研究速報

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[†]（正員）

A Block Slicing Method Using Lightweight Dynamic Information

Tomonori TAKADA[†,††], *Nonmember* and Katsuro INOUE[†], *Regular Member*

[†] 大阪大学大学院情報科学研究科，豊中市
Graduate School of Information Science and Technology, Osaka University, Toyonaka-shi, 560–8531 Japan

[††] 日本電信電話株式会社，横須賀市
Nippon Telegraph and Telephone Corporation, Yokosuka-shi, 239–0847 Japan

あらまし　静的な制御依存情報と動的なデータ依存情報を利用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，更にオーバヘッドを削減することが求められている．本論文では，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，実験を通じて，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可能であることを確認した．

キーワード　準動的解析，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

1. まえがき

プログラムスライシング技術を用いることでソースプログラムのうちフォールトに関連する部分のみに注目することが可能である．プログラムスライス（スライス）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．スライスを利用することでデバッグを効率的に行うことができる [3]．スライシング技術は大きく静的スライシングと動的スライシングの二つに分類される．

Weiser [5] によって提案された静的スライスは，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．静的スライシングを計算するためには，プログラム依存グラフ（Program Dependence Graph, PDG）と呼ばれる有向グラフを用いる．PDG は文間の依存関係を表したグラフであり，その節点はプログラム中の文または条件式を，辺は制御依存関係（ある文の実行有無が条件式の実行結果に依存する関係）・データ依存関係（変数の定義–参照関係）を表している．

Agrawal ら [1], [2] によって提案された動的スライスは，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．

スライスを抽出するためには，実行前解析，実行時解析，スライス抽出の 3 段階の計算が必要となる．実行前解析時間はプログラムの長さとその構造に依存する．実行時解析時間は実行系列の長さに依存し，スライス抽出時間は，解析した依存関係の大きさに依存する．このため，繰返し等が多用されるプログラムにおいては，実行系列が大きくなり，実行時解析時間が非常に大きくなる．

デバッグ作業においては，動的スライシングが適しているが，上記の問題から，解析に必要な時間が非常に大きくなる．

この問題を解決するために，静的解析と動的解析を組み合わせた準動的解析手法が提案されている．依存キャッシュスライシング [6] は，このような準動的解析手法であり，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実行時解析時間を短縮するとともに，静的スライシングと比べスライスサイズを小さくすることができる．しかし，その実行時間（実行時解析時間とプログラム自体の実行時間の和）は静的スライシングの 3〜10 倍程度であり，現実のデバッグ環境においては，更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もとに，更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．

2. ブロック単位スライシング

2.1 概　　要

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するために，複数の文をブロックとして扱い，ブロック単位でデータ依存関係を求める．この手法をブロック単位スライシングと呼ぶ．

以下に，ブロック単位スライシングの計算手順を示す．

［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，制御依存辺のみをもちデータ依存辺をもたない PDG の部分グラフ PDG_{BL} を静的に生成する．

まず，**2.2** 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．

次に，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，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，**2.3** 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$_{BL}$ 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．

2.2 ブロック化アルゴリズム

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

このアルゴリズムでは，ブロック化因子 N に基づき，N 個の文を一つのブロックとしてブロック化を行う．ブロック化因子 N はユーザによって指定可能であり，N の値を変化させることにより，ユーザは任意の粒度でのブロック化が可能である．

2.3 データ依存関係収集アルゴリズム

図 2 に **2.1** 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．

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）された際，

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すべ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，関数境界は超えないものとする）
（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超える場合），ブロック境界を超えないような文を s_k とすると，$B_j := \{s_i, \cdots, s_k\}$，$i := k + 1$
（c） $BS := BS \cup B_j$

図 1 ブロック化アルゴリズム
Fig. 1 Algorithm for creating blocks.

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 2 データ依存関係収集アルゴリズム
Fig. 2 Algorithm for computing data dependence relations.

$C(v)$ 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（既に存在しなければ）追加する．一方，s で v が定義された際，

$C(v)$ は $B(s)$ 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ついては，依存キャッシュスライシングと同様，すべての要素に対してキャッシュを用意する．

3. 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及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搬することはない．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とも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（basic block）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，ユーザが特にブロック化因子を指定する必要のない場合に有用である．また，制御構造の境界を超えることがなくなり，無駄なブロックが少なくなると考えられる．ここで，基本ブロックとは，連続する分岐のない文の列のうち，可能な限り大きいものを指す．

4. 実　　験

4.1 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の測定を行った．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 [6] ため，C 言語で記述されたサンプルプログラムを対象とし，動的データ依存収集の動作を追加した．

サンプルプログラムとして，多様なプログラムで用いられるソートプログラムを対象とし，$P1$，$P2$ の二つのプログラムを用いた．対象プログラムとしては，プログラム長に対し実行系列が大きくなるもの，つまり繰返しを多く含むものとするため，ソートプログラムを選んだ．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，キーとデータの二つの構成要素をもつ構造体の配列に対し，キーをもとに整列を行う．

$P1$ に対して 10,000 の要素，$P2$ に対して 100,000 の要素のソートを行う処理について，元プログラム（実行時解析を行わない．静的スライシングの場合も含む），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，**3.** 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

4.2 考　　察

表 1 より，依存キャッシュスライシングでは元プログラム（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約 3〜6 倍程度となっている．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わかる．

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．PDG の節点数を表 2 に示す．

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ない場合，解析を行わない．これは，頻繁に呼び出さ

表 1　平均実行時間（秒）
Table 1　Average execution time (s).

	$P1$	$P2$
元プログラム	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 600 MHz CPU with 256 MB Memory)

表 2　PDG 節点数
Table 2　The number of nodes in PDG.

	$P1$	$P2$
ブロック化前	27	45
ブロック化後	20	34

表 3　実行時に解析した定義–参照関係数
Table 3　The number of analysed Def-Use relation during executing sample programs.

	$P1$	$P2$
依存キャッシュスライシング	2,300,000	20,000,000
ブロック単位スライシング	380,000	6,450,000

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．本実験では，$P2$ において，関数内局所変数に関するキャッシュの参照数を平均約 1,600,000 回省略することが可能であった．

スカラ型変数の解析については，ブロック単位スライシングでは実行前解析において解析しているため実行時に解析する必要がない．表 3 に，実行時に解析した定義–参照関係の数について示す．スカラ型変数の実行前解析により，大幅に実行時解析のコストを抑えることが可能であることがわかる．

5. むすび

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．本論文では，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，**4.2** で示したブロック内静的依存解析についても検討，及び，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 [4] 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．

文　献

[1] H. Agrawal and J. Horgan, “Dynamic program slicing,” SIGPLAN Notices, vol.25, no.6, pp.246–256, 1990.
[2] B. Korel and J. Laski, “Dynamic program slicing,” Inf. Process. Lett., vol.29, no.10, pp.155–163, 1988.
[3] 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.
[4] 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.
[5] M. Weiser, “Program slicing,” Proc. Fifth International Conference on Software Engineering, pp.439–449, 1981.
[6] 高田智規，井上克郎，芦田佳行，大畑文明，“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（D-I），vol.J85-D-I, no.2, pp.228–235, Feb. 2002.

（平成 14 年 6 月 17 日受付，9 月 17 日再受付）