# 非常に偏った条件分岐が存在するプログラムのデータフロー最適化

古関 聰[†] 小松秀昭[†]

共通部分式の削除，定数・複写伝播などの最適化は，プログラムの実行速度を高めるための有効な手段である．これらの最適化は，一般にデータフロー方程式を解きながら，プログラム全体に適用される．このような最適化はデータフロー最適化と呼ばれる．ところが，プログラム中には，条件分岐による制御の流れの変更があり，データの伝播性や計算式の到達性が制限されることがある．この制限は，条件分岐が存在した場合に避けることはできないが，条件分岐の中には非常に希にしか起こらないものも存在し，そのような分岐中のほとんど実行されないコードの影響を考慮するために，データフロー最適化が少なからず制限されてしまう．このような制限を緩和する方法の１つに，分岐の合流点から先をすべてコピーしてしまうという手法がある．しかしながら，このようなコピーは分岐の組合せによる爆発を起こしてしまうため，実用的とはいえない．本論文では，分岐の偏りを利用し，希にしか実行されないコードに影響される部分のコピーを抑制することで，コード量の爆発を防ぎながら，データフロー最適化の制限を除去する手法を提案する．

# Data-flow Optimizations in the Presence of Rarely-taken Paths

Akira Koseki[†] and Hideaki Komatsu[†]

Common subexpression elimination and copy/constant propagation are very effective methods for achieving good program performance. These optimizations are usually modeled as data-flow equations, and are therefore called data-flow optimizations. However, data propagatability and subexpression reusability are hindered by the control flow merges due to conditional branches. This limitation of optimization is inevitable in the presence of such control flow merges. Yet, even though some conditional branches are very rarely taken, the effectiveness of data-flow optimizations is still hindered at their corresponding control flow merge points. One solution to this problem is to copy the parts of a program which follow the merge points of conditional branches. However this is not practical because it causes exponential code size explosion. This paper describes a method to keep data-flow optimizations effective by suppressing copying the parts of a program that are affected by the rarely-executed code and thereby avoiding code size explosion.

## 1. はじめに

共通部分式の削除，定数・複写伝播，冗長コードの削除などの最適化[1)~4)]は，プログラムの実行速度を高めるための有効な手段である．これらの最適化は，一般にデータフロー方程式を解きながら，プログラム全体に適用される．このような最適化はデータフロー最適化と呼ばれており，我々が開発している Java[5)]用の Just-in-time コンパイラでの主要な最適化手法となっている[6),7)]．データフロー最適化は，一般的に，データフロー方程式を解くことで，演算結果の下方到達性や，演算の上方移動性を計算することにより行われる．ここでは，前者を下向きのデータフロー，後者を上向きのデータフローと呼ぶことにする．

プログラムには条件分岐による制御の流れの変更があり，これらの存在により，前述の下方到達性や上方移動性が制限されることがある．たとえば，変数の値の伝播を考えてみる．ある条件分岐のそれぞれの方向で同じ変数が定義されている場合，その分岐の合流点を超えて，その変数へ格納されるデータを下向きに伝播することはできない．この制限は，条件分岐が存在した場合に避けることはできないが，条件分岐の中には非常に希にしか起こらないものも存在し，そのような分岐中のほとんど実行されないコードの影響を考慮するために，データフロー最適化の適用性が制限されているのが現状である．後述のように，我々が開発しているコンパイラでは静的に分岐の偏りが明らかに

† 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.

判明していることも多く，頻繁に実行される部分へのデータフロー最適化の適用性をどのように向上させるかが課題となっている．

そこで，本論文では，そのような分岐が存在した場合に，下向きのデータフロー最適化適用性低下を緩和する手法を提案する．基本的には，下向きのデータフローは分岐の合流によって適用性が落ちてしまうので，合流後のコードをコピーすることがその解決方法となる．しかし，直列した分岐のすべての組合せを考慮してコピーを行ってしまうとコード量の爆発を招いてしまう．これに対し，本手法では，分岐の偏りの情報を利用し，実行確率が低いパスのコピーを抑制して行うことでコード量の爆発を回避する．

## 2. プログラムに現れる分岐の偏り

一般的に，プログラムのコンパイル中に，ある分岐についてその分岐がほとんど 1 方向にしか起こらないということが判定できることが多い．これは，以下のような方法で行われる．

- プロファイル情報を利用する．
- プログラムの構造から分岐の偏りを判定する．

前者は，プロファイラなどを活用して分岐の振舞いを分析することで分岐の偏りを発見する方法である．特に，Just-in-time コンパイラのようなダイナミックコンパイラでは，コンパイルを行う前にインタプリタなどで実行を行い，何度も実行される部分だけを選択的にコンパイルするのが一般的である．このような方式ではプロファイルデータは実行時にインタプリタで収集される．いずれの方式にせよ，なんらかの方法で記録されたそれぞれの分岐の履歴をコンパイル中に分析することで，分岐の偏りを発見することができる．

後者は，プログラムの静的な構造から分岐の偏りを推測する方法である．まず，プログラムの specialization などの方法でコンパイラ自身が偏りのある分岐を生成することがある．specialization は，ある汎用のアルゴリズムをインプリメントしたコードと，ある特定の条件が成立したときに実行されるスペシャルコードを両方用意しておいて，それらを実行時に選択して実行する手法である．このとき，その特定の条件が成立する確率が高いことが推測できる場合，その分岐が偏っていることが分かる．ループのバージョニングや後に詳しく述べる devirtualization[8)~11)]を行ったときに現れる分岐がその例である．また，コントロールフロー解析により分岐の偏りを推測する方法として，ループを検出し，ループから脱出する分岐を偏った分岐とする，あるいは，例外を明示的に上げる命令に必ず到達する分岐を偏った分岐とするなどの方法がある．

これらの中で，オブジェクト指向である Java をターゲットとする我々のコンパイラが最も頻繁に遭遇するものが，devirtualization 時に現れる分岐である．以下に，devirtualization 時に現れる分岐を例にとり，偏った分岐とデータフロー最適化について詳しく述べる．

### 2.1 devirtualization 時に現れる分岐の偏りとデータフロー最適化

devirtualization は，メソッド呼び出しの際にどのメソッドが呼び出されるかを予測し，実行時にその推測が当たっているかどうかをテストしたうえで，推測が当たったパス（ファストパス）には予測したメソッドが呼び出されることを前提とした最適化を施したコードを配置し，また，推測が外れたパス（スローパス）にはオリジナルの呼び出しを行うコードを配置する手法である．ファストパスにおける最適化は様々なものが考えられるが，通常は予測したメソッドのインラインが行われることが多い．

呼び出されるメソッドの予測は，コンパイル時に，そのメソッドのレシーバとなるオブジェクトのクラス階層においてメソッドが唯一である場合に，そのメソッドを選択する（この唯一性は後に他のクラスがロードされたときに失われる可能性がある），あるいは，レシーバオブジェクトのクラス階層の中で最も最近ロードされたものを選択するなどの方法で行われる．特に前者の場合は，特殊な振舞いをする少数のクラスを除いて，その予測は非常に高い確率で当たる．つまり，コンパイル時に単一であったメソッドの多くは，プログラムの実行の最後まで単一である可能性が高いということができる．これは実際，Java の主要なベンチマークの実行においても確認されており，メソッドの書き換えを前提としたクラス（たとえば GUI などのフレームワークを提供する一連のクラス）を除いて，コンパイル時に単一性が確認されたメソッドの単一性が実行時に失われることはほとんどない．

devirtualization を利用してインライニングを行うことで，動的束縛によるメソッド呼び出しが多用されるようなプログラムの有効な最適化が可能になることは確かである．しかしながら，インラインされたコードを実行するために条件判定を行わなければならないことは問題である．なぜなら，インライニングの効果は関数呼び出しのオーバヘッドの削減よりも，むしろインラインされたコードそのものの最適化によるところが大きいからである．インラインされたコードを最適化する主要なものはデータフロー最適化であるが，インラインを可能にしたその分岐によって，データフ

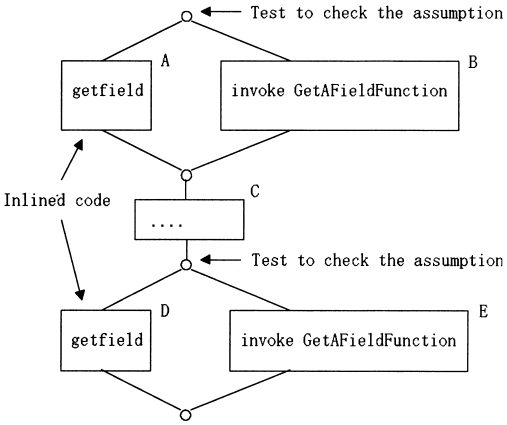


**図 1** Devirtualization を利用したインライニング
Fig. 1 Inlining with devirtualization.

ロー最適化が十分に適用されなくなってしまう．

**図 1** は，devirtualization を利用したインライニングの例である．ここでは，ある Java のメソッドに GetAFieldFunction というメソッドを 2 回インラインしている．また，コンパイル時におけるレシーバオブジェクトのクラス階層の分析により，GetAField-Function の最も頻繁に実行される実装が判明しており，そのコードは単に getfield を行うものであることが分かっていて，そのコードによってインライニングが行われているとする．ここで，インラインコードの前には，呼び出されるメソッドが予測したものと等しいかどうかを判定するテストが配置されている．

同図では，2 回目の getfield を 1 回目の getfield の結果によって置き換えることでプログラムを最適化できると考えられる．しかしながら，この例では，先に現れる getfield が後の getfield に到達するパスの中にスローパスとの合流があるため，そのような置き換えはできない．なぜならば，getfield 間の最適化を行い，かつ，GetAFieldFunction がまったく意味の異なったメソッドにオーバライドされた場合，B-C-D のパスのプログラムの意味が異なってしまうからである．このようにして，このプログラムはほとんど A-C-D のパスのみが実行されるにもかかわらず，最適化の機会が失われてしまってるといえる．

## 3. 基本アイディア

### 3.1 分岐の偏りを考慮したコードのコピー

前述のように，下向きのデータフローの適用性の低下は，分岐の合流から先のコードをコピーすることで回避できる．たとえば，ある条件分岐のそれぞれの方向で同じ変数が定義されている場合でも，その合流点から先の部分をすべてコピーすれば，個別に定義されるそれぞれの値を下向きに伝播できる．

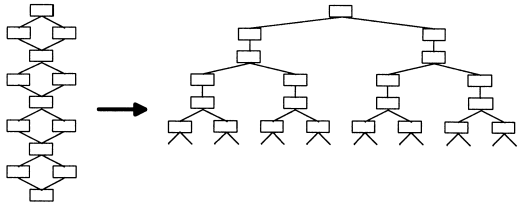

**図 2** コピーによるコード量爆発
Fig. 2 Code size explosion by copying.

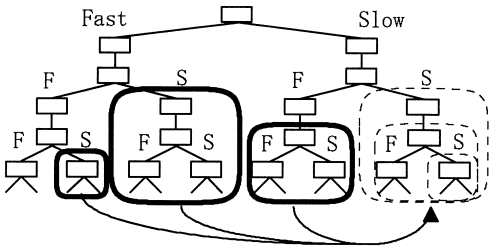


**図 3** 実行頻度の低いパスのコピーの縮退
Fig. 3 Reduction of copying rarely-taken paths.

しかしながら，単純なコードのコピーを行うということは，条件分岐が直列する場合に分岐のすべての組合せに合わせたコードが出てしまうことを意味する．この場合，コードのコピーの数は 2 の分岐数乗になってしまうので，この方法は実用的なコンパイラに採用できる方式とはいい難い（**図 2**）．

一方，これまで述べてきたように，プログラムには前もって分岐頻度の偏りが分かっている分岐が少なからず存在する．そこで，我々は，分岐が非常に偏っている場合は，スローパスにおけるコピーは必ずしも必要ではないことに注目する．つまり，スローパスに対する最適化の適用性の低下を考慮しなければ，スローパス内での合流を除去するためのコピーは必要がないことになる．これにより，コード量の増加を抑制しながら，ファストパスへの下向きのデータフロー最適化の適用性を高めることが可能となる．

この手法は，すべての組合せを用意する代わりに，1 つでもスローパスが含まれるパスの組合せを 1 つの代表的なパスに縮退させていると捉えることができる．**図 3** は，図 2 の組合せがどのように縮退されるかを表したものである．この縮退により，コピー後の全体のコード量は多くても 2 倍で抑えられる．

### 3.2 コントロールフローの変形

スローパスのコピーの縮退を考慮したコントロールフローグラフの変形を**図 4** に示す．図 4 の左のフロー

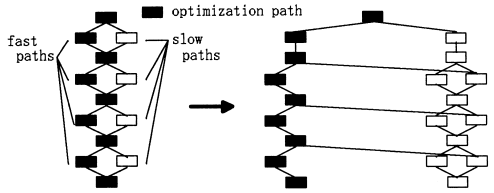


**図 4** コントロールフローの変形
Fig. 4 Control-flow change.

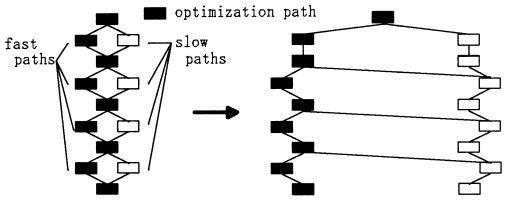


**図 5** コントロールフロー変形の特例
Fig. 5 Special case of control-flow change.

グラフは変形前のものであり，最適化の適用が期待されるパスを黒く示してある．また，同図の右のフローグラフは変形後のものである．同フローグラフにおいて，ファストパスへの合流はすべて除去され，合流の除去を最小限に補償するためのコピーが限定的に行われているのが分かる．この変形により，コード量の爆発を抑えながら，ファストパスに対する下向きのデータフロー適用性を高めることが可能となる．

### 3.3 コントロールフロー変形の特例

プログラムの分岐の中には，分岐の 1 方向の意味が他の方向に含まれているものがある．前に述べた specialization や devirtualization を行った場合に現れる分岐はその性質を持つ．なぜなら，スペシャルコードは，オリジナルコードの特定の場合を抜き出したものであるからである．このような場合，プログラムの意味の包含関係を利用してコードのコピー量を削減できる．すなわち，スローパスにおける最適化は度外視するという前提なので，コピーされたパスの中でファストパスに相当する部分を削除することでコード量のさらなる削減が可能になる．この様子を**図 5** に示す．

### 3.4 データフロー最適化の適用例

これまでに説明したコントロールフローグラフ変形によって得られたプログラムに対する下向きのデータフロー最適化適用例を**図 6**，**図 7** に示す．図 6，図 7 では，o.f() というバーチャルコールが devirtualization を利用してインラインされている．それぞれの図の左のプログラムはコントロールフローグラフ変形の前の状態である．ここで，ブロック C，F，I はスローパスであり，オリジナルのメソッド呼び出しが配置されているものとする．また，ブロック B，E，H はファストパスでありインラインされたコードが配置されているものとする．また，各図の右のプログラムは，図 5 で示したコントロールフローグラフ変形を行った後，データフロー最適化を適用したものである．

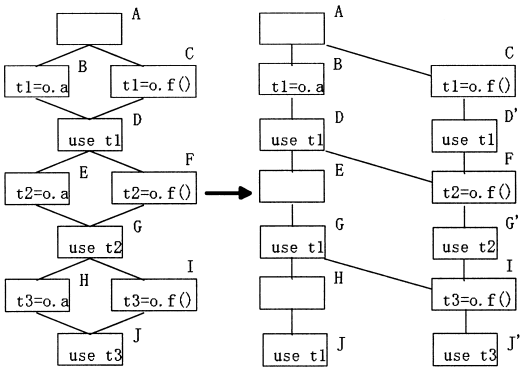


**図 6** Commoning 適用例
Fig. 6 Application of commoning.

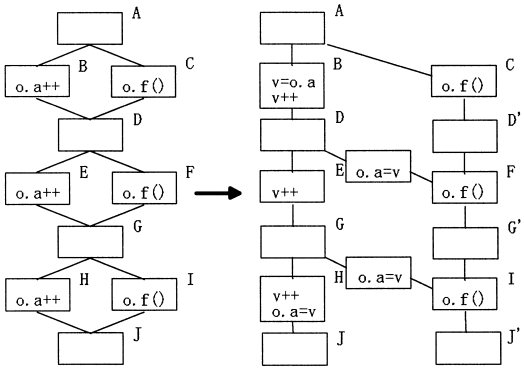


**図 7** Privatization 適用例
Fig. 7 Applicaiton of privatization.

図 6 は，共通部分式の削除の適用例である．このプログラムでは，devirtualization の推測が当たっている限りにおいて，ブロック B，E，H のフィールドアクセスはすべて同じ値を読みにいく．したがって，2 回目，3 回目のフィールドアクセスは冗長なのでこれらの除去が期待される．しかしながら，左のプログラムでは，これまで述べたように，インラインされたコードの結果を伝播し，再利用することができない．これに比べ，フローグラフ変形後の右のプログラムでは，ファストパスへの合流が取り除かれているため，ブロック E，H のコードを共通式として除去することができる．

次に，図 7 は，privatization の適用例である．このプログラムでは，devirtualization の推測が当たっている限りにおいて，ブロック B，E，H では同じフィールドに対するインクリメントが行われる．したがって，

このフィールドの値を変数にキャッシュしておくことで演算のコストを低めることができる．しかしながら，これまでと同様に左のプログラムのままでは最適化の効果は得られない．これに比べ，フローグラフ変形後の右のプログラムでは，ファストパスへの合流が取り除かれているため，フィールドの値のキャッシングが有効である範囲がブロック E，H に到達し，それらの範囲における演算のコストを低減することができている．

図 6，図 7 は，簡単のため，同じメソッドの繰返しによる単純な構造からなるプログラムを用いたが，本手法の本質は，最適化対象となるパスへの流入を除去することであり，制御構造がもっと複雑である（たとえばループを含む）プログラムも本手法の対象となる．次章において，一般的な制御構造に本アイディアを具体化するアルゴリズムについて述べる．また，本例では同じメソッドをインラインしたコード間のデータフロー最適化を示したが，データフロー最適化の機会はこのような場合に限られず，異なったメソッドをインラインしたコードを含むパスにも最適化の機会は多く存在する．たとえば，同じ配列をアクセスする異なったメソッドがインラインされている場合に，アクセスされる配列の境界条件テストやアドレスの計算がインラインされたコード間で最適化できるケースなど，様々な機会が考えられる．

## 4. アルゴリズム

本章では，3 章で説明したファストパスとスローパスの合流点の除去を一般的に行うアルゴリズムを示す．

### 4.1 Brute-force な手法

コードをほぼ 2 倍にすることをいとわなければ，本手法は非常に簡単に実装できる．そのアルゴリズムを以下に示す．

(1) プログラム全体をコピーし，コントロールフローグラフを二重化する．1 つを最適化グラフ，もう 1 つを非最適化グラフと呼ぶ．
(2) 最適化グラフの中で，最適化の対象となる偏った分岐を検出する．これらを対象分岐と呼ぶ．
(3) 対象分岐のスローパスをグラフから削除し，非最適化グラフの中でこのスローパスに対応する部分を検出する．
(4) 対象分岐からスローパスへ流出していたエッジを，検出したスローパスに接続する．
(5) 最適化グラフの先頭からグラフの到達性を計算し，非最適化グラフの中の非到達領域を削除する．
(6) コントロールフロー変形の特例が適用できる場合，非最適化グラフからファストパスをすべて削除する．

以上の操作により，最適化グラフ内のスローパスからの合流を除去できる．

### 4.2 Resource-constrained な手法

Brute-force な手法は実装が容易であるが，つねにコード量が 2 倍近くに増えてしまう．我々が開発しているコンパイラは，実行時にコンパイルを行うため，コンパイル時間や使用メモリの無制限な増大は大きな問題となる．そこで，本手法を限定的に適用する手段が必要となる．以下に，コントロールフローグラフの変形をコード量の増加の制約のもとで実現するアルゴリズムを示す．

(1) あるファストパスとスローパスの合流点から直接到達可能なファストパスとスローパスの分岐点あるいはプログラムの終端を検出する．検出されなければ終了する．
(2) プログラムの終端が検出された場合は (11) へ．ファストパスとスローパスの合流点が検出された場合は (3) へ．
(3) (1) で検出された合流点から，同じく (1) で検出された分岐点から始まる合流点までのすべてのパスを検出する．
(4) (3) で検出されたパスのコード量を調べ，それらのパスのコピーにより，全体のコード量がコード量制限を超えてしまう場合，アルゴリズムを終了する．
(5) (3) で検出されたパスをコピーする．ただし，いままでの過程においてコピーが行われてない部分にのみ行う．また，コントロールフロー変形の特例が適用できる場合，コピーされたものからファストパスを削除する．
(6) (1) で検出された合流点を削除し，合流点に流入していたファストパスを，(3) で検出されたパスにつなげる．また，同じく合流点に流入していたスローパスを (5) でコピーされたパスにつなげる．
(7) (3) で検出されたパスの分岐点に対し，そのスローパスを除去し，(4) でコピーされたものにつなげる．
(8) (3) で検出されたパスのスローパス以外の部分に外から流入するエッジがあり，それがスローパスからのエッジだった場合，(5) でコピーされたパスの対応部分につなぎかえる．そうでない場合は何もしない．

(9) (3) で検出されたパスから外へ流出するエッジがあった場合，エッジの始点のブロックと対応するコピーされたブロックとの新しい合流点をつくり，その合流点からのエッジで上記エッジを置き換える．
(10) (1) へ．
(11) (1) で検出された合流点から，プログラムの終端までのすべてのパスを検出する．
(12) (11) で検出されたパスのコード量を調べ，それらのパスのコピーにより，全体のコード量がコード量制限を超えてしまう場合，アルゴリズムを終了する．
(13) (11) で検出されたパスをコピーする．ただし，いままでの過程においてコピーが行われてない部分にのみ行う．
(14) (1) で検出された合流点を削除し，合流点に流入していたファストパスを，(11) で検出されたパスにつなげる．また，同じく合流点に流入していたスローパスを (13) でコピーされたパスにつなげる．
(15) (11) で検出されたパスのスローパス以外の部分に外から流入するエッジがあり，それがスローパスからのエッジだった場合，(11) でコピーされたパスの対応部分につなぎかえる．そうでない場合は何もしない．
(16) (11) で検出されたパスから外へ流出するエッジがあった場合，エッジの始点のブロックと対応するコピーされたブロックとの新しい合流点を作り，その合流点からのエッジで上記エッジを置き換える．
(17) (1) へ．

### 4.3 Resource-constrained アルゴリズムの適用例

下記の一連の図に本アルゴリズムの適用例を示す．ただし，ここでは，3.3 節で示したコントロールフロー変形の特例が適用できるものとする．

**図 8** の段階 1 のコントロールフローグラフにおいて，(1) に従い，合流点 2 と分岐点 3 が検出される．ここでは，(3) に従い，ブロック D，E，F，G を含んだパスがコピー対象として検出される．次に (5) に従い，コピーを行う．本例ではコントロールフロー変形の特例が適用できるとされたので，実際にはブロック D，E，G のパスがコピーされる．次に (6) に従い，合流点 2 に流入していたファストパスとスローパスはそれぞれ，オリジナルのパスとコピーされたパスに接続される．また，(7) に従い，ブロック G を除去し，

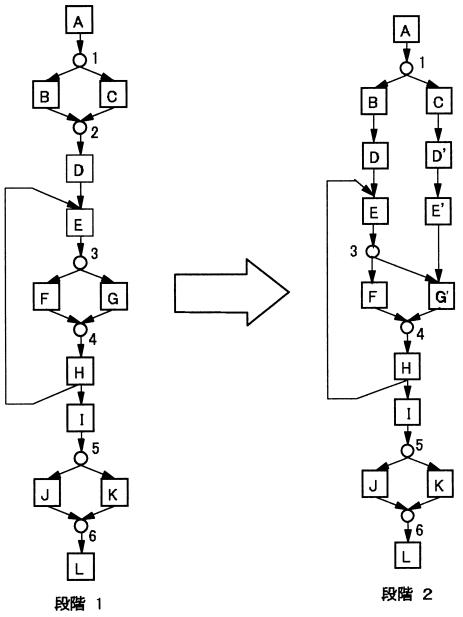


**図 8** 変形の過程 1
Fig. 8 Process of the change 1.

分岐点 3 からのスローパスをブロック G' につなげる．E には外から流入するエッジがあるが，これはスローパスからのエッジではないので，(8) に従い，何も行わずに次に進む．こうしてできたコントロールフローグラフが段階 2 である．

**図 9** において，合流点 4 から直接到達可能な分岐点は 2 つある．すなわち，分岐点 3 と分岐点 5 である．まず，分岐点 3 が先に選択された場合を説明する．まず，コピーの対象となるのは，ブロック H，E，F，G' を含むパスであるが，ブロック E，F，G はすでに処理対象に 1 度なったため，ここではブロック H のみがコピーされ，H'-E'-G' のパスが形成される．そして合流点 4 が取り除かれ，合流点 4 に流入していたファストパスとスローパスはそれぞれ，オリジナルのパスとコピーされたパスに接続される．このとき，コピー対象となったパスから外に流出するエッジがあるので（H から I のエッジ），(9) に従い，H と H' の新しい合流点を 1 つ作り，その合流点からのエッジで，H から I のエッジを置き換える．これが段階 3-A である．

一方，合流点 4 に対し，分岐点 5 が先に検出される可能性もある．この場合は，ブロック H，I，J，K を含むパスがコピー対象となり，これまで説明したように，ファストパスとスローパスの形成が行われる．また，コピー対象となったパスから外に流出するエッジがあるので（H から E のエッジ），H と H' の新しい合流点を 1 つ作り，その合流点からのエッジで，H か

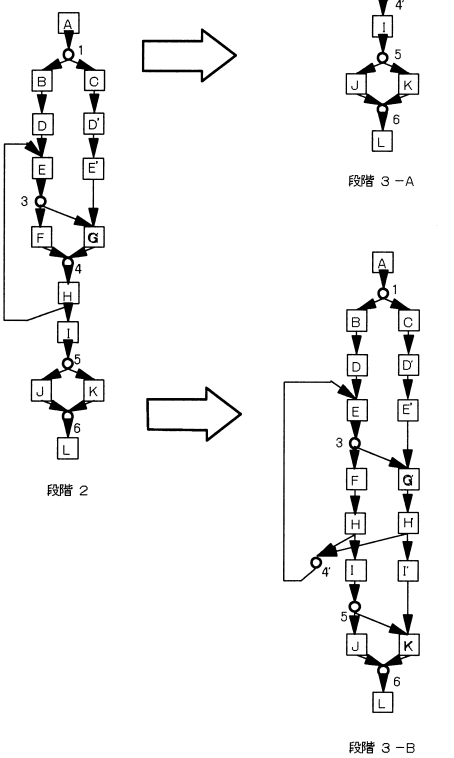


**図 9** 変形の過程 2
Fig. 9 Process of the change 2.

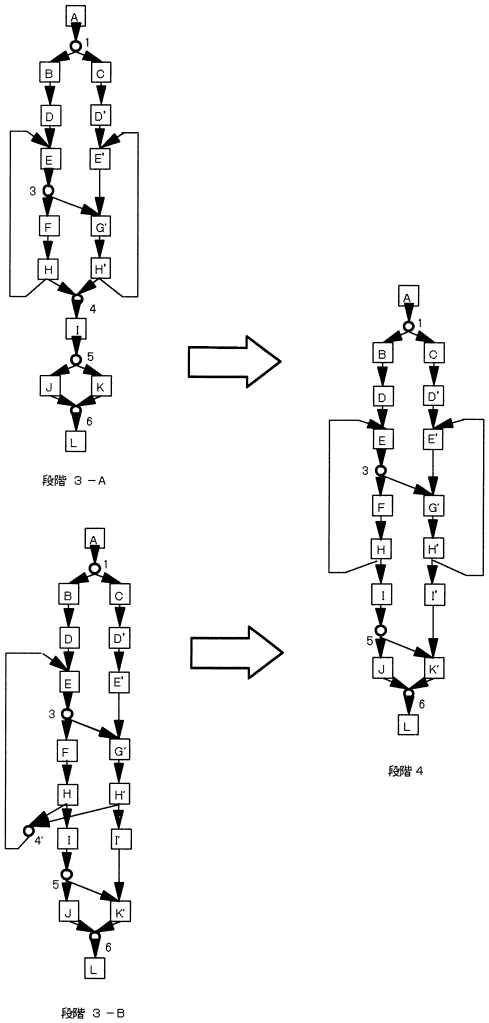


**図 10** 変形の過程 3
Fig. 10 Process of the change 3.

らＥのエッジを置き換える．これが段階 3-B である．このように選択可能性が複数あった場合，異なった途中段階が得られるが，アルゴリズムが最後まで繰り返される場合は同じ最終段階が得られる．本例においては，次の過程において異なった途中段階からの集約が見られる．

次に，**図 10** では，段階 3-A または段階 3-B から段階 4 のように変形が行われる．段階 3-A からの場合は，合流点 4’ に対して分岐点 5 が検出され，ブロックＩ，Ｊ，Ｋを含むパスがこれまでと同様にコピーされる．また，段階 3-B からの場合は，合流点 4’ に対し分岐点 3 が検出され，ブロックＥ，Ｆ，Ｈ，Ｇ’，Ｈ’を含むパスがこれまでと同様にコピーされる．また，ブロックＥには外から流入するエッジがあるが，これはスローパスからのエッジではないので何も行わない．

最後に，**図 11** において，段階 4 から最終段階である段階 5 のように変形が行われる．ここでは，合流点 6 からプログラムの終端までのパスに沿ったブロックＬが複製され，ブロックＪからブロックＬに，ブロックＫからブロックＬ’にそれぞれエッジが接続される．

## 5. 評　　価

本章では，4.2 節で示したアルゴリズムを実装し，本手法の有効性の評価を行う．ここでは，アルゴリズムの実装を次のように限定して，IBM Developer’s Kit for Windows，Java Technology Edition，Version 1.3[14] 上に実現した．

(1) 対象とする分岐を devirtualization の予測の正

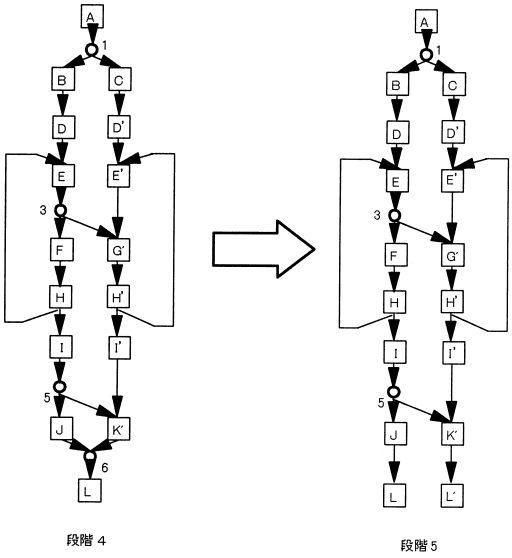


**図 11** 変形の過程 4
Fig. 11 Process of the change 4.

```
class testClass
{
  static final
    int a = setA();
  static final
    int b = setB();

  int getValue () {
    return a*b;
  }
}
```

```
public class test
{
  static testClass obj;
  public static void main (String args[]) {
    int sum = 0;
    obj = new testClass () ;

    for(int i = 1; i < 100000000; i++){
      sum = sum + obj.getValue (); (A)
    }
  }
}
```

**図 12** テストプログラム
Fig. 12 Test program.

しさを判定するのものに限定.

(2) 変形を行う範囲にループが含まれないように限定.

### 5.1 テストプログラムによる実験

**図 12** は，我々が用いたテストプログラムである．このプログラムでは，変数 sum に対し，testClass のオブジェクトの getValue メソッドの結果が繰り返し加算される．我々は，本手法による下向きのデータフローの効果を調べるため，図中の (A) の個数をさまざまに変えて，繰返し部分に費やされる実行時間を測定した．

我々のコンパイラによって最適化が行われた場合，このプログラムの繰返し部分に関しては，getValue は devirtualization を利用してインラインされ，**図 13** 左で示したような構造が得られる．すなわち，インラインされたファストパスでは，オブジェクトからの 2 つのフィールドの読み出しと，それらの値の乗算が行われ，スローパスではナイーブなバーチャルコールが行われる．また，ファストパスとスローパスの前には devirtualization の推測の正しさを確かめるテストが配置される．ここで，オブジェクトの 2 つのフィールドは static final で宣言されているため，インラインされた部分で変数 sum に足される値はすべて同じ値となる☆．一方，オブジェクト obj は test クラスのスタティックフィールドに格納されており，他のスレッドがこの内容を書き換える可能性がある．したがって，(A) の部分を実行している間に，多のスレッドが異なった getValue の実装を持った testClass のサブクラスを新たにロードし，そのサブクラスのオブジェクトを test.obj に格納する可能性がある．したがって，図 13 における devirtualization の推測が正しいかどうかのテストは毎回行われなければならない．

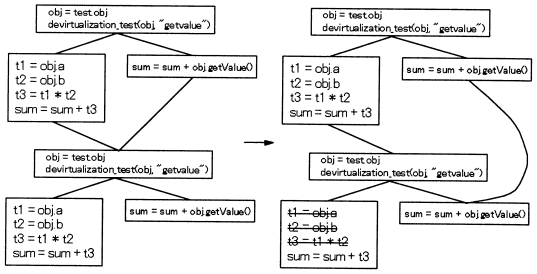


**図 13** テストプログラムの最適化
Fig. 13 Optimization of test program.

このプログラムにおいて，(A) の個数が 2 以上の場合，devirtualization の推測が正しい限り getValue の値はすべて再利用できる．しかしながら，これまで述べてきたとおり，本手法を適用する前の状態では，各所にファストパスとスローパスの合流があるため，値を下方に伝播することはできない．一方，本手法を適用した場合，プログラムは図 13 右のように変形され，2 回目以降のフィールドアクセス，乗算はすべて 1 回目のものと共通化される．

**図 14** は (A) の個数を 1 から 10 に変化させたときの繰返し部分の実行時間である．同図より，個数が多ければ多いほど，本手法の効果が大きく現れていることが分かる．これは，(A) の回数が多くなれば，フィールドアクセスや乗算の共通化の機会が多くなるためである．結論として，実験結果より，本手法は devirtualization をともなったインラインが多数行われるケースにおいて，下向きのデータフロー最適化の適用性を

☆ ここで，簡単のためにフィールドを static final で宣言して値が書き換わらないようにしてあるが，他のスレッドがオブジェクトのフィールドを書き換えるような場合でも，オブジェクトを privatize することで演算を共通化できる．

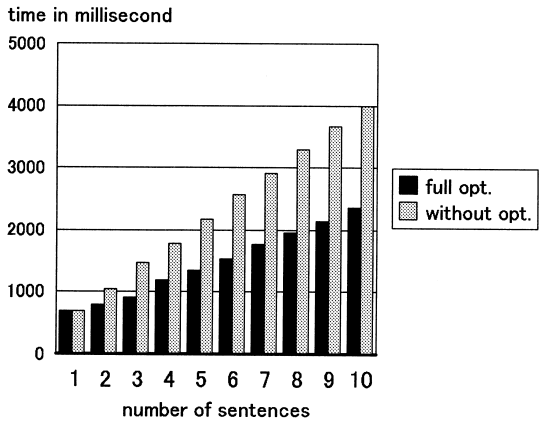


**図 14** 測定結果
Fig. 14 Result of measurement.

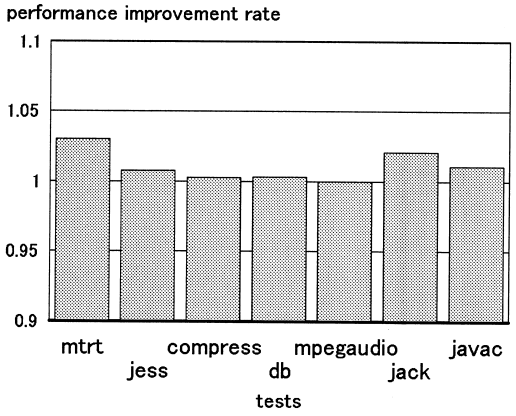


**図 15** SpecJVM98 におけるパフォーマンス改善
Fig. 15 Performance improvement in SpecJVM98.

高めることに有効であることが確認できた．

### 5.2 SpecJVM98

次に，SpecJVM98[15]の各テストに対する本手法の効果とコンパイル時間の増加を測定した．

**図 15** は，本手法を用いなかった場合に比べ，本手法が適用された場合にどれだけのパフォーマンス改善が得られたかを示す図である．同図より，mtrt，jess，jack，javac の各テストにおいて数パーセントのパフォーマンス改善が得られている．これは，本コンパイラにおいて，SpecJVM98 における，共通部分式の削除をはじめとするデータフロー最適化によるゲインが 7〜8%であることを考えると，比較的大きな効果が現れているといえる．これらの中で，最も効果の大きかった mtrt のカーネル部分を分析してみると，果たしてバーチャルコールが数多く行われており，それらのほとんどが devirtualization を利用してインラインされていた．本手法の適用の結果，インラインされたパスを含んだ，プログラムの実行に重要なパスに

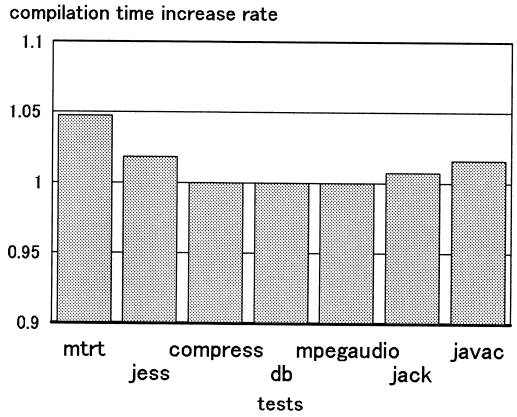


**図 16** SpevJVM98 におけるコンパイル時間増加
Fig. 16 Compilation time increase in SpecJVM98.

下向きのデータフローが適用され，特に，定数/複写の伝播，冗長な NullPointer チェックの除去において顕著な改善が得られていたことが確認できた．一方，compress，db，mpegaudio では，本手法による改善はみられなかった．これらのテストを分析してみると，各カーネルは 1 つか 2 つのメソッドにより構成されており，そもそもインライニングによる効果が期待できないプログラム構成であることが判明した．実際，バーチャルコールのインライニングを停止して測定したところ，実行時間の変化は 0.1%以下であった．したがって，本手法はこれらのテストのパフォーマンスにほとんど影響を与えないことが分かった．

また，**図 16** は，本手法を適用した場合にどれだけコンパイル時間が増大したかを示す図である．図 16 より，compress，db，mpegaudio においてはコンパイル時間がまったく変化していない．したがって，これらのテストに関しては，本手法によるコントロールの変形の適用自体がほとんど行われていないと推測できる．一方，図 15 において改善がみられた残りのテストに関しては，数パーセントのコンパイル時間の増加が確認される．これらの増加は，コントロールフローの変形にかかる時間というよりは，むしろ，変形によりベーシックブロック数やコード数が増えたことにより，本手法の後に適用されるデータフロー最適化などのコンポーネントの処理時間が増加したことによる．この測定は，コード量の増加の影響は全体のコンパイル時間の数パーセントの変化に及ぶことを示しており，本手法によるコンパイル時間への影響は決して少なくない．したがって，本論文で述べた手法を実際に実装する場合は，4.2 節で述べたような resource-constrained な方法が適切であると結論できる．

## 6. 関連研究

本手法は，コントロールフローを変形して，データフロー最適化の効果を高める手法であるが，分岐の合流点の先をコピーしてコントロールフローを変形する手法という点では，Trace Scheduling[16]やHyperblock[17]構築におけるコントロールフロー変形と共通の操作を行っているといえる．

Trace Schedulingでは，Trace上の命令を分岐や合流を超えて移動したときにコントロールフローの変形をともなった命令のコピーが行われる．本手法では，最適化対象パスへの合流点を後方に移動するという観点でコピーが行われるのに対し，Trace Schedulingでは，これに加えてTraceからの分岐を前方に移動するためのコピーも行われる．このため，本手法に比べてコピー量が多くなる．また，分岐命令どうしの順番を入れ替える場合は，組合せ爆発を起こすようなコピーが行われる可能性がある．

一方，Hyperblockの構築では，Hyperblock中への流入を除去するためのTail Duplicationと呼ばれる操作が行われる．これは，本手法と目的を同一にする手法であり，文献中に，コピー量の爆発が問題となることは指摘されているが，それを防ぐための具体的な手法については述べられていない．我々の手法は，コピー量の爆発を防ぎながら最適化対象パスへの合流を除去できるので，Tail Duplicationと本手法を組み合わせることで効果的なHyperblockの構築が行えるものと考えらる．実際，我々が開発しているコンパイラではHyperblock構築に本手法を応用する予定である．

次に，プログラムのスペシャライズという観点から比較を行うと，本手法が分岐を明示的に表現する手法であるのに対し，プログラムにあたかもファストパスしか存在しないと仮定してコンパイルを行う方式がある．この方式では，スローパスに分岐すると判明した時点で，プログラムのコンパイルのやり直し（リコンパイル）が行われる[18]．リコンパイル方式では，最適化対象となるパスへの合流というものは字面上存在しないので，下向きのデータフローの制限を取り除くことができるという点で本手法とまったく同じ効果がある（上向きのデータフローに関する制約は依然として存在するので，この点に関しては本手法と変わりはない）．ただし，リコンパイル方式は，コンパイルをもう一度最初からやり直すため，万一スローパスの実行の必要性が生じたときのオーバヘッドが存在する．または，on-the-flyのリコンパイルを行わずに，安全性が保証されたコード（つまりスローパスのみのコード）を用意しておき，スローパスに分岐すると判明した時点で，そのコードに飛ぶという方法も考えられる．しかし，この方式では，安全性が保証されたコードとスタックやローカル変数のイメージを合わせるために，最適化コードにおける変数の生存区間が伸び，レジスタプレッシャーの増加によってコードの質が落ちることがある．

一方，本方式はリコンパイル方式に比べ，非常に簡単に実装可能である．リコンパイル方式の実装は様々なものが考えられるが，たとえば，あるメソッドの実行中にそのメソッド自体のリコンパイルを許すような方式は，実装が非常に難しい．また，リコンパイルの対象を限定して，メソッドの実行中には推測の真偽が変化しないものだけを対象にする方法もある．この方式は，メソッドの実行が始まる前にのみ，予測が正しいかどうかを判定し，予測が外れていればコンパイルをしなおすだけでよいので比較的実装が簡単である．

そこで，実装の容易さと，高い最適化能力を同時に得るために，リコンパイル方式の後者の方法と本手法を合わせたハイブリッドな方式も考えられる．実際，我々はこの方向で開発を進めている．たとえば，メソッドのインライニングを考えると，メソッドの実行中に予測の正しさが変化しないものについてはリコンパイルを前提としたインライニングを，また，そうでないものに関しては，devirtualizationを利用したインライニングを行い，本論文で提案したコントロールフローグラフの変形を適用するというような戦略が妥当であると考えられる．これについては，別の機会に詳しく論証したい．

## 7. おわりに

本論文では，実際のプログラムには偏った条件分岐がよく現れることを示し，この分岐が下向きのデータフロー最適化の適用性を低下させてしまうことを述べた．しかし，これらの分岐の偏りを利用することで，コード量の爆発を避けながら，頻繁に実行される部分から分岐の合流点を取り除けることを明らかにした．また，この除去手法を開発中のコンパイラに実装し，その有効性を確認した．今回の本手法の適用はヒューリスティクスを利用したアルゴリズムによるものであったが，今後は，そのヒューリスティクスを変化させた場合に，コンパイル時間と実行時間にどのような影響が現れるかを分析していく予定である．

**謝辞** 本研究を行うに際して数々の示唆と助言をいただいた日本IBM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の中谷登志男マネー

ジャはじめ同グループの研究員の方々に感謝します．また，主に本手法の実装を手がけてくれた百瀬浩之元同研究所研究員に深く感謝いたします．

## 参 考 文 献


1) Gupta, R.: Optimizing Array Bound Checks Using Flow Analysis, *ACM Letters on Programming Languages and Systems*, Vol.2, Nos.1–4, pp.135–150 (1993).
2) Morel, E. and Renvoise, C.: Global Optimization by Supression of Partial Redundancies, *CACM*, Vol.22, No.2, pp.96–103 (1979).
3) Knoop, J., Ruthing, O. and Steffen, B.: Lazy Code Motion, *ACM SIGPLAN Conference on Programming Language Design and Implementaion*, pp.224–234 (1992).
4) Knoop, J., Ruthing, O. and Steffen, B.: Optimal Code Motion: Theory and Tools, *ACM Trans. Programming Languages and Systems*, Vol.17, No.5, pp.777–802 (1995).
5) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison-Wesley (1996).
6) Ishizaki, K., Kawahito, M., Yasue, T., Takeuchi, M., Ogasawara, T., Onodeda, T., Komatsu, H. and Natakani, T.: Design, Implementaion, and Evaluation of Optimizations in a Just-in-Time Compiler, *ACM Java Grande Conference*, pp.119–128 (1999).
7) Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H. and Nakatani, T.: Overview of the IBM Java Just-in-Time Compiler, *IBM Systems Journals, Java Performance Issue*, Vol.39, No.1, pp.175–193 (2000).
8) Grove, D., Dean, J., Garrett, C. and Chambers, C.: Profile-Guided Receiver Class Prediction, *Conference on Object Oriented Programming Systems, Languages & Applications*, pp.108–123 (1995).
9) Aigner, G. and Holzel, U.: Eliminating Virtual Function Calls in C++ Programs, *10th European Conference on Object-Oriented Programming*, pp.142–166 (1996).
10) Holzel, U.: Adaptive Optimizations for SELF: Reconciling High Performance with Exploratory Programming, Ph.D Thesis, Stanford Universwity (1994).
11) Dean, J., Grove, D., Garrett, C. and Chambers, C.: Optimizations of Object-Oriented Programs Using Static Class Hierarchy, *9th European Conference on Object-Oriented Programming*, pp.77–101 (1995).
12) Fernandez, M.F.: Simple and Effective Link-Time Optimizations of Modula-3 Programs, *ACM SIGPLAN Conference on Programming Language Design and Implementaion*, pp.103–115 (1995).
13) Detlefs, D. and Agesen, O.: Inlining of Virtual Methods, *13th European Conference on Object-Oriented Programming*, pp.258–278 (1999).
14) International Business Machines Corp.: IBM Developer's Kit for Windows, Java Technology Edition, Version 1.3.
15) The Standard Performance Evaluation Corp., SPEC JVM98 Benchmarks, http://www.spec.org/osg/jvm98/
16) Fisher, J.A.: Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Trans. Comput.*, Vol.C-30, No.7, pp.478–490 (1981).
17) Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E. and Bringmann, R.A.: Effective Compiler Support for Predicated Execution Using the Hyperblock, *25th International Symposium on Microarchitecture*, pp.45–54 (1992).
18) Holzel, U., Chambers, C. and Ungar, D.: Debugging Optimized Code with Dynamic Deoptimization, *ACM SIGPLAN Conference on Programming Language Design and Implementaion*, pp.32–43 (1992).


(平成 12 年 5 月 26 日受付)
(平成 12 年 9 月 8 日採録)

**古関 聰**（正会員）

1969 年生．1994 年早稲田大学大学院理工学研究科電気工学専攻修士課程修了．1998 年同大学院理工学研究科電気工学専攻博士課程修了．同年日本 IBM 入社．以来，同社東京基礎研究所において，Java just-in-time コンパイラの開発に従事．工学博士．

**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）．