

## 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

高田 智規<sup>†</sup> 井上 克郎<sup>††</sup> 大畑 文明<sup>††</sup> 芦田 佳行<sup>†††</sup>

A Program Slicing Method Using Lightweight Dynamic Information

Tomonori TAKADA<sup>†</sup>, Katsuro INOUE<sup>††</sup>, Fumiaki OHATA<sup>††</sup>, and Yoshiyuki ASHIDA<sup>†††</sup>

あらし

大規模なプログラムのデバッグを行なう際、デバッグの対象となるソースプログラム全体からバグがあると思われる部分を小さな範囲に限定することができれば作業効率が向上する。プログラムから注目した一部分を抽出する技術としてプログラムスライシング技術があるが、静的スライシングはプログラムから抽出したスライスが大きくなり、動的スライシングは実行オーバーヘッドが大きくなるという問題がある。本稿では、静的手法と動的手法との中間に位置する、依存キャッシュスライシング手法の提案を行なう。また、このアルゴリズムを実装し、サンプルプログラムを用いて実行データの収集を行なった。この結果、配列やポインタを使用したプログラムに対し、依存キャッシュスライシングは既存手法と比較して約 30-90%のスライスサイズ削減を得ることができ、要したオーバーヘッドも現実的な値であった。この手法は、効果的にデバッグを行なう上で有効と考えられる。

キーワード 静的スライス、動的スライス、実行時オーバーヘッド、依存キャッシュスライシング

## 1. はじめに

ソフトウェアのテストや保守工程において、ソースプログラム中に存在するフォールト位置の特定は非常に時間を要する作業である。このとき、ソースプログラム全体を見るのではなくフォールトに関連する部分のみ着目することができれば、作業効率を改善することができる。このようなフォールト位置特定を行う手法の一つに、プログラムスライシング (*Program Slicing*, 以降、スライシングと略す)[12]がある。プログラムスライス (*Program Slice*, 以降、スライスと略す)とは、直観的には、関心のある文に含まれる変数に影響を与える文の集合を指す。作業者はスライスに含まれる文のみに着目すればよく、デバッグを効率的に行うことができる [9]。

Weiser による研究 [12] をきっかけに、スライシングに関する多くの研究と応用がなされている。

スライシング技術は大きく静的スライシング (*Static Slicing*) と動的スライシング (*Dynamic Slicing*) の 2 つに分類される。

Weiser [12] によって提案された静的スライス (*Static Slice*) は、特定のプログラム文中のある変数の値に影響を与える可能性のある文の集合である。静的スライスは一般にサイズが大きく、極端な場合、ソースプログラム全体がスライスとして抽出される。これは静的スライシングが、すべての入力データ、つまりすべての実行経路パターンを想定しているためである。また、静的スライシングにおいて、変数名の別名 (*Aliasing*) の判定、配列および構造体要素の区別、ポインタ変数の参照先の追跡などの解析は容易ではない。さらにオブジェクト指向プログラムでは、識別子に対応するメソッドがプログラム実行時に決定される点も無視できない。

Agrawal ら [1], [6] によって提案された動的スライス (*Dynamic Slice*) は、注目した文中の変数の値に実際に影響を与えた実行文の集合である。動的スライスは特定の入力データに基づいて導出されるため、ソースプログラムのうち実行されなかった部分は自動的に除かれる。このため、スライスサイズは静的スライスと比べ一般的に小さくなり、フォールト位置特定には好

<sup>†</sup> 西日本電信電話株式会社, 大阪市

Nippon Telegraph and Telephone West Corporation, Osaka-shi, 550-0001, Japan

<sup>††</sup> 大阪大学 大学院基礎工学研究科, 豊中市

Graduate School of Engineering Science, Osaka University, Toyonaka-shi, 560-8531 Japan

<sup>†††</sup> NTT ソフトウェア株式会社, 横浜市

NTT Software Corporation, Yokohama-shi, 231-8554, Japan

```

1  program Square_Cube(input, output);
2  var a, b, c, d : integer;
3  function Square(x : integer) : integer;
4  begin
5    Square := x * x
6  end;
7  function Cube(x : integer) : integer;
8  begin
9    Cube := x * x * x
10 end;
11 begin
12  writeln("Squared Value ?");
13  readln(a);
14  writeln("Cubed Value ?");
15  readln(b);
16  writeln("Select Feature! Square: 0 Cube: 1");
17  readln(c);
18  if c = 0 then
19    d := Square(a)
20 else
21    d := Cube(b);
22  if d < 0 then
23    d := -1 * d;
24  writeln(d)
25 end.

```

図1 サンプルプログラム  
Fig.1 Sample program

```

1  program Square_Cube(input, output);
2  var a, b, c, d : integer;
3  function Square(x : integer) : integer;
4  begin
5    Square := x * x
6  end;
7  function Cube(x : integer) : integer;
8  begin
9    Cube := x * x * x
10 end;
11 begin
12
13  readln(a);
14
15  readln(b);
16
17  readln(c);
18  if c = 0 then
19    d := Square(a)
20 else
21    d := Cube(b);
22  if d < 0 then
23    d := -1 * d;
24  writeln(d)
25 end.

```

図2 静的スライシング基準 (24, d) に対する静的スライス  
Fig.2 Static slice for slicing criteria (24, d)

ましい。しかし、動的スライシングでは動的にプログラム文間の依存関係を追跡する必要があるため、多くのメモリや時間を必要とする。

本稿では、静的スライシングと動的スライシングの中間に位置する、準動的 (Semi-Dynamic) 手法の一つを提案する。以降、2.節では、静的・動的スライシングについての簡単な説明および既存の準動的手法の分類を行う。3.節では、新たな準動的手法である依存キャッシュスライシングの提案を行う。Osaka Slicing System を用いた提案手法の実現および評価実験を4.節で示し、5.節で検証結果の考察と関連研究について述べる。最後に、6.節でまとめと今後の課題について述べる。

## 2. スライシング手法の分類

本節では、静的スライシングと動的スライシングに関し、その抽出過程を中心に述べ、その具体例として、図1のプログラムに対する静的スライスおよび動的スライスを、図2、図3にそれぞれ示す。また、これらの中間に位置する準動的手法がいくつか提案されており、それらの紹介および分類を行う。

### 2.1 静的スライシング および その分類

ソースプログラム  $p$  中の文  $s_1$  と  $s_2$  について考える。

$s_1$  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決定されるとき、文  $s_1$  から  $s_2$  に対し制御依存 (Control Dependence, CD) があるといい、 $s_1 \dashrightarrow s_2$  と記述する。

$s_1$  から  $s_2$  へ変数  $v$  を再定義しない実行経路が少なくとも1つ存在し、 $s_1$  において  $v$  が定義され、 $s_2$  において  $v$  が参照されるとき、文  $s_1$  から  $s_2$  に対し変数  $v$  に関するデータ依存 (Data Dependence, DD) があるといい、 $s_1 \xrightarrow{v} s_2$  と記述する。

プログラム依存グラフ (Program Dependence Graph, PDG) は辺が文間の依存関係を表し、節点が制御文・代入文などの文を表す有向グラフである。

文  $s$  における変数  $v$  に関する静的スライスとは、文  $s$  に対応する PDG 節点から PDG 辺を逆向きに (制御依存辺は無条件に、データ依存辺は最初は着目している変数名に対応するもののみで以降無条件に) 辿ることで到達可能な節点集合に対応する文の集合である。なお、組  $(s, v)$  を静的スライシング基準 (Static Slicing Criteria) と呼ぶ。

静的スライシングの解析手法は、以下のように分類できる。

- 制御フロー解析の有無 (Flow Sensitiveness / Insensitiveness)

- 関数・手続き解析時に実際の引数・大域変数情報を用いるかどうか (Context Sensitiveness / Insensitivity)

本稿では、特に明示しなければ、制御フロー解析を行ないかつ関数・手続き解析時に実際の引数・大域変数情報を用いない (呼び出し文と呼び出し先とを独立して解析する) 手法を採用する。

## 2.2 動的スライシング

静的スライシングではソースプログラムコードを対象に依存関係を解析し、スライスを抽出していたが、動的スライシングでは、依存関係の抽出対象は実行系列 (Execution Trace) になる。実行系列とは、ある入力を与えプログラムを実行したときの、実際に実行された文の列をいう。また、実行系列中の  $r$  番目の文の実行のことを実行時点  $r$  と呼ぶ。

実行系列  $e$  中の実行時点  $r_1, r_2$  について考える。

$r_1$  が制御文であり、 $r_1$  の結果によって  $r_2$  が実行されるかどうかが決まるとき、実行時点  $r_1$  から  $r_2$  に対し動的制御依存 (Dynamic Control Dependence, DCD) があるという。

$r_1$  から  $r_2$  へ変数  $v$  を再定義する実行時点がなく、 $r_1$  において  $v$  が定義され、 $r_2$  において  $v$  が参照されるとき、実行時点  $r_1$  から  $r_2$  に対し変数  $v$  に関する動的データ依存 (Dynamic Data Dependence, DDD) があるという。

そして、これらの動的依存関係を元に動的依存グラフ (Dynamic Dependence Graph, DDG) を構築する。

入力値の組  $\mathcal{X}$  が与えられたときの実行時点  $r$  における変数  $v$  に関する動的スライスとは、実行時点  $r$  に対応する DDG 節点から DDG 辺を逆向きに辿ることで到達可能な節点集合に対応する文の集合である。なお、組  $(\mathcal{X}, r, v)$  を動的スライシング基準 (Dynamic Slicing Criteria) と呼ぶ。

### 2.3 準動的な手法 および その分類

テスト・保守工程でのデバッグ作業において、プログラムスライシング技術はフォールト位置特定に有効である [9]。このとき、テストケースは一般に有限個であり、入力データも特定のものに限定される。本来、デバッグ作業では対象プログラムの実行は必要不可欠であり、そこから得られる情報を利用することで、静的スライスよりも小さい (正確な) スライスを抽出することができる。

動的スライシングはそのようなアプローチの一つであるが、実行経路 および 各実行時点での変数の状態

```

1  program Square_Cube(input, output);
2  var a, b, c, d : integer;
3  function Square(x : integer) : integer;
4  begin
5      Square := x * x
6  end;
7
8
9
10
11 begin
12
13     readln(a);
14
15
16
17     readln(c);
18     if c = 0 then
19         d := Square(a)
20
21
22
23
24     writeln(d)
25 end.
```

図3 動的スライシング基準 ( $\{a=2, b=3, c=0\}, 24, d$ ) に対する動的スライス

Fig.3 Dynamic slice for slicing criteria ( $\{a=2, b=3, c=0\}, 24, d$ )

など) プログラム実行に関する全履歴を必要とするため、プログラム実行時のオーバーヘッドが非常に大きい。

そこで、実行時オーバーヘッドの削減を目的として、静的解析と動的解析を組み合わせるスライスを抽出する、準動的な手法が提案されている。

#### 2.3.1 実行経路の収集に着目した手法

プログラムの実行経路に関する情報を収集・利用することでスライスサイズの削減を行う。軽量の実行時オーバーヘッドで経路情報を収集することに重点が置かれる。

- Profiling Method [1]

動的スライシングを単純化したものとして提案された。各文の実行の有無を記録し、静的に生成された PDG から実行されなかった文を削除する。この手法では、実行系列の保存は不要である。

- コールマークスライシング (Call-Mark Slicing) [10]

実行時、関数・手続き呼び出し文の実行の有無のみ記録する。しかし、静的解析により導出される実行経路情報と組み合わせることで、呼び出し文以外でかつ確実に実行されることのない文もスライス計算対象から排除できる。この手法は、Profiling Method と比べ、

```

1: a[0] := 0;
2: a[1] := 1;
3: readln(i);
4: c := a[i];

```

図4 配列変数に関するデータ依存  
Fig. 4 Data dependence for array variables

実行時に記録する文が少なくなる。

- ハイブリッドスライシング (Hybrid Slicing) [3]

ユーザの設定したブレークポイントの実行履歴または各関数・手続きの呼び出し履歴を記録することで、実行経路の予測を行う。有効な実行経路情報を得るため、ユーザはブレークポイントの設定位置に注意を払う必要がある。

### 2.3.2 データ依存関係の収集に着目した手法

プログラムの実行経路を静的解析により予測できれば、整数やブールなどの単純型の変数に関するデータ依存関係は容易に抽出できる。しかし、配列やポインタ変数に関するデータ依存関係の抽出には限界がある。

例として、図4に示すプログラム断片を考える。この場合、実行経路は唯一に定まるが、それだけでは文4が文1および文2にデータ依存するかを決定することはできない。

通常、この問題の解決には、完全実行履歴 (Full Execution History) を用いて DDG 構築を行なう。DDG ではすべての配列要素が展開されており、完全なデータ依存関係を保持することができるが、完全実行履歴の蓄積に要するオーバーヘッドは極めて大きい。そこで、データ依存関係の正確性と実行時オーバーヘッドとのトレードオフを考慮した手法が提案されている。

- Reduced DDG Method [1]

DDG 中に存在する同一構造を持つ部分グラフを一つに集約し、DDG 全体の大きさを削減する。これにより DDG のサイズは小さくなるが、実行時オーバーヘッドは類似性チェックのため増加する。

- 依存キャッシュスライシング

本稿でこの手法を提案する。単純なキャッシュを利用して動的に抽出したデータ依存関係と、静的に解析した制御依存関係を組み合わせるスライシングの計算を行なう。この詳細は3.節で示す。

## 3. 依存キャッシュスライシング

### 3.1 概要

通常、ポインタ変数や配列要素のデータ依存関係を静的解析により得ることは非常に難しい[4], [5], [7]。

一方、ある入力データを与えプログラムを実行し、その際にポインタ変数の参照先や配列の添字値を把握する機構を実現することは容易である。しかし、プログラム実行のために非常に大きなオーバーヘッドを必要とする。

そこで我々は、依存関係抽出の正確性と実行時オーバーヘッドとのトレードオフを考慮したスライシング手法である、依存キャッシュスライシング (Dependence Cache Slicing) 手法を提案する。以下は依存キャッシュスライシングの計算手順の概略である。

#### Step 1 静的制御依存関係解析

PDG の部分グラフ  $PDG_{DS}$  を静的に生成する。

まず、静的スライシングで利用する PDG と同様、文または制御文に対応する節点を用意する。そして、文間に制御依存関係が存在すれば、対応する節点間に制御依存辺を引く。ただし、 $PDG_{DS}$  にデータ依存辺は加えない。

#### Step 2 動的データ依存関係解析

対象プログラムをある入力データで実行する。

実行の際、次節で示すデータ依存関係抽出アルゴリズムに基づき、動的なデータ依存関係を計算し、 $PDG_{DS}$  にデータ依存辺を追加する。プログラム実行が終了した時点で、 $PDG_{DS}$  の完成となる。

#### Step 3 $PDG_{DS}$ によるスライシング計算

$PDG_{DS}$  を用いて、静的スライシングと同様の方法でスライシング計算を行う。

例えば、スライシング基準  $(s_c, v)$  に関する依存キャッシュスライシングを抽出する場合、まず、 $s_c$  に対応する節点から制御依存辺および  $v$  に関するデータ依存辺を逆向きに辿ることで到達可能な節点集合を導出する。そして、この節点集合に対応する文が求めるスライシングとなる。

### 3.2 データ依存関係収集アルゴリズム

図5に3.1節のStep2で使われるデータ依存関係抽出アルゴリズムを示す。まず、プログラム中で用いられる全ての変数  $v$  に対して、キャッシュ  $(C(v))$  と記す) を用意する。大域変数など静的な変数はプログラム実行開始時に、スタック上の Automatic 変数やヒープ中の変数など動的な変数はその変数の割付け時にキャッシュを用意する。プログラムの各実行時点において、 $C(v)$  は最も新しく  $v$  を定義した文に対応する節点を保持し、文  $s$  において  $v$  が使用 (参照) された時、 $C(v)$  が保持する節点から  $s$  に対応する節点に対してデータ依存辺を (すでに存在しなければ) 追加す

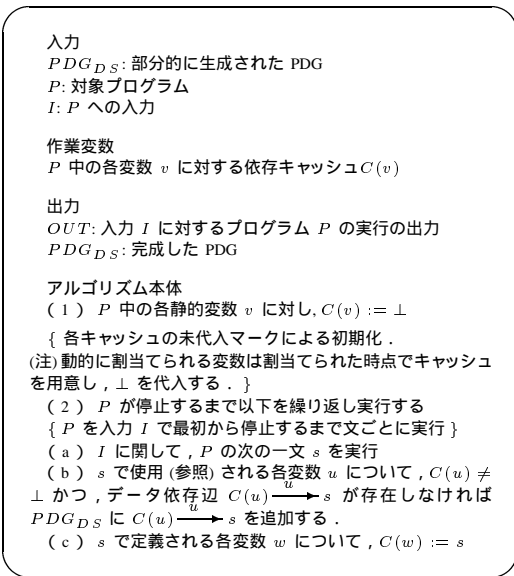


図5 データ依存関係収集アルゴリズム

Fig.5 Algorithm for computing data dependence relations

る．一方,  $s$  で  $v$  が定義された時,  $C(v)$  は  $s$  に対応する節点に更新する．これらを全ての変数に対して行う．

配列変数や構造体に対しては, 全ての要素に対してキャッシュを用意する．例えば,  $A[1], A[2], \dots, A[10]$  という 10 個の要素を持つ配列変数  $A$  は, キャッシュ  $C(A[1]), C(A[2]), \dots, C(A[10])$  を持つ．ポインタ変数  $p$  が文  $s$  において使用された場合, 使用された  $p$  自身だけでなく,  $p$  を介して間接参照された変数  $p \uparrow$  をも考慮しなければならない．つまり, 直接と間接の参照 ( $C(p) \xrightarrow{p} s$  と  $C(p \uparrow) \xrightarrow{p \uparrow} s$  の両者) をデータ依存辺に含めなければならない．また, 文  $t$  における  $q \uparrow := \dots$  のようなポインタ変数  $q$  を介した間接的な代入については, キャッシュ  $C(q \uparrow)$  が  $t$  において更新され, また  $t$  において  $q$  が使用されたと考える．

動変数に対しては, 個々のインスタンスごとにキャッシュを用意する．また, 配列や構造体では, 参照される要素ごとにキャッシュが必要である．例えば, 要素  $v_1$  と  $v_2$  から構成される構造体  $v$  に対しては, キャッシュ  $C(v_1), C(v_2)$  を用意する．各  $v_1, v_2$  への定義及び参照時の操作は図5のアルゴリズムと同じである．文  $s$  で構造体  $v$  全体への定義が行なわれる時,  $C(v_1) := s, C(v_2) := s$  を行なう．また,  $s$  で  $v$  全体の参照が行なわれる時は, データ依存辺  $C(v_1) \xrightarrow{v_1} s$  及び  $C(v_2) \xrightarrow{v_2} s$  を  $PDG_{DS}$  に追加する．(注: 辺上のラベル  $v_1, v_2$  は静的に決まりうる要素名)

このアルゴリズムでは, プログラム実行中の変数の使用状況に比例したキャッシュ空間と, 変数へのアクセス回数に応じた実行時間のオーバーヘッドが必要である．

## 4. 実験

### 4.1 Osaka Slicing System の概要

様々なスライシングアルゴリズムを評価するため, 我々は *Osaka Slicing System* [11] と呼ぶソフトウェア開発・デバッグ環境を構築している．対象言語は Pascal である．

このシステムはプログラムの実行とデバッグを行なうことができる．また, 静的スライシング, 動的スライシングの機能を持つ．今回, コールマークスライシング, そして 3.2 節で提案した依存キャッシュスライシングの機能を追加した．

### 4.2 サンプルプログラムの実行

このシステムを用い, 様々なプログラムを実行し幾つかの評価尺度を得た．プログラム  $P1$  はカレンダープログラム,  $P2$  は在庫管理プログラム,  $P3$  はプログラム  $P2$  の在庫管理プログラムの拡張版である．

表1は3つのサンプルプログラムのスライスサイズである (表中, CM はコールマークスライス, DC は依存キャッシュスライスを示す)．これらの値はスライシング基準や入力値によって変動する．ここでは典型的なデバッグ状況に対していくつかの基準と入力を用いた平均値を示している．

表2は実行前解析に要した時間を示している．静的スライシングの場合, この値はPDG構築に要した時間である．コールマークスライシングではPDGの構築と実行経路情報の静的解析の両方に要した時間である．依存キャッシュスライシングでは初期の  $PDG_{DS}$  構築に要した時間である．また, 動的スライシングでは実行前解析は不要である．

表3に実行時間を示す．静的スライシングの場合, 対象プログラムは実行時オーバーヘッド無しで実行されるため, この値は対象プログラム自身の実行時間である．動的スライシングの実行はDDGの生成時間が含まれている．依存キャッシュスライシングの実行時間は依存関係をキャッシュする時間と  $PDG_{DS}$  を生成する時間が含まれている．また, コールマークスライシングでは呼び出し文を記録する時間が含まれている．

表4は依存グラフを用いてスライスを計算する時間を示している．静的スライシングの場合, これは実行前に完了している．動的スライシングでは, 動的依存

表 1 スライスサイズ (行)

Table 1 Size of slice (lines)

プログラム	静的	CM	DC	動的
P1(85 行)	21	17	15	5
P2(387 行)	182	162	16	5
P3(871 行)	187	166	61	8

表 2 実行前解析時間 (ms)

Table 2 Pre-execution analysis time (ms)

プログラム	静的	CM	DC	動的
P1	11	14	5	N/A
P2	213	215	19	N/A
P3	710	698	48	N/A

(Celeron-450MHz CPU with 128MB Memory)

表 3 実行時間 (ms)

Table 3 Execution time (ms)

プログラム	静的	CM	DC	動的
P1	47	47	51	174
P2	43	43	45	4,540
P3	4,700	4,731	4,834	206,464

(Celeron-450MHz CPU with 128MB Memory)

表 4 スライス計算時間 (ms)

Table 4 Slice computation time (ms)

プログラム	静的	CM	DC	動的
P1	0.4	0.6	0.3	76.0
P2	1.9	1.8	0.7	101.0
P3	3.0	3.0	1.2	24,969.3

(Celeron-450MHz CPU with 128MB Memory)

関係を探る時間が計測されている。また、コールマークスライシングと依存キャッシュスライシングでは、それぞれの Step 3 に要した時間である。

次節でこれらの結果について議論を行なう。

## 5. 考察

### 5.1 実験データの解釈

#### • スライスサイズ

表 1 はスライスのサイズを示している。依存キャッシュスライスのサイズは静的スライスの 9-71% となっている。

依存キャッシュスライスのサイズは静的スライスと動的スライスの中間となっている。これは常に静的スライスより小さく(良く)、動的スライスより大きい(悪い)。また、依存キャッシュスライスはコールマークスライスよりも良い(小さい)といえる。これは、コールマークスライシングが静的に解析したデータ依存関係から実行されない部分を削除するだけなのに対し、依存キャッシュスライシングが特定の実行パスに依存するデータ依存関係を反映するためである。依存キャッ

シススライスのサイズの削減度合はプログラム P2 と P3 では、P1 に比べ大きい。これは P1 が整数やブールなどの単純型の変数しか使用していないのに対し、P2 と P3 が依存キャッシュスライシングもしくは動的スライシングのみが詳細に解析できる配列変数を使用しているためである。

#### • 実行前解析

表 2 で示したように、依存キャッシュスライシングでは、静的スライシングに比べ短い実行時間で実行前解析が可能である。これは、静的スライシングが制御依存関係とデータ依存関係の解析を行なっているのに対し、依存キャッシュスライシングでは制御依存関係のみの解析を行なえばよいためである。

#### • 実行時間

表 3 で示した実行時間は動的スライシングのオーバーヘッドが極端に大きいことを示している。ループを繰り返し実行するなどしてプログラムの実行が長くなると、深刻な性能低下を引き起こす。依存キャッシュスライシングは静的スライシングよりも実行時間が長く、動的スライシングよりも短くなっている。

ここで示した実行時間はインタープリタ型のシステムに基づいているため、これらはインタープリタのオーバーヘッドによって隠蔽されている可能性がある。この問題については 5.2 節で議論する。

#### • スライス計算時間

表 4 で示すように、動的スライシングはスライス結果を収集するのに長時間を必要とする。依存キャッシュスライシングでは、静的スライシングよりも短くなっている。これは、依存キャッシュスライシングが PDG よりも小さい PDG<sub>DS</sub> を用いているためである。動的スライシングについては、巨大な DDG を探索するのに非常に時間がかかる。

### 5.2 依存キャッシュスライスの適用領域と限界

4. 節で示した実験はインタープリタ型の実行システムの下で行なわれた。ここでは、スライシングに対する解析・実行時間の特徴がコンパイラ環境でも同様かどうかを考察する。

C でマージソートのプログラムを書き、実行中に依存キャッシュの更新とデータ依存関係を収集するようにソースプログラムを変更した。このプログラムの実行時間は変更前の 8.6 倍であった。これは依存キャッシュスライシングのオーバーヘッドがコンパイラ環境で

は大きくなることを示している<sup>(注1)</sup>。しかし、単純型変数のデータ依存情報は静的解析で容易に決定できるため、データ依存情報を配列とポインタ変数についてのみ動的に収集することで高速化が可能である。この考えに基づき、プログラムを再び変更した。この部分的依存キャッシュスライシングプログラムの実行時間は変更前の3.4倍となり、これは実用的に許容できると考える。

表1からわかるように、依存キャッシュスライスは常に動的スライスよりも大きくなっている。これは依存キャッシュスライシングは同じ文の複数回の出現を区別せず、キャッシュの中に直近のDef-Use関係を保持しているからである。

### 5.3 他手法との関連

データフロー情報の収集に注目した動的手法についてはほとんど研究されていない。2.3節で紹介したReduced DDG Methodはこれらのうちの一つであり、動的スライシングの正確さと、解析に必要なメモリ空間の削減を実現している。しかし、DDGの重複チェックのために実行オーバーヘッドは削減されず、増加する可能性もある。

文献[4],[5]では、ポインタや配列変数の静的解析について述べられている。これらは不確実性を残している[13]のに対し、依存キャッシュスライシングは動的情報を利用しており、軽量のオーバーヘッドでどんな種類の変数に対しても、実用的なスライスの正確さを実現している。

文献[2]では、静的と動的スライスを一般化させた、制約スライス (*Constrained Slice*) が提案されている。これはプログラムの入力の部分集合をプログラムの実行と捉え、この入力制限を用い、依存関係の再計算を行なう。しかし、このアプローチの実行効率やその実用性については不明である。

## 6. まとめ

プログラムの注意をソフトウェアの小さな一部分に特化させることは、プログラムのデバッグや保守の効率を改善するために非常に重要である。伝統的なプログラムスライシングは精度と効率の十分なトレードオフを提供していない。

我々は依存キャッシュスライシングという軽量の準

動的スライシング手法を提案した。

スライス結果は同一スライシング基準での動的スライスより大きくなるが、静的スライスより小さくなる。我々はこれらのスライシングアルゴリズムを実験インタープリタシステムに実装した。また、様々なサンプルプログラムを実行し、我々のアプローチを検証した。

我々は現在のインタープリタシステムではなくコンパイラシステムに基づくデバッグ環境の構築を予定している。そのシステムでは様々なデバッグ機能を結合させ、依存キャッシュスライスが計算可能となる。このコンパイラは依存キャッシュ機能を持ったオブジェクトコードを生成し、そのコードは自動的に動的データ依存関係を収集する。この情報はスライスまたは他のデバッグ補助情報としてユーザから要求された際に表示され、フォールト位置特定のための有用なデバッグ情報を提供する。

謝辞 本研究は、一部文部省科学研究費補助金特定領域研究(A)(2)(課題番号:10139223)の補助を受けた。

### 文献

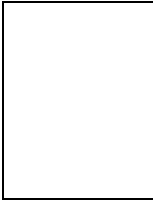
- [1] Agrawal, H. and Horgan, J.: "Dynamic Program Slicing", *SIGPLAN Notices*, Vol.25, No.6, pp. 246–256, 1990.
- [2] Field, J., and Ramalingam, G.: "Parametric Program Slicing", *Proc. of 22nd ACM Symposium on Principles of Programming Languages*, pp. 379–392, San Francisco, USA, January (1995).
- [3] Gupta, R., Soffa, M.L., and Howard, J.: "Hybrid Slicing: Integrating Dynamic Information with Static Analysis", *ACM Transaction on Software Engineering and Methodology*, Vol. 6, No. 4, pp. 370–397, 1997.
- [4] Hind, M., Burke, M., Carini, P., and Choi, J.: "Interprocedural Pointer Alias Analysis", *ACM Trans. on Programming Languages and Systems*, Vol.21, No. 4, pp. 848–894 (1999).
- [5] Horwitz, S. Pfeiffer, P., and Reps, T.: "Dependence Analysis for Pointer variables", *Proceedings of SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp.28–40, *SIGPLAN Notices* Vol. 24, No. 6 (1989).
- [6] Korel, B., and Laski, J.: "Dynamic Program Slicing", *Information Processing Letters*, Vol.29, No.10, pp. 155–163 (1988).
- [7] Liang, D., and Harrold, M. J.: "Efficient Points-To Analysis for Whole-Program Analysis", *Proc. of 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp.199–215, Toulouse, France (1999).
- [8] Ning, J. Q., Engberts, A., Kozaczynski, W. V.: "Automated Support for Legacy Code Understanding", *Communications of the ACM*, Vol. 37, No. 5, pp.50–57, May (1994).
- [9] 西松 顕, 楠本 真二, 井上 克郎: "フォールト位置特定におけるプログラムスライスの実験的評価". 信学技報 SS98-3, pp.17-24, Mar 1998
- [10] Nishimatsu, A., Jihira, M., Kusumoto, S. and Inoue, K.: "Call-Mark Slicing: An Efficient and Economical Way of Reducing

(注1): これは、実行中に DDG の探索が必要な動的スライシングではオーバーヘッドが許容できないほど大きくなることも示唆している。

Slice”, *Proceedings of The 21st International Conference on Software Engineering*, pp.422-431, Los Angeles, CA, USA, 1999.

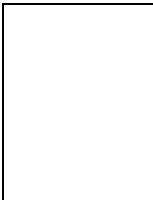
- [11] 佐藤 慎一, 飯田 元, 井上 克郎: “プログラムの依存関係解析に基づくデバッグ支援ツールの試作”, *情処論* Vol.37, No.4, pp.536-545, Apr 1996
- [12] Weiser, M.: “Program Slicing”, *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439-449 (1981).
- [13] Ramalingam, G.: The Undecidability of Aliasing, *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 5, pp. 1467-1471 (1994).

(平成年月日受付, 月日再受付)



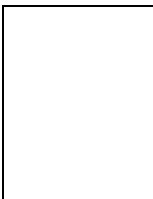
#### 高田 智規

平 7 阪大・基礎工・情報卒．平 9 同大大学院修士課程了．現在西日本電信電話株式会社勤務．在学中，プログラムスライスの研究に従事．



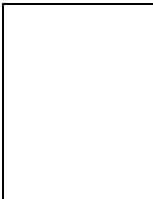
#### 井上 克郎 (正員)

昭 54 阪大・基礎工・情報卒．昭 59 同大大学院博士課程了．同年同大・基礎工・情報・助手．昭 59～昭 61 ハワイ大マノア校・情報工学科・助教授．平 1 阪大・基礎工・情報・講師．平 3 同学科・助教授．平 7 同学科・教授．工博．ソフトウェア工学の研究に従事．



#### 大畑 文明

平 10 阪大・基礎工・情報中退．平 12 同大大学院修士課程了．現在同大学院博士課程在学中．プログラム構造解析の研究に従事．



#### 芦田 佳行

平 10 阪大・基礎工・情報中退．平 12 同大大学院修士課程了．現在 NTT ソフトウェア株式会社勤務．在学中，プログラムスライスの研究に従事．