Javaバイトコードの動的依存解析情報を用いたスライシングシステムの実現

Implementation of Bytecode-based Java Slicing System

梅森 文彰 † 誉田 謙二 †† 横森 励士 †† 井上 克郎 †
Fumiaki UMEMORI Kenji KONDA Reishi YOKOMORI Katsuro INOUE
† 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
†† 大阪大学大学院基礎工学研究科
Graduate School of Engineering Science, Osaka University
{umemori,inoue}@ist.osaka-u.ac.jp,
{konda,yokomori}@ics.es.osaka-u.ac.jp

デバッグを効率良く行なう手法の一つに，スライシングがある．一般にスライスの計算には文間の依存関係解析が前提となっており，解析方法によって静的スライシング，動的スライシングがある．我々の研究グループでは両者の手法を組み合わせた DC スライシングを提案している．本論文では，実行時決定要素を多く含む Java に対して有効な DC スライシングシステムを実現する．システムでは，Java バーチャルマシンの実行中に動的に依存関係を抽出することでバイトコードを単位とした細粒度のスライス抽出を実現した．

1 はじめに

プログラムデバッグを効率よく行う手法の一つに，**プログラムスライシング**(*Program Slicing*) がある．Weiser[6] によって提案されたプログラムスライシングによって，プログラム中のある文のある変数 v に関するスライスを抽出することで，v の値に影響を与える可能性のあるすべての文を抽出できる．一般に，スライスの計算にはプログラム文間の**依存関係**（*Dependence Relation*）の抽出が前提となっており，抽出の方法によって静的スライシング [5]，動的スライシング [3] などが存在する．我々の研究グループでは，静的な解析と動的な解析を組み合わせることで解析コストの削減を実現した DC スライシングを提案した [2, 8]．オブジェクト指向言語の利用の高まりに伴い，オブジェクト指向言語特有の概念を考慮したスライシング手法が提案されている．オブジェクト指向言語では，多くの実行時決定要素が含まれるため，DC スライシングを用いることで，低コストで高精度なスライスを抽出することができるが，実用的なシステムは実現されていない．

そこで本論文では Java を対象とした DC スライシングシステムを実現する．実現したシステムではバイトコードを単位とする依存関係解析を提案することでさらなる精度向上を実現している．実現したシステムは，バイトコード・ソースコード対応表出力を行う **Java コンパイラ**（*Java Compiler*），バイトコード間の動的データ依存関係解析を行う **Java バーチャルマシン**（*Java Virtual Machine*，以下 JVM），バイトコードを対象とする静的制御依存関係解析ツール，**プログラム依存グラフ**（*Program Dependence Graph*，以下 PDG）[4] に基づくスライサで構成される．バイトコードに対して計算された DC スライスを，コンパイラによって出力された対応表を利用することでソースコードに対応付ける．

以降，2 節でプログラムスライスおよび，今回実現するバイトコードを対象とした DC スライシングについて説明する．3 節で実現した DC スライシングシステムについて述べ，4 節で提案手法の有効性を従来のスライシング手法と比較し，最後に 5 節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

プログラムのある文のある変数に関連する文を抽出するための手法としてプログラムスライシング [6] が提案されている．**プログラムスライス**（*Program*

Slice，以下，スライス）の計算では，一般的に PDG を用いた手法が用いられる．PDG では，プログラムの変数間の情報を表す**データ依存関係**と実行制御に関する情報を表す**制御依存関係**を用いてプログラムに関する情報をグラフ化する．

PDG を用いてスライスを計算する場合の手順は以下の通りである．

Phase 1: 依存関係解析

各プログラム文に対し，
(a) **制御依存関係解析**
(b) **データ依存関係解析**
を行う．

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．PDG の節点はプログラム文やバイトコード命令など，スライス抽出における解析の粒度を表し，辺は節点間の依存関係を表す．

Phase 3: スライス計算

スライス基準 [6] 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を計算し，その節点集合に対応する文をスライスとする．

2.1 静的スライス

静的スライシング [6] は，PDG の節点をプログラム文として，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，現実には短い時間で計算される．しかし，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，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．

2.2 動的スライス

動的スライシング [5] は，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．特定の入力を与えてプログラムを実行させ，その実行系列における各**実行時点**（*Execution Point*）を節点として依存関係解析を行った上で PDG を構築し，スライスを計算する．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のみを考慮するため，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り，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，実行中に発生する各依存関係をすべて記憶する必要がある．そのため，入力データによっては実行系列が非常に大きくなり，多大な空間コストと時間コストを要する場合が存在する．

2.3 DC(Dependence-Cache) スライス

DC スライスを計算する際には [2, 7, 8]，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．これにより，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．また，データ依存関係解析において実行系列の保存を必要としないため，動的スライシングに比べ解析コストを抑えることができる．DC スライシングにおける手法として，PDG の節点をプログラム文とした手法が提案されているが，以下では，今回提案するバイトコードを対象とした DC スライシングについて説明する．

2.3.1 DC スライシングにおける制御依存関係解析

DC スライシングは，Phase 1(a) において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，本論文では，バイトコードの制御依存関係を次のように定義する．

この制御依存関係は，図 1 のアルゴリズムをメソッド単位で適用することで抽出できる．

定義

バイトコードの 2 命令 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が存在するという．

1. s は分岐命令である
2. t は s から直接到達する**基本ブロック**（*Basic Block*）[1] 内の命令である

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) バイトコードの命令列を基本ブロックに分割する ($\mathcal{N}$: 基本ブロックの集合)
(2) **foreach** n **in** $\mathcal{N}$ **begin**
(3) **if** n の最後の命令が分岐命令なら **then**
(4) n から制御が移動する基本ブロック集合 $\mathcal{N}'$ を導出
(5) **foreach** n' **in** $\mathcal{N}'$
(6) n の最後の命令と n' の各命令の対を制御依存関係として抽出する
(7) **end**

図 1: 静的制御依存関係解析アルゴリズム

2.3.2 DC スライシングにおけるデータ依存関係解析

DC スライシングでは，Phase 1(b) において JVM 上でバイトコードを実行し，それと並行して動的データ依存関係解析を行う．解析時には，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および JVM におけるスタックそれぞれに対してキャッシュを用意する．

JVM 上でバイトコードを実行中に各データの値が参照された場合，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する．また，値が定義された場合にはキャッシュの内容を実行中の命令に更新する．なお，あるデータ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．

動的データ依存関係解析アルゴリズムを図 2 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，各インスタンスごとに独立してデータ依存関係解析を行う．解析の例として，図 4 のバイトコードに対して動的データ依存関係解析を行なった場合のキャッシュの推移と抽出されるデータ依存関係を表 1 に示す．

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 2: 動的データ依存関係解析アルゴリズム

表 1: 図 4 のバイトコードにおけるキャッシュの推移

命令	ローカル変数 [0]	スタック [0]	スタック [1]	依存関係
1	-	1	-	
2	2	-	-	1 → 2
3	2	3	-	
4	2	3	4	2 → 4
5	2	-	-	3,4 → 5
6	6	-	-	2 → 6
7	6	-	-	
9	6	9	-	6 → 9
10	6	-	-	9 → 10

2.3.3 PDG 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．DC スライシングにおいて構築される PDG の例を図 4 に示す．DC スライシングでは依存関係辺に関する情報の抽出のみで十分であるため，実行系列を保存する必要はなく，動的スライシングに比べ解析コストは十分に小さい．

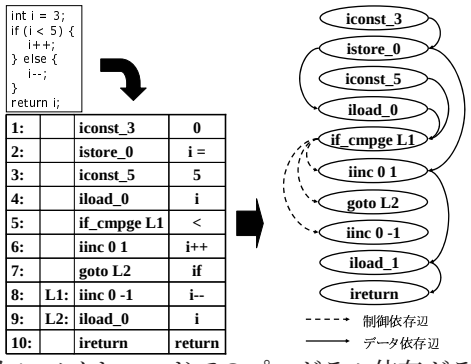

図 4: バイトコードでのプログラム依存グラフ

2.3.4 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．

2.4 手法の比較

静的スライシング，動的スライシング，DC スライシングにおける計算手法の違いを表 2 に示す．

表 2: 各スライシング手法の違い

	静的スライシング	DC スライシング	動的スライシング
CD 解析	静的	静的	動的
DD 解析	静的	動的	動的
PDG 節点	プログラム文	プログラム文 またはバイトコード	実行時点

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つことが知られている [7, 8].

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

各手法におけるスライス計算結果の例を，図 3 に示す．これは，解析対象プログラムのスライス基準 $<8, b>$ に対するスライスである．

なお，DC スライシングおよび動的スライシングにおいては，入力として変数 c に 2 を与えた実行を想定している．図 3 より，DC スライシングは，静的スライシングより高い精度のスライスを計算できる手法であることが分かる．

3 システム構成

本論文では，バイトコードを対象とした DC スライシング手法に対して，システムの実装を行った．実

```
プログラム                静的スライス              DC スライス               動的スライス
1:  a[1] = 1;             1:  a[1] = 1;             1:  a[1] = 1;             1:  a[1] = 1;
2:  a[2] = 2;             2:  a[2] = 2;             2:  a[2] = 2;             2:  a[2] = 2;
3:  a[3] = 3;             3:  a[3] = 3;             3:  a[3] = 3;             3:  a[3] = 3;
4:  read(c);              4:  read(c);              4:  read(c);              4:  read(c);
5:  while (c > 0) {       5:  while (c > 0) {       5:  while (c > 0) {       5:  while (c > 0) {
6:      b = a[c] + 1;     6:      b = a[c] + 1;     6:      b = a[c] + 1;     6:      b = a[c] + 1;
7:      c = c - 1;        7:      c = c - 1;        7:      c = c - 1;        7:      c = c - 1;
 :  }                      :  }                      :  }                      :  }
8:  write(b);             8:  write(b);             8:  write(b);             8:  write(b);
```

図 3: 各手法によるスライスの比較

現したシステムの構成を図 5 に示す．図 6 は実現したシステムのメインウィンドウである．

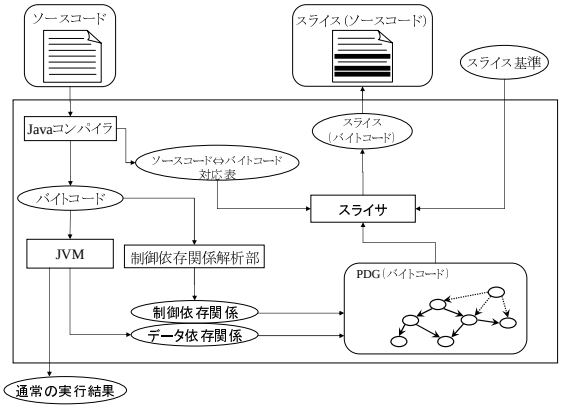

図 5: システム構成

解析においてはまず，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．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と節点とする 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 探索に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．

4 評価

本節では，実現した DC スライシングシステムに対する評価実験を行なう．実験では，表 3 のプログラムに対して，スライスサイズ及び解析コストに関する評価を行なった．

表 3: スライス対象プログラム

プログラム	クラス数	総行数
P1（データベースシステム）	4	262
P2（ソートアルゴリズム）	5	231

4.1 スライスサイズ

Java を対象とした静的スライシング，動的スライシング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．フォールト位置の特定を利用方法として考えた場合，得られたスライスのサイズが小さいほどフォールト位置を特定しやすく，精度が高いといえる．表 4 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．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．

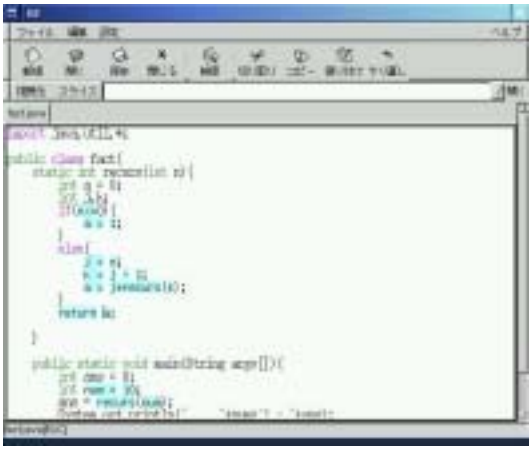

図 6: システムのメインウィンドウ

表 4: スライスサイズ [行]

	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	24	30
P1-スライス基準 2	19	14	15
P2-スライス基準 1	79	51	51
P2-スライス基準 2	27	23	25

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た．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．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るため，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．しかし，クラスの継承やメソッドのオーバーライド，オー

表 5: 解析コスト

プログラム	JVM 実行時間 [ms]		JVM 実行時使用メモリ量 [Kbytes]		PDG 節点数	PDG 構築時間および
	通常時	動的 DD 関係解析時	通常時	動的 DD 関係解析時		スライス計算時間 [ms]
P1	325	2,058	3,780	15,980	34,966	525
P2	341	3,089	4,178	26,091	34,956	450

バーロードがより多く含まれる大規模プログラムでは，静的スライスでは全ての場合を考慮する必要があるため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，実際に実行された継承やオーバーライド，オーバーロードのみを考慮するため，精度の低下は発生せず有効性が期待できる．

4.2 解析コストに関する考察

表 3 の各プログラムに対し，動的データ依存関係解析における実行時間およびメモリ計算量，PDG 節点数，PDG 構築およびスライス計算時間を測定した．表 5 は計測結果で，全く解析を行わずに通常実行を行う場合と比較して，多くの解析コストを必要とした．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，JDK ライブラリを含めた全てのバイトコードに対して動的データ依存関係解析を実行していることによると考えられる．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，表 6 で示すように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る．DC スライスが動的スライスと比較し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ことから，提案手法は非常に有用であるといえる．

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が挙げられる．しかし現段階でも十分実用的に動作するスライスシステムとして，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．

表 6: 構築される PDG の節点数

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
P1	34,966	1,198,596	1 : 34.3
P2	34,956	1,808,051	1 : 51.7

5 まとめ

本論文では，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，本来ソースコードに対して行われていた DC スライシングをバイトコードに適用することで細粒度の DC スライス計算を行う手法を提案した．提案手法では，各バイトコードを PDG の節点としてその PDG における依存関係を定義し，動的なデータ依存関係と静的な制御依存関係を抽出する手法を実現した．

また，提案手法をスライス計算ツールとして実現した．さらに，実現したシステムを用いて評価実験を行ない，提案手法により動的スライシングより格段に小さな解析コストで，静的スライシングより高い精度のスライスを得られることを確認した．

今後の課題としては，以下があげられる．

- JVM での動的データ依存関係解析の高速化
- 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

参考文献

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

[2] F.Ohata, K.Hirose and K.Inoue: “A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information”, Proceedings of Eighth Asia-Pacific Software Engineering Conference (APSEC2001), pp.273–280, Macau, China, December (2001).

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

[4] 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).

[5] 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).

[6] M.Weiser: “Program Slicing”, IEEE Transaction on Software Engineering, 10(4), pp.352–357 (1984).

[7] 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, Vol.J85-D-I, No.2, pp.228-235, 2002 年 2 月.

[8] Y.Ashida, F.Ohata and K.Inoue: “Slicing Methods Using Static and Dynamic Information”, Proceedings of the 6th Asia Pacific Software Engineering Conference (APSEC’99), pp.344–350, Takamatsu, Japan, December (1999).