイル結果として生成されるバイトコードを解釈，実行する機構である JVM を利用した動的データ依存関係解析の実現を考える．JVM 上でバイトコードが実行される際には，上記の実行時決定要素のすべてを把握することが可能で，バイトコードに対し，動的データ依存関係解析を行うことができる．

また一般に，Java で記述されたソースコードの各文は，コンパイラにより複数のバイトコード命令に変換される．そのため，ソースコードの文単位で依存関係解析を行うのではなく，バイトコードの命令単位で依存関係解析を行うことで，細粒度の解析の実現が期待できる．

本研究では，本来ソースコード対して提案されてきた DC スライスをバイトコードに適用する．具体的には，バイトコードの各命令を節点とする PDG を構築し，バイトコード上での DC スライスの計算を行う．そして，バイトコードに対して得られた DC スライスをソースコードへ反映させる．

ソースコードとバイトコードの対応付けは，コンパイラの機能拡張により実現する．通常，Java コンパイラは Java で記述されたソースコードをバイトコードに変換することが主な役割であるが，本手法ではさらに，図 9 に示す，バイトコードの各命令とソースコードのトークン集合との対応表の生成を Java コンパイラにより行う．この対応表を用いることで，

- ユーザにより指定されたソースコード上でのスライス基準と，それに対応するバイトコードとの対応付け
- バイトコードにおいて計算された DC スライスと，それに対応するソースコードとの対応付け

を行うことが可能となる．

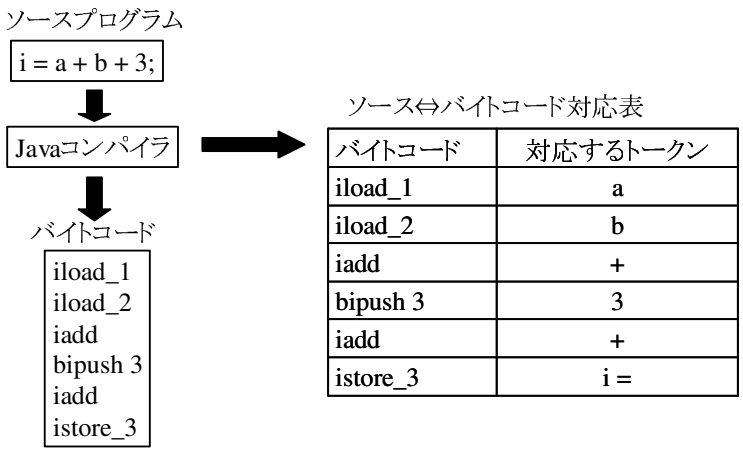


図 9: コンパイラによる対応表の出力

以降，DC スライスのバイトコードへの適用について述べる．

3.2 DC スライスのバイトコードへの適用

バイトコードに対する DC スライスの計算は以下の手順により実現される．

バイトコードに対する DC スライス計算手順

Phase 1: 依存関係解析

各バイトコード命令に対し，

(a) 静的制御依存解析

(b) 動的データ依存解析

を行う．(a)3.2.1. で，(b) については 3.2.2. で詳しく述べる．

Phase 2: PDG 構築

Phase 1 で求めた依存関係を利用し，バイトコードの各命令を節点とした PDG を構築する．

Phase 3: スライス抽出

指定されたスライス基準に対応する節点からグラフ探索を行い，バイトコードでのスライスを計算する．

以降，各 Phase での処理について詳細を述べる．

3.2.1 静的制御依存関係解析

Phase 1(a) では，与えられたバイトコードに対し，制御依存関係解析を静的に行う．その際，ソースコードにおける条件節とその述部という概念に基づく制御依存関係の定義を，そのままバイトコードに適用することは困難であるため，本研究では，バイトコードでの制御依存関係を [3] に従い定義する．以降，まずこれからの議論に必要な用語について説明し，バイトコードにおける制御依存関係の定義を行う．

制御フローグラフ

制御フローグラフ (*Control Flow Graph* ，CFG) とは，プログラムの制御の流れをグラフで表したものである．制御フローグラフは，プログラム中の分岐も合流もない部分 (基本ブロック[2]，*basic block*) を節点とし，それらの間を分岐や合流を表す有向辺で表した有向グラフである．

また，制御フローグラフにおいて，節点 X から節点 Y に向かって有向辺が引かれているとき，X は Y の先行ノード (*predecessor node*) という．X の先行ノードの集合を $Pred(X)$ と表す．

支配木

X，Y を制御フローグラフの節点としたとき，その入口節点から Y に至る全ての経路 (パス) に X が現れるとき，X は Y を支配する (*dominate*) という．支配関係は反射的であり，また，推移的であるため，X は自分自身を支配する．また，X が Y を支配し，Y が Z を支配するならば X は Z を支配する．

X が Y を支配し，かつ X≠Y のとき，X は Y を厳密に支配する (*strictly dominate*) という．X が Y を厳密に支配し，X から Y への経路に X 以外に Y を厳密に支配する節点がないとき，X は Y を直接支配するという．節点間の直接支配の関係を辺で表した木構造を支配木 (*dominator tree*) という．支配木の根は入口節点となる．

Dominance Frontier

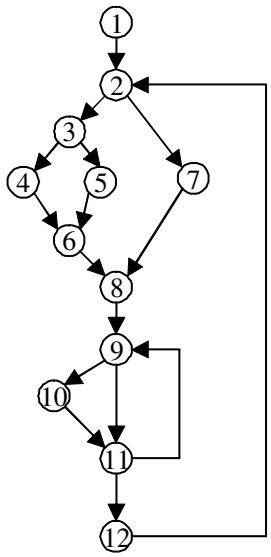
制御フローグラフ上で，節点 X からグラフを辿り，初めて X の支配から外れた節点の集合を X の **Dominance Frontier** という．制御フローグラフの節点 X の Dominance Frontier とは，次の 2 つの条件を満たす節点 Y の集合である．

1. X は Y の先行ノード集合 $Pred(Y)$ の中の少なくとも 1 つの節点を支配する．
2. X は Y を厳密に支配しない．

支配木を構築するアルゴリズム [6][12] および Dominance Frontier を構築するアルゴリズム [5] は既に提案されている．図 10 に制御フローグラフの例を示す．この例において，

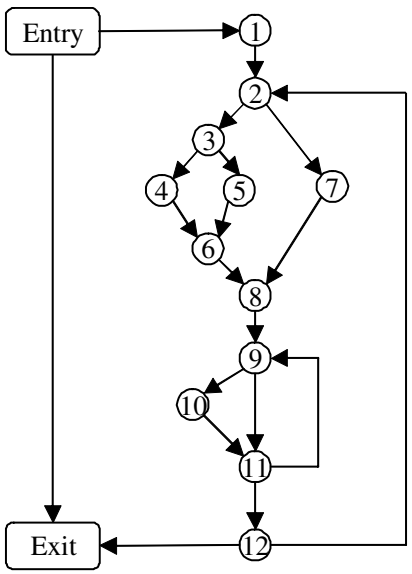
$Pred(2) = \{1, 12\}$ である．また，図 10 の制御フローグラフに入口節点・出口節点を加え，その節点間に有向辺を引いた制御フローグラフに対して構築した支配木を図 11 に示す．

```
     iconst_1          (1)
     istore_0          (1)
L8:  iconst_1          (2)
     iload_0           (2)
     if_icmpne L2      (2)
     iconst_2          (3)
     iload_0           (3)
     if_icmplt L3      (3)
     iconst_3          (4)
     goto L4           (4)
L3:  iconst_2          (5)
L4:  istore_1          (6)
     goto L5           (6)
L2:  iload_0           (7)
     iconst_3          (7)
     iadd              (7)
     istore_1          (7)
L5:  iload_0           (8)
     istore_1          (8)
L7:  iload_1           (9)
     iconst_2          (9)
     if_icmpgt L6      (9)
     iload_1          (10)
     istore_0         (10)
L6:  iload_0          (11)
     iconst_1         (11)
     if_icmpne L7     (11)
     goto L8          (12)
```

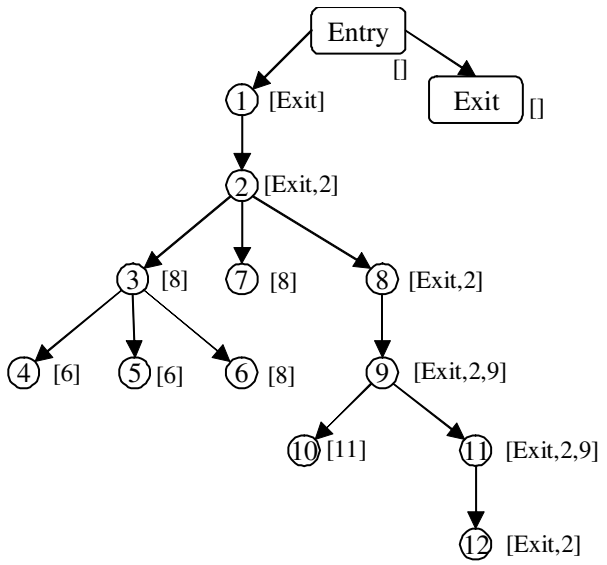


サンプルコード　　構築される制御フローグラフ

図 10: 制御フローグラフの例



(a):制御フローグラフ



(b):(a) に対する支配木 ([ ] 内は Dominance Frontier)

図 11: 制御フローグラフと支配木

本研究では，バイトコードにおける制御依存関係を次のように定義する．

バイトコードにおける制御依存関係

バイトコードに対して構築された制御フローグラフの節点 X に属する命令 s，節点 Y に属する命令 t に関して，以下の条件を満たすとき，s から t の間に制御依存関係が存在するという．

1. s は節点 X の最終命令であり，分岐命令である．
2. 節点 X から U と V への分岐があり，U から出口へ Y を通らないパスがあり，V から出口へのパスが必ず Y を通る．

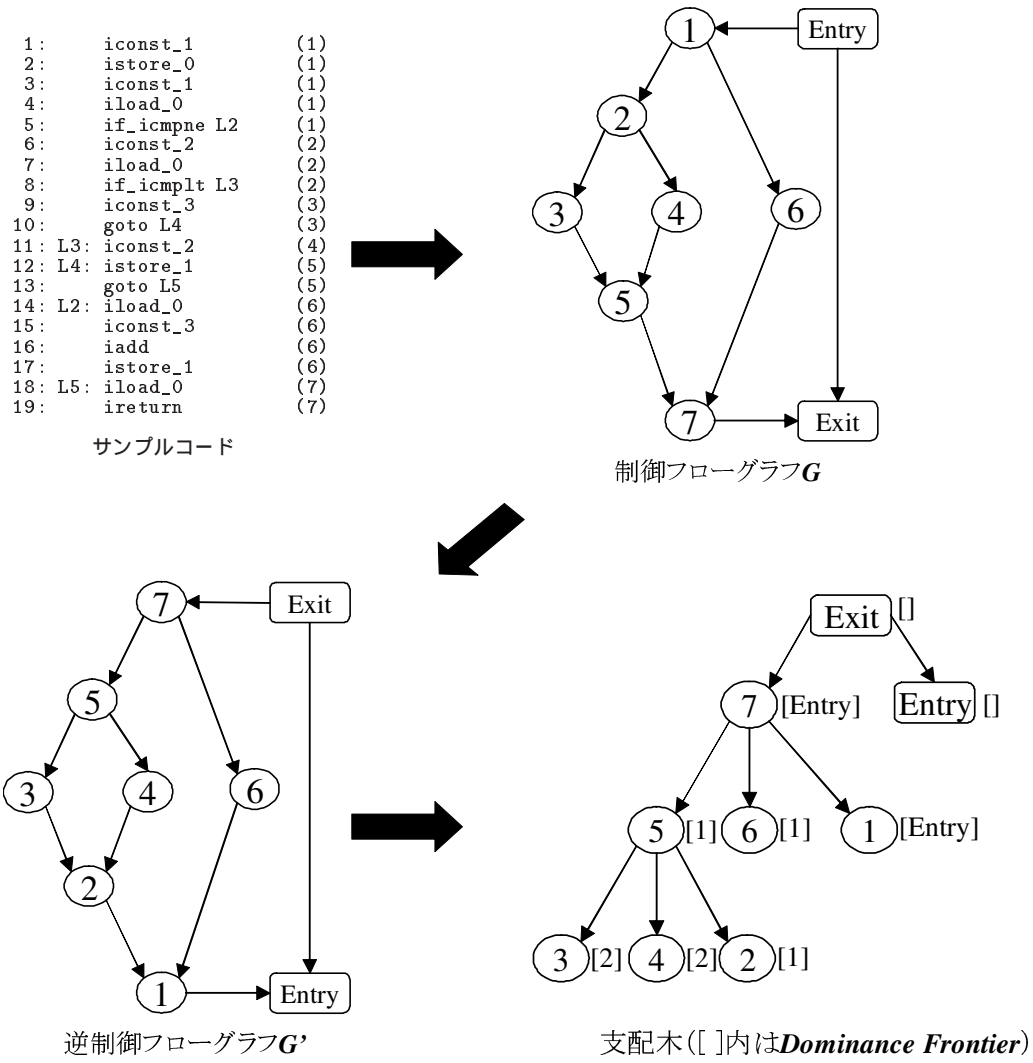
このように定義したバイトコードにおける制御依存関係は，図 12 に示すアルゴリズムにより抽出できる．なお．このアルゴリズムはメソッド単位に適用する．

入力 バイトコード
出力 命令間に存在する制御依存関係
処理 バイトコードにおける静的制御依存関係を抽出する

(1) バイトコードを基本ブロックに分割し，制御フローグラフ G を構築する
(2) G に入口節点 R，出口節点 E を追加し，R から G の最初の節点へ，G の最後の節点から E へ，R から E にそれぞれ辺を追加する
(3) G に対し，逆制御フローグラフ G' を構築する ($\mathcal{N}$: G' の節点集合)
(4) G' に対し，支配木を構築する (根は G の出口節点となる)
(5) **foreach** x **in** $\mathcal{N}$ **begin**
(6) Dominance Frontier $DF_G'[x]$ を計算する
(7) **foreach** y **in** $DF_G'[x]$
(8) x の最終命令と y の各命令の対を制御依存関係として抽出する
(9) **end**

図 12: バイトコードにおける静的制御依存関係解析アルゴリズム

このアルゴリズムを適用することによって抽出される制御依存関係の例を図 13 に示す．



制御依存関係のある命令の組	
(5: if_icmpne L2 , 6: iconst_2)	(5: if_icmpne L2 , 7: iload_0)
(5: if_icmpne L2 , 8: if_icmplt L3)	(5: if_icmpne L2 , 12: istore_1)
(5: if_icmpne L2 , 13: goto L5)	(5: if_icmpne L2 , 13: goto L5)
(5: if_icmpne L2 , 14: iload_0)	(5: if_icmpne L2 , 15: iconst_3)
(5: if_icmpne L2 , 16: iadd)	(5: if_icmpne L2 , 17: istore_1)
(5: if_icmpne L2 , 17: istore_1)	(8: if_icmplt L3 , 9: iconst_3)
(8: if_icmplt L3 , 10: goto L4)	(8: if_icmplt L3 , 11: iconst_2)

図 13: バイトコードに対して抽出される制御依存関係

3.2.2 動的データ依存関係解析

Phase 1(b) では，JVM 上でバイトコードを実行し，それと並行してバイトコードに対する動的データ依存関係解析を行う．ソースコードに対する DC スライスの計算においては，各変数に対してキャッシュを用意した．バイトコードにおいてもそれと同様に，メソッド内のローカル変数やインスタンスのメンバ変数などのデータ領域それぞれに対してキャッシュを用意する [9]．ただし，JVM にはスタックを介した演算が存在するため，スタックをデータ領域とみなし，それに対するキャッシュも用意する．

そして，各データ領域に関して，値が参照された場合にはキャッシュに保存されている命令と実行中の命令間に発生したデータ依存関係を抽出し，値が定義された場合にはキャッシュの内容を更新する．なお，あるデータ領域の値が定義されたとき，それに対応するキャッシュが存在しない場合には新たにキャッシュを生成する．

以上の方針に基づき定義された，バイトコードにおける動的データ依存関係解析アルゴリズムを図 14 に示す．このアルゴリズムは，JVM 上でバイトコードの一命令が実行されるたびに適用される．本手法では，同一クラスから生成された複数のインスタンスはそれぞれ独立にキャッシュを保持しており，各インスタンス独立にデータ依存関係解析が行われることになる．

入力 バイトコードの命令 s
出力 s の実行により発生するデータ依存関係
処理 バイトコードにおける動的データ依存関係を抽出する

(1) **foreach** n **in** s で参照される変数
(2) n のキャッシュに保持されている命令と s の対をデータ依存関係として抽出する
(3) **foreach** n **in** s で定義される変数 **begin**
(4) **if** n のキャッシュがなければ **then**
(5) n のキャッシュを生成する
(6) n のキャッシュを s に更新
(7) **end**

図 14: バイトコードにおける動的データ依存関係解析アルゴリズム

例として，図 15 のようなバイトコードに対して動的データ依存関係解析を行う場合を考

える．図 15 に示すバイトコードを実行させたときの各実行時点における各データ領域 (スタック，ローカル変数，フィールド) のキャッシュ$C(v)$の推移を表 3 に示す．ただし，表 3 においては，スタック [0] が底，スタック [1] がスタックトップを表す．

```
.method public ddd()I
   .limit stack 2
   .limit locals 1
 1: iconst_3
 2: istore_0
 3: iconst_2
 4: putstatic A.value
 5: iload_0
 6: iload_0
 7: iadd
 8: ireturn
.end method
```

図 15: 動的データ依存関係解析を行うバイトコード

表 3: 図 15 におけるキャッシュの推移

実行命令	スタック [0]	スタック [1]	ローカル変数 [0]	A.value
1	1	-	-	-
2	-	-	2	-
3	3	-	2	-
4	-	-	2	4
5	5	-	2	4
6	5	6	2	4
7	7	-	2	4
8	-	-	2	4

命令 1 では，スタックに定数 3 が積まれるため，$C($ スタック $[0]) = 1$ となる．命令 2 の実行においては，ローカル変数 [0] の値が定義され，$C($ ローカル変数 $[0]) = 2$ となると同時に，スタック [0] の値が参照される．つまり，命令 2 の実行時に命令 $C($ スタック $[0])$，つまり命令 1 で定義した値が参照される．そのため，命令 1 と命令 2 の間にデータ依存関係が発生することとなる．以降，このようにデータ依存関係解析を続けることによって，表 4 に示すデータ依存関係が抽出される．

表 4: 図 15 のプログラムの実行において抽出されるデータ依存関係

データ依存関係が発生する命令の組
(1: iconst_3 , 2: istore_0)
(3: iconst_2 , 4: putstatic A.value)
(2: istore_0 , 5: iload_0)
(2: istore_0 , 6: iload_0)
(5: iload_0 , 7: iadd)
(6: iload_0 , 7: iadd)
(7: iadd , 8: ireturn)

3.2.3 PDG の構築

静的制御依存関係解析，動的データ依存関係解析により抽出された依存関係を用いて PDG を構築する．構築される PDG の例を図 16 に示す．PDG の節点はバイトコードの各命令であるため，実行系列を保存する必要はなく，動的スライスに比べ解析コストは十分に小さくなる．

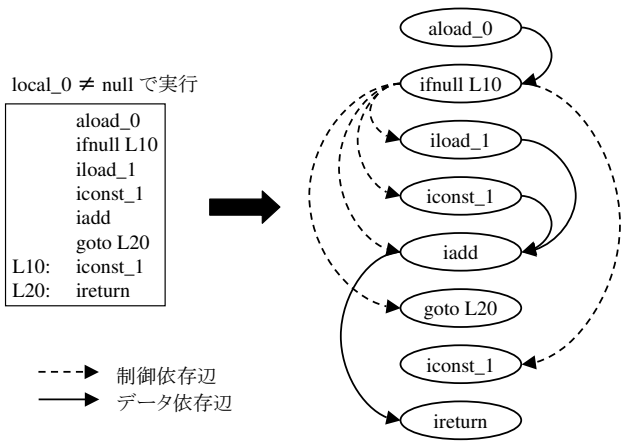


図 16: バイトコードにおけるプログラム依存グラフ

3.2.4 スライス計算

PDG を用いてスライス計算を行う．バイトコードに対するスライス計算も従来手法と同じく，スライス基準に対応する PDG 節点から PDG 辺を逆に辿り，到達可能な節点集合を求めることによる．

たとえば，図 16 の PDG において、命令 ireturn に対応する節点をスライス基準としてバイトコードにおけるスライス計算を行うと，図 17 で斜線で示された節点が到達可能であり，バイトコードに対して計算されたスライス結果となる．

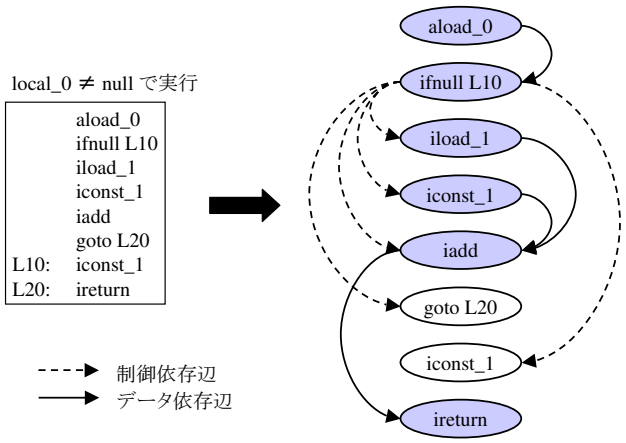


図 17: バイトコードにおける PDG によるスライス計算

3.3 バイトコードとソースコードの対応

バイトコード・ソースコード間の変換は，コンパイラによって出力される対応表を参照することにより行う．

バイトコードとソースコードの対応表の例を図 18 に示す．この対応表を用いることにより，ソースコード上で指定されたスライス基準をバイトコードに変換し，PDG 探索を行い，バイトコードでのスライスを計算する．そして再び対応表を用いることで計算されたスライスをソースコードに対応付ける．

例として，図 18 のソースコードを実行させ，スライスを計算する過程を図 19 に示す．まず，バイトコードに対する静的制御依存関係解析・動的データ依存関係解析により PDG が構築される．次に，ソースコード上で指定されたスライス基準を，図 18 の対応表を参照することでバイトコードに変換し，対応する PDG 節点を求める．求められた PDG 節点からグラフ探索を行うことで，バイトコードにおけるスライスが計算される．最後に，計算されたバイトコードでのスライスを，再び対応表を参照することでソースコードに対応付ける．

```
int i = 2;
if (i > 2){
   i++;
} else {
   i--;
}
int j = i;
```

バイトコード	対応するトークン集合
iconst_2	"2"
istore_1	"i ="
iload_1	"i"
iconst_2	"2"
if_icmple L13	">"
iinc 1 1	"i++"
goto L18	"if"
L13: nop	""
nop	"if"
iinc 1 -1	"i−−"
L18: nop	"if"
iload_1	"i"
istore_2	"j ="
return	"main"

図 18: バイトコードとソースコードの対応表

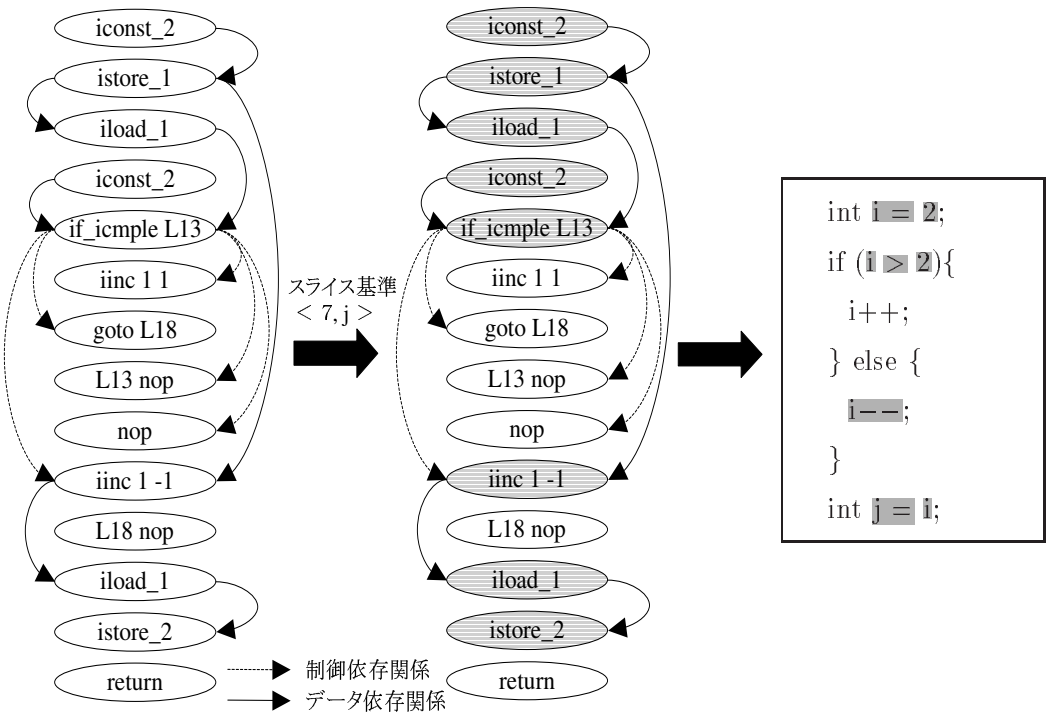


図 19: 構築される PDG とスライス計算

4　考察

4.1　関連研究

Java プログラムに対する動的解析に関して，本研究以外にもいくつか研究がなされている．ここでは，その中の 2 つの関連研究を挙げる．

4.1.1　バイトコードの埋め込み方式による動的解析

BIT (Bytecode Instrumenting Tool) [11] は，バイトコードに，それ自身を解析するためのバイトコード命令を挿入するライブラリ群である．BIT のアーキテクチャを図 20 に示す．

ユーザは，解析対象となるバイトコードから，必要な動的情報を抽出するためのコードを，BIT が提供するライブラリ群を用いて記述し，解析対象のバイトコードに，それ自身の解析用命令を挿入する．そのようにして生成したバイトコードを通常の JVM で実行することにより，必要な動的情報が出力されることになる．

BIT を用いるにあたり，ユーザは，挿入すべきバイトコードを自ら記述する必要があり，バイトコードについて，特別な知識を要求される．また，結果として得られる動的情報としては，実行命令のトレースや分岐の成功比率など，統計データの計測を目的とするものが多く，依存関係のような複雑な計算を必要とする情報の抽出は困難である．

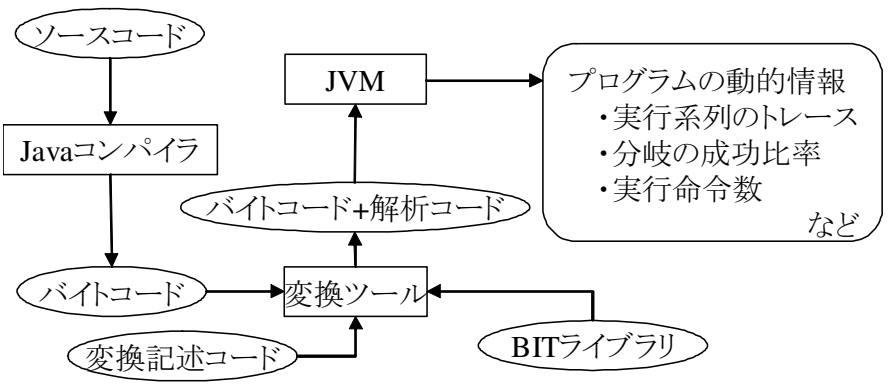


図 20: BIT アーキテクチャ

4.1.2　ソースコードの埋め込み方式による動的解析

広瀬ら [13] は，本研究と同様，Java に対して DC スライスを計算する手法を提案している．そのアーキテクチャを図 21 に示す．

[13] では，まず，プリプロセッサにより，ソースコードに対して，それ自身を解析するた

めのソースコードを挿入する．プリプロセッサを経たのち，通常と同様にコンパイルおよび実行を行うことで，ソースコードの各文を節点とした，実行時に発生する動的データ依存関係に基づく PDG が構築される．このようにして構築された PDG を用いて DC スライスを計算する．

DC スライスの計算を行うにあたり，ユーザは特別な知識を必要としないが，PDG を構築する際に，ソースコードの埋め込みによる動的依存関係解析の実現を行っているため，Java の構文制約による埋め込み位置の制限が厳しく，適切な解析コードの埋め込みが困難であり，結果として十分な解析精度が得られない問題がある．また，ソースコードの変更を前提とするため，ソースコードの存在しないクラスを介した依存関係の解析が不可能である．その結果，ソースコードの存在しないクラス，例えば JDK ライブラリを介したスライス計算を行う場合，極端に解析程度が低下してしまう．

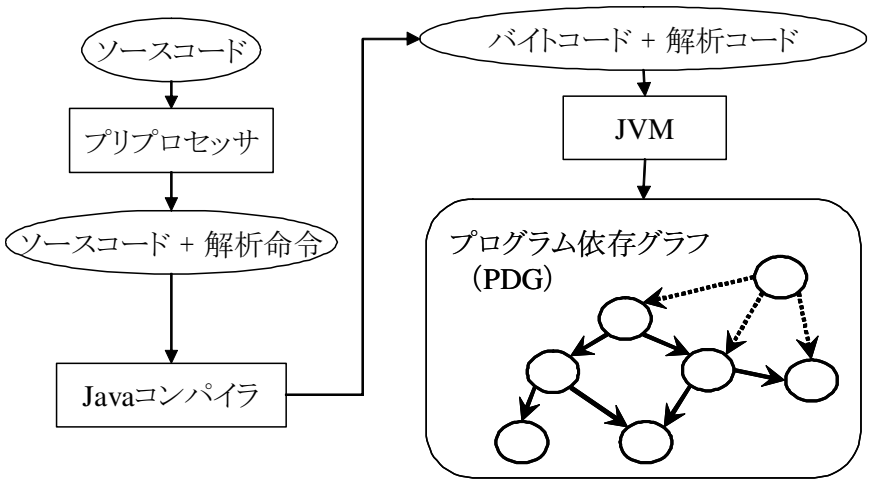


図 21: ソースコードの埋め込みによる DC スライス計算

4.2 提案手法の有効性

本研究で提案した手法では，本来ソースコードに対して行われる依存関係解析をバイトコードに対して行い PDG を構築する．そして，バイトコードに対して計算された DC スライスを，コンパイラが生成する対応表をもとにソースコードに反映させる．

提案手法を用いることにより，既存の研究に比べ，以下の点において有効であると考えられる．

- 解析精度の向上
  コードの埋め込みによる実現では，構文制約の影響を受け，解析命令の埋め込みに限

界があり，そのため解析精度の低下を引き起こすことがある．しかし提案手法では，コードの埋め込みを行わないため，解析精度の低下を防ぐことができる．また，ソースコードの存在を前提としないため，クラスファイルのみ存在するようなクラスを介した依存関係も解析可能となる．これにより JDK ライブラリなど利用した多くのプログラムに対しても精度を保ったままスライス計算ができる．

- 細粒度の解析が可能
  一般にソースコードの各文は複数のバイトコードの命令へとコンパイルされる．本研究においては，PDG 節点をソースコードの文単位ではなく，バイトコードの命令単位としたことにより，細粒度の解析結果を得ることができる．

- ユーザへの負担が軽い
  ユーザは DC スライスの計算にあたり，本システムについて特別な知識を必要とせず，通常のコンパイル，実行を行うだけで DC スライス計算に必要な情報を取得することができる．

5　提案手法の実現

5.1　システム構成

本研究では，提案した手法のシステムとしての実装を行った．実現したシステムの構成を図 22 に示す．

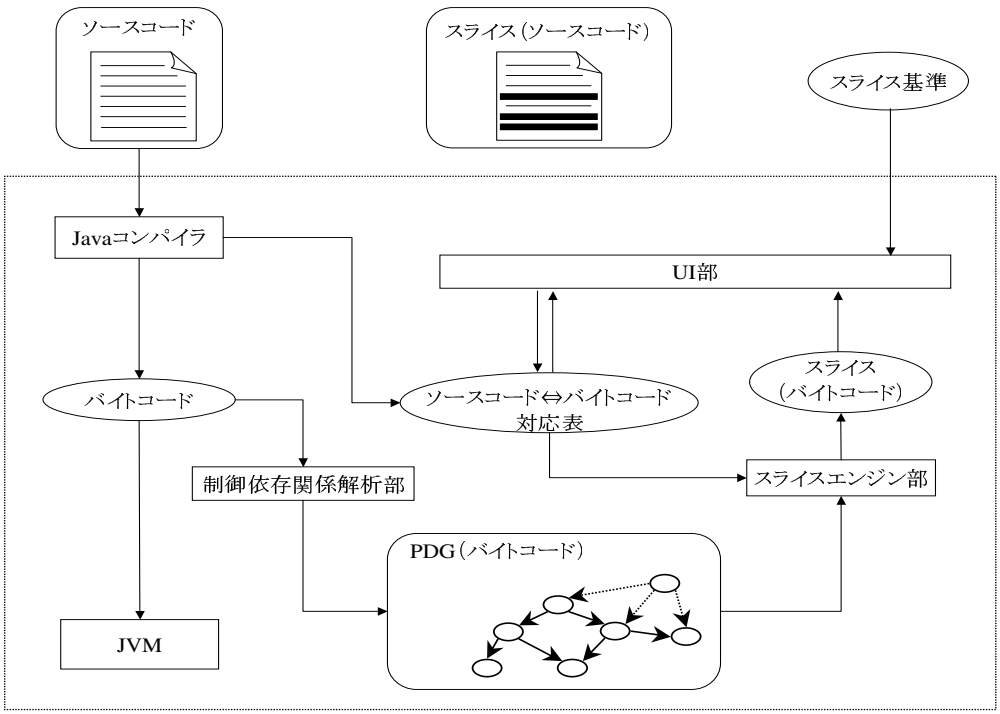


図 22: システム構成

ソースコードのコンパイル時にバイトコードとソースコードの対応表を生成する．そして，静的に制御依存関係解析を行ったのち，JVM 上でバイトコードの実行を行いながら動的にデータ依存関係解析を行う．これらにより抽出された依存関係を元にバイトコードの各命令が節点となる PDG を構築する．ユーザによりソースコードにおけるスライス基準が指定されると，対応表を用いてそれをバイトコードにおけるスライス基準に変換し，PDG 探索よるスライス計算を行う．最後に，対応表を参照しながらスライス結果をソースコードに対応付ける．システムのメインウィンドウを図 23 に示す．以降，各処理部の実装について順に説明する．

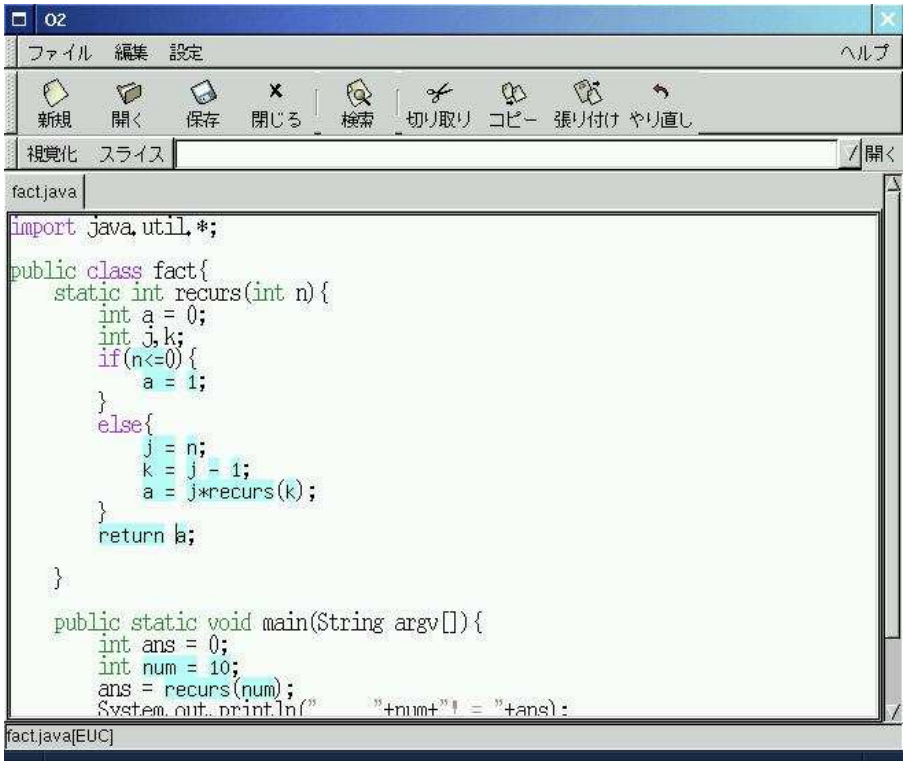


図 23: システムのメインウィンドウ

Java コンパイラ

入力： Java ソースコード

出力： バイトコード・ソースコードのトークン集合とバイトコードの対応表

処理： 入力ファイルを解析し，通常のコンパイラと同様にコンパイルし，クラスファイルを出力する．また，生成されるバイトコードの各命令と，それに対応するソースコードのトークン集合の対応表を出力する．

概要： JDK 付属の javac に対する機能拡張として実装．

実装言語： Java

クラス数： 174 (149)

行数： 約 42,000 行 (約 3,000 行)

( ) 内は今回の機能拡張によって追加・修正されたコードを表す．

データ依存関係解析部 (JVM)

入力： バイトコード

出力： 実行時に動的に発生したバイトコードの命令間のデータ依存関係

処理： 入力されたバイトコードを実行し，通常の JVM と同様の実行結果を出力する．また，実行時に各命令間に発生するデータ依存関係を動的に抽出し，その結果を出力する．

概要： JDK 付属の java に対する機能拡張として実装．

実装言語： C

ファイル数： 約 1200 (12)

行数： 約 520,000 行 (約 1,100 行)

( ) 内は今回の機能拡張によって追加・修正されたコードを表す．

制御依存関係解析部

入力： バイトコード

出力： 静的に解析することによって抽出できるバイトコードの命令間の制御依存関係

処理： 入力されたバイトコードに対し，静的に制御依存関係解析を行い，各命令間に存在するデータ依存関係を静的に抽出し，その結果を出力する．

概要： バイトコードダンプツール jaca[8] に対する機能拡張として実装．

実装言語： C

ファイル数： 30 (19)

行数： 約 4,000 行 (約 1,600 行)

( ) 内は今回の機能拡張によって追加・修正されたコードを表す．

スライスエンジン部

入力： 依存関係解析の結果・ソースコードのトークン集合とバイトコードの対応表

出力： ソースコードにおけるスライス

処理： 入力された依存関係解析の結果からバイトコードでの PDG を構築する．また，ユーザから指定されたスライス基準を，対応表を用いることでバイトコードの命令に変換し，バイトコードにおけるスライスを計算する．計算されたスライスを，再び対応表を用いることでソースコードにおけるスライスに変換する．

実装言語： C

ファイル数： 15

行数： 約 1,200 行

UI 部

入力： ユーザにより定められたスライス基準

出力： 指定されたスライス基準から計算されるスライス

処理： スライスを計算するための操作を総合的に管理する．

- Java ソースコードの読み込み，表示，編集する．
- ユーザから指定されるスライス基準および読み込んだソースコードに対し，外部プログラム (Java コンパイラ・依存関係解析部・スライスエンジン部) を呼び出すことで，求めるスライスを計算する．
- スライスを表示する．

実装言語： C++

ファイル数： 43

クラス数： 18

行数： 約 8,000 行

5.2 評価

前節で述べたシステムを用いて提案手法の有効性を評価した．ここでは，以下の点について比較を行う．

1. 計算されるスライスサイズ
2. 解析コスト (解析時間・解析時使用メモリ量)

スライスサイズについては，静的スライス，動的スライスとの比較を行った．スライス計算を行うシステムの実装は，提案手法のみであるため，提案手法のスライスは実装システムを用いて求め，他の手法のスライスは手計算で求めた．

解析コストは，各処理にかかる時間・使用メモリ量を計測することにより評価を行った．コンパイル・JVM による動的データ依存関係解析については通常のコンパイル・JVM での実行との比較を行い，制御依存関係解析・PDG 構築およびスライス計算については各処理に必要な時間と使用メモリ量を計測することで評価を行った．また，動的スライスの計算コストとの比較を行うため．提案手法で構築される PDG 節点数と，動的スライス計算の際に保存される実行系列に含まれる命令数の比較も行った．

スライス対象プログラムの概要を表 5 に示す．P1 は，ファイルからデータを読み込み，そのデータを操作し，再び保存することの出来る簡易データベースシステムである．また，P2 は 50 個の整数値に対し，バブルソート・双方向バブルソート・クイックソートを適用するプログラムである．

表 5: スライス対象プログラム

プログラム	クラス数	オーバーライドメソッド数	総行数
P1	4	0	262
P2	5	3	231

5.2.1 スライスサイズ

Java を対象とした静的スライス，動的スライスによって得られたスライスと提案手法によって得られたスライスのサイズの比較を行った．各手法ともに，計算の結果スライスに含まれるべき文含まれないということはないため，得られたスライスのサイズが小さいほど精度が高い (スライスに含まれるべきではない文がスライスに含まれる割合が小さい) といえる．

計測値は，各プログラムに対し，任意に定めた 2 つのスライス基準について，それぞれの手法で得られたスライスのサイズである．実装は，本手法のみ行ったため，提案手法のスライスは実装したシステムを用いて求め，他の手法のスライスは手計算で求めた．計測値を表 6 に示す．

表 6: スライスサイズ [行] (全体に対するスライスの割合)

	静的スライス	動的スライス	提案手法
P1-スライス基準 1	60 (22.9%)	24 (10.3%)	30 (11.4%)
P1-スライス基準 2	19 (7.3%)	14 (5.4%)	15 (5.7%)
P2-スライス基準 1	79 (34.2%)	51 (22.1%)	51 (22.1%)
P2-スライス基準 2	27 (11.7%)	23 (10.0%)	25 (10.8%)

提案手法で得られるスライスは，静的スライスの約 50%から 93%のサイズであり，静的スライスより高精度なスライスを得られることがわかった．今回の実験は，スライス対象プログラムが比較的小規模であったため，提案手法によるスライスと静的スライスの差は少なかったと考えられる．

しかし，クラスの継承やメソッドのオーバーライド，オーバーロードがより多く含まれる大規模プログラムでは，静的スライスの精度がより低下することが推測できる．それに対し，提案手法では，継承やオーバーライド，オーバーロードの数に関わらず，それによる精度の

低下は発生しない．また，今回の実験においては，提案手法では動的スライスとほぼ同等のスライスが得られた．

5.2.2 解析コスト

各処理について，処理時間および使用メモリ量を計測することでスライス計算に必要なコストの評価を行った．実行環境を表 7 に示す．

表 7: 実験環境

CPU	Pentium 4 1.5GHz
メモリ	512MB
OS	FreeBSD 5.0-CURRENT
Java	JDK 1.2.2

以降，それぞれの処理について，計測したコスト値を順に示す．また，各計測値は，実行を 10 回行った際の平均値である．

Java コンパイラ・JVM

処理時間・使用メモリ量の評価を，通常のコンパイラおよび JVM との比較により行った．計測結果を表 8・表 9・表 10・表 11 に示す．

表 8: コンパイル時間

プログラム	通常 [ms]	対応表出力 [ms]	対応表出力/通常
P1	1,114	2,405	2.16
P2	974	1,325	1.36

表 9: コンパイル時の使用メモリ量

プログラム	通常 [Kbytes]	対応表出力 [Kbytes]	対応表出力/通常
P1	11,490	11,824	1.03
P2	10,394	11,792	1.14

表 10: JVM 実行時間

プログラム	通常 [ms]	動的 DD 関係解析 [ms]	動的 DD 関係解析/通常
P1	325	2,058	6.33
P2	341	3,089	9.06

表 11: JVM 実行時の使用メモリ量

プログラム	通常 [Kbytes]	動的 DD 関係解析 [Kbytes]	動的 DD 関係解析/通常
P1	3,780	15,980	4.22
P2	4,178	26,091	6.24

制御依存関係解析

処理時間・使用メモリ量の評価を，表 5 に示したスライス対象プログラムおよび約 4,800 個のクラスからなる JDK ライブラリを解析した際の解析時間と使用メモリ量の計測によって行った．計測値を表 12・表 13 に示す．

表 12: 静的制御依存関係解析時間

プログラム	総クラス数	解析時間（全体） [ms]	平均解析時間（一クラス） [ms]
P1	4	25.79	6.45
P2	5	24.56	4.92
JDK ライブラリ	4,807	48,060	9.99

表 13: 静的制御依存関係解析時の使用メモリ量

クラス名	サイズ [bytes]	使用メモリ量 [Kbytes]
java.security.Principal.class	264	124
sun.io.CharToByteCp866.class	7,269	181
sun.io.CharToByteCp933.class	193,709	2,779

PDG 構築・スライス計算

処理時間・使用メモリ量の評価を，表 5 に示したスライス対象プログラムに対し，あるスライス基準からのスライス計算時の PDG 構築・スライス計算 (PDG 探索) 時間の計測によって行った．計測結果を表 14・表 15 に示す．

表 14: PDG 構築・スライス計算時間

プログラム	PDG 節点数	PDG 構築時間 [ms]	スライス計算時間 [ms]	合計時間 [ms]
P1	34,966	346	179	525
P2	34,956	352	98	450

表 15: PDG 構築・スライス計算時の使用メモリ量

プログラム	PDG 節点数	使用メモリ量 [Kbytes]
P1	34,966	5,426
P2	34,956	4,047

PDG 節点数の評価

動的スライスを計算する場合のコストとの比較を行うために，表 5 に示したスライス対象プログラムに対して，提案手法によって構築された PDG の節点数と，動的スライス計算の際に保存される実行系列に含まれる命令数を計測した．計測結果を表 16 に示す．

表 16: 構築される PDG の節点数と解析したクラス数

プログラム	提案手法	動的スライス	提案手法/動的スライス	解析したクラス数
P1	34,966	1,198,596	0.0292	147
P2	34,956	1,808,051	0.0193	148

以上より，本システムは全く解析を行わずに通常実行を行う場合と比較し，多くの解析コストを要するといえる．特に，動的データ依存関係解析を行う JVM の実行については，実行時間においておよそ 6 倍から 9 倍，使用メモリ量についておよそ 4 倍から 6 倍を要してい

る．これは，ソースコードの文単位ではなく，バイトコードの命令単位による細粒度の解析を実現していることと，ユーザが入力として与えたコードだけではなく，実行されるJDKライブラリ全体に対してデータ依存関係解析を行っていることによるものだと考えられる．また，コンパイラにおいてソースコードの対応関係を容易に取得できるように，バイトコードの最適化を一切行っていないことも原因の一つである．

しかし，動的スライスの計算には，節点数の保存のためにおよそ30倍から50倍のコストが必要となり，5.2.1.で行った実験においてほぼ同等のスライス結果が得られたことを考慮すると，提案手法を用いることで動的スライスより格段に小さな解析コストで，静的スライスより高い精度のスライスの計算が可能となるといえる．

提案手法の今後の課題として，解析速度の点について改良を行い，より低コストでスライスの計算を行うことを挙げることができる．しかし現段階で実用的に動作するスライスシステムは本システム以外にはなく，その点においても提案手法を実現した本システムは非常に有用であるといえる．

6　むすび

本研究では，オブジェクト指向言語 Java で記述されたプログラムに対し，バイトコード単位での動的データ依存関係解析・静的制御依存関係解析に基づいた DC スライス計算を行う手法を提案した．

提案手法は，本来ソースコードに対して行われていた依存関係解析をバイトコードに対して適用した．データ依存関係解析を動的に行うことにより，静的スライスに比べ精度の高いスライスを得ることが可能となった．一方，制御依存関係解析を静的に行うことにより，動的スライスで見られるような実行系列の保存による解析コストの増大を抑えることができた．

また，提案手法をスライス計算ツールとして実装し，その有効性を確認した．

今後の課題としては，以下がある．

- バイトコードの最適化に伴うソースコードとの対応表の整合性の維持
- JVM での動的データ依存関係解析の高速化
- native メソッド実行時の動的データ依存関係解析の実現
- 実行時に決定される制御フローへの動的解析の適用（メソッド内サブルーチンなど）

Java では，例外処理など，制御フローが動的にしか決定しないことがある．そのような場合に発生する制御依存関係については動的に解析を行うことで，さらに精度の高い解析が実現できると考えている．

謝辞

本研究の全過程を通して，常に適切な御指導および御助言を賜りました 大阪大学大学院 基礎工学研究科 情報数理系専攻 井上 克郎 教授に心より深く感謝致します．

本論文を作成するにあたり，逐次適切な御指導および御助言を賜りました 大阪大学大学院 基礎工学研究科 情報数理系専攻 楠本 真二 助教授に心から感謝致します．

本論文を作成するにあたり，適切な御指導，御助言を賜りました 大阪大学大学院 基礎工学研究科 情報数理系専攻 松下 誠 助手に心から感謝致します．

本研究を通して，適切な御助言を頂きました 大阪大学大学院 基礎工学研究科 情報数理系専攻 大畑 文明 氏に深く感謝致します．

本研究において，御協力を頂きました 大阪大学 基礎工学部 情報科学科 梅森 文彰 氏に深く感謝致します．

最後に，その他様々な御指導，御助言等を頂いた 大阪大学大学院 基礎工学研究科 情報数英系専攻 井上研究室の皆様に深く感謝いたします．

参考文献

[1] H.Agrawal and J.Horgan: “Dynamic Program Slicing”, SIGPLAN Notices, Vol.25, No.6, pp.246–256 (1990).

[2] A.V.Aho , R.Sethi , J.D.Ullman: ”Compilers Principles,Techniques, and Tools”, Addison-Wesley Publishing Company(1986) .

[3] A.W.Appel and M.Ginsburg: “Modern Compiler Implementation in C”, Cambridge University Press, Cambridge (1998).

[4] Y.Ashida, F.Ohata and K.Inoue: “Slicing Methods Using Static and Dynamic Information”, Proceedings of the 6th Asia Pacific Software Engineering Conference , pp.344–350, Takamatsu, Japan, December (1999).

[5] R.Cytron, J.Ferrante, B.K.Rosen, M.N.Wegman and F.K.Zadeck: “Efficiently Computing Static Single Assignment Form and the Control Dependench Graph”, ACM Transactions on Programming Languages and Systems, Vol.13, No.4, pp.461–486, October (1991).

[6] D.Harel: “A liner time algorithm for finding dominator in flow graphs and related problems”, Proceedings of 17th ACM Symposium on Theory of computing, pp.185–194, May (1985).

[7] K.Inoue, F.Ohata and Y.Ashida: “Lightweight Semi-Dynamic Methods for Efficient and Effective Program Slicing”, Technical Report of Osaka University, Department of Information and Computer Scineces, Inoue Laboratory, June (2000).

[8] E.Kawai: jaca, “http://minatow3.aist-nara.ac.jp/oie-lab/person/eiji-ka/research/my_jvm/jaca/”.

[9] 誉田 謙二, 大畑 文明, 井上 克郎: “Java バイトコードにおけるデータ依存解析手法の提案と実現”, コンピュータソフトウェア, Vol.18, No.3, pp.40–44, (2001).

[10] L.Larsen and M.J.Harrold: “Slicing Object-Oriented Software”, Proceedings of the 18th International Conference on Software Engineering, pp.495–505, Berlin, March (1996).

[11] H.B.Lee and B.G.Zorn: “BIT:A Tool for Instrumenting Java Bytecodes”, Proceedings of the USENIX Symposium on Internet Technologies and Systems, pp.73–83, Montray, California, December (1997).

[12] T.Lengauer and E.Tarjan: “A fast algorithm for finding dominators in a flow graph”, ACM Transactions on Programming Languages and Systems, Vol.1, No.1, pp.121–141, July (1979).

[13] F.Ohata, K.Hirose and K.Inoue: “A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information”, Proceedings of 8th Asia Pacific Software Engineering Conference , pp.273–280, Macau, China, December (2001).

[14] K.J.Ottenstein and L.M.Ottenstein: “The program dependence graph in a software development environment”, Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp.177–184, Pittsburgh, Pennsylvania, April (1984).

[15] B.Steensgaard: “Points-to analysis in almost linear time”, Technical Report MSR-TR-95-08, Microsoft Research (1995).

[16] 高田 智規, 井上 克郎, 大畑 文明, 芦田 佳行: “制限された動的情報を用いたプログラムスライシング手法の提案”, 電子情報通信学会論文誌 D-I(採録決定).

[17] R.Ueda, K.Inoue and H.Iida: “A Practical Slice Algorithm for Recursive Programs”, Proceedings of the International Symposium on Software Engineering for the Next Generation, pp.96–106, Nagoya, Japan, February (1996).

[18] M.Weiser: “Program Slicing”, Proceedings of the 5th International Conference on Software Engineering, pp. 439–449 (1981).

[19] J.Zhao: “Dynamic Slicing of Object-Oriented Programs”, Technical Report SE-98-119, Information Processing Society of Japan , pp.11–23, May (1998).