制限された動的情報を用いたブロック単位スライシング手法の提案

高田智規[†]　井上克郎[†]

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ヘッドが大きくなるという問題がある．これまでに，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った．依存キャッシュスライシングは動的スライシングと比べ，少ない実行時オーバヘッドでスライスを求めることが可能である．しかし，実際のデバッグ段階で使用するためには，さらにオーバヘッドを削減することが求められている．

そこで，本稿では，依存キャッシュスライシングのオーバヘッドを削減させたブロック単位スライシングの提案を行う．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なった．この結果，ブロック単位スライシングは依存キャッシュスライシングに比べ約 50%実行時オーバヘッドを削減することが可能であった．

A Block Slicing Method Using Lightweight Dynamic Information

TOMONORI TAKADA[†] and KATSURO INOUE[†]

When we try to debug a large program effectively, it is very important to separate suspicious program portions from the overall source program. Program slicing is a promising technique to extract a program portion; however, it remains difficult issues. Static slicing sometimes produces a large portion of the source program, especially for a program with array and pointer variables. Dynamic slicing requires unacceptably huge run-time overhead. we proposed a dependence-cache slicing method which uses both static and dynamic information, and can compute program slices more effective than dynamic slicing. However, the further reduction of the run-time overhead is required.

In this paper, we propose a slicing method named a block slicing which can reduce the run-time overhead compared to a dependence-cache slicing. This algorithm has been implemented, and execution data for sample programs have been collected. The result shows block slicing reduces the run-time overhead by about 50% from the dependence-cache slicing.

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）[10] 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[7]．

Weiser による研究[10] 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[10] 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

† 大阪大学 大学院基礎工学研究科
Graduate School of Engineering Science

```
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

```
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

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),5)] 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．

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 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 / Insensitiveness）

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[7]．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手法が提案されている．

```
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
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

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)[8)]
  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，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)[2)]
  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．

```
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

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[11)]
  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なう．

本稿では，依存キャッシュスライシングの実行時オーバヘッドを削減させた手法である，**ブロック単位スライシング**の提案を行う．

3. ブロック単位スライシング

3.1 概　　要

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[3),4),6)]．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．

ポインタ変数や配列変数のデータ依存関係を効率良く解析する手法として，依存キャッシュスライシング[11)] がある．依存キャッシュスライシングでは，簡単なキャッシュを用いることによって，動的スライシングと比べ小さいオーバヘッドでデータ依存関係を求めることができる．

しかし，その実行時間は静的スライシングの 3 倍〜10 倍程度であり，現実のデバッグ環境においては，さらなるオーバヘッドの削減が求められる．

そこで，依存キャッシュスライシングの基本的なアイデアを基に，さらなる効率化を行った，ブロック単位スライシングの提案を行う．以下は，ブロック単位スライシングの計算手順である．

STEP1　実行前解析 (ブロック化・静的制御依存解析)
ソースプログラムから，以下の手順により，PDG の部分グラフ PDG_{BL} を静的に生成する．
まず，3.2 節に述べるブロック化アルゴリズムに従い，文の集合とブロックとの対応関係を得る．
次に，ブロックに対応する節点を用意し，ブロック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，データ依存辺は加えない．

STEP2　実行時解析 (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，3.3 節で示すデータ依存関係抽出アルゴリズムに基き，動的なデータ依存関係を計算し，PDG$_{BL}$ 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$_{BL}$ の完成となる．

STEP3　スライス計算
PDG$_{BL}$ 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スとなる．

3.2 ブロック化アルゴリズム

3.1 節の Step 1 で使われるブロック化アルゴリズムを図 5 に示す．

このアルゴリズムでは，ユーザの設定したブロック化因子 N に基づき，任意の粒度でのブロック化が可能である．

3.2.1 ブロック化の実例

図 6 に示すプログラムに対して，ブロック化因子を変化させた際の実例を以下に示す．

3.2.2 ブロック化因子 $N = 2$ の場合

$s1, s2$ は制御構造を含まないため，$B1 = \{s1, s2\}$ と

入力
　P: ソースプログラム (文 $s_1, s_2, \cdots, s_n$ が含まれているものとする)
　N: ユーザによって指定されたブロック化因子
出力
　BS: ブロックの集合
アルゴリズム本体
(1)　$BS := \perp$, $i = 1$
(2)　$i < n$ である限り，次を実行
　(a)　$s_i, \cdots, s_{i+N-1}$ が同じ制御ブロックに含まれる場合，または制御文を含むがその配下の文が全てこれらの文集合に含まれる場合 $B_j := \{s_i, \cdots, s_{i+N-1}\}$, $i := i + N$
　(ただし，関数境界は越えないものとする)
　(b)　$s_i, \cdots, s_{i+N-1}$ が上記以外の場合 (制御ブロックの境界を越える場合), ブロック境界を越えないような文を s_k とすると，$B_j := \{s_i, \cdots, s_k\}$, $i := k + 1$
　(c)　$BS := BS \cup B_j$

図 5　ブロック化アルゴリズム

```
s1: ...
s2: ...
s3: if (..)  then begin
s4:  ...
s5:  ...
s6:  ...
    end
    else begin
s7:  ...
    end
s8: ...
s9: ...
```

図 6　ブロック化サンプルプログラム

なる．次に，$s3$ は制御文であり，かつ $s3, s4$ は $s3$ 配下の文全てでは無いため制御ブロックの境界を越える場合と見なされ，ブロック $B2 = \{s3\}$ となる．$s3$ 配下の文 $s4, s5$ には制御構造を含まないため，$B3 = \{s4, s5\}$ となる．$s6, s7$ は制御ブロックの境界を越えるため，$B4 = \{s6\}$ となる．$s7, s8$ は制御ブロック境界を越えるため，$B5 = \{s7\}$ となり，$B6 = \{s8, s9\}$ となる．

以上から，
B1: s1, s2
B2: s3
B3: s4, s5
B4: s6
B5: s7
B6: s8, s9

という B1〜B6 の 6 つのブロックが得られる．

3.2.3　ブロック化因子 $N = 5$ の場合

$N = 2$ の時と同様に，5 つの文を対象にブロック化を進めていく．この結果は
B1: s1, s2
B2: s3
B3: s4, s5, s6
B4: s7
B5: s8, s9

となる．

3.2.4　ブロック化因子 $N = 7$ の場合

$\{s4, \cdots, s7\}$ は全て $s3$ の if 文の配下であり，$N = 7$ の場合は，これら全ての文が同じブロックに含まれる．そのため，この結果は
B1: s1, s2, s3, s4, s5, s6, s7
B2: s8, s9

となる．

入力
　PDG_{BL}: 部分的に生成されブロック化された 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_{BL}: 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 $C(u) \xrightarrow{u} B(s)$ が存在しなければ PDG_{BL} に $C(u) \xrightarrow{u} B(s)$ を追加する.
　ここで，$B(s)$ は s を含むブロックを表す.
　(c)　s で定義される各変数 w について，$C(w) := B(s)$

図 7　データ依存関係収集アルゴリズム

3.3　データ依存関係収集アルゴリズム

図 7 に **3.1** 節の 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の属するブロックを保持し，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 の属するブロックに対応する節点 ($B(s)$) に対してデータ依存

辺を (すでに存在しなければ) 追加する．一方，s で v が定義された時，$C(v)$ は $B(s)$ 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，$A[1], A[2], \ldots, A[10]$ という 10 個の要素を持つ配列変数 A は，キャッシュ $C(A[1]), C(A[2]), \ldots, C(A[10])$ 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（$C(p) \xrightarrow{p} B(s)$ と $C(p\uparrow) \xrightarrow{p\uparrow} B(s)$ 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，参照されう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 7 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，$C(v_1) := B(s), C(v_2) := B(s)$ 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 $C(v_1) \xrightarrow{v_1} B(s)$ 及び $C(v_2) \xrightarrow{v_2} B(s)$ を PDG_{BL} 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，およびブロック数に応じた実行時間のオーバヘッドが必要である．

3.4 ブロック単位スライシングの例

図 1 に示したサンプルプログラムに対して，入力 $a = 2, b = 3, c = 0$，ブロック化因子 $N = 2$ としたときのブロック単位スライスを図 8 に示す．このときのブロックは，

B1: 5
B2: 9
B3: 12, 13
B4: 14, 15
B5: 16, 17
B6: 18
B7: 19
B8: 21
B9: 22
B10: 23
B11: 24

となっている．

```
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
8
9
10
11  begin
12     writeln("Squared Value ?");
13     readln(a);
14
15
16     writeln("Select Feature! Square: 0 Cube: 1");
17     readln(c);
18     if c = 0 then
19        d := Square(a)
20
21
22
23
24     writeln(d)
25  end.
```

図 8 スライシング基準 ({a = 2, b = 3, c = 0}, 24, d) に対するブロック単位スライス ($N = 2$)

4. ブロック単位スライシングの拡張

ブロック単位スライシングの実行時効率および利便性を向上させるため，以下を考える．

- スカラ型変数の静的解析
  配列やポインタ型の変数のデータ依存関係を静的に解析することは非常に困難であるが，スカラ型変数については比較的容易に解析することが可能である．
  そこで，PDG_{BL} 作成時に，スカラ型変数のデータ依存関係についても解析し，実行時にはポインタ・配列・構造体などの変数についてのみ依存関係を解析することで，実行時オーバヘッドを削減できる．
- ブロック内局所変数の解析省略
  ブロック内でのみ使用される変数 (関数の局所変数など) は，ブロック外へ依存関係が伝播することは無い．そこで，このような変数についてはキャッシュの作成・データ依存辺の追加を共に省略することにより，実行時間・消費メモリを削減することができる．
- 基本ブロック単位のブロック化
  ブロック化因子の特別な場合として，基本ブロックを一つのブロックとして計算すれば，ユーザが

```
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    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
21
22
23
24    writeln(d)
25  end.
```

図 9 スライシング基準 ({a = 2, b = 3, c = 0}, 24, d) に対するブロック単位スライス (基本ブロック単位でブロック化)

特にブロック化因子を指定する必要の無い場合に有用である．また，制御構造の境界を越えることが無くなり，無駄なブロックが少なくなると考えられる．図 1 に示したサンプルプログラムに対し，基本ブロック単位でブロック化を行い，入力 $a = 2, b = 3, c = 0$ としたときのブロック単位スライスを図 9 に示す．

5. 実　　験

5.1 概　　要

ブロック単位スライシングの有効性を確認するため実行時間に関する実験を行った．依存キャッシュスライシングのオーバヘッドはコンパイラ型言語の場合に特に大きくなる[11] ため，C 言語で記述された 2 つのプログラム $P1$，$P2$ を対象とし，動的データ依存収集の動作を追加した．

プログラム $P1$ はマージソートを行うプログラムであり，配列に格納されたデータの整列を行う．プログラム $P2$ はクイックソートを行うプログラムであり，キーとデータの 2 つの構成要素を持つ構造体の配列に対し，キーを基に整列を行う．

これらに対して，静的スライシング，依存キャッシュスライシング，ブロック単位スライシング の実行時解析時間を複数測定し，その平均時間を求めた．なお，ブロック単位スライシングについては，4 節で示した拡張を取り入れたアルゴリズムを使用した．

この実験の結果を表 1 に示す

表 1 平均実行時間 (秒)

	$P1$	$P2$
静的スライシング	0.017	0.266
依存キャッシュスライシング	0.141	2.692
ブロック単位スライシング	0.058	1.413

(M-PentiumIII 600MHz CPU with 256MB Memory)

5.2 考　　察

表 1 より，依存キャッシュスライシングでは静的スライシングの約 9〜10 倍程度の実行時間となっているのに対し，ブロック単位スライシングでは静的スライシングの約 3〜6 倍程度のとなっている．また，ブロック単位スライシングは依存キャッシュスライシングの実行時間は約 1/2 に短縮できていることが分かる．

この理由について以下に簡単な考察を行う．

ブロック単位スライシングでは，ブロック化を行うことにより PDG の節点数を減らすことができる．そのため，実行時に追加するデータ依存辺は依存キャッシュスライシングに比べ少なくなり，また，依存辺の追加に関するオーバヘッドも少なくなる．

また，関数内局所変数については，依存キャッシュスライシングではキャッシュを作成し，そのデータ依存辺をも追加しているが，ブロック単位スライシングではその局所変数が単一ブロックの中でしか使用され無い場合，解析を行わない．これは，頻繁に呼びだされ制御構造が単純な小さな関数などについて非常に有効であると考えられる．

今回の実験では，実行時間のみに注目し，ブロック単位スライシングの有効性を示したが，本来は実行時間とスライスの正確性が共に達成される必要がある．現状のブロック単位スライシングでは，スライスの粒度がブロック単位であり，スライスサイズは他手法と比べ大きくなる．

この問題は，スライス計算時に抽出されたブロックの内部を，静的解析によって文単位で抽出することで解決可能である．本アルゴリズムでは，関数境界を越えないようにブロック化を行う．そのため，ブロック内部のデータ依存関係について複雑な解析を行う必要はなく，変数の定義・参照関係および制御依存関係がわかれば解析が可能である．これは静的スライシングの関数内解析と同じ処理であり，ブロック内部の正確性については静的スライシングと同じとなる．従って，ブロック単位スライスのサイズは，最大の場合 (全て

のブロックがスライスに含まれている場合) においても静的スライシングと同程度となる．

また，本アルゴリズムでは，ブロック化を行った後に制御依存解析を行っている．ブロック単位スライスを一度のみ求める際にはこの方法が適していると考えられるが，ブロック単位スライスのブロック化因子を何度も変更するような場合には，あらかじめ制御依存解析を行った PDG の部分グラフを作成しておき，この PDG に対して節点集約 (ブロック化) を行うことで，実行前解析の時間を短縮できると考えられる．

6. ま と め

一般に，配列やポインタなどを含んだプログラムのデータ依存関係を静的に解析するのは非常に困難であり，また，動的に解析するのは非常に実行時オーバヘッドが必要となる．本稿では，ブロック単位スライシングというアルゴリズムを提案した．この方式では，複数の文をブロックとして扱い，ブロック単位で動的にデータ依存関係を求めることで，実行時オーバヘッドを大幅に削減できる．

今後は，ブロック単位スライシングの正確性について実験を行うとともに，5.2 節で示したブロック内静的依存解析についても検討，および，ブロック化因子を様々な数値に変動させた場合の，実行時間と正確性についての考察をも行う．また，静的・動的・依存キャッシュスライシング等の機能を実装したデバッグシステム[9] への実装を行い，実ユーザによるデバッグ実験などの検証も行う予定である．

参 考 文 献

1) Agrawal, H. and Horgan, J. : “Dynamic Program Slicing”, *SIGPLAN Notices, Vol.25, No.6*, pp. 246–256, 1990.
2) 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.
3) 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).
4) 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).
5) Korel, B., and Laski, J. : “Dynamic Program Slicing”, *Information Processing Letters*, Vol.29, No,10, pp. 155–163 (1988).
6) 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).
7) 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
8) Nishimatsu, A., Jihira, M., Kusumoto, S. and Inoue, K. : “Call-Mark Slicing: An Efficient and Economical Way of Reducing Slice”, *Proceedings of The 21st International Conference on Software Engineering*, pp.422-431, Los Angeles, CA, USA, 1999.
9) 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
10) Weiser, M.: “Program Slicing”, *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439–449 (1981).
11) 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”, 信学論 (掲載手続中)