動的コンパイラにおける実行時経路情報の構造的収集手法の提案

安江俊明[†] 菅沼俊夫[†]
小松秀昭[†] 中谷登志男[†]

本論文では，構造的パスプロファイル収集手法という，主に動的コンパイラを対象とした新しいパスプロファイル収集手法を提案する．パスプロファイル情報は，プログラムの実行挙動を把握するうえで非常に重要な情報の１つであるとともに，実行状態に適するようにプログラムの変形操作を実施するような最適化において非常に有用な情報となる．特に動的コンパイラではプログラムの実行中にプロファイルを収集し，かつ最適化に反映できるので，従来の静的コンパイラに比べてより進んだ最適化の実施が可能となる．しかしながら，従来の動的コンパイラでは，主にプロファイル収集コストが高いことから，パスプロファイル情報が利用されてこなかった．本論文で提案する構造的パスプロファイル収集手法では，パスプロファイル情報を，階層化された構造グラフを用いて収集することで，従来手法と同等の精度のパスプロファイルをより少ないオーバヘッドで収集することを可能とした．本手法を IBM Java Just-in-Time コンパイラに実装して評価したところ，プロファイルが頻繁に実施される状況においても，全実行を通じて収集したプロファイルと比較して約 90%の精度のパスプロファイル情報を，平均で 2〜3%のオーバヘッドで収集できることを示した．

Structural Path Profiling: An Efficient Online Path Profiling Framework for Dynamic Compilers

Toshiaki Yasue,[†] Toshio Suganuma,[†] Hideaki Komatsu[†]
and Toshio Nakatani[†]

This paper describes a new path profiling technique, structural path profiling, used in the dynamic compilers. A path profile is one of the important sources of information for understanding the application behavior and should be useful for restructuring and optimizing the target program for higher performance. However, dynamic compilers have not been able to take advantage of this information due to the high overhead for collecting path profile online. Our technique has the advantage that we can collect the path profiling information for the whole regions of the target method with low profiling overhead by using the structure graphs for path profiling. The experimental results show that our technique can achieve efficient path profiling with overhead of around 2-3% on average in the active profiling phase, while it provides the path profile for program optimizations with sufficient accuracy of around 90% compared to the offline complete path profiles.

1. はじめに

近年 Java 言語の普及にともなって，Java Just-in-Time コンパイラに代表される動的コンパイラが脚光を浴びている．動的コンパイラ[2),4),10),19),22)]は，実行中にプログラムをコンパイルするためにコンパイル時間が実行時間に含まれてしまう問題点を持つが，その反面，実行中のプログラムの実行環境や実行挙動などの情報を用いることができる点で，従来の静的コンパイラよりも優れた実行コードを生成できる可能性を持っている．実際，これらの動的コンパイラは，たとえば実行頻度の高いメソッド（ホットメソッド）の検出[2),4),10),19),22)]，仮想呼び出しの最適化[2),19),22)]，基本ブロックの並べ替え[2)]などの様々なプロファイル情報を利用した最適化を実装している．これらにより静的コンパイラよりも高速に実行されるプログラムも少なくない．

プログラムの実行挙動を求めるプロファイル手法として，ノードプロファイル（vertex profiling）手法とエッジプロファイル（edge profiling）手法[5)]がある．ノードプロファイル手法は各基本ブロックの実行頻度を，エッジプロファイル手法は制御フローエッジの実行頻度をそれぞれ収集し，分岐予測や基本ブロックの

† 日本アイ・ビー・エム株式会社東京基礎研究所
Tokyo Research Laboratory, IBM Japan, Ltd.

並べ替え，実行頻度の高い経路（ホットパス）の予測などのために用いられてきた．しかし，近年提案されている，データフロー解析を用いた最適化の性能向上のを目的としたプログラム変形[1),8)]やコード移動[13),14)]，実行頻度の高い経路の分離処理[24),25)]などの最適化では，複数の基本ブロック間にまたがる経路の実行状況を把握する必要があるために，ノードプロファイルやエッジプロファイルでは不十分で，より正確な実行挙動を得られるパスプロファイル（path profile）情報が必要となる．

パスプロファイル情報はプログラムの実行挙動を詳細に表現できるプロファイルの 1 つであり，プログラム中の有限長の経路に対する実行頻度として表現される．既存のパスプロファイル収集手法[6),25)]では，対象プログラム中に存在する有限長の実在経路を用いてパスプロファイル情報を収集する．本論文ではこのように実経路を用いて定義される経路を自然な経路と定義する．

自然な経路を用いたパスプロファイル収集手法では，各経路で収集される情報はその経路の実際の実行回数となる．パスプロファイル収集処理により増加する実行時間（以下，収集コスト）は収集される実行経路数に比例すると考えると，自然な経路を用いたパスプロファイル収集手法の収集コストはメソッド中で実行される経路の数に比例して増加する．つまり非常に実行回数の多いループがあるメソッドでは収集コストがループの実行回数に比例して爆発的に増大してしまう．もし収集コストを抑えるために，メソッドの実行の途中でパスプロファイルを終了すると，それ以降の部分のプロファイルが実施されず，結果として収集情報が不完全となり，プロファイル情報の精度が低下してしまう．この性質は動的コンパイラにおけるプロファイル手法として適さない．

一般に，動的コンパイラにおいて用いられるプロファイル手法は，収集される情報を用いて実施される最適化による速度短縮よりも少ない時間しか費やすことができない．収集情報の精度は最適化結果に影響することから，プロファイルに要する時間と収集情報の精度とをバランスさせながら，必要十分な精度の情報を少ない時間で収集することが必要である．

本論文では，この問題を解決する手法として構造的パスプロファイル収集手法（SPP: Structural Path Profiling）を提案する．SPP の基本アイデアは，プログラムを各ループネストレベルごとに分割し，それぞれのループネストに対して独立にパスプロファイル収集処理を実施することである．各ループネスト中に 1 つの 1 重ループしか存在しないように，内側ループを縮退させながらグラフを階層的に分割することで，各ループネストごとの収集コストが内側ループの実行状態と独立に決定できるようになる．つまり各ループネストごとに収集する実行経路の総数を適切に設定することで，少ないオーバヘッドで十分な精度のプロファイルを収集することが可能となる．

本論文で提案するフレームワークは，既存の 2 つのパスプロファイル収集手法のいずれも扱うことができるが，本論文では収集コストが軽い点から Ball と Larus によるパスプロファイル収集手法を用いて SPP を IBM Java Just-In-Time コンパイラに実装し，評価を行った．各メソッドで収集される実行経路数を固定した場合のプロファイル情報の精度を測定した結果，SPP は，収集経路数が 1000 の場合に 90%の精度を得られるとともに，Ball らの手法と動的インストルメンテーション手法[17),23)]を単純に組み合わせた手法に比べて少ない収集量で高い精度を出すことを示した．さらに実際の実行に対するオーバヘッドでも，非常に多くのメソッドに対してプロファイルが実施されている状況でも 2〜3%のオーバヘッドしか要しないことを示した．

本論文の主張点をまとめると以下の 2 点となる．

- 動的コンパイラのための新しいパスプロファイルフレームワークとして構造的パスプロファイル収集手法を提案した．
- 提案手法の効果を評価し，その有効性を示した．

以下では，まず 2 章においてパスプロファイル情報の有効性を述べる．続いて 3 章で動的コンパイラにおいてパスプロファイル収集手法が満たすべき要件について示し，この要件を満たす新手法である構造的パスプロファイル収集手法を 4 章で説明する．続いて 5 章において提案手法を IBM Java Just-In-Time コンパイラに実装して行った評価結果を示す．その後，6 章で関連研究を解説し，最後に 7 章で本論文の結論を示す．

2. パスプロファイル情報の有効性

従来より，エッジプロファイル情報を用いてメソッド中で実行頻度の高い経路（ホットパス）を求める手法[7)]が提案されている．本章では，このエッジプロファイルを基にした手法に対するパスプロファイル情報の優位性について述べる．

ホットパスを検出するという目的では，従来よりエッジプロファイル情報に基づく手法で必要十分な効果をあげられることが示されている．たとえば Ball らは，

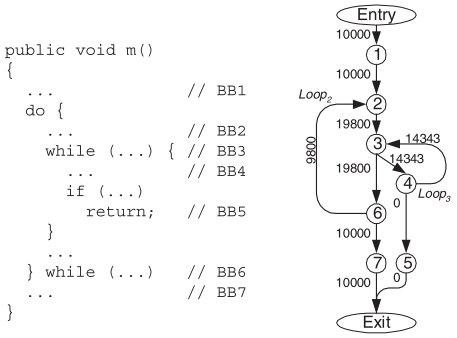
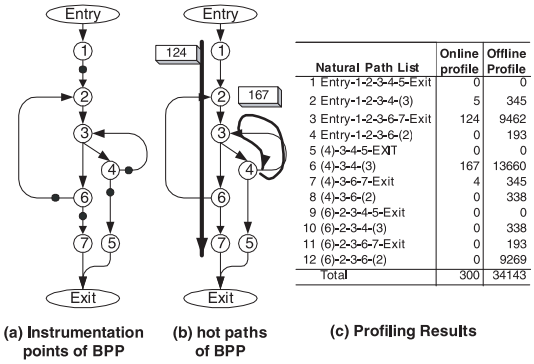


図 1 2 重ループを持つプログラムとそのエッジプロファイルの例
Fig. 1 Example program with a nested loop and its corresponding control flow graph with edge profiles.



Natural Path List	Online profile	Offline Profile
1 Entry-1-2-3-4-5-Exit	0	0
2 Entry-1-2-3-4-(3)	5	345
3 Entry-1-2-3-6-7-Exit	124	9462
4 Entry-1-2-3-6-(2)	0	193
5 (4)-3-4-5-EXIT	0	0
6 (4)-3-4-(3)	167	13660
7 (4)-3-6-7-Exit	4	345
8 (4)-3-6-(2)	0	338
9 (6)-2-3-4-5-Exit	0	0
10 (6)-2-3-4-(3)	0	338
11 (6)-2-3-6-7-Exit	0	193
12 (6)-2-3-6-(2)	0	9269
Total	300	34143

図 2 BPP によるプロファイルの例
Fig. 2 Example of BPP.

エッジプロファイル情報からホットパスを予測する手法[7]を提案している．確かに数値計算系のプログラムでは予測適中率が高いが，これは実行経路のばらつきが少なくいためであり，実行経路が広く分散する非数値計算系のプログラムでは適中率が下がってしまう傾向にある．

また，変形操作を実施する場合，個々の経路の実行頻度を識別する必要があるために，エッジプロファイル情報では不十分である．たとえば，**図 1** の例では，2 重ループを持つプログラムに対するエッジプロファイル情報を示している．エッジプロファイル情報から，外側のループは平均で約 2 回実行されると予測される．しかし後述のパスプロファイル情報から，このループは実行の大半はループを 1 回しか実行しない反面，一度繰り返されると多数回実行される．この予測結果の違いは最適化結果に大きく影響する．

他方，Ball と Larus のパスプロファイル収集手法[6]では，その収集コストは，経路数が増大した場合に用いられるハッシュ手法を使用しない場合にはパスプロファイル処理の収集コストがエッジプロファイルと同等かそれよりも少ないコストになることが報告されている．コスト以外の観点ではパスプロファイル処理に必要なバッファの増大の問題が残るが，この問題もパスプロファイル処理を選択的に適用することでプログラムの実行に重大な影響をあたえる状況を回避できる．

以上の点から，収集コストについてほとんど差がなく，情報量の多い点で，あえてエッジプロファイル情報を利用するよりもパスプロファイル情報を利用する方が最適化にとって有利であるといえる．

3. 解決すべき課題

動的コンパイラにおけるパスプロファイル収集手法で満たすべき要件は，少ない収集コストで必要十分な精度のプロファイルを収集することである．

たとえば，**図 2** では，Ball-Larus の手法に対して動的インストルメンテーション手法を適用して指定した数の経路実行数を収集した時点でプロファイルを終了するように実装した手法（BPP: Baseline Path Profiling）を用いて，300 の実行経路数を収集した場合の例を示している．図 2 (c) のプロファイル結果から分かるとおり，BPP では本来ホットパスとして検出されるべき経路 12 がまったく検出できていない．これは，経路 3 と経路 6 で示される経路が先行して実行されるために経路 12 の実行が始まる前にパスプロファイル処理が終了してしまったことが原因である．しかしこのような状況を避けるためにプロファイル期間を長くとると，非常に多くの収集コストが発生し，許容される以上のオーバヘッドを生じてしまう．

このような問題が生じる理由は，従来のパスプロファイル収集手法が自然な経路を用いて経路情報を収集することに起因する．正確なパスプロファイル情報を得るためにはメソッドの実行途中でパスプロファイル処理を止めることができずに収集コストの増大を招き，逆に収集コストを低減するためにメソッドの実行途中でプロファイルを終了すると，終了時点以降のプロファイルが収集されず，プロファイル情報の精度の低下を引き起こしてしまう．

プロファイル収集のオーバヘッドを軽減する従来手法として，Arnord らの Instrumentation sampling 手法[3]を利用して BPP を実施する方法が考えられる．確かに彼らの手法を用いることで前述の問題を緩和することは可能である．しかし彼らの方法では，オーバ

ヘッドとプロファイルの精度をサンプリング・レートにより制御するため，サンプリング・レートをすべてのループに適した値に設定することは困難である．実際，サンプリング・レートが低いとプロファイル収集時間が長くなり，最適化コンパイルの機会を遅延させてしまう．一方，サンプリング・レートが高いと前述の問題が生じてしまい，プロファイルの精度が落ちてしまう．

4. 構造的パスプロファイル収集手法

本章では，動的コンパイラにおいて前章の問題を解決する新しいパスプロファイル収集手法として構造的パスプロファイル収集手法（SPP）を提案する．SPPでは，まずコンパイル時にメソッド内のプログラムからループネストごとに階層化したグラフ（構造グラフ）を作成し，各構造グラフごとにインストルメンテーション命令を生成する．続く実行時にはインストルメンテーション命令を各構造グラフを単位として制御することで各構造グラフに対応するパスプロファイル情報を収集する．最後に収集したパスプロファイル情報を補正して処理を完了する．

構造グラフを用いる利点は，各グラフでは内側ループがすべて縮退されており，最大で1つの1重ループしか存在しないことである．つまり，内側ループの実行と独立にパスプロファイル情報を収集できるので，そのグラフ中のループの実行回数に無関係に収集する実行経路数を決めてもグラフ内の全域に対して実際の実行頻度に比例したパスプロファイル情報を収集できることを保証できる．このため，各構造グラフに対して適切にプロファイル量を設定することで，少ないオーバヘッドで十分な精度のプロファイルを収集することが可能となる．

以下では，まず構造グラフを定義した後，SPPの各処理について説明する．続いて収集されたパスプロファイルを用いた最適化の例を示した後，構造グラフ上で定義される構造的経路について考察する．

4.1 構造グラフの定義

構造グラフは，始点ノード，終点ノート，基本ブロックを表すノード，および強連結領域を表すループノードと，それらの間の制御フロー関係を表すエッジにより構成される．**図3**は，図1で示されるプログラムを構造グラフで表したものである．図中の太線のノードはループノードであり，点線のエッジは仮想エッジである．仮想エッジはもともとの制御フローグラフには存在しないエッジであるが，ループへ出入りするエッジに対応しており，そのラベルで対応する元のエッジが示されている．構造グラフは外郭構造グラフとループ構造グラフがある．

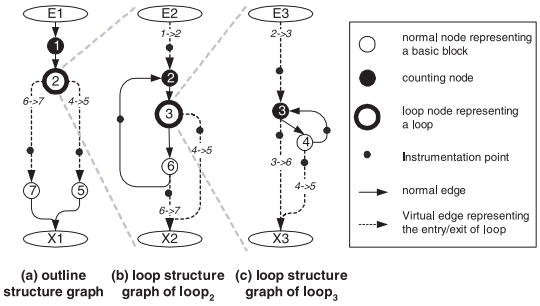


図3 構造グラフの例
Fig. 3 Example of structure graph.

外郭構造グラフは元の制御フローグラフ中のすべての強連結領域をループノードに縮退したグラフとして定義される．また，メソッドの開始基本ブロックにあたるノードを外郭構造グラフに対するカウントノードと定義する．もしこのノードが強連結領域内に位置する場合は，その強連結領域を縮退したループノードをカウントノードとする．図3(a)が外郭構造グラフの例である．

ループ構造グラフは，その内側ループをすべてループノードに縮退させた強連結領域として定義される．また，その強連結領域を構成するループのヘッダノードをループ構造グラフのカウントノードと定義する．図3(b)がノード2をヘッダノードとするループネストに対応するループ構造グラフを表し，図3(c)がノード3をヘッダノードとするループネストに対応するループ構造グラフを表している．構造グラフの階層構造関係は，その内包関係により定義される．

4.2 構造グラフの生成

SPPでは，まず対象メソッドの制御フローグラフから構造グラフを生成する．**図4**に構造グラフの生成アルゴリズムを示す．メソッド m を引数として関数 $makeStrGraph$ を呼ぶことでメソッド m に対する階層化された構造グラフを構築する．まず関数 $genCFG$ によりメソッド m に対する制御フローグラフを作成する．次に関数 $naturalLoopDetection$ により自然なループを検出する．処理中にループを形成するバックエッジを検出したら，バックエッジに識別フラグを設定するとともに，ターゲットノードであるヘッダノードにループ番号を1から順に設定する．ループが存在する場合は関数 $makeLoopStrGraph$ で各ループネストごとのループ構造グラフとそれらの間の階層構造を構築する．関数 $makeLoopStrGraph$ では，まず関数 $makdDfsList$ を用いて，制御フローグラフ

```
makeStrGraph (Method m) {
 og = genCFG (m);
 n_backedge = naturalLoopDetection (g);
 if (n_backedge > 0) {
   makeLoopStrGraph (m, g);
 }
}

makeLoopStrGraph (Method m, Graph osg) {
 dfsList = makeDfsList (osg);
 // find loop region
 for (dfs_index = 1 to numNode (g)) {
   n = dfsList (dfs_index);
   for (e in incomming edges of n)
     if (e is a loop backedge) {
       hdr = targetNode (e);
       src = sourceNode (e);
       if (loopIndex (src) != loopIndex (src)) {
         clearTraversedFlagsOfAllNodes (g);
         traverseLoopRegion (hdr, src);
       }
     }
 }
 // find loop exit edges
 for (lsg in loop structure greaphs in m)
   for (n in lsg)
     for (e in the outgoing edges of n)
       if (targetNode (e) is outside lsg)
         registerExitEdge (lsg, e);
 // make virtual edges
 for (g in depth-first-post-order of
      the structure graphs in m) {
   for (lsg in innerLoopStrGraph (g))
     setLoopEntryEdges (lsg, g);
   for (lsg in innerLoopStrGraph (g))
     setLoopExitEdges (lsg, g);
   if (g is loop structure graph)
     setEntryAndExitEdges (g);
 }
}

traverseLoopRegion (Node hdr, Node n) {
 setTraversedFlag (n);
 if (loopIndex (n) > 0)
   constructHierarchy (g, strGraph (n));
 else
   loopIndex (n) = loopIndex (hdr);
 for (e in incomming edges of node) {
   src = srcNode (e);
   if (dfsIndex (src) < dfsIndex (hdr))
     registerEntryEdge (g, e);
   else if (src is not traversed &&
     loopIndex (src) == loopIndex (hdr))
     traverseLoopRegion (hdr, src);
 }
}
```

図 4 構造グラフの生成アルゴリズム
Fig. 4 Algorithm to construct structure graphs.

中のノードを深さ優先帰りがけ順の逆順に並べたリスト *dfsList* を作成する．続いてこのリスト順にノードをたどりながら関数 *traverseLoopRegion* を用いてループ領域を決定する．同関数中の dfsIndex の比較は既約ループ（irreducible loop）におけるヘッダノード以外のノードへ入るエッジを検出するための処理である．この dfsIndex を用いてループ領域を検出する方法は，基本的には Havlak のループ検出手法[15]と同様の手法である．次に，ループの出口エッジを検出した後，ループノードにつながるエッジと，各構造グラフの入り口エッジ，出口エッジを構築する．

本アルゴリズムでは，ループ後方分岐からヘッダノードを決定し，ヘッダノードごとにループ構造グラフを作成する．プログラムの中には複数のネストレベルで 1 つのヘッダノードを共有する多重ループが存在する．しかし制御フローグラフ上では異なるネストかどうかを判断する手段がないため，このようなプログラムに本アルゴリズムを適用すると，これらのループネストは 1 つのループとして扱われてしまう．このようなループを識別できる場合には，本アルゴリズムを適用する前に，ヘッダノードの前に空のノードを生成し，外側ループの後方分岐をそのノードに向けるような変形を施す処理を実施することで対処できる．

4.3 インストルメンテーション命令の生成

続いて制御フローグラフから生成された構造グラフを用いて対象メソッドにインストルメンテーション処理を挿入する．まず各構造グラフに対して Ball-Larus のパスプロファイル収集手法を適用して，各グラフに対するパスプロファイル処理のためのインストルメンテーション情報を生成する．たとえば．図 3 (a)–(c) ではエッジ上にある黒点がインストルメンテーションポイントを示している．

次に各構造グラフごとのプロファイル期間を制御するためのエントリーカウンタの情報をカウントノードに生成する．ループが既約ループ☆の場合はさらにループ外からカウントノード以外のノードへ入るエントリーエッジに対しても同様のカウンタを生成する．既約ループの場合でも，繰返し以降の経路は必ずカウントノードを通過するので，カウントノードの実行回数に加えて外部からのエッジの実行回数をカウントすることで全経路数をカウントすることができる．

最後に各構造グラフのインストルメンテーション情報を，メソッドの元の制御フローグラフ上の対応位置

☆ Java では例外ハンドラや JSR によりしばしば既約ループが形成される．

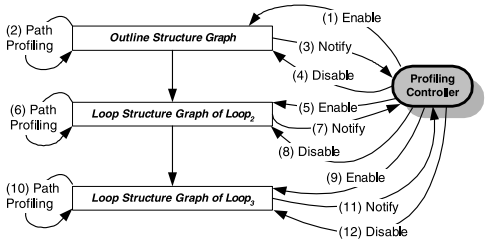


図 5 プロファイル制御機構
Fig. 5 Profile controller.

にインストルメンテーション命令として挿入する．このとき，仮想エッジ上のインストルメンテーション情報はそのラベルで示されるエッジに対して挿入が行われる．また，エッジ上に割り当てられているインストルメンテーション情報は可能な限りその前後の基本ブロックに移動して挿入し，移動できない場合は新しい基本ブロックを作成して挿入する．

その後，実行コード生成時にインストルメンテーション命令に対する実行コードを生成する．この実行コードはメソッドのコードとは別の場所に生成しておき，実行時に動的インストルメンテーション手法を用いて各挿入点の命令を無条件分岐命令に書き換えることで実行状態の切替えを行えるようにしておく．同時に，実行時に各構造グラフのパスプロファイル処理を制御するためにグラフの階層構造の情報も生成する．

Ball-Larus の手法では，3 種類のインストルメンテーション（経路番号の初期化，経路番号の更新，経路カウンタの更新）を経路上に配置し，実行にともなってこれらを順に実行することで経路情報を収集する．このため，プロファイル開始時点ですでに実行中のコンテクストが経路上に存在すると，経路番号の初期化が行われないために不正な経路番号が経路カウンタに到達してしまう．このため経路カウンタの更新では，経路番号が経路数内であるかどうかを検査する処理を用いて不正な値でのカウンタの更新を回避する．不正な経路番号が偶然経路数内の値をとることもあるが，これは最大でも同時に実行されるスレッド数以下であり，ほとんどの場合に無視できる．

4.4 プロファイル情報の収集

パスプロファイル情報は，生成したコードを用いてメソッドを実行することで収集される．構造的パスプロファイル収集処理は，各構造グラフに対するパスプロファイル収集処理をプロファイル制御機構で制御しながら進行する．**図 5** に，図 3 の例に対するプロファイル制御の例を示す．図中の数字は処理順序を表している．まずタイマー・サンプリング・プロファイラに

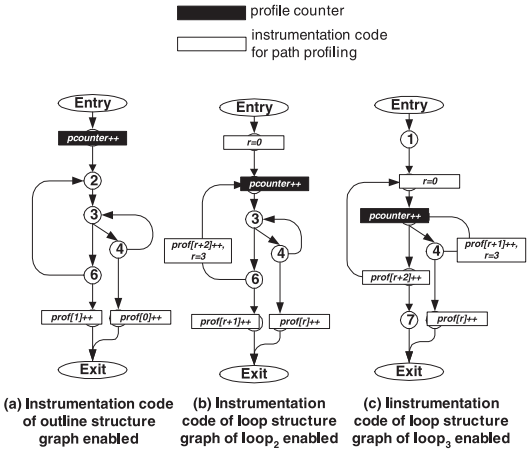


図 6 インストルメンテーション処理の例
Fig. 6 Examples of instrumentation code.

より検出されたホットメソッドの外郭構造グラフに対して開始される．そのエントリーカウンタが閾値に達した時点で外郭構造グラフに対するパスプロファイル処理を停止し，構造グラフの階層構造に従って，下階層へ向かって順にパスプロファイル処理を実施する．すべての構造グラフに対してパスプロファイルの収集が終了した時点で，メソッドに対するパスプロファイル処理が完了する．**図 6** (a)–(c) は，各構造グラフに対してインストルメンテーションを有効にした状態を示しており，図 5 の (2)，(6)，(10) において有効化されるインストルメンテーション処理にそれぞれ対応している．

4.5 プロファイルの補正

最後に収集した各構造グラフのパスプロファイル情報を補正して，実際の実行頻度に相当するプロファイルを生成する．補正方法は次のとおりである．ある構造グラフにおけるパスプロファイル情報は，そのループの外からループへ入る経路の実行頻度に対する相対的な頻度と考えることができる．したがって，ループ外からループへ入る経路の実行頻度と，上位層の構造グラフ中にあるループノードの実行回数との比（補正係数）を計算し，この値をその構造グラフの各パスプロファイル情報の値に掛け合わせることで，この 2 つの構造グラフ間のパスプロファイル情報の値が対等な関係となる．以下に，ループ構造グラフ X の補正係数 A_X の計算式を示す．ここで，C_p は，経路 p に対して収集された実行回数のプロファイル値，$P_X(a)$ は構造グラフ X 中のノード a を通る経路の集合，$P_X(entry)$ は構造グラフ X においてループ外からループへ入る経路の集合，Y は X の上位層の構造グラフ，N_X は構造グラフ X に対応する上位階層

Graph	idx	Path List	Online Profile: Local Profile	Online Profile: Global Profile	Offline Profile	index of BPP
Outline Structure Graph	1	E1-1-2-7-1	100	100.0	10000	3,(2,4,7,11)
	2	E1-1-2-5-1	0	0.0	0	1,(2,4,5,9)
Loop Structure Graph of Loop2	3	E2-2-3-X2	0	0.0	0	1,(5)
	4	E2-2-3-6-X2	50	98.0	9800	3,(2,7)
	5	E2-2-3-6-(2)	1	2.0	200	4,(8)
	6	(6)-2-3-X2	0	0.0	0	9,(5)
coefficient	7	(6)-2-3-6-X2	1	2.0	200	11,(7)
1.96	8	(6)-2-3-6-(2)	48	94.1	9600	12,(8)
Loop Structure Graph of Loop3	9	E3-3-4-X3	0	0.0	0	1,9
	10	E3-3-4-(3)	2	6.8	683	2,10
	11	E3-3-X3	56	189.3	19117	3,4,11,12
	12	(4)-3-4-X3	0	0.0	0	5
coefficient	13	(4)-3-4-(3)	40	135.2	13660	6
3.38	14	(4)-3-X3	2	6.8	683	7,8
Total			300	634.1	63943	

(a) Profiling Result

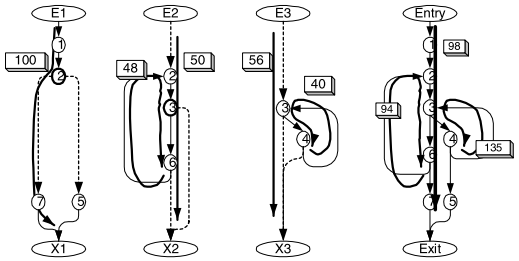


図 7 SPP の例
Fig. 7 Example of SPP.

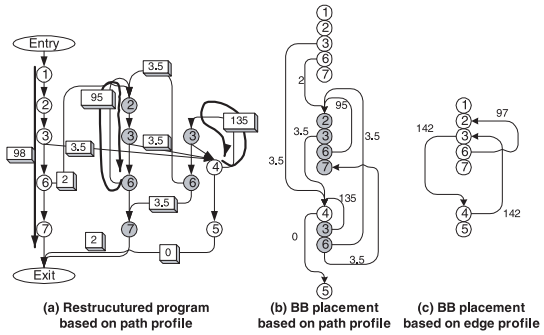


図 8 パスプロファイル情報を用いた最適化の例
Fig. 8 Example of program restructuring using path profiles.

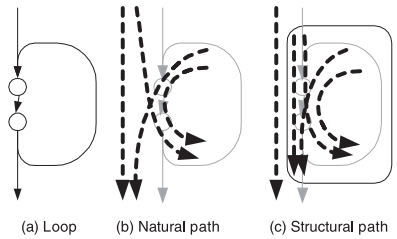


図 9 自然な経路と構造的経路の違いの例
Fig. 9 Example of natural path and structural path.

の構造グラフ中のループノード，をそれぞれ表すものとする．

$$A_X = \begin{cases} 1 & \text{if } X \text{ is outline} \\ \dfrac{A_Y \cdot \sum_{q \in P_Y(N_X)} C_q}{\sum_{p \in P_X(entry)} C_p} & \text{otherwise} \end{cases}$$

補正処理の例として，**図 7** (a) に収集したプロファイルとその補正結果を示す．表の各欄は，各構造グラフごとに，その経路リスト（Structural Path List），収集したプロファイル（Local Profile），補正したプロファイル（Global Profile），プログラムの全実行を通じて収集した場合のプロファイル（Offline Profile），および対応する自然な経路の番号（NP idx），を示している．自然な経路の番号でカッコに入っている番号は経路の一部分が対応していることを示している．Loop2 では，補正係数が $(1 * 100)/51 (= 1.96)$ となるので，各経路のプロファイル値に 1.96 をかけて補正値を得る．Loop1 では補正係数 $(1.96 * 100)/58 (= 3.38)$ を各経路のプロファイル値にかけて補正値を計算している．本例では，補正値の比率がプログラムの全実行に対するプロファイルの比率に非常に近いことが分かる．得られたプロファイルから得られるホットパスをグラフ上に太線で示したものが図 7 (b)–(e) である．

4.6 パスプロファイル情報を用いた最適化の例

収集したパスプロファイル情報を用いてプログラムを最適化する例を**図 8** に示す．まず図 7 (a) の構造的パスプロファイル情報からループを繰り返さない経路 $(4, 11)$ とループを繰り返す経路 $(8, 13)$ がそれぞれ独立したホットパスであることが分かり，Loop Peeling を適用してこれらのホットパスを分離する（図 8 (a)）．次に，変換後のパスプロファイル情報から生成したエッジプロファイルを用いて Pettis と Hansen の基本ブロックの並べ替えアルゴリズム[20]を用いてホットパスが連続になるように配置する（図 8 (b)）．図 8 (c) はエッジプロファイルを用いて基本ブロックの並べ替えのみを行った結果である．(b) と (c) の比較により次の 3 つの点でプログラムの実行速度が改善されることが分かる．第 1 点は分岐命令の分岐頻度が特定の方向に大きく偏る（ノード 3 とノード 6）ことで，ハードウェアの分岐予測の適中率が向上する．第 2 点は独立した 3 つの Hot Path がそれぞれ連続になるように配置することで，I–キャッシュ効率が上がる．第 3 点は無条件分岐や前方分岐の数が激減している．

4.7 構造的経路の特徴

ここで制御フローグラフ上の実経路を用いて定義される経路（自然な経路）と構造グラフ上の経路を用いて定義される経路（構造的経路）の差異について議論

する．2つの経路の定義の違いは，**図 9** (b)，(c) で示されるようにループにまたがる経路の扱いである．自然な経路では，ループの直前までの経路とループの 1 回目の実行の経路，ループの最後の実行とループの直後からの経路，およびループの直前の経路からループを繰り返さずにループの直後へつながるの経路，という 3 種類の経路がループにかかわる経路である．一方構造的経路では，ループの外の経路とループの中の経路の連続性はまったくない．しかし，ループの直前までの経路とループの直後以降の経路はつねに連続の経路となるとともに，4.6 節で示したように Loop Peeling などのループの変形に必要な情報は維持している．プログラムのホットパスが主にループで得られることから，ループの内外での経路の関連性よりもループの同じネストレベルのコード間の振舞いの関連の方が最適化には有用である場合が多く，またループの前後の経路の関連を必ず収集できる点から，構造的経路の方が優位であるといえる．

5. 評　　価

本章では SPP を IBM Java Just-In-Time コンパイラ上に実装して実施した評価結果を示す．

5.1 評 価 方 法

すべての評価は，IBM Aptiva A-series 6832（Pentium 4 2.0 GHz Uniprocessor 512 MB）で動作する Windows 2000 SP2 上で測定した．Java JVM は，IBM Developer Kit for Windows，Java Technology Edition version 1.3.1 prototype build を使用した．評価に用いたベンチマークは SPECjvm98 1.04 [21] を interactive mode，default input size，初期ヒープサイズ 256 MB，最大ヒープサイズ 256MB で実行した．

SPP では，メソッド中の各構造グラフに対して，指定された閾値をグラフ数で割った値分だけ経路情報を収集した．たとえば構造グラフが 4 つあるメソッドで，閾値 1000 が指定されている場合，各構造グラフではそれぞれ 250 の経路情報を収集し，メソッド全体で 1000 の経路を収集する．

比較対象として，Ball-Larus の手法を動的に適用するように変更した方法（BPP）を使用する．BPP は対象メソッドに対して，Ball-Larus の手法に対して各インストルメンテーション処理を動的インストルメンテーション手法で実装する．さらにインストルメンテーションコードのうちの各経路カウンタの更新処理に対して，総経路数をカウントする処理を付加し，総経路数が閾値に達した時点でプロファイルを終了する．

SPP，BPP のいずれも，タイマー・サンプリング・プロファイラにより検出されたホットメソッドで，かつ経路数が 1000 以内のメソッドのみにプロファイルを実施する．ホットメソッドであるが経路数が 1000 を超えるメソッドに対してはエッジプロファイル処理を実施した．エッジプロファイルを実施したメソッドは SPECjvm98 ベンチマーク全体で数個程度であった．

表 1 経路数とインストルメンテーションポイント数の比較
Table 1 Size of paths and instrumentation points.

	Total num. of Paths		*Total num. of Inst.*	
	SPP	BPP	SPP	BPP
mtrt	2,617	3,370	909	807
jess	4,332	5,534	1,642	1,420
compress	2,142	2,829	627	560
db	2,324	2,943	755	671
mpegaudio	2,673	3,255	1,021	879
jack	4,379	5,197	1,365	1,224
javac	12,234	13,563	4,376	3,780
GM	3,616	4,469	1,232	1,083

5.2 静 的 情 報

ここではメモリの使用量の評価として，SPP と BPP の静的な経路数とインストルメンテーションポイント数の比較を**表 1** に示す．最初の欄は，SPP と BPP の各ベンチマークにおける静的な経路の総数を，次の欄は SPP と BPP の各ベンチマークにおけるインストルメンテーションポイントの総数を示している．プロファイルに必要なバッファの大きさは，経路数が多いほど大きくなる．インストルメンテーションポイントの数は実行コードサイズに影響する．メソッドによっては構造グラフに分解することで経路数が増加する場合が存在するが，評価結果より SPP の経路数は BPP に対して平均で約 20%減少することが分かる．経路数は分岐の数に指数関数的に影響するために，グラフを分割することで経路を減少させていることが分かる．一方 SPP のインストルメンテーションポイント数は BPP に対して約 14%増加している．これは，グラフが分割されることで主に経路番号の初期化処理が増加したことが主な原因である．

5.3 精　　度

収集されたパスプロファイル情報の精度を評価する方法として，収集したプロファイルにおける各経路の実行頻度の比率が，プログラムの全実行を通じて収集した場合の比率とどの程度類似しているかという点で比較する．類似度の評価方法として，overlap percentage という評価手法[12]を用いた．overlap percentage は，各項の比率で表される 2 つの多項分布に対して重なる領域の面積を計算することでそれらの類似度を求める手法である．各パスプロファイルの大小関係や頻

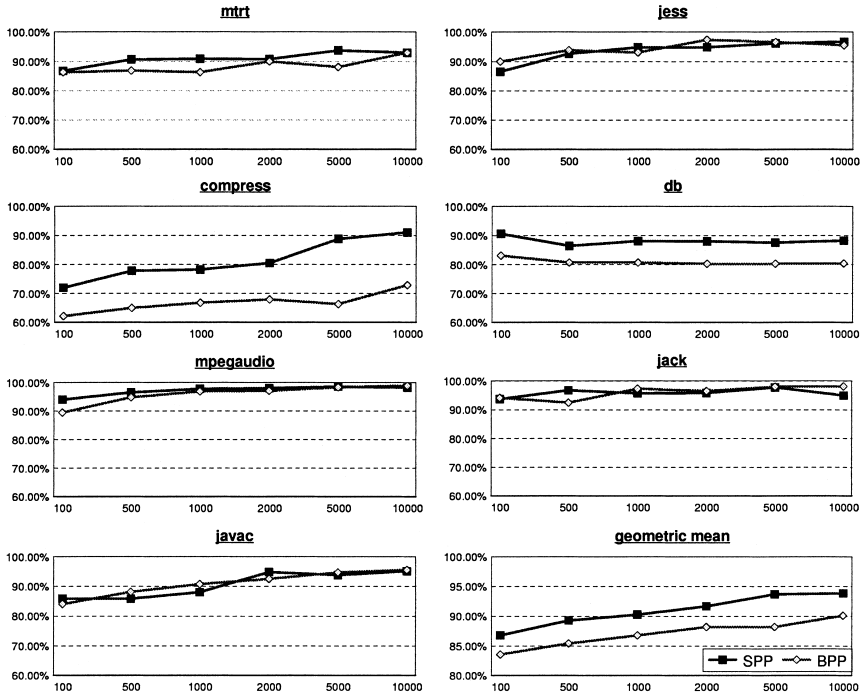


図 10 Overlap Percentage による各閾値での SPP と BPP の精度の比較
Fig. 10 Comparison of accuracy (overlap percentage) for each threshold of profile counts.

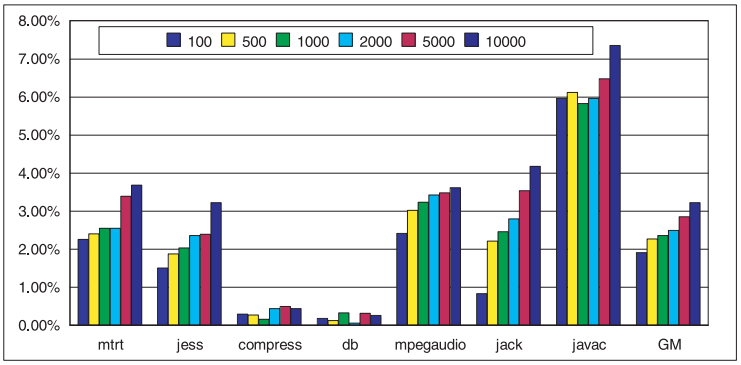


図 11 SPECjvm98 の初回実行における SPP のオーバヘッド
Fig. 11 Performance impact of SPP in the first run of SPECjvm98 benchmarks (smaller is better).

度の低い部分の類似性も算定に加えられるため，最適化を目的としたパスプロファイルの精度を評価するうえで適した評価尺度の 1 つといえる．たとえば図 7 (a) で示される SPP の overlap percentage は 99.7%であり，図 2 (c) で示される BPP の overlap percentage は 69.7%となる．

図 10 は，収集する実行経路数の閾値を 100 から 10000 まで変化させた場合の，overlap percentage による SPP と BPP の精度を表している．評価結果より，compress と db は全域にかけて SPP の精度が BPP の精度を上回っていることが分かる．これは BPP がいくつかのホットパスをプロファイルし損ねているためである．また，mpegaudio では SPP は少ない閾値でも高い精度を出していることが分かる．全体としてみると，SPP は閾値 1000 で 90%の精度を出しているが，同じ精度を出すために BPP では閾値を 10000 にしなければならないことが分かる．パスプロファイル情報の収集コストは同じ閾値の場合は SPP も BPP もほとんど変わらないので，SPP は BPP よりも少ないオーバヘッドで高い精度のプロファイルを収集できることが分かる．

5.4 オーバヘッド

SPP が実際の実行に与えるオーバヘッドとして，実行速度の低下を閾値を 100 から 10000 まで変化させた場合について測定した結果を**図 11** に示す．ここで SPP のみを評価しているのは，SPP と BPP は各メソッドに対して同数の実行経路数を収集するので，同じ閾値に対する実行時オーバヘッドはほとんど変わらないためである．またコンパイル時間の差についても，SPP は BPP に比べてループの検出と構造グラフの生成処理を余分に要するが，基本的には線形時間の処理でありコンパイル時間全体に対する影響の差は無視できる．

図 11 の各グラフは，各ベンチマークを 1 回だけ実行したときに，プロファイルを実施しない場合に対するプロファイルを実施した場合の速度低下率を表している．つまり，この速度低下率は，インストルメンテーションコード生成にかかる時間，実行中のパスプロファイル収集処理にかかる時間，およびパスプロファイルの実施により生じる最適化のための再コンパイルの遅延による損失時間など，SPP の実施にともなうすべてのオーバヘッドが含まれた値を示している．また 1 回目の実行を測定するのは，1 回目の実行では多数のメソッドに対してプロファイルが実施されるため，プロファイルに対する速度低下の影響が著しく現れるためである．実行の回を重ねて定常状態に達した時点では，ほとんどのホットメソッドは際コンパイルされた状態となり，新たなプロファイル処理が発生しないため SPP によるオーバヘッドは発生しない．

compress と db はホットメソッドが 10 個程度しか存在しないために，ほとんどオーバヘッドが現れていない．一方，javac は約 60 個のホットメソッドに対してプロファイルが実施されるために，速度遅延が大きく現れている．平均で見ると，プロファイルが頻繁に起こる状況下で 2〜3%のオーバヘッドと非常に少ないオーバヘッドであることを示している．

6. 関 連 研 究

これまでに，静的コンパイラのためのパスプロファイル収集手法として，2 つの手法が提案されている．1 つは Ball と Larus による手法[6]であり，もう 1 つは Young と Smith による手法[25]である．Ball-Larus の手法は，ループを含まない経路に対するパスプロファイル処理を高速に実行する手法である．各分岐エッジに割り当てた経路番号を計算するための定数値を，グラフ中の木に対する弦となるエッジに集約させることでインストルメンテーションポイント数を最小化しているとともに，経路途中の処理は経路番号を計算するために変数に定数値を加算する処理なので，処理コストが平均で約 31%と非常に少ない．Young-Smith の手法は，プロファイルする経路として，あるノードに到達するまでの過去 K 個の分岐ノードの履歴の列（K-bounded general path）を用いている．実行にともない path CFG と呼ばれるグラフを構築しながらプロファイルを実施する．path CFG は，各ノードが元の CFG における経路を表し，各エッジがその始端ノードが指す経路の最後のノードの分岐エッジを表す．彼らの手法は Ball-Larus の手法に比べてそのオーバヘッドが非常に大きいかわりに，個々のノードに対してより詳細な情報が得られる．彼らはさらに K-bounded general path を用いていくつかの最適化手法[24),25)]も提案し，その有効性を示している．

インストルメンテーションによるプロファイルのオーバヘッドを軽減する試みも提案されている．Arnold と Ryder は Instrumentation sampling 手法を提案している[3)]．この手法では，元のコードを複製した後，複製コードにのみインストルメンテーションコードを挿入する．元のコードには，その開始点と各ループの後方分岐エッジにカウンタを用いたチェックコードを挿入し，定期的に実行を遷移させてインストルメンテーションコードを実行することで，インストルメンテーションコードの実行頻度を下げている．3 章で述べたように，彼らの手法では，動的コンパイラにおけるパスプロファイルの問題を本質的に解決することはできない．しかしながら，彼らの手法は実行時のパスプロファイル処理のオーバヘッドを軽減して，実行への影響を軽減できるとともに，実行の局所的な偏りによるプロファイルの精度の低下を回避できる点で有効である．

プログラム中の実行頻度の高い経路を検出する手法もいくつか提案されている．Chillimbi と Hirzel は実行頻度の高いメモリアクセスの順序列（Hot Data Stream）を検出し，動的にプリフェッチ命令を挿入する手法を提案している．彼らは Instrumentation sampling 手法を改良した Bursty Tracing 手法[16)]を用いて，メモリアクセスの履歴を収集し，これを Sequitur アルゴリズム[18)]を用いて圧縮しすることで頻度の高い部分列を抽出する．Duesterwald らの提案する NET[11)]では，あらかじめプログラム中に存在する後方分岐の到達点（start-of-trace）にカウンタを挿入しておき，カウンタ値が閾値を超えた時点における，その位置からの実行経路（next execution trace）をホットパスとして検出する．これらの処理は実行頻度の高い領域

を検出する方法であるが，パスプロファイル収集手法のように具体的な実行頻度が得られるわけではない．

7. おわりに

本論文では，動的コンパイラのための実用的なパスプロファイル収集手法として，構造的パスプロファイル収集手法を提案した．本手法では，プログラムをループネストごとに構造グラフに分割したのち，各構造グラフに対して動的インストルメンテーション手法を用いてパスプロファイル処理を実施することで，プロファイルの収集コストをループの実行回数とは独立に制御することを実現した．これにより，十分な精度のプロファイルを少ないオーバヘッドで収集することを可能とした．本手法を IBM Java Just-In-Time コンパイラに実装して評価したところ，各メソッドに対して実行経路数を 1000 収集した場合に，そのプロファイル精度が約 90%となるとともに，自然な経路を用いたパスプロファイル収集手法に比べて少ないプロファイル量で高い精度を上げられることを示した．また実際のオーバヘッドは，プロファイルが頻繁に行われる状況でもオーバヘッドが約 2〜3%と非常に少ないことを示した．

謝辞 本研究を行うに際して貴重なご助言をいただいた日本アイ・ビー・エム（株）東京基礎研究所ネットワーク・コンピューティング・プラットフォームグループ諸氏に感謝いたします．

参考文献

1) Ammons, G. and Larus, J.R.: Improving Data-flow Analysis with Path Profiles, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '98*), pp.72–84 (1998).
2) Arnold, M., Fink, S., Grove, D., Hind, M. and Sweeney, P.F.: Adaptive Optimizations in the Jalapeno JVM, *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications* (*OOPSLA '00*), pp.47–65 (Oct. 2000).
3) Arnold, M. and Ryder, B.G.: A Framework for Reducing the Cost of Instrumented Code, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '01*), pp.168–179 (Jun. 2001).
4) Bala, V., Duesterwald, E. and Banerjia, S.: Dynamo: A Transparent Dynamic Optimization System, *Proc. ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pp.1–12 (Jun. 2000).
5) Ball, T. and Larus, J.R.: Optimally profiling and tracing programs, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.4, pp.1319–1360 (1994).
6) Ball, T. and Larus, J.R.: Efficient Path Profiling, *Proc. 29th International Conference on Microarchitecture* (*MICRO-29*), pp.46–57 (Dec. 1996).
7) Ball, T., Mataga, P. and Sagiv, M.: Edge Profiling versus Path Profiling: The Showdown, *Proc. 25th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (*POPL '98*), pp.134-148 (1998).
8) Bodik, R., Gupta, R. and Soffa, M.L.: Complete Removal of Redundant Expressions, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '98*), pp.72–84 (1998).
9) Chilimbi, T.M. and Hirzel, M.: Dynamic Hot Data Stream Prefetching for General-Purpose Program, *Proc. ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, pp.199–209 (Jun. 2002).
10) Cierniak, M., Lueh, G.Y. and Stichnoth, J.M.: Practicing JUDO: Java Under Dynamic Optimizations, *Proc. ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pp.13–26 (Jun. 2000).
11) Duesterwald, E. and Bala, V.: Software Profiling for Hot Path Prediction: Less is More, *Proc. 9th International Conference on Architectural Support on Programming Languages and Operating Systems* (*ASPLOS '00*), pp.202–211 (Nov. 2000).
12) Feller, P.T.: Value profiling for instructions and memory locations, Master Thesis CS98-581, University of California, San Diego (Apr. 1998).
13) Gupta, R., Berson, D.A. and Fang, J.Z.: Path Profile Guided Partial Dead Code Elimination Using Predication, *Proc. Conference on Parallel Architectures and Compilation Techniques* (*PACT '97*), pp.102–113 (Nov. 1997).
14) Gupta, R., Berson, D.A. and Fang, J.Z.: Path Profile Guided Partial Redundancy Elimination Using Speculasion, *Proc. IEEE International Conference on Computer Languages*, pp.230–239 (May 1998).
15) Havlak, P.: Nesting of Reducible and Irreducible Loops, *ACM Trans. Prog. Lang. Syst.*, Vol.19, No.4, pp.557-567 (1997).
16) Hirzel, M. and Chilimbi, T.: Bursty Tracing: A Framework for Low-Overhead Temporal Profiling, *Workshop on Feedback-Directed and Dynamic Optimizations* (*FDDO*) (2001).

17) Hollingsworth, J.K., Miller, B.P., Goncalves, M.R., Naim, O., Xu, Z. and Zheng, L.: MDL: A Language and Compiler for Dynamic Program Instrumentation, *Proc. International Conference on Parallel Architectures and Compilation Techniques* (*PACT '97*), pp.201–212 (Nov. 1997).
18) Nevill-Manning, C.G. and Witten, I.H.: Linear-time, incremental hierarchy inference for compression, *Proc. Data Compression Conference* (*DCC'97*), pp.3-11 (1997).
19) Paleczny, M., Vick, C. and Click, C.: The Java HotSpot Server Compiler, *Proc. Java Virtual Machine Research and Technology Symposium* (*JVM '01*), pp.1–12 (Apr. 2001).
20) Pettis, K. and Hansen, R.C.: Profile Guided Code Positioning, *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp.16-27 (Jun. 1990).
21) Standard Performance Evaluation Corporation: SPECjvm98. available http://www.spec.org/osg/jvm98
22) Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H. and Nakatani, T.: A Dynamic Optimization Framework for a Java Just-In-Time Compiler, *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications* (*OOPSLA '01*), pp.180–194 (Oct. 2001).
23) Traub, O., Schechter, S. and Smith, M.D.: Ephemeral Instrumentation for Lightweight Program Profiling, Technical Report, Harvard University (1999).
24) Young, C. and Smith, M.D.: Better Global Scheduling Using Path Profiles, *Proc. 31st International Conference on Microarchitecture* (*MICRO-31*), pp.115–123 (1998).
25) Young, C. and Smith, M.D.: Static Correlated Branch Prediction, *ACM Trans. Prog. Lang. Syst.*, Vol.21, No.5, pp.1028–1075 (1999).

(平成 15 年 2 月 18 日受付)
(平成 15 年 4 月 23 日採録)

安江 俊明（正会員）

1991 年早稲田大学大学院理工学研究科電気工学専攻修了．1995 年同大学院博士後期課程中退後，日本 IBM 東京基礎研究所入社．最適化コンパイラ，並列処理の研究に従事．

菅沼 俊夫（正会員）

1982 年京都大学大学院工学研究科数理工学専攻修了．1992 年ハーバード大学大学院計算機学科修了．同年日本 IBM 東京基礎研究所入社．HPF，Java JIT 等のコンパイラに関する研究に従事．

小松 秀昭（正会員）

1960 年生．1985 年早稲田大学大学院理工学研究科電気工学専攻修了．同年日本 IBM 東京基礎研究所入社．コンパイラ，アーキテクチャ，並列処理の研究に従事．博士（情報科学）．

中谷登志男（正会員）

1975 年早稲田大学理工学部数学科卒業．同年日本 IBM（株）野洲工場入社．1983 年から米国プリンストン大学大学院（コンピュータ・サイエンス学科）．1985 年同大学から M.S.E. および M.A.，1987 年同大学から Ph.D. 同年より，日本アイ・ビー・エム（株）東京基礎研究所に移り，VLIW コンパイラ，HPF コンパイラ，JIT コンパイラ等のプロジェクトを担当．一貫して，プログラムを最適化して高速に実行させるための新しいソフトウェア技術について研究開発している．現在，IBMDistinguished Engineer，ネットワーク・コンピューティング・プラットフォーム担当．