論　文

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[†,††]　　井上　克郎[†]　　大畑　文明[†]　　芦田　佳行[†††]

A Program Slicing Method Using Lightweight Dynamic Information

Tomonori TAKADA[†,††], Katsuro INOUE[†], Fumiaki OHATA[†], and Yoshiyuki ASHIDA[†††]

あらまし　大規模なプログラムのデバッグを行う際，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ヘッドが大きくなるという問題がある．本論文では，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，実行時オーバヘッド，依存キャッシュスライシング

1. まえがき

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．

† 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
†† 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
††† NTT ソフトウェア株式会社，横浜市
NTT Software Corporation, Yokohama-shi, 231–8554 Japan

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）の二つに分類される．

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，配列及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．更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．

本論文では，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.** では，静的・動的スライシングについての簡単な説明及び既存の準動的手法の分類を行う．**3.** 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及び評価実験を**4.** で示し，**5.** で検証結果の考察と関連研究について述べる．最後に，**6.** でまとめと今後の課題について述べる．

2. スライシング手法の分類

本章では，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及び動的スライスを，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，それらの紹介及び分類を行う．

2.1 静的スライシング及びその分類

ソースプログラム p 中の文 s_1 と s_2 について考える．

```
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.

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一つ存在し，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）たど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．

```
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 critetia (24, d).

```
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 critetia ($\{a = 2,\ b = 3,\ c = 0\}$, 24, d).

- 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き数・大域変数情報を用いるかどうか（Context Sensitiveness / Insensitiveness）

本論文では，特に明示しなければ，制御フロー解析を行いかつ関数・手続き解析時に実際の引き数・大域変数情報を用いない（呼出し文と呼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定されるとき，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もとに動的依存グラフ（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 辺を逆向きにたど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 準動的手法及びその分類

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，（実行経路及び各実行時点での変数の状態など）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ヘッドが非常に大きい．

そこで，実行時オーバ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てスライスを抽出する，準動的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．軽量な実行時オーバ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，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 及び文 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も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

本論文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てスライスの計算を行う．この詳細は **3.** で示す．

```
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.

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ス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} によるスライス計算

PDG_{DS} を用いて，静的スライシングと同様の方法でスライス計算を行う．

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スを抽出する場合，まず，s_c に対応する節点から制御依存辺及び v に関するデータ依存辺を逆向きにたど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スとなる．

3.2　データ依存関係収集アルゴリズム

図 5 に **3.1** の Step 2 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すべての変数 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

入力
PDG_{DS}: 部分的に生成された PDG
P: 対象プログラム
I: P への入力

作業変数
P 中の各変数 v に対する依存キャッシュ$C(v)$

出力
OUT: 入力 I に対するプログラム P の実行の出力
PDG_{DS}: 完成した PDG

アルゴリズム本体
（1） P 中の各静的変数 v に対し, $C(v) := \perp$
{ 各キャッシュの未代入マークによる初期化.
(注) 動的に割当てられる変数は割当てられた時点でキャッシュを用意し，$\perp$ を代入する. }
（2） P が停止するまで以下を繰り返し実行する
{ P を入力 I で最初から停止するまで文ごとに実行 }
（a） I に関して，P の次の一文 s を実行
（b） s で使用 (参照) される各変数 u について，$C(u) \neq \perp$ かつ，データ依存辺 $C(u) \xrightarrow{u} s$ が存在しなければ PDG_{DS} に $C(u) \xrightarrow{u} s$ を追加する.
（c） s で定義される各変数 w について，$C(w) := s$

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.

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）されたとき，$C(v)$ 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（既に存在しなければ）追加する．一方，s で v が定義されたとき，$C(v)$ は s に対応する節点に更新する．これらをすべての変数に対して行う．

配列変数や構造体に対しては，すべての要素に対してキャッシュを用意する．例えば，$A[1]$, $A[2]$, ..., $A[10]$ という 10 個の要素をもつ配列変数 A は，キャッシュ $C(A[1])$, $C(A[2])$, ..., $C(A[10])$ をもつ．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 $q\uparrow := \ldots$ 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．

動的変数に対しては，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，参照され得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 全体への定義が行われるとき，$C(v_1) := s$, $C(v_2) := s$ を行う．また，s で v 全体の参照が行われるときは，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 は静的に決まり得る要素名）．

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もつ．今回，コールマークスライシング，そして **3.2** 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いく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 は三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ヘッドなしで実行され

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-450 MHz CPU with 128 MB 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-450 MHz CPU with 128 MB 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-450 MHz CPU with 128 MB Memory)

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，依存キャッシュスライシング，動的スライシングの場合，それぞれ PDG，PDG_{DS}，DDG を探索する時間である．また，コールマークスライシングでは，実行されなかったとわかる文の頂点を PDG から削除し，探索するのに要した時間である．

次章でこれらの結果について議論を行う．

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．依存キャッシュ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若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，これらはインタプリタのオーバヘッドによって隠ぺいされている可能性がある．この問題については **5.2** 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_{DS} 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. で示した実験はインタプリタ型の実行システム

のもとで行われた．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(注1)．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** 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，軽量なオーバ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とらえ，この入力制限を用い，依存関係の再計算を行う．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．

(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．

6. むすび

プログラマ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な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も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] H. Agrawal and J. Horgan, “Dynamic program slicing,” SIGPLAN Notices, vol.25, no.6, pp.246–256, 1990.

[2] J. Field and G. Ramalingam, “Parametric program slicing,” Proc. 22nd ACM Symposium on Principles of Programming Languages, pp.379–392, San Francisco, USA, Jan. 1995.

[3] R. Gupta, M.L. Soffa, and J. Howard, “Hybrid slicing: Integrating dynamic information with static analysis,” ACM Trans. Software Engineering and Methodology, vol.6, no.4, pp.370–397, 1997.

[4] M. Hind, M. Burke, P. Carini, and J. Choi, “Interprocedural pointer alias analysis,” ACM Trans. Programming Languages and Systems, vol.21, no.4, pp.848–894, 1999.

[5] S. Horwitz, P. Pfeiffer, and T. Reps, “Dependence analysis for pointer variables,” Proc. SIGPLAN ’89 Conference on Programming Language Design and Implementation, pp.28–40, SIGPLAN Notices, vol.24, no.6, 1989.

[6] B. Korel and J. Laski, “Dynamic program slicing,” Information Processing Letters, vol.29, no,10, pp.155–163, 1988.
[7] D. Liang and M.J. Harrold, “Efficient points-to analysis for whole-program analysis,” Proc. 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on Foundations of Software Engineering, pp.199–215, Toulouse, France, 1999.
[8] J.Q. Ning, A. Engberts, and W.V. Kozaczynski, “Automated support for legacy code understanding,” Commun. 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，March 1998.
[10] A. Nishimatsu, M. Jihira, S. Kusumoto, and K. Inoue, “Call-mark slicing: An efficient and economical way of reducing slice,” Proc. 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,” 情処学論，vol.37, no.4, pp.536–545, April 1996
[12] M. Weiser, “Program slicing,” Proc. Fifth International Conference on Software Engineering, pp.439–449, 1981.
[13] G. Ramalingam, “The undecidability of aliasing,” ACM Trans. Programming Languages and Systems, vol.16, no.5, pp.1467–1471, 1994.

（平成 13 年 4 月 16 日受付，8 月 27 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，平 13 より阪大大学院基礎工学研究科博士課程在学中．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 59〜61 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．