コード書換えによる動的メソッド呼び出しの直接 devirtualization

石崎一明[†] 安江俊明[†]
川人基弘[†] 小松秀昭[†]

本論文では，Java 等の動的クラスローディングをともなう言語において，実装が容易な動的メソッド呼び出しの直接 devirtualization 手法を提案する．本手法では，動的メソッド呼び出しに対して直接 devirtualization されたコードと，メソッドがオーバライドされた場合に実行する動的メソッド呼び出しの 2 種類のコードをコンパイル時に生成する．最初は前者を実行し，メソッドのオーバライドが起きたときにコードを書き換えて後者を実行する．本手法では，コード書換えによって直接 devirtualization されたコードを無効化するので，脱最適化のような再コンパイルのための複雑な実装が不要である．一方，再コンパイルを不要にするためにコンパイル時に 2 種類のコードを用意するため，制御フロー上に合流点が生成される．一般に制御フローの合流点はコンパイラの最適化を妨げるが，本論文では合流点が存在しても十分な最適化を可能にする手法を示す．また本手法と他の devirtualization 手法を組み合わせて Java の Just-In-Time コンパイラに実装し評価を示す．その結果，devirtualization を行わない場合に比べ，SPECjvm98 と SPECjbb2000 において 0〜181%（平均 24%）性能を改善できることを示す．

A Direct Devirtualization Technique with the Code Patching Mechanism

Kazuaki Ishizaki,[†] Toshiaki Yasue,[†] Motohiro Kawahito[†]
and Hideaki Komatsu[†]

This paper presents a direct devirtualization technique for a language such as Java with dynamic class loading. The implemetation of this technique is easy. For a given dynamic method call, a compiler generates the inlined code of the method, together with the code of making the dynamic call. Only the inlined code is actually executed until our assumption about the devirtualization becomes invalidated, at which time the compiler performs code patching to make the code of dynamic call executed subsequently. This technique does not require complicated implementations such as deoptimization to recompile the method that is active on the stack. Since this technique prevents some optimizations across the merge point between the inlined code and the dynamic call, we have furthermore proposed optimization techniques effectively. We made some experiments to understand the effectiveness and characteristics of the devirtualization techniques in our Java Just-In-Time compiler. To summarize our result, we improved the execution performance of SPECjvm98 and SPECjbb2000 ranging from 0% to 181% (with the geometric mean of 24%).

1. はじめに

オブジェクト指向言語において，プログラミングの柔軟性を提供するために，実行時に与えられるレシーバのクラスに基づいて呼び出し先メソッドを決定する動的メソッド呼び出しのサポートは必須である．動的メソッド呼び出し☆は，複数のメソッドを呼び出し先として持つ可能性があるため，実行時に呼び出し先のメソッドの検索が必要となる．この検索が実行時オーバヘッドとなる．コンパイル時にメソッド検索を行うことで，このオーバヘッドを削減する手法は devirtualization と呼ばれる．これまでに多くの devirtualization の手法が提案されてきた．devirtualization は，その方法に基づいて，間接 devirtualization，直接 devir-

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

☆ 動的メソッド呼び出しは，Java バイトコードでは invokevirtual, invokeinterface に相当する．静的メソッド呼び出しは，Java バイトコードでは invokestatic, invokespecial に相当する．

tualization の 2 つに分けられる．

間接 devirtualization は，動的メソッド呼び出しを行う際にレシーバのクラスの型テストを行い，このテストが成功したとき呼び出し先メソッドの検索なしにメソッド呼び出しを実行する[1]．Self のような動的な型付けを行う言語では，呼び出し先メソッドの検索のオーバヘッドが大きいので，この方法は有効である．しかし，Java のような静的な型付けを行う言語では，呼び出し先メソッドの検索のオーバヘッドが小さいため，効果が小さいことが知られている[2]．

直接 devirtualization は，動的メソッド呼び出しを行う際にレシーバのクラスの型テストを行わず，呼び出し先メソッドの検索なしにメソッド呼び出しを実行する．つまり，静的メソッドと同じコストでメソッド呼び出しを実行できる．さらにメソッドのインライン展開の適用範囲を拡大することができる．したがって，静的な型付けを行う言語でも大きな効果を得られる．これまで提案された直接 devirtualizaton の多くは，プログラム実行前にクラス階層解析[3]を行うため，C++のように動的クラスローディングをともなわない言語に適用されていた．Java のように動的クラスローディングをともなう言語では，実行時にクラス階層が変更されることがある．このような言語で直接 devirtualization を行うためには，実行時もクラス階層解析を行い，動的クラスローディングによってメソッドのオーバライドが起きて直接 devirtualization を適用した際のクラス階層の仮定が成立しなくなったときに，直接 devirtualization されたコードを無効化しなければならない．この無効化を行うために，実行中のメソッドを再コンパイルする脱最適化[4]が提案されているが，この方法は従来では複雑な実装を必要としている．

本論文では，動的クラスローディングをともなう言語において，実装が容易な直接 devirtualization 手法[5]を提案する．動的メソッド呼び出しに対して，直接 devirtualization されたコードと，動的メソッド呼び出しのコードをコンパイル時に生成する．最初は，直接 devirtualization されたコードを実行する．動的クラスローディングによって動的メソッド呼び出しが複数のメソッドを呼び出し先として持ったとき，直接 devirtualization されたコードの先頭命令を書き換えて，動的メソッド呼び出しのコードを実行する．本手法では，コード書換えによって直接 devirtualization されたコードを無効化するので，脱最適化のような複雑な再コンパイル機構が不要である．一方，コンパイル時に複数のコードを用意するので，制御フロー上に合流点が生成される．一般に制御フロー上の合流点はコンパイラが行う最適化を妨げるが，本論文では制御フロー上に合流点が存在しても十分に最適化できる手法を示す．本手法と従来提案された devirtualization 手法を組み合わせて Java Just-In-Time コンパイラに実装し，SPECjvm98 と SPECjbb2000 ベンチマークプログラムにおいて，devirtualization 手法を適用しない場合に比べて 0〜181%（平均☆24%）の性能改善を得た．

以下，2 章で関連研究を述べ，3 章で動的メソッド呼び出しの devirtualizaiton 手法について述べる．4 章では，プログラムを用いて本方式を評価した結果を示す．5 章でまとめを述べる．

2. 関 連 研 究

本章では，従来提案された devitualization 手法を分類して示す．動的メソッド呼び出しの devitualization は，その方法に基づいて，レシーバのクラスの型テストを必要とする間接 devirtualization と，テストを必要としない直接 devirtualization の 2 つに分けられる．

2.1 間接 devirtualization

間接 devirtualization では，動的メソッド呼び出しにおいてコンパイル時に呼び出し頻度の高いメソッドを決定する．動的メソッド呼び出しを実行する際に，メソッドの静的なクラスとレシーバのクラスの型テストを行う．このテストが成功した場合，呼び出し先メソッドの検索なしに高速にメソッド呼び出しを実行できる．

クラステスト[6]は，インライン展開されたメソッドのクラスとレシーバのクラスについて型テストを行い，一致した場合にはインライン展開されたコードを実行する．この場合，呼び出し先のメソッドを高速に実行できる．テストが失敗した場合には，呼び出し先メソッドを検索後メソッド呼び出しを行う．コード例を，**図 1** (a) に示す．メソッドテスト[7]は，インライン展開されたメソッドとレシーバのクラス内のメソッドについて型テストを行い，一致した場合にインライン展開されたコードを実行する．テストが失敗した場合はクラステストと同様である．コード例を，図 1 (b) に示す．これらの方式では，1 つの動的メソッド呼び出しに対してコンパイル時に 2 種類以上のコードを生成するので，制御フロー上に合流点が生成される．

レシーバのクラスが，インライン展開されたメソッドのサブクラスであり，そのクラスでメソッドをオー

☆ 文で用いる平均は，相乗平均を表す．

```
r0 = <receiver object>
r1 = load(r0 + <offset-of-class-in-object>)

if (r1 == <address-of-proper-class>) {
    <inlined code>
} else {
    r2 = load(r1 + <offset-of-method-in-class>)
    call r2
}
```

a) クラステストのコード例

```
r0 = <receiver object>
r1 = load(r0 + <offset-of-class-in-object>)
r2 = load(r1 + <offset-of-method-in-class>)
if (r2 == <address-of-inlined-method>) {
    <inlined code>
} else {
    call r2

}
```

b) メソッドテストのコード例

図 1 クラステストとメソッドテストのコード例[7)]
Fig. 1 Examples of class test and method test [7)].

バライドしていない場合，クラステストでは失敗となるが，メソッドテストでは成功となる．したがって，メソッドテストの方が型テストの成功率が高い．しかし，メソッドテストはクラステストよりロードが 1 つ多いため，オーバヘッドが大きい☆．

2.2 直接 devirtualization

動的メソッド呼び出しのレシーバがとりうるクラス集合で，コンパイル時に呼び出しメソッドを一意に決定可能か，クラス階層解析等の静的解析手法を用いて調べる．もし一意に決定できれば，静的メソッド呼び出し，またはインライン展開したメソッドを，レシーバのクラスの型テストなしに実行できる．

直接 devirtualization は，再コンパイルを必要とする方式と，必要としない方式に分けられる．

2.2.1 再コンパイルを必要としない devirtualization

局所クラス解析[8)]は，データフロー解析において，オブジェクト生成式（たとえば Java の new 式）を元に変数から型集合への写像をつくり，制御フローに沿って伝搬する．制御フローの合流点では入力の型集合の和をとり出力へ伝搬する．この結果，動的メソッド呼び出しのレシーバの型がオブジェクト生成式と同一であれば，レシーバがとるクラスは唯一であるので，動的メソッド呼び出しを静的メソッド呼び出しに変換できる．さらに，変換された静的メソッド呼び出しのコードのみを生成すればよいので，制御フロー上に合流点を生成しない．この変換は動的クラスローディングがある言語でもつねに成り立ち，再コンパイルは不要である．

動的メソッド呼び出しがとりうるクラス集合内で呼び出しメソッドがオーバライドされていなければ，メソッドを一意に決定できるので，直接 devirtualization できる．この判定には，プログラム全体のクラス階層においてどのメソッドが定義されているかを調べる，クラス階層解析[3)]が必要である．プログラム実行前にクラス階層が決定し実行時に変化しない C++のような言語では，直接 devirtualization を適用した際の解析結果は不変であり，再コンパイルは不要である．

2.2.2 再コンパイルを必要とする devirtualization

動的クラスローディングによってプログラム実行中にクラス階層が変化する Java のような言語では，直接 devirtualization を適用した際のクラス階層内でメソッドのオーバライドがないという仮定が無効になることがある．以下に述べる 2 つの方式では，devirtualization されたコードのみをコンパイル時に生成するので，生成されたコードを無効化する再コンパイルが必要である．

脱最適化[4)]は，直接 devirtualization を適用した際の仮定が，メソッド実行中のクラス階層の変化によって無効になったとき，コンパイル時に設定した安全な地点に到達するまでコードを実行する．そこで，現在実行中のスタックに積まれた最適化されたコードのためのフレームを最適化前のコードに対応するフレームに変換する．この後，直接 devirtualization される前のコードを実行し，動的メソッド呼び出しを正しく実行する．この手法で用いられるフレームへの変換は，最適化時の多くの情報を実行時まで持つ必要がある．さらに，インライン展開された 1 つのメソッドのフレームをインライン展開前の複数メソッドのフレームに対応づける等，実装が非常に複雑である．

クラス階層の変更がメソッド実行中に発生し，直接 devirutalization を適用した際のクラス階層の仮定が無効になった場合においても，動的メソッド呼び出しのレシーバが実行中のメソッドの呼び出し前に生成されていたならば，クラス階層の変化に影響されずに直接 devirutalization されたコードを実行できる．この性質を preexistence [7)]と呼ぶ．Preexistence が成り立つこと示すためには，動的メソッド呼び出しのレシー

☆ コンパイラが 2 つのロードをコード移動や部分式共通化等の最適化対象にすることで，オーバヘッドを低減できる場合もある．

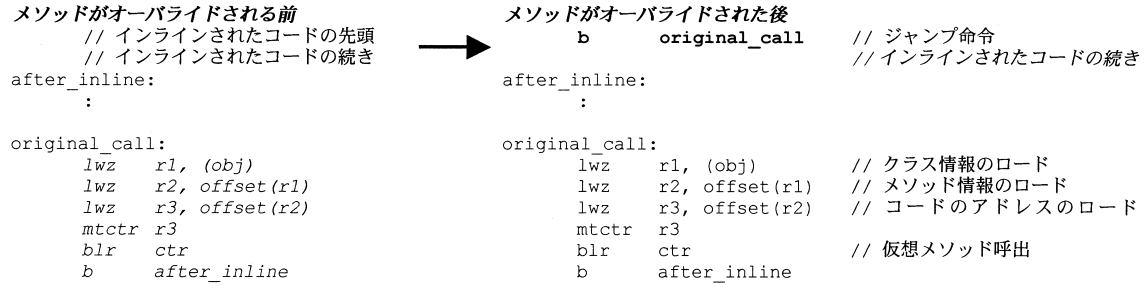


図 2 直接 devirtualization された仮想メソッド呼び出しのコード
Fig. 2 A directly devirtualized code of a virtual method call.

バがメソッドの実行中に変化しない場合を検出する．この方式では，直接 devirtualization を無効化するメソッドの再コンパイルは次回のメソッド呼び出し時に行えばよい．したがって，脱最適化のような複雑な実装を必要としない．

3. 動的メソッド呼び出しの devirtualization

本章では，我々が提案する，動的クラスローディングをともなう言語において実装が容易な動的メソッド呼び出しの直接 devirtualization 手法[5]について述べる．その後，コンパイラにおける動的メソッド呼び出しの devirtualization を実装する際の戦略を述べる．

3.1 コード書換えを用いた devirtualization

本節では，我々が実現した動的クラスローディングをともなう言語において動的メソッド呼び出しの直接 devirtualization を行う方法を示す．以下では，Java の仮想メソッド呼び出しとインタフェースメソッド呼び出しを例に用いて述べる．

コンパイル時のクラス階層解析によって仮想メソッド呼び出しの呼び出し先を一意に決定できると判断したとき，コンパイラはレシーバのクラスの型テストなしに，メソッドをインライン展開するか，静的メソッド呼び出しを実行するコードを生成する．同時に，呼び出し先メソッドの検索を行う仮想メソッド呼び出しのコードも生成する．このとき，テストなしに実行できる命令列の先頭アドレスを記録する．

インライン展開を行った場合のコード例を，PowerPC の命令セットを用いて，**図 2** に示す．まず，メソッドがオーバライドされていないときは，左のコードのように直接 devirtualization されてインライン展開されたメソッドのコードが実行され，イタリック文字の仮想メソッド呼び出しのコードは実行されない．動的クラスローディングによってクラス階層内でメソッドのオーバライドが起きて，仮想メソッド呼び出しが複数のメソッドを呼び出し先として持った場合，コンパイル時に記録されたテストなしに実行できる命令列の先頭アドレスの命令が，仮想メソッド呼び出しを実行するように `b` 命令で書き換えられる．この結果が右のコードであり，インライン展開されたコードは，もはや実行されない．このように，実行中の任意の時点で動的メソッド呼び出しのレシーバがとるオブジェクトのクラス階層が変更されても正しく実行されることを保証する．したがって，本方式はコンパイル時にオーバライドされていないすべての動的メソッド呼び出しに対して適用できる．

Java では，多重継承を実現するためにインタフェースクラスを用意している．インタフェースメソッド呼び出しも，クラス階層解析を用いて最適化できる．クラス階層解析によって，インタフェースクラスを実装するクラスが 1 つだけであることが分かれば，直接 devirtualization を適用してインタフェースメソッド呼び出しから実装しているクラスについての仮想メソッド呼び出しに変換できる．さらに，インタフェースクラスを実装しているクラスのサブクラスでメソッドをオーバライドしていないならば，変換された仮想メソッド呼び出しを直接 devirtualization できる．一般にインタフェースメソッド呼び出しでは，ループを実行して呼び出し先のメソッド検索を行うのでオーバヘッドが大きい．したがって，この最適化は非常に有効である．このとき，コンパイラは，以下の 3 種類のコードを生成する．

1. 呼び出し先が一意に決定されたメソッドを直接 devirtualization したコード
2. 仮想メソッド呼び出しのコード
3. インタフェースメソッド呼び出しのコード

インライン展開を行った場合のコード例を，**図 3** に示す．まず，インタフェースクラスを実装するクラスが 1 つで，メソッドがオーバライドされていないときは，左のコードのように直接 devirtualization によっ

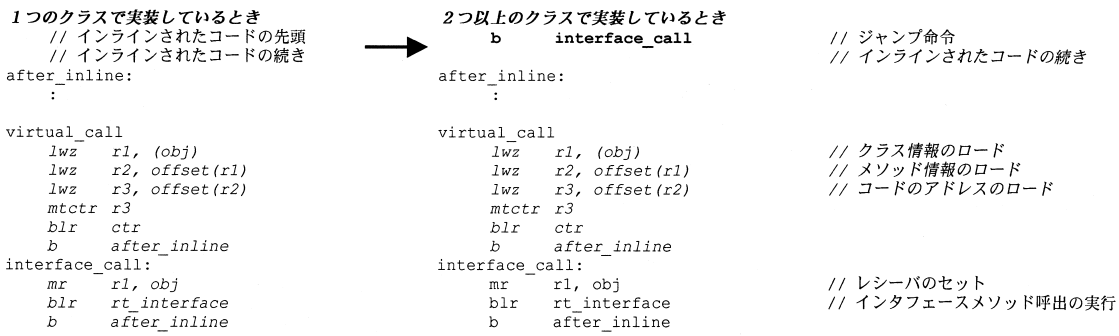


図 3 直接 devirtualization されたインタフェースメソッド呼び出しのコード
Fig. 3 A directly devirtualized code of an interface method call.

てインライン展開されたコードが実行され，イタリック文字のコードは実行されない．もし，インタフェースクラスを実装するクラスが 1 つでも，動的クラスローディングによって実装しているクラス階層内でメソッドのオーバライドが起きたら，インライン展開されたコードの先頭の命令が仮想メソッド呼び出しを実行するために b 命令で書き換えられて，インライン展開されたコードは実行されない．また，動的クラスローディングによってインタフェースクラスを実装するクラスが 2 つ以上になった場合は，インライン展開されたコードの先頭命令がインタフェース呼び出しを実行するために b 命令で書き換えられる．この結果，右のコードのように，インライン展開されたコードも，仮想メソッド呼び出しのコードも実行されない．

3.2 制御フローの合流点を持つコードの最適化

本手法では，コード書換えによって直接 devirtualization されたコードを無効化し，あらかじめ用意された動的メソッド呼び出しのコードを実行する．したがって，実装が容易である．なぜなら，脱最適化における実行中の最適化されたフレームの変換をともなうメソッドの再コンパイルという複雑な機構が不要なためである．一方，再コンパイルを不要にするために，直接 devirtualization されたコードと動的メソッド呼び出しのコード，と 2 種類以上のコードをコンパイル時に用意するので，制御フロー上に合流点が生成される．一般に制御フロー上の合流点では，データフロー解析において 1 つの変数に関する複数の情報が合流するため，データフロー最適化が妨げられる．さらに，本手法が生成する動的メソッド呼び出しのコードを含むパスでは，呼び出し先のメソッドが不定であるため副作用に影響されるすべての変数についてデータフローの情報を不定とする．この結果，制御フローの合流点においてスカラへの置換え，コード移動が制限される．また，メソッド呼び出しの結果で得られる型情報は（特にテンプレートを持たない Java では）非特定的であることが多いので，合流点で型情報が失われやすい．この結果，局所クラス解析の結果が不定になる．

このようなコードに対して，フロー合流点の除去，例外等の副作用の除去，を行う以下の最適化を適用し，フロー解析における情報を一意にする．

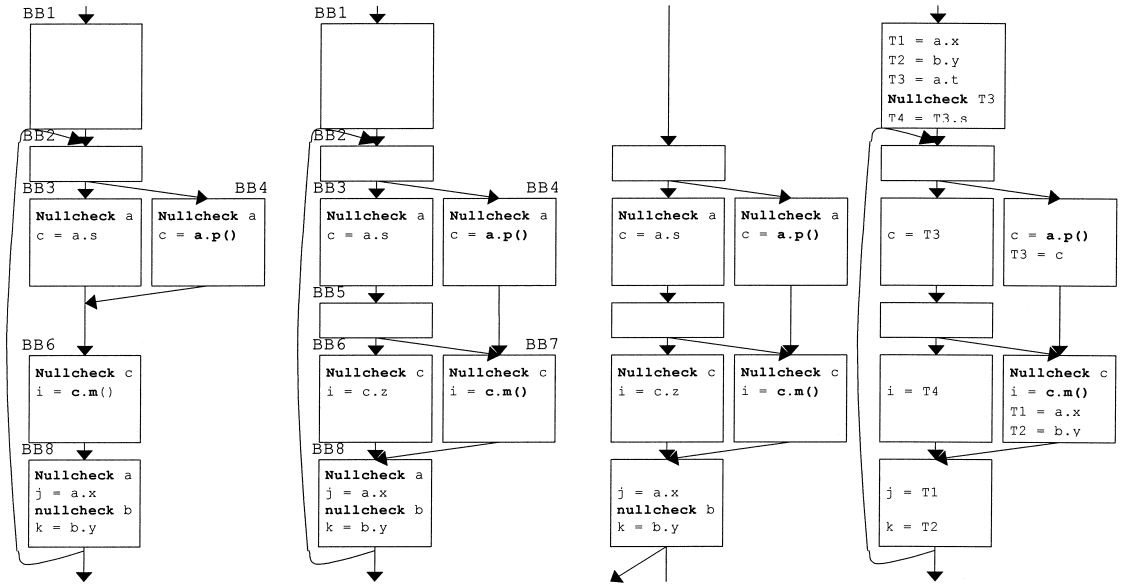
1. 型情報に関する制御フローの合流点の除去
2. ループ皮むき
3. 後方移動によるポインタチェックの除去
4. 実行確率情報を利用した補償コードの生成

この結果，スカラへの置換え等，他の最適化の適用機会を増やすことができる．以下，**図 4** (a) のプログラムを用いて，制御フローの合流点が存在しても最適化できる手法を示す．まず，本方式による直接 devirtualization を S1 に適用する．その結果が図 4 (b) である．Basic Block（BB）2 は，コード書換え点にあたる仮想的な制御フローの分岐点で，実際にコードは生成されない．BB3 と BB4 の合流点で c に関するクラスの型情報が失われている．本方式を S2 に適用後，分岐の偏りを考慮したコードの複製[9]を適用して，BB3 と BB4 のフローの合流点を BB6 の下へ移動する．この結果，BB3 → 5 → 6 は c に関して特定された型情報が到達する．その結果が，図 4 (c) である．BB5 は，コード書換え点にあたる仮想的な制御フローの分岐点で，実際にコードは生成されない．

次に，a.x，b.y の参照に対してスカラへの置換えを適用したい．そのため a と b に関する null ポインタチェックを除去したいが，Java では例外発生の順序は正確に守られなければならないので，単に nullcheck a や nullcheck b をループ外に移動する最適化は適用できない．a，b はループ内不変変数であるので，ループ皮むきを適用後，前方到達による null ポインタチェック除去を適用する．これによって，ループ内に存在する a，b に関する null ポインタチェックを除去できる．

```
Class Bar extends Foo {
  int m() { return this.z;}
}
class Foo {
  Bar s;
  int x, y, z;
  Foo p() { return this.s;}
  int m() { return this.z+1;}
  int caller(Foo a, Foo b) {
    Foo c;
    do {
S1:   c = a.p();   // BB3 and 4
S2:   i = c.m();   // BB6 and 7
S3:   j = a.x;     // BB8
S4:   k = b.y;     // BB8
    } while (cond)
    ...
  }
}
```

a) ソースコード



b) 直接 devirtualization 後の中間表現　c) コードの複製後の中間表現　d) 最終結果の中間表現

図 4 最適化の適用例
Fig. 4 An example of applying optimizations.

この結果，a.x，b.y の参照に対してスカラへの置換えを適用できる．

最後に，c.z の参照に対してスカラへの置換えを適用したい．そのため，後方移動による null ポインタチェック除去[10]を用いて nullcheck c を loop 外に移動する．さらに，BB7 はメソッドのオーバライドが起きたときにだけ実行されるため実行確率が低いと考えられる．したがって，部分冗長性除去を用いて BB7 の invoke 命令の直後に補償コードを生成して，スカラへの置換えを適用する．最終結果が，図 4 (d) である．

この例では，メソッドのオーバライドが起きないときに実行される BB2 → 3 → 5 → 6 → 8 のコードは，制御フロー上に合流点が存在しないコードと同等である．

3.3 脱出解析の適用

脱出解析の Java への適用[11]が提案されている．脱出解析では以下の条件が満たされるならば，参照されるオブジェクトは，生存区間がメソッド内部かつスレッド内で閉じているので，捕捉されていると判断する．

1. 変数が単一代入である．
2. 変数の定義が new 式によって行われる．
3. 変数のすべての参照が，動的メソッド呼び出しのレシーバや引数となる等による脱出を起こさない．

この結果，捕捉されているオブジェクトの各フィールドの定義・参照に対してスカラへの置換えを適用できる．この結果，オブジェクトをヒープに確保するオーバヘッドが削減される，変換後のプログラムに対して他の最適化を行う機会が増す等の利点が得られる．

```
class Foo {
  int x, y;
  void m() { this.x++;}
  int caller() {
    Foo f = new Foo();

    f.x = 1;
    f.m();









    f.y = f.x + 2;
    System.out.println(f.y);




  }
}
```

a) ソースコード

```
class Foo {
  int x, y;
  void m() { this.x++; }
  int caller() {
    Foo f;
    Boolean heap_allocated = false;
    int t1 = 1;
    if (!is_code_patched) {
      t1++;
    } else {
      if (!heap_allocated) {
        f = new Foo();
        heap_allocated = true;
      }
      f.x = t1;
      f.y = t2;
      f.m();
      t1 = f.x;
      t2 = f.y;
    }
    t2 = t1 + 2;
    System.out.println(t2);
    if (heap_allocated) {
      f.x = t1;
      f.y = t2;
    }
  }
}
```

b) 脱出解析後のコード

図 5 脱出解析の適用例
Fig. 5 An example of applying escape analysis.

動的メソッド呼び出しのレシーバや引数によって変数の参照が起きた場合は，前記の条件 3. によって，その変数から参照されるオブジェクトは捕捉されていないと通常は判断される．しかし，以下に示す手順で補償コードを生成することにより，動的メソッド呼び出しから脱出しているオブジェクトも捕捉されていると扱える．

1. new 式があった場所に，オブジェクトをヒープに確保したかを示す論理型変数を定義し，偽の値で初期化する．
2. 本方式が生成する動的メソッド呼び出しの前で，以下の 2 つの処理を行う．
   a) 1. で定義した論理型変数が偽ならば，new 式でヒープにオブジェクトを確保し，論理型変数に真の値を代入する．
   b) スカラに置き換えられたフィールド内の値をヒープ上のオブジェクトのフィールドへ代入する．
3. 本方式が生成する動的メソッド呼び出しの後ろで，ヒープ上のオブジェクト内のフィールドの値をスカラに置き換えられたフィールドへ代入する．
4. return 文，同一メソッド内でとらえられない例外等メソッドから脱出する直前で，1. で定義した論理型変数が真ならばスカラに置き換えられたフィールド内の値をヒープ上のオブジェクト内のフィールドへ代入する．

通常，プログラム中に元から存在する動的メソッド呼び出しの実行確率は高いと考えられるので，これらの補償コードを生成することはその実行のオーバヘッドに見合わないと考えられる．4.1 節の実験結果が示すように，コンパイル時にメソッドのオーバライドがない動的メソッド呼び出しは，メソッドのオーバライドが起きることは非常に少ない．本方式が生成する動的メソッド呼び出しは，動的クラスローディングによってメソッドがオーバライドされた後のみ実行されるので，この動的メソッド呼び出しの実行確率は低いと考えられる．したがって，補償コードの実行オーバヘッドは無視できる．補償コードの生成によって，本方式が生成する動的メソッド呼び出しから脱出していたオブジェクトでも捕捉されているとして扱うことができる．この例を**図 5** に示す．このように，本方式による直接 devirtualization によって動的メソッド呼び出しが残った場合でも，その特性を利用して補償コードを生成することで，脱出解析による最適化の機会が十分に得られる．

以上のように，本論文で提案したコード書換え方式による直接 devirtualization は実装が容易であるという利点を持つ．また，制御フロー上に合流点を生成するが，メソッドのオーバライドが起きない場合のコードの質を，各種の最適化によって制御フローの合流点を持たない場合と同等にできることを示した．

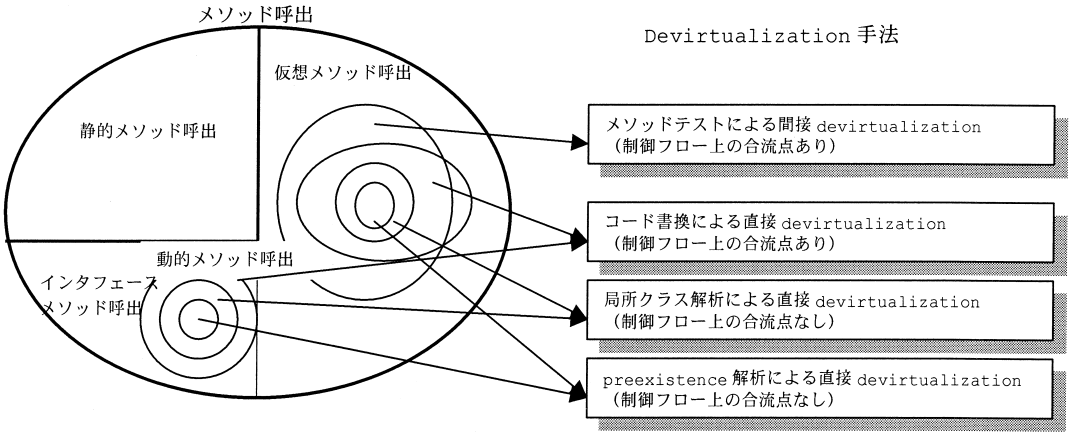


図 6 devirtualization 手法の適用分類
Fig. 6 Venn diagram of applicable categories of devirtualization techniques.

3.4 devirtualization の戦略

本節では，コンパイラに動的メソッド呼び出しの devirtualization を実装する場合の様々な devirtualization 方式の組合せについて述べる．

前節で提案したコード書換えによる直接 devirtualization は，実装の容易さと，preexistence 解析や局所クラス解析よりも広い適用範囲を特徴とする．制御フローに合流点を生成することが，各種最適化によるコンパイル時間の増加や，補償コードによるコード量の増加を招く可能性がある．したがって，できる限り制御フロー上に合流点を生成しない devirtualization 方式を適用したい．制御フロー上に合流点を生成しない脱最適化方式は，本方式と同様にコンパイル時にメソッドがオーバライドされていないすべての動的メソッド呼び出しに適用可能である．しかし，脱最適化は 2 章の関連研究でも述べたように実装が非常に複雑であるため，手間を考えた場合，候補にあげにくい．

関連研究で述べたように，制御フロー上に合流点を生成しない devirtualization 方式に，preexistence 解析や局所クラス解析がある．適用範囲は本方式や脱最適化より狭いが，これらの方式は実装が容易である．preexistence 解析によって必要になるメソッドの再コンパイルは，メソッドの次回の起動時に行えばよい．局所クラス解析はメソッドの再コンパイルの必要がなく，解析は静的型付けを行う言語ではデータフロー方程式を解くだけでよい．これらの方式は，制御フローの上に合流点がないため，脱最適化と同等のコードを生成できる．もし，動的メソッド呼び出しに対してこの 2 つの方式が適用できない場合，本方式を適用する．

最後に，コンパイル時にすでに，動的メソッド呼び出しがとるレシーバのクラス階層において，メソッドがオーバライドされていた場合，メソッドテストによる間接 devirtualization を適用する．この方式では型テストが必要になる．この場合も，制御フローの合流点が生成される．しかし，3.1 節で述べた最適化を行うことで，このオーバヘッドを低減できる．

したがって，コンパイラに動的メソッド呼び出しの devirtualization を実装する場合，実装の容易さと最適化能力を考慮して，以下の手順で devirtualization を適用するのが適切である．

1. preexistence 解析を適用し，制御フローの合流点を生成しない，直接 devirtualization を行う．
2. 局所クラス解析を適用し，制御フローの合流点を生成しない，直接 devirtualization を行う．
3. 本方式を適用し，制御フローの合流点を生成する，直接 devirtualization を行う．
4. 動的メソッド呼び出しが複数のメソッドを呼び出し先として持つ場合，メソッドテストによる間接 devirtualization を行う．

上記の方式の動的メソッド呼び出しへの適用範囲の分類を，**図 6** に示す．

4. 実験結果

本章では，前章で述べた戦略に基づいてコンパイラに実装した動的メソッド呼び出しの devirtualization の効果について述べる．実験のために，IBM Developers Kit for AIX，Java Technology Edition，Version 1.3 の Just-In-Time コンパイラに，3.4 節で述べた devirtualization 方式を実装した．このコンパイラは，静的メソッド呼び出しのインライン展開，動的メソッド呼び出しの devirtualization，脱出解析，複写除去・定数伝搬・不要コード除去・共通部分式除去・スカラへの

表 1 静的メソッド呼び出しと動的メソッド呼び出しの特性
Table 1 Characteristics of static and dynamic method calls.

Program	Static Call	Virtual Call	Monomorphic Virtual Call %		Interface Call	Monomorphic Interface Call %	
			Lib.	App.		Lib.	App.
compress	225,978,320	12,461	47.0%	24.8%	480	43.3%	56.2%
jess	78,377,418	36,871,886	0.2%	83.8%	706,539	0.0%	0.7%
db	30,487,129	52,529,637	0.1%	97.1%	14,931,573	0.0%	100.0%
javac	56,771,254	48,356,179	5.1%	62.3%	3,381,214	0.0%	99.8%
mpegaudio	98,393,731	9,853,928	0.2%	33.2%	182,254	0.1%	99.9%
mtrt	17,336,758	269,740,831	0.3%	90.6%	436	48.6%	51.4%
jack	24,273,005	25,219,317	20.3%	59.5%	4,155,349	0.0%	55.0%
jbb	118,207,158	170,208,179	15.4%	81.0%	3,949,449	0.1%	99.9%
平均			74.2%			49.7%	

Static Call: 静的メソッド呼出の総実行回数.
Virtual Call: 仮想メソッド呼出の総実行回数.
Monomorphic Virtual Call: 単一メソッドを呼ぶインタフェースメソッド呼出が実行された割合.
Interface Call: インタフェースメソッド呼出の総実行回数.
Monomorphic Interface Call: 単一メソッドを呼ぶインタフェースメソッド呼出が実行された割合.
Lib.: メソッド呼出が Java ライブラリクラス内で実行された割合.
App.: メソッド呼出がアプリケーションクラス内で実行された割合.

置換え・不要な例外チェックの除去等のデータフロー解析等，多くの最適化を実装したコンパイラである．本論文で述べた最適化は，ループ皮むき，分岐の偏りを考慮したコードのコピーによって得られるより正確な型情報に基づいた直接 devirtualization，脱出解析のための補償コード生成を除いてすべて実装した．

すべての測定は，IBM 社の RS/6000 44P モデル 170（POWER3-II 400 MHz，メモリ 768 MB），AIX 4.3.3 を使用して行った．ベンチマークプログラムには，SPECjvm98 [12] に含まれる 7 つのプログラム（compress，jess，db，javac，mpegaudio，mtrt，jack）を size=100 で実行，SPECjbb2000 [13] を warehouse=1 で実行したものを用いた．この処理系には，頻繁に実行されるメソッドだけをコンパイルする選択コンパイル機能が実装されているが，今回の実験ではすべてのメソッドをコンパイルする設定で行った．

4.1 動的メソッド呼び出しの特性

まず，最適化前のプログラムの静的メソッド呼び出し，仮想メソッド呼び出し，インタフェースメソッド呼び出しの特性を**表 1** に示す．33.4〜97.2%（平均 74.2%）の仮想メソッド呼び出しが，単一メソッドを呼び出している．mpegaudio を除いて，アプリケーションのクラス内の多くの仮想メソッド呼び出しが単一メソッドを呼び出している．この結果から，多くの仮想メソッド呼び出しに直接 devirtualization を適用する機会があると予想される．compress は静的メソッド呼び出しが非常に多いので devirtualization の効果がないと予想される．また，db ではインタフェースメソッド呼び出しの回数が多い．

次に，メソッドテスト，コード書換え，局所クラス解析，preexistence 解析の devirtualization 手法を適用したときの動的メソッド呼び出しの特性を調べた．**図 7** は，動的メソッド呼び出しに devirtualization 手法を適用した場合，どの devirtualization 方式によって動的メソッド呼び出しの実行回数が減少したかを示すグラフである．左側のバーが devirtualization 手法をまったく適用しない場合，右側のバーがメソッドテスト，コード書換え，局所クラス解析，preexistence 解析の 4 つの devirtualization 手法を適用した場合を示す．なお，SPECjbb2000 は一定時間あたりのスループットを測るベンチマークであるため，devirtualization 手法の適用前後を比較できない．したがって，グラフから除いた．コード書換えによる実行回数の減少は，jess，db，mtrt，jack で大きい．特に，db，mtrt では，インスタンス変数または配列要素の参照を行う小さなメソッドを呼び出す頻度が非常に多く，このメソッド呼び出しが直接 devirtualization によってインライン展開された．局所クラス解析による実行回数の減少は，db，javac，jack で大きい．特に，db では，局所クラス解析の結果を用いたクラス階層解析によっ

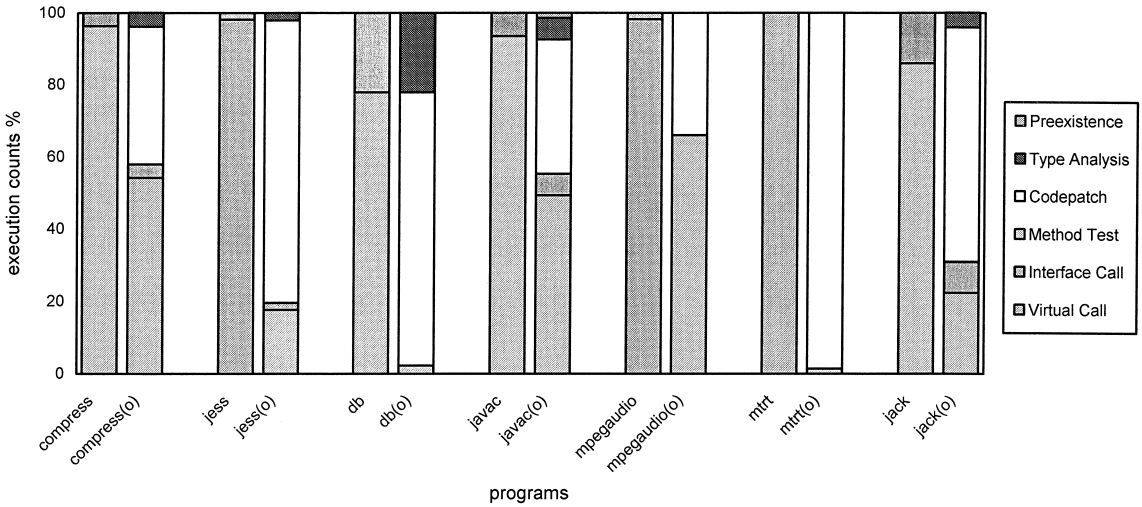


図 7 それぞれの devirtualization 手法によって減少したメソッド呼び出し実行回数
(o) なしは devirutalization 手法適用前，(o) ありは devirutalization 手法適用後
Fig. 7 Breakdown of call sites optimized by each devirtualization technique (in execution counts). without (o): before applying devirtualization techniques, with (o): after applying devirtualization techniques.

表 2 実行回数の減少率に基づく各 devirtualization 手法の効果
Table 2 Effectiveness of devirtualization techniques (in execution counts).

Program	N から MCTP での削減率			M から MCTP での削減率	MC から MCTP での削減率	MCT から MCTP での削減率
	仮想メソッド呼出	インタフェースメソッド呼出	両方	メソッドテストによってテスト無しで実行できるコード	コード書換えによってテスト無しで実行できるコード	コード書換えによってテスト無しで実行できるコード
compress	43.8%	0.6%	42.2%	100.0%	19.4%	23.4%
jess	82.0%	0.7%	80.4%	100.0%	6.5%	26.0%
db	97.1%	99.9%	97.7%	100.0%	0%	0%
javac	47.2%	8.7%	44.7%	91.0%	10.0%	27.1%
mpegaudio	32.8%	99.8%	34.0%	100.0%	4.5%	93.1%
mtrt	98.6%	0.7%	98.6%	100.0%	6.4%	31.1%
jack	74.0%	39.8%	69.1%	100.0%	3.0%	22.8%
平均	63.0%	7.2%	43.0%	98.7%	2.2%	11.8%

N: devirtualization 適用無し．
M: メソッドテスト．
MC: メソッドテスト，コード書換え．
MCT: メソッドテスト，コード書換え，局所クラス解析．
MCTP: メソッドテスト，コード書換え，局所クラス解析，preexistence 解析．

てインタフェースメソッド呼び出しを仮想メソッド呼び出しに変換でき，さらにその仮想メソッド呼び出しを直接 devirtualization できた．preexistence 解析は，主に制御フローの合流点を持つ直接 devirtualization のコードから合流点を取り除く変形に使われるため，javac を除いて動的メソッド呼び出しの実行回数はほとんど変化しない．

次に，メソッドテスト，コード書換え，局所クラス解析，preexistence 解析の devirtualization 手法を順々に適用していったときの，静的メソッド呼び出し，動的メソッド呼び出しの特性を調べた．**表 2** は，メソッドテスト，コード書換え，局所クラス解析，preexistence 解析，と 4 つの devirtualization 手法の適用によって，動的メソッド呼び出し回数と直接 devirtualization されたコードの実行回数がどのように減少したかを示した表である．ここでも，SPECjbb2000 は表から除いた．4 つの手法すべてを適用した場合には，34.0〜98.6%（平均 43.0%）動的メソッド呼び出しの回数が減少した．動的メソッド呼び出しの実行回数の多いベンチマークでは，mtrt での動的メソッド呼び

表 3 直接 devirtualization された動的メソッド呼び出しサイトの特性
Table 3 Characteristics of directly devirtualized call sites.

Program	テスト無しに実行可能なコードの実行割合	コード書換えがおきた動的メソッド呼出しサイト数	再コンパイルされたメソッド数
compress	96.5%	11	4
jess	100%	13	4
db	100%	11	4
javac	99.8%	14	15
mpegaudio	99.9%	11	4
mtrt	100%	11	4
jack	99.5%	15	4
jbb	100%	12	2
平均	99.5%	12.2	4.3

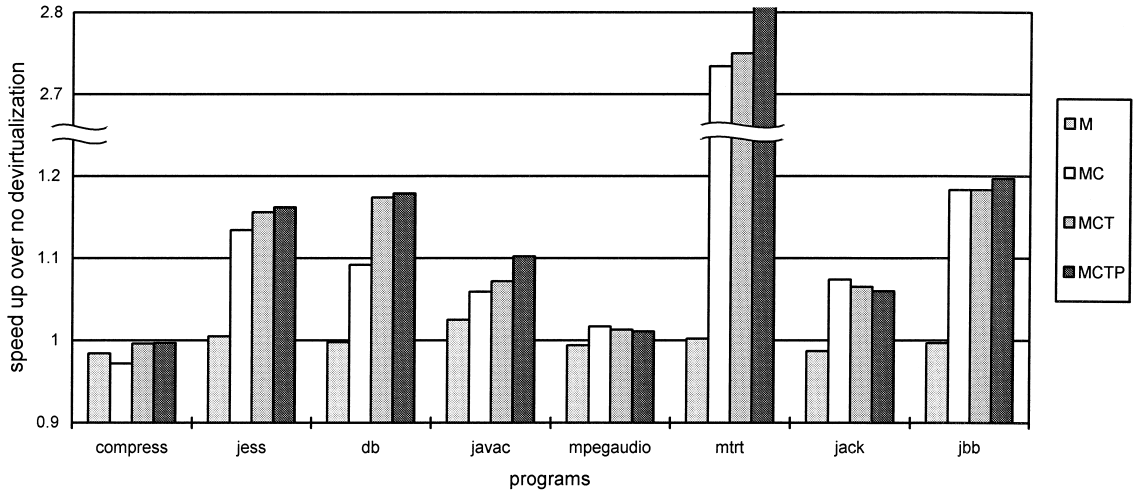


図 8 SPECjvm98 と SPECjbb2000 の性能向上割合
Fig. 8 Performance improvement for SPECjvm98 and SPECjbb2000.

出し回数の減少が特に大きい．このプログラムでは，インスタンス変数の参照を行う小さなメソッドの呼び出し頻度が非常に多い．このメソッド呼び出しが直接 devirtualization によってインライン展開されたため，メソッド呼び出しの回数が減少した．また，コード書換えによって直接 devirtualization された動的メソッド呼び出しのうち 0.0〜93.1%（平均 11.8%）が，preexistence 解析によって制御フロー上の合流点を生成しない devirtualization に変換可能である．

表 3 は，メソッドテスト，コード書換え，局所クラス解析，preexistence 解析と 4 つの手法をすべて適用した場合に，コード書換えによって直接 devirtualization されたコードがどれくらいの割合で実行されるか，書き換えられる動的メソッド呼び出しサイトの数，preexistence 解析によって起きる再コンパイルの数である．直接 devirtualization されたコードが実行される割合は，96.5〜100%（平均 99.5%）と非常に高い．つまり，最初のメソッド呼び出し時にオーバライドされていない動的メソッド呼び出しは，後からオーバライドされることが非常に少ない．また，コード書換えが起きる呼び出しサイト数や再コンパイルされるメソッドの数も非常に少ない．

4.2 性能評価

各種 devirtualization 手法の組合せによる，性能向上を測定した結果を**図 8** に示す．すべての値は，devirtualization を行わなかったときの実行時間に対する速度向上比である．グラフ内のそれぞれのバーは，以下の devirtualization を適用したときの値である．

- **M** メソッドテスト
- **MC** メソッドテスト，コード書換え
- **MCT** メソッドテスト，コード書換え，局所クラス解析

- **MCTP** メソッドテスト，コード書換え，局所クラス解析，preexistence 解析

間接 devirtualization によるメソッドテストだけを適用した場合には，ほとんど性能に変化がない．compress，mpegaudio，jack，jbb では，性能が悪化している．これは，すでにメソッドがオーバライドされたメソッド呼び出しにおいて，インライン展開されたコードが実行されずに，動的メソッド呼び出しが実行されてテストの分オーバヘッドが増加したためと考えられる．

コード書換えによる直接 devirtualization も適用した場合には，平均で 21%性能が改善された．特に mtrt での改善が大きい．4.1 節で述べたように，このプログラムでは，インスタンス変数の参照を行う小さなメソッドの呼び出し頻度が非常に多い．このメソッド呼び出しが直接 devirtualization によってインライン展開されて，オーバヘッドなしで実行可能になったため，性能が大きく改善された．

さらに，局所クラス解析による直接 devirtualization も適用した場合には，平均で 23%性能が改善された．特に，db での改善が大きい．これは，4.1 節で述べたように，局所クラス解析の結果を用いたクラス階層解析によってインタフェースメソッド呼び出しを仮想メソッド呼び出しに変換でき，さらにその仮想メソッド呼び出しを直接 devirtualization してインライン展開されたコードを実行できたためである．さらに，jess も性能改善が大きい．これは，局所クラス解析の結果を用いてコード書換えによる直接 devirtualization が可能になり，インライン展開されたコードを実行できたためである．

最後に，preexistence 解析による直接 devirtualization も適用し，3.4 節で述べた devirtualization の戦略をすべて実行した場合には，平均で 24%性能が改善された．mtrt でさらに性能が改善された理由は以下のとおりである．preexistence 解析を用いた devirtualization によって，頻繁に呼び出されるメソッド内にある動的メソッド呼び出しが，脱出解析においてオブジェクトの脱出点となる動的メソッド呼び出しを生成することなく直接 devirtualization された．その結果，捕捉されたオブジェクトの各フィールドに対してスカラへの置換えを適用できたためである．javac でさらに性能が改善された理由は，図 7 で示したように preexistence 解析で動的メソッド呼び出しの実行回数が減少したためである．

5. ま と め

本論文では，動的クラスローディングをともなう言語における，実装が容易な動的メソッド呼び出しの直接 devirtualization 手法を提案した．動的メソッド呼び出しに対して，直接 devirtualization されたコードとメソッドがオーバライドされた場合の 2 種類のコードをコンパイル時に生成し，最初は前者を実行し，メソッドのオーバライドが起きたときにコード書換えによって後者を実行する手法を述べた．本手法では，コードの書換えによって直接 devirtualization されたコードを無効化するので，脱最適化のような複雑な実装が不要である．一方，再コンパイルを不要にするために，コンパイル時に 2 種類のコードを用意するので，制御フロー上に合流点が生成される．一般に合流点はコンパイラの最適化を妨げるが，制御フロー上に合流点が存在しても最適化できることを示した．さらに，本手法と他の devirtualization を Java の Just-In-Time コンパイラに実装し，SPECjvm98 と SPECjbb2000 において devirtualization を行わない場合に比べ平均 24%性能を改善できることを示した．

謝辞 本研究を進めるにあたり，貴重なご意見をいただいた日本 IBM 東京基礎研究所のネットワーク・コンピューティング・プラットフォームグループの皆様に深く感謝いたします．

参 考 文 献

1) Grove, D., Dean, J., Garrett, C. and Chambers, C.: Profile-Guided Receiver Class Prediction, *Proc. Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA '95*, pp.108–123 (1995).
2) Lee, J., Yang, B.-S., Kim, S., Lee, S., Chung, Y.C., Lee, H., Lee, J.H., Moon, S.-M., Ebcioglu, K. and Altman, E.: Reducing Virtual Call Overheads in a Java VM Just-In-Time Compiler, *4th Annual Workshop on Interaction between Compilers and Computer Architectures*, pp.21–33 (2000).
3) Dean, J., Grove, D. and Chambers, C.: Optimization of object-oriented programs using static class hierarchy, *Proc. 9th European Conference on Object-Oriented Programming — ECOOP '95*, Volume 952 of Lecture Notes in Computer Science, pp.77–101, Springer-Verlag (1995).
4) Holzle, U., Chambers, C. and Ungar, D.: Debugging optimized code with dynamic deoptimization, *Proc.ACM SIGPLAN '92 Conference*

on Programming Language Design and Implementation, pp.32–43 (1992).
5) Ishizaki, K., Kawahito, M., Komatsu, H. and Nakatani, T.: A Study of Devirtualization Techniques for a Java Just-In-Time Compiler, *Proc. Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA '00*, pp.294–310 (2000).
6) Calder, B. and Grunwald, D.: Reducing Indirect Function Call Overhead in C++ Programs, *Proc. ACM SIGPLAN '94 Symposium on Principles of Programming Languages*, pp.397–408 (1994).
7) Detlefs, D. and Agesen, O.: Inlining of Virtual Methods, *Proc. 13th European Conference on Object-Oriented Programming — ECOOP '99*, Volume 1628 of Lecture Notes in Computer Science, pp.258–278, Springer-Verlag (1999).
8) Palsberg, J. and Schwartzbach, M.I.: Object-Oriented Type Inference, *Proc. Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA '91*, pp.146–161 (1991).
9) 古関，小松：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，情報処理学会論文誌：プログラミング，Vol.42 No.SIG2 (PRO 9), pp.26–36 (2001).
10) Kawahito, M., Komatsu, H. and Nakatani, T.: Effective Null Pointer Check Elimination Utilizing Hardware Trap, *Proc. 9th International Conference on Architectual Support for Programming Langauges and Operating Systems*, pp.139–149 (2000).
11) Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V.C. and Midkiff, S.: Escape Analysis for Java, *Proc. Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA '99*, pp.1–19 (1999).
12) The Standard Performance Evaluation Corp.: SPECjvm98 Benchmarks. available at http://www.spec.org/osg/jvm98/
13) The Standard Performance Evaluation Corp.: SPECjbb2000 Benchmarks. available at http://www.spec.org/osg/jbb2000/

(平成 13 年 2 月 16 日受付)
(平成 13 年 11 月 14 日採録)

石崎 一明（正会員）
1992 年早稲田大学大学院理工学研究科修士課程修了．同年日本 IBM 入社．以来，東京基礎研究所において，並列処理，最適化コンパイラに関する研究に従事．

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．

川人 基弘（正会員）
1968 年生．1991 年早稲田大学理工学部電子通信学科卒業．同年日本 IBM に入社．現在，同社東京基礎研究所に所属．コンパイラの研究に従事．

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）．