実行環境が異なる 2 つのコード間の遷移を行う際の効果的な最適化手法

川人基弘[†] 小松秀昭[†]

動的コンパイラにおいて，メソッドの途中で再コンパイルを行い，より高いレベルのコンパイルコードに実行を移すことが有効な場合が多々ある．しかし，遷移先メソッドをコンパイルする際には，合流点に飛びこむ制御フローのエッジを考慮する必要があり，このエッジが合流点をまたいだ最適化を妨げることがある．本稿は，まず最初に合流点以前の 2 つの異なるレベルのコード領域が意味上は同じであることを利用して，最適化の抑制を解消する方法を述べる．続いて，この手法と補償コードを生成する手法を組み合わせることにより，コピー伝播の抑制を解消する方法を述べる．我々は本稿で述べる手法を IA-64 上の IBM Java JIT Compiler 上で実装し，いくつかのベンチマークやアプリケーションを使って評価を行った．その結果，これらの最適化の抑制を解消することにより，大きくパフォーマンスを改善することができた．

Effective Optimization Techniques for Transferring between Two Code Blocks in Different Execution Environments

Motohiro Kawahito[†] and Hideaki Komatsu[†]

In a dynamic compiler, there are many opportunities for transferring execution to the higher-level compiled code in the middle of a method by recompiling the method. However, the control flow edge that jumps into a merge point may suppress several optimizations along the normal path in the target method. In this paper, we first present how to avoid suppressions of several optimizations by utilizing the fact that the semantics of the two code blocks before a merge point is the same. Next, we present how to avoid suppression of copy propagation by combining this semantic-based technique and a technique for generating a compensation code. We implemented these techniques in the IBM Java JIT Compiler on IA-64 and evaluated them using several benchmarks and applications. As a result, we can greatly improve performance by avoiding these suppressions.

1. はじめに

近年 Java の普及により動的コンパイラが注目されている．しかし，動的コンパイラの持つ問題点としてコンパイル時間が全体の実行速度に直接影響する点があげられる．このため，動的コンパイラだけを使ってすべてのメソッドをコンパイル・実行する方法では，強力な最適化をなかなか適用することができなかった．

この問題を解決するために，1 つのメソッドを実行頻度によって多段階の最適化レベルで実行する方法が提案されている．たとえば，我々のシステムや Sun の HotSpot は個々のメソッドについて最初はインタプリタで実行させ，ある程度頻繁に実行されると判断できるメソッドについて動的コンパイラでコンパイル・実行を行っている[7),10),11)]．さらに，我々のシステムは動的コンパイラ内でも複数の最適化レベルを使っている[11)]．

多段階の最適化レベルを使って実行することにより，多くのコンパイル時間が必要となる強力な最適化を実装することができるようになった．また，これらの方法はコンパイル時間の削減だけにとどまらず，最適化にとってもメリットがある．たとえば，クラスの初期化チェックは，基本的にはメソッド呼び出しとして扱う必要があるため，メモリの最適化を抑制する[12)]．多段階の最適化レベルを使うことにより，低い最適化レベルのコードで constant pool の resolution やクラスの初期化を行うことができるため，高い最適化レベルでは resolution やクラスの初期化が行われた状態でメソッドのコンパイルを行うことができるといった利点がある．

我々は，最初の多段階の最適化レベルとして，イン

† 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.

タプリタと動的コンパイラの組合せを実装した際に，メソッドの入り口にカウンタを設け，そのメソッドがある閾値以上呼ばれたときに動的コンパイラに遷移するという方法を実装した[10]．しかし，この方法はメソッドが終了するまでは実行がインタプリタ上で行われてしまうという欠点があった．そのため，科学計算処理などの深いネストを持つループを含むプログラムでは，深刻なパフォーマンスの低下を引き起こしていた．これは，インタプリタとコンパイルされたコードの速度性能の差が 6 倍から 10 倍[11]と大きく異なるためである．

この問題を解決するために，我々はループのバックエッジでもカウンタを更新し，ループの途中でインタプリタから JIT コンパイラに実行を遷移するように変更を行った[10]．同様の方法は，HotSpot も使っている[7]．この方法によって，インタプリタ上で長い間プログラムが実行され続けるという問題は解決された．一般的にいうと，低い最適化レベルから高い最適化レベルへ遷移する際に，この 2 つのレベルの実行性能が大きく異なる場合には，途中遷移を行って高い最適化レベルに移行したほうが全体のパフォーマンスが向上する．インタプリタからコンパイラへの遷移は，この 2 つのレベルの実行性能が大きく異なる典型的な例である．

また，動的コンパイラならではの最適化技術として，仮定に基づく最適化（assumption based optimization）がある．たとえば，仮想メソッドの呼び出し（`invokevirtual`）をインラインする方法として，(1) ガードつきでインラインする方法[2]，(2) コンパイル時のクラス階層を仮定して，ガードなしでインラインし，仮定が崩れたときに再コンパイルするという方法[7]がある．後者の方法を採用したときには，メソッドがオーバロードされた場合などはコンパイルした時点の仮定が崩れるため，正しさを保障するためにメソッドの途中であっても新しい仮定でコンパイルされたコードに遷移しなければならない．

別の例をあげると，実行時のプロファイル情報を使った最適化[12]を適用する場合には，コンパイルした時点のプロファイル情報が時間の経過とともに不正確になっていくことがある．この状態を放置するとパフォーマンスが徐々に低下していく．パフォーマンスの低下が大きくなったときには，本来の最適化性能であるコードにできるだけ早く遷移するためにメソッドの途中で再コンパイルして遷移を行ったほうがよい．

このように，動的コンパイラを用いる際に非常に重要となる技術が，メソッドの途中で実行環境が異なる

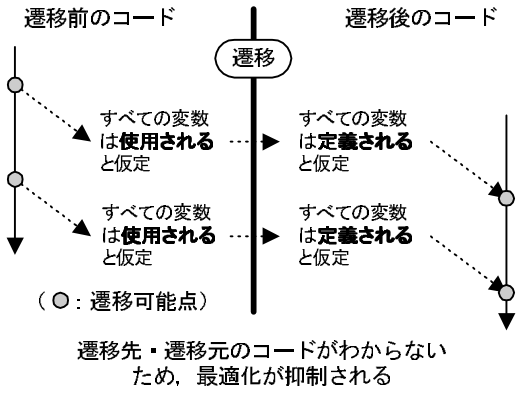

図 1 メソッドの途中遷移の問題点
Fig. 1 A problem of transferring into a middle of a method.

2 つのコード間の遷移（以下，**メソッドの途中遷移**と呼ぶ）である．今までの話をまとめると，メソッドの途中遷移が必要となる場合は次の 2 種類ある．

- 仮定に基づく最適化の**正しさを保障**するため
- 実行性能が高いコードに**できるだけ早く遷移する**ため

このような場合にメソッドの途中遷移は必要となるが，途中遷移を仮定して最適化すると，遷移前および遷移後のコードに対して最適化が抑制されるという問題が新たに発生する．この理由は，異なるコード間を遷移する場合には，一般的に遷移先や遷移元のコードがどのようになっているか分からないためである．**図 1** の「遷移前のコード」では，遷移先の変数の使用状況が分からないため，すべての変数が使用されると仮定する必要がある．また，「遷移後のコード」では，遷移元の変数の定義状況が分からないため，すべての変数が定義されていると仮定する必要がある．

遷移前のコードと遷移後のコードに対する最適化に対する影響度という意味では，遷移後のコードに対する影響度のほうが大きい．これは，コピー伝播や共通部分式の除去といった多くの最適化は，遷移可能点での仮定的な変数定義によって最適化が止まるためである．これらの仮定的な変数定義が，遷移元からの合流エッジに沿って流入してくるために，多くの最適化を抑制する．しかし，実行されるプログラムが遷移前と遷移後で変わらないという性質を利用すると，最適化の抑制を解決することができる．本稿では，遷移後の実行環境で最適化が抑制されてしまう問題を解決する方法を提案する．

2. 関連研究

実行環境が異なる 2 つのコード間の遷移を行う方法

として，Hölzle らはプログラムのデバッグのために脱最適化（Deoptimization）[1]を提案している．この方法は，最適化されたコードと最適化されていないコードの 2 つのバージョンを用意して，通常は最適化されたコードが実行されるようにする．デバッグのためにユーザがインタラプトを発生させると，あらかじめ決められた遷移可能点まで実行を継続し，そこで最適化されていないコードに遷移することにより，ソースレベルのデバッグが可能となる．遷移可能点はメソッドの入り口とループのバックエッジが対象となっている．最適化されたコード内では，遷移可能点では必ずソースコードとの整合性がとれている状態になっている．Hölzle らの手法は，遷移先が「最適化されていないコード」という，途中遷移の形態の中でも特殊なケースを扱っているため，遷移によって遷移先の最適化が抑制されてしまう問題そのものが存在しない．問題としては，遷移前の最適化が抑制されるという点だが，現実的には先ほど述べたように，遷移前の最適化が抑制される問題はそれほど大きなものではないと考えられる．

我々は，選択的コンパイルを行うために動的コンパイラとインタプリタを組み合わせている．さらに我々は，動的コンパイラ向けのコンパイル方法として，複数の最適化レベルでメソッドをコンパイルする方法を提案している[11]．この方法は，インタプリタから遷移した場合には，そのメソッドをまず低い最適化レベル（文献 11) 上の Quick に対応）で 1 回コンパイルする．そして，この版に対してプロファイルをとり，頻繁に実行されると判断されたメソッドについては，高い最適化レベル（文献 11) 上の Full に対応）で再コンパイルを行い，前の版と置き換える．これにより，このメソッドの**次の実行**からは，より強力な最適化を行った版が実行される．さらには，メソッドの特殊化（specialization）を行うこともできる．

この方法の長所は，高い最適化レベルでメソッドをコンパイルした版には遷移可能点が含まれていないため，メソッドの途中遷移の問題が低い最適化レベルでコンパイルされた版に閉じるという点である．そのため，高い最適化レベルで再コンパイルされたコードについては，本稿で述べる最適化手法は影響を与えない．しかし，この方法の短所としてはメソッドが複数回呼ばれることを前提にしている点とアップグレードが行われることを前提にしている点があげられる．科学計算処理などの 1 つのメソッドが長時間実行されるようなプログラムでは，この方法だけではパフォーマンスが悪くなる場合がある．また，プログラムによっては明確なホットスポットが存在せず，プログラム全体が平均的に実行されるようなものも存在する．このようなプログラムに対しては，アップグレードコンパイルがまったく行われない場合も多い．このようなときには，やはりこの方法だけではパフォーマンスが悪くなる．

我々のシステムでは，低い最適化レベルでもいくつか最適化を行っているために，低い最適化レベルと高い最適化レベルの実行性能の差は平均すると 15%程度[11]にとどまっている．しかし，たとえばまったく最適化を行わない実行環境を低い最適化レベルとするような場合には，メソッドによっては高い最適化レベルとの実行性能が大きく異なってくる（我々の実験では平均約 2 倍[11]）．最適化レベルによって実行性能が大きく異なるようなメソッドについては，先ほど述べたように途中遷移を行って高い最適化レベルに移行したほうがパフォーマンスが向上する．このようなときには，やはり途中遷移によって最適化が抑制されるため，本稿で述べる最適化が必要になってくる．

また，文献 11) の再コンパイルの手法はコンパイラのデバッグをより困難にする点も，実用上は重要な問題となる．この一番大きな原因は，アップグレードコンパイルを決定するプロファイリングをタイマを使って非同期に取得しているために，問題が起きた場合にその発生した状況をなかなか再現できない場合があるためである☆．4 章の実験結果では，タイミングによって再コンパイルの挙動が変わるケースを紹介する．

3. 我々の手法

この章では，我々がどのようにメソッドの途中遷移の問題を解決しているかについて説明する．なお以下の説明では，主にインタプリタからの遷移についての問題点と解決法について述べているが，「インタプリタ」という言葉を「遷移元」と読み替えれば，インタプリタからの遷移に限らず，異なる 2 つの実行環境間を遷移する場合一般に対して適用可能である．

3.1 式を後方移動させる際の考慮点

図 2 にメソッドの途中遷移を行う際に最適化が抑制される例を示す．図 2 (a) の途中遷移が行われない場合には，ループ不変の「nullcheck a」および後続するメモリアクセス「a.F」はループの外に移動する．

一方，図 2 (b) の途中遷移が行われる場合には，このような変形を行うことはできない．仮に，図 2 (a)

☆ 本稿を書いた時点では，我々の JIT コンパイラはオプションを明示的に指定しない限り，文献 11) の手法は有効にならない．

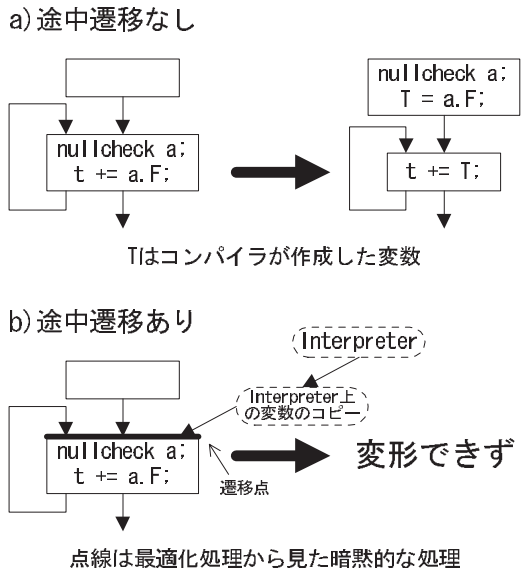

図 2 メソッドの途中遷移によって最適化が抑制される例
Fig. 2 Example of suppressing an optimization by transferring into a middle of a method.

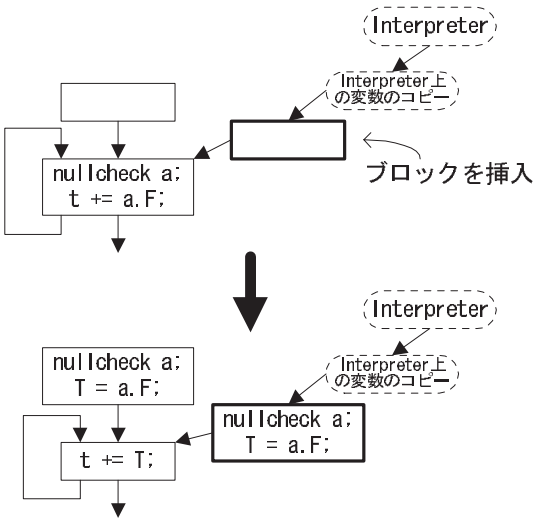

図 3 図 2 (b) の問題点の解決方法
Fig. 3 A solution of Fig. 2 (b).

のように変形を行うと，インタプリタ上では「a.F」の結果を保持している変数 T の定義がないため，遷移したときに間違った結果となる．そのため，メソッドの途中遷移を行う場合には，遷移可能点を持つループについて図 2 (a) のような変形を行うことができなくなる．

式を実行とは逆向きに移動させる際に移動が阻害される問題は，**図 3** のようにループ本体に合流する直前に補償コード用のブロックを挿入することにより回避できる．このブロックを挿入することにより，既存の

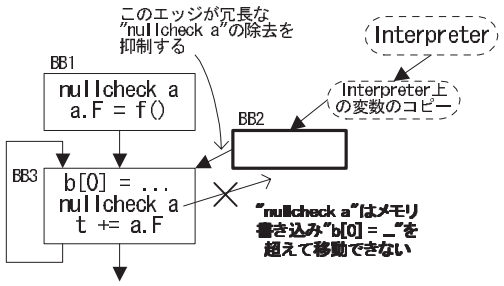

図 4 ヌルチェック除去が抑制される例
Fig. 4 Example where null check elimination is suppressed.

Partial Redundancy Elimination （PRE）[4),6),12)]の手法を使えば，図 3 のように「nullcheck a」や「a.F」をループ外に移動させることができる．

3.2 前進データフロー解析への考慮点

この節では，メソッドの途中遷移が前進データフロー解析（forward dataflow analysis）を使った最適化に悪影響を及ぼす問題を考える．

3.2.1 冗長な例外チェックの除去

図 3 の例は，補償コード用のブロックを挿入することで解決できたが，ブロックの挿入だけでは解決できない場合がある．ヌルチェックの最適化[4)]は実行とは逆向きにコード移動を行い挿入点を求めるフェーズと，前進データフロー解析を使って冗長なヌルチェックを除去するフェーズの 2 つに分かれる．ここで問題となるのは，実行とは逆向きにコード移動を行う場合に移動を止めなければならない条件と，冗長性を実行方向に伝播する場合に伝播できなくなる条件が違うという点である．

図 4 にヌルチェックの除去が抑制される例を示す．Java 言語に対して例外を起こす命令を移動する場合には，多くの命令が移動を阻害する．たとえば，メモリ書き込みなどを越えてヌルチェックを移動することはできない．そのため，この例では補償コード用のブロックへ BB3 の「nullcheck a」を移動することができない．この例で問題がさらに深刻なのは，BB3 の「nullcheck a」がループ内に残ることによって後続するループ不変のメモリロード「a.F」もループ内に残ってしまう点である．

一方，仮にメソッドの途中遷移がなければ，BB1 のヌルチェックで変数 a についてヌルではないことが保障されているために，BB3 の「nullcheck a」や後続するメモリロード「a.F」は除去することができる．しかし，メソッドの途中遷移がある場合には単純に冗長性を実行方向に伝播すると，BB2 から BB3 のパスに「nullcheck a」という式が含まれていないため，

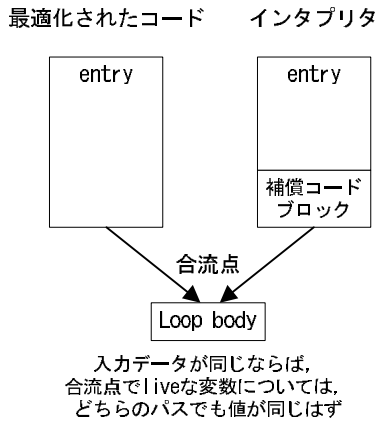

図 5 途中遷移における性質
Fig. 5 Characteristic for transer.

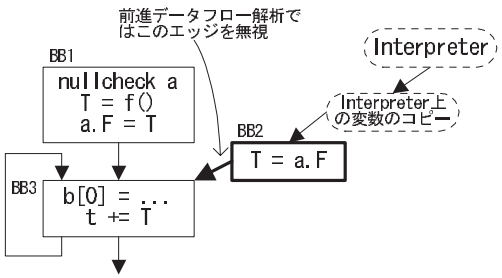

図 6 図 4 に対する最適化結果
Fig. 6 Result of Fig. 4.

BB3 内に a が非ヌルであるという情報が伝播されない．つまりこれは，BB2 の前にある仮定的な変数定義「Interpreter 上の変数のコピー」によって，a の値に関する情報が BB2 に伝播されていないため，最適化が抑制されていることを意味している．

しかしここでの重要な観察は，インタプリタ上でも遷移前に BB1 内の「nullcheck a」という式は実行されているという事実である．そのため，この例ではインタプリタから遷移するパスにおいても，a はヌルではないことが実際には保障されている．

もっと一般的にいうと，**図 5** に示すように，入力データがまったく同じときには，合流点で live な変数についてはインタプリタ・コンパイルされたコードどちらの entry からの実行でも，**同じ値**になるはずであるということである．

よって，例外チェックの最適化では，合流点においてインタプリタからのパスを無視することにより，最適化の抑制を防ぐことができる．図 4 の例では，BB2 から BB3 のエッジを無視して最適化することにより，BB3 の「nullcheck a」を除去することができる．その結果，ループ内のメモリロード「a.F」も PRE を

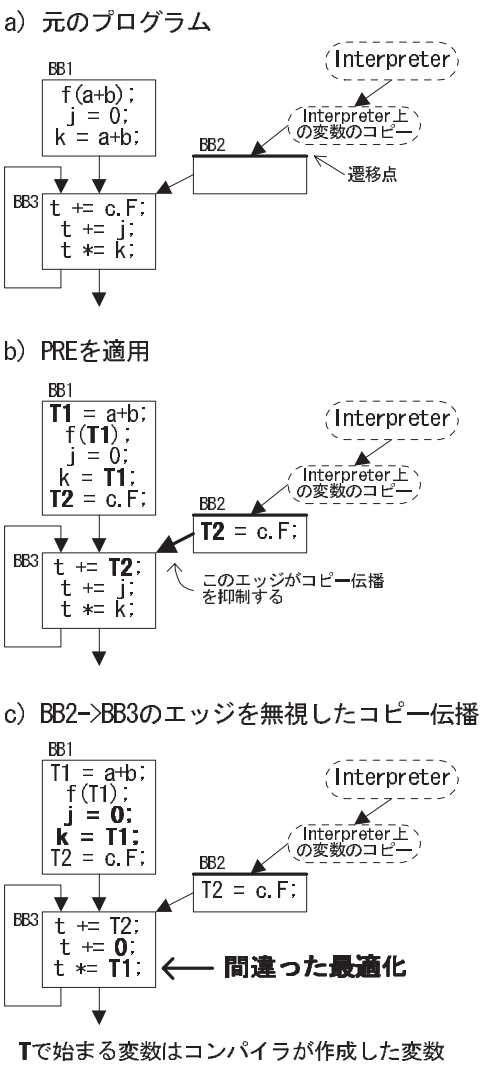

図 7 コピー伝播が抑制される例
Fig. 7 Example where copy propagation is suppressed.

使うことによって，最終的には**図 6** のようにメソッドの途中遷移がない状態と同等に最適化を行うことができる．

3.2.2 コピー伝播

前項で合流点においてインタプリタからのパスを無視することによって，最適化の抑制を防ぐことができる例を説明した．この項では，単純にインタプリタからのパスを無視すると間違った最適化を行ってしまう例およびその解決方法について述べる．

図 7 にコピー伝播が抑制される例を示す．図 7 (b) で，仮にメソッドの途中遷移がなければ，(c) のように BB1 に含まれる 2 つのコピー命令は BB3 内に伝播される．しかし，メソッドの途中遷移がある場合には単純にコピー伝播を行うと，BB2 から BB3 のパス

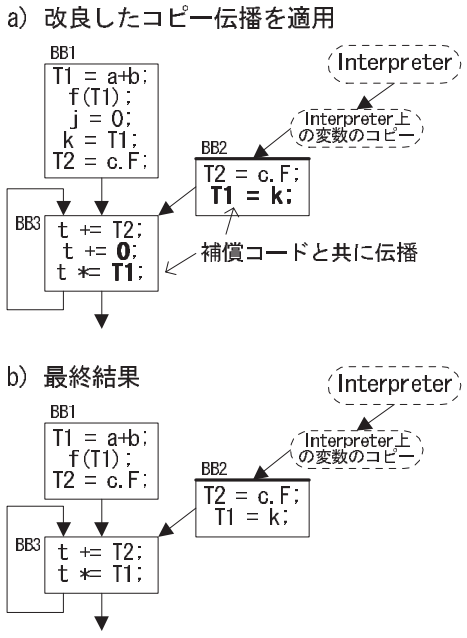

図 8 図 7 の解決方法
Fig. 8 A solution of Fig. 7.

で BB1 に含まれる 2 つのコピー命令が含まれていないため，BB3 内に伝播されない．

この問題を前項のように，BB2 から BB3 のエッジを無視して最適化すると，“k = T1” のコピーが伝播されることによって，図 7 (c) のように誤った最適化結果を生み出す．これは，インタプリタからのパスでは T1 が初期化されていないためである．

ここでの重要な観察は，図 5 で議論したように，入力データがまったく同じときには，変数 j，k の値はインタプリタ・コンパイルされたコードどちらの entry からの実行でも同じ値（j=0，k=a+b）になるという点である．問題は，T1 という変数はインタプリタ上では存在しないという点である．そのため，「j = 0」は伝播してもかまわないが，「k = T1」は単純には伝播できない．

我々は，「k = T1」が伝播しない問題を，**図 8** (a) に示すように，BB2 に補償コードを生成することにより BB3 に伝播することを可能にしている．別のいい方をすると，変数 k にはインタプリタからきたパスでも，正しい値が入っていることを利用して「T1 = k」というコピー命令を補償コードとして生成することにより伝播させても正しく動作させることができる．具体的には，**図 9** のアルゴリズムで示すように，まずインタプリタからのパスを無視してコピー伝播を行って，次に合流点の先頭で求まる伝播したコピーの集合を使って必要な補償コードを生成する．

```
＜インタプリタからのパスを無視してコピー伝播を行う．＞
MERGE_IN = 合流点の先頭で求まる伝播したコピーの集合
for (each C ∈ MERGE_IN){
   if (Cの出力 ∈ インタプリタ上に存在する変数 &&
       Cの入力 ∈ インタプリタ上に存在しない変数){
      ＜合流点に対応する補償コード用ブロックの最後に
       Cの入力と出力を反転させたコピー命令を生成＞
   }
}
```

図 9 コピー伝播の補償コードの生成アルゴリズム
Fig. 9 Algorithm for generating compensation code of copy propagation.

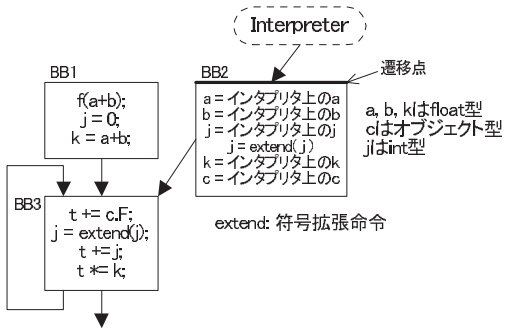

図 10 Interpreter 上の変数のコピーの明示的な生成
Fig. 10 Generating copy instructions explicitly from interpreter's variable image.

一般的には，変数の live range を実行方向に伸ばす最適化に対しては，コンパイラが作成した変数の live range が最適化後に合流点をまたぐ場合に，その変数に対して正しい値を代入する補償コードを生成することにより，最適化を抑制する問題を解決することができる．それ以外の前進データフロー解析を使った最適化は，前節で述べたように単純にインタプリタからのパスを無視して最適化すれば，最適化を抑制する問題を解決することができる．

3.2.3 そのほかの考慮点

この項では，そのほか考慮しなければいけない問題について述べる．実装によっては，今までの例で暗黙的な処理として扱ってきた仮定的な変数定義「Interpreter 上の変数のコピー」を，最適化処理に対して明示的な命令として表したほうが都合がよい場合がある．

たとえば，「Interpreter 上の変数のコピー」は変数の型（float，int，object，...）によって生成する命令が異なる．**図 10** は 64-bit アーキテクチャに対して変数のコピーを明示的に生成した例を示す．この例で，変数 a，b，k は float 型なので float のコピー（ロードも含む）命令を，変数 c はオブジェクト型なので整数型のコピー命令を，変数 j は int 型なので整数型のコピー命令と符号拡張命令を BB2 に生成する必要が

ある．

変数の型を見つけるためにはいくつか実装方法が考えられる．一例をあげると初期状態では変数の型は未定義で仮のコピー命令を BB2 に生成し，その後 definition-use chain を作成し使用される変数の型を元に BB2 内のコピー命令の変更を行う方法などが考えられる．このような実装形態の場合には，補償コードブロック内にコピー命令を明示的な命令として生成したほうが都合がよい．我々の実装では，IA-64 向けの JIT コンパイラでは，このような変数のコピーを明示的な命令として表し，そのほかのアーキテクチャ向けの JIT コンパイラでは明示的な命令として表していない．

さらには，明示的に命令を生成したおかげで「Interpreter 上の変数のコピー」を行う命令を減らせる可能性がある．たとえば，補償コードブロック内の符号拡張命令（extend）は場合によっては除去できる場合があり☆，合流点以降で使われない変数のコピー命令も除去することができるようになる．

しかし，補償コードブロック内に変数の定義が明示的に見えることで，最適化が抑制される場合がある．たとえば，図 10 に対して 3.2.2 項で述べたコピー伝播を使わずに use-definition chain を使って定数伝播を行う例を考えてみる．この場合，BB3 内の j の定義は BB1 と BB2 内に 2 つあり，BB1 内の定義値は 0 と分かるが BB2 内の定義値は変数のため特定できない．しかし，図 5 での議論を踏まえると，BB2 内の j の定義値も 0 のはずである．

この問題を解決するためには，「Interpreter 上の変数のコピー」を行う命令を，「変数定義を無視できる命令」として扱えるようにすればよい．図 10 を例にとると BB2 内の a の変数定義から c の変数定義までの命令すべてに「変数定義を無視できる命令」というフラグを立て，最適化はこのフラグを見て無視することが可能かを判断すればよい．

このような処理を行うと，先ほどの use-definition chain を使った定数伝播の例では，j の BB1 内の定義値は 0 と分かり，BB2 内の定義値は無視できると分かる．その結果，BB3 の j の値は 0 と判断できるため 0 と置き換えることができる．念のためにいっておくと，図 8 (b) の BB2 内の命令のように，最適化によって生成された補償コードの変数定義は無視してはいけない．そのために「変数定義を無視できる命令」というフラグはブロックに設定するのではなく，命令ごとに設定する必要がある．

☆ 図 10 の例では実際には BB3 内（ループ内）の符号拡張命令が除去され，BB2 内の符号拡張命令については除去されない[5]．

4. 実験結果

我々の手法の評価を行うために CaffeineMark 3.0[8]，jBYTEmark，SPECjvm98[9] の 3 つのベンチマークと一太郎 Ark[3]，Java 2 SDK 1.2.2 に含まれる Java2Demo の 2 つのアプリケーションを使って測定を行った．すべての測定結果は，IA-64 向けの IBM Java JIT Compiler に最適化を実装し，IBM IntelliStation Z Pro model 689412X（Itanium 800 MHz 2 個，2 GB RAM）上で測定した．測定方法はベンチマークやアプリケーションを複数回独立に実行させ，それらのスコアの平均を最終的な結果とした．本稿で述べた手法の比較を行うために，次の版を作成し測定を行った．それぞれの版の最適化レベルは，再コンパイルを行う版の低い最適化レベルは文献 11) の Quick に対応し，それ以外は文献 11) の Full に対応する．

JIT Only インタプリタでの実行を止めて，すべての実行を JIT 上で行う版．つまり，メソッドの途中遷移は起こらず，合流点も存在しない．なお，この版では 1 章で述べたように，クラスの初期化チェックなどによって最適化が抑制されている可能性がある．

Mixed インタプリタから JIT へメソッドの途中で遷移を行うが，本稿で述べた手法は行わない版．遷移点は，最初にインタプリタから遷移をしてきた 1 カ所のみ作成する．仮に，複数のスレッドからほぼ同時に遷移してきた場合には，最初に処理を行ったスレッド以外はインタプリタに実行を戻し，次にそのメソッドが呼ばれたときにコンパイルされたコードに移る．

Mixed+New Dataflow Mixed に加えて，本稿で述べた手法を行う版．

Recomp. Mixed に加えて，複数の最適化レベルで再コンパイルさせる版．文献 11) で提案されている方法．なお，IA-64 向けの再コンパイルによるメソッドの特殊化はまだ実装されていない．

Recomp.+New Dataflow Recomp. に加えて，本稿で述べた手法を行う版．

4.1 CaffeineMark の測定結果

CaffeineMark には通常版とグラフィックベンチマークを除いた Embedded 版があるが，今回の測定では JIT による性能比較のため Embedded 版を用いて測定した．**図 11** に JIT Only を 100%としたときの，CaffeineMark に対するそれぞれの版の相対的なパフォー

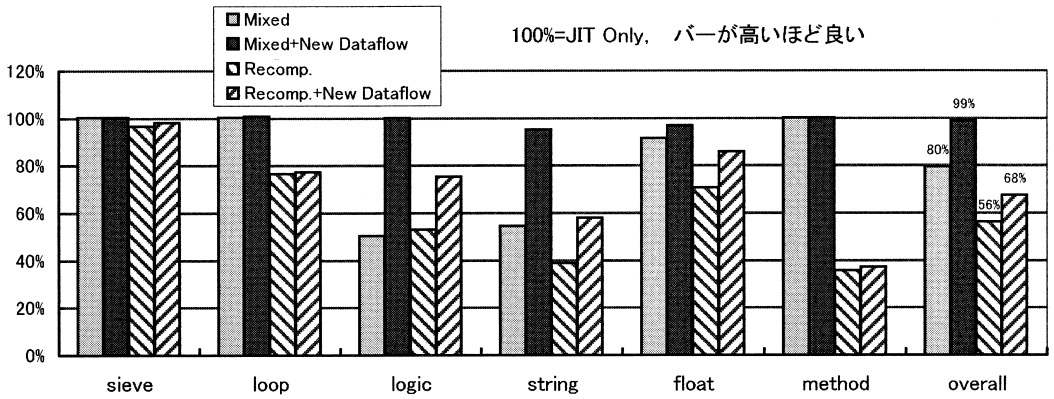

図 11 CaffeineMark に対する相対的なパフォーマンス（JIT Only=100%）
Fig. 11 Relative performance of CaffeineMark (JIT Only=100%).

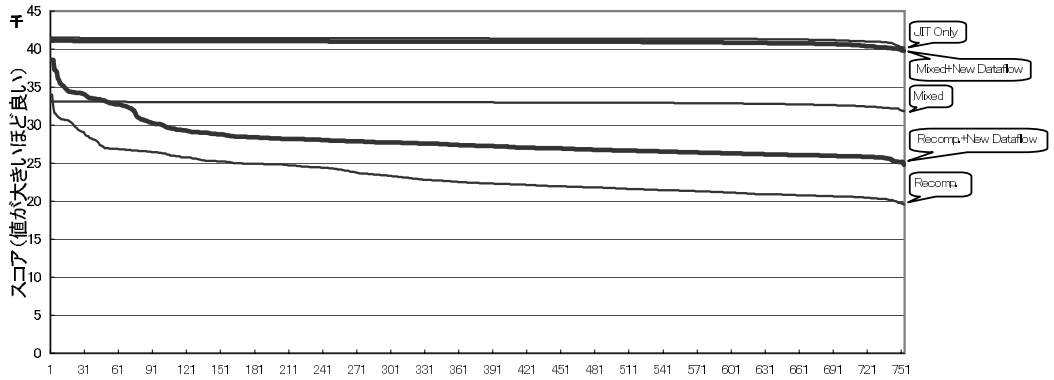

図 12 CaffeineMark の Overall のスコアを良い順にソートした結果（太線は本稿の手法を使っている版）
Fig. 12 Result in which score of “Overall” of CaffeinMark is sorted in good order (thick lines denote versions using “New Dataflow”).

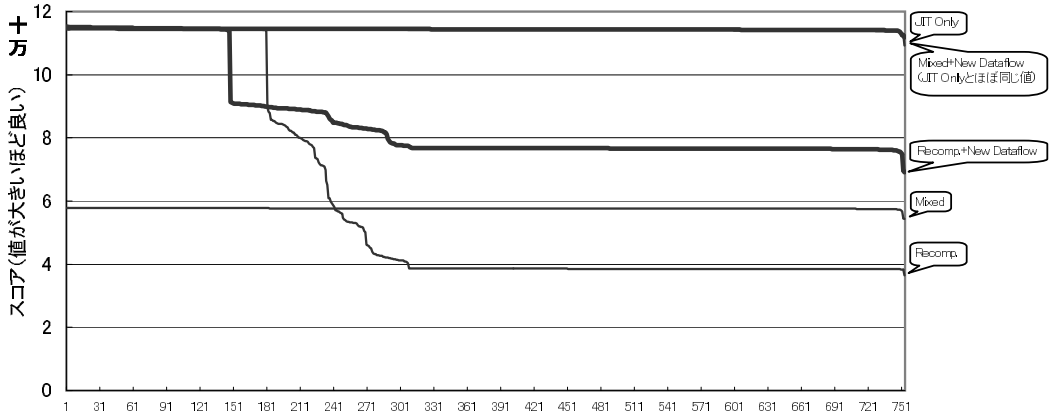

図 13 CaffeineMark の Logic のスコアを良い順にソートした結果（太線は本稿の手法を使っている版）
Fig. 13 Result in which score of “Logic” of CaffeinMark is sorted in good order (thick lines denote versions using “New Dataflow”).

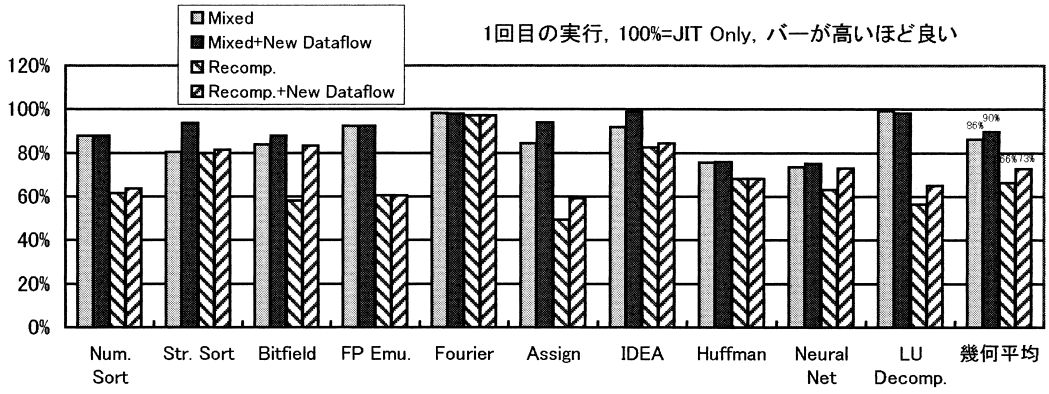

図 14 jBYTEmark の 1 回目の実行時間に対する相対的なパフォーマンス（JIT Only=100%）
Fig. 14 Relative performance of the first execution for jBYTEmark (JIT Only=100%).

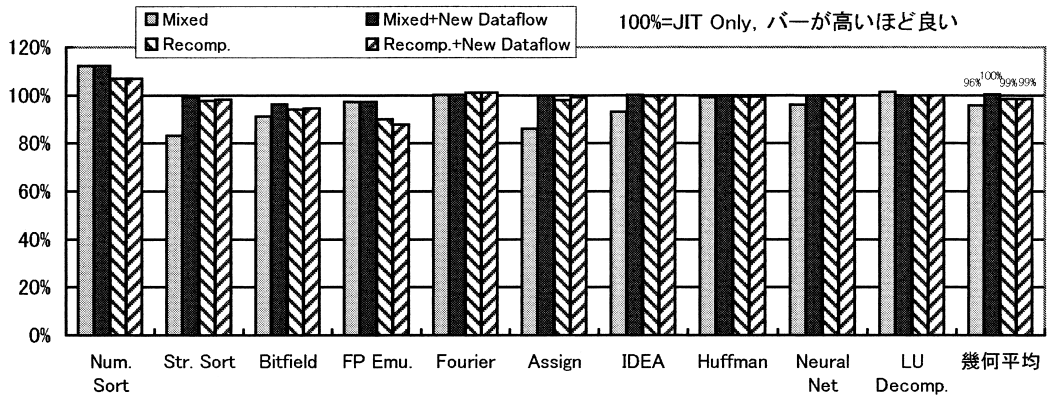

図 15 jBYTEmark の最終スコアに対する相対的なパフォーマンス（JIT Only=100%）
Fig. 15 Relative performance of the final score for jBYTEmark (JIT Only=100%).

マンスを示す．再コンパイルを行う方法は再コンパイルを行わないものよりもパフォーマンスが悪いという結果になった．

この原因は，基本的には最初に低い最適化レベルでメソッドをコンパイルし，再コンパイルされずにそのコンパイルされたコードが測定時間の大部分の間実行されたためと考えられる．ただし，特に CaffeineMark では，再コンパイルを行う 2 つの版についてスコアに大きなばらつきが見られた．

図 12 に CaffeineMark を 753 回独立に実行させ，Overall のスコアについて結果が良い順にソートして値をプロットしたグラフを示す．先ほど述べたように，図 11 の比較はこの 753 回の平均を元に計算したものである．独立な実行を繰り返しているため，理想的にはスコアはすべての実行で同じ値になるはずである．図 12 のグラフを見ると，再コンパイルをしない 3 つの版については，それほどばらつきが見られないが，再コンパイルを行う 2 つの版はスコアに大きなばらつきがあることが分かる．

この原因を調べてみたところ，タイミングによって再コンパイルが行われる場合と行われない場合があることが分かった．これは，タイマを使ったプロファイル収集を行っているためと考えられる．典型的な例として，**図 13** に CaffeineMark の中の Logic ベンチマークについて，図 12 と同様の方法でスコアをプロットしたグラフを示す．このグラフを見ると，再コンパイルを行う 2 つの版（Recomp. と Recomp.+New Dataflow）は，スコアが主に 2 つの値に分かれていることが分かる．この 2 つの値は，それぞれ再コンパイルが行われない場合と行われた場合のスコアを示している．また，本稿で述べた手法によって，再コンパイルが行われなかった場合（低いスコア）について改善されていることも分かる．この結果から，複数回呼ばれる頻度が高いメソッドであっても，タイミングによってはアップグレードされない場合があることが分かる．

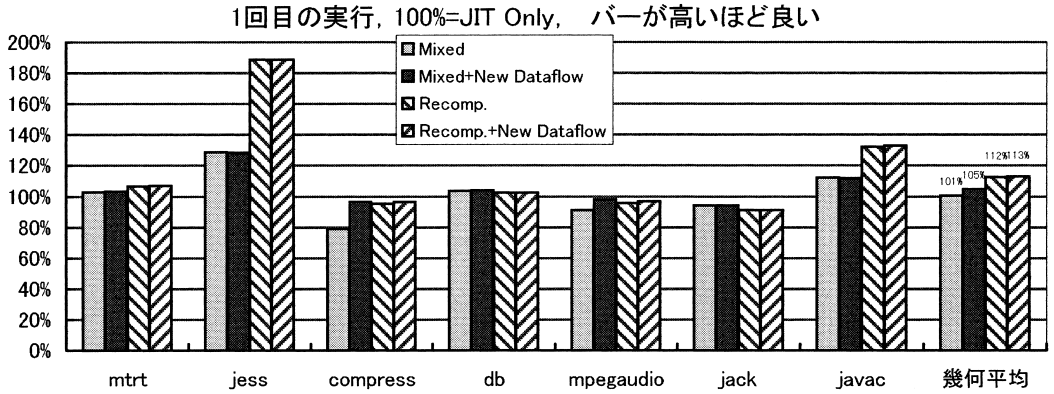

図 16 SPECjvm98 の 1 回目の実行時間に対する相対的なパフォーマンス（JIT Only=100%）
Fig. 16 Relative performance of the first execution for SPECjvm98 (JIT Only=100%).

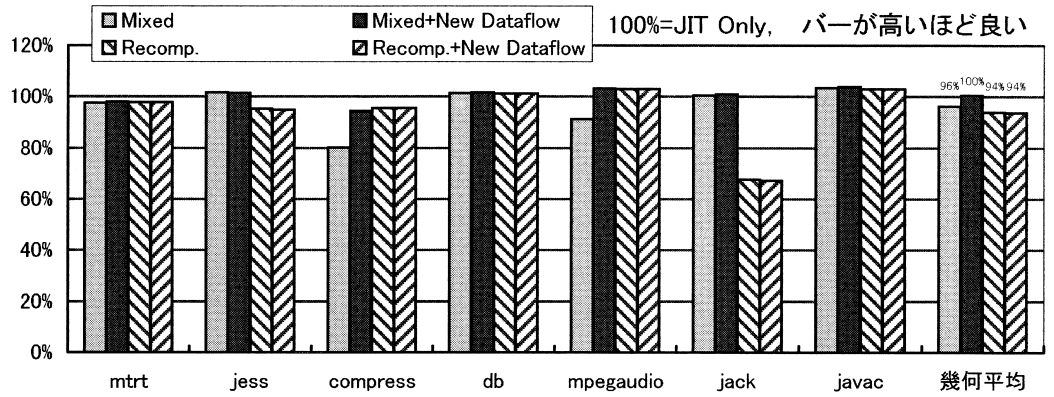

図 17 SPECjvm98 のベストスコアに対する相対的なパフォーマンス（JIT Only=100%）
Fig. 17 Relative performance of the best score for SPECjvm98 (JIT Only=100%).

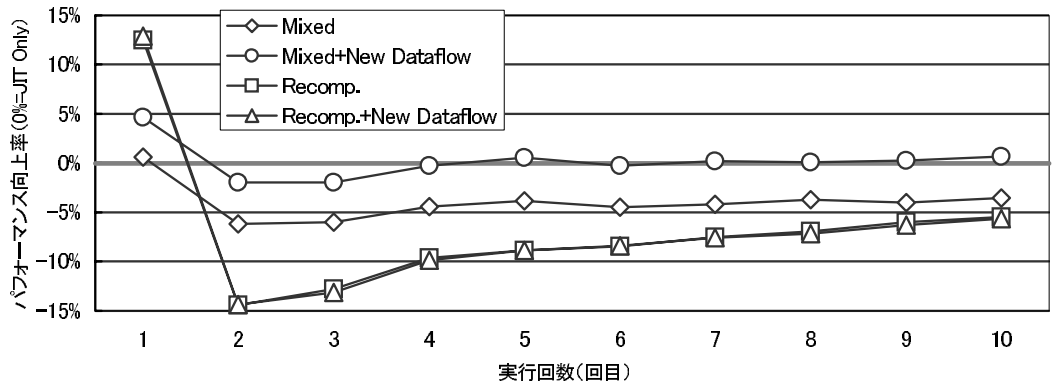

図 18 SPECjvm98 の各ベンチマークを 10 回ずつ実行したときの 7 つのベンチマークの幾何平均の変化の様子（JIT Only=0%）
Fig. 18 Performance changes of the geometric mean of seven benchmarks in SPECjvm98 when each benchmark runs 10 times (JIT Only=0%).

図 11 の overall を比較すると，本稿で述べた手法は，再コンパイルしない場合・する場合ともに 20%程度パフォーマンスを改善することが分かる．また，“Mixed+New Dataflow” だけが JIT Only とほぼ同等のパフォーマンスを得られた．これらの結果は，再コンパイルの手法が万能ではないことを示している．

4.2 jBYTEmark の測定結果

jBYTEmark は，1 回の独立な実行につき，測定用のプログラムを何回か連続で実行を行ってそれらの結果からスコアを計算している．2 章で述べたように，高い最適化レベルで再コンパイルされた後のコードには本稿の手法は影響を与えない．しかし，数値計算のようなプログラムは，実際の用途では 1 つのメソッドを 1 回だけ実行して，ループを長時間実行するようなものが多い．一方，1 回しか実行されないようなメソッドについては，再コンパイルの手法では低い最適化レベルの状態で実行され続けてしまうという問題点がある．jBYTEmark は数値計算を行うプログラムが多いことから，今回我々は 1 回しか実行されないような状況を想定して，1 回目の実行時間についても計測を行った．また，各ベンチマークのサイズはそれぞれ値を明示的に指定した．

図 14 に，jBYTEmark の 1 回目の実行時間に対するそれぞれの版の相対的なパフォーマンスを示す．幾何平均を見ると，CaffeineMark のときと同様の理由で，再コンパイルを行う方法は再コンパイルを行わないものよりもパフォーマンスが悪くなった．また，本稿で述べた手法は，再コンパイルしない場合・する場合ともにパフォーマンスを改善した．

図 15 に，jBYTEmark の最終的なスコアに対するそれぞれの版の相対的なパフォーマンスを示す．幾何平均を見ると，本稿で述べた手法は，再コンパイルしない場合のみパフォーマンスを改善させた．再コンパイルする版では，実行頻度が高いメソッドの多くが再コンパイルされたため，本稿で述べた手法によってパフォーマンスが改善されなかったと考えられる．最終的なスコアは，Mixed 以外の 3 つの版について JIT Only とほぼ同等の結果を得られた．このことは，jBYTEmark においては再コンパイルの手法も効果的だったことを示している．

4.3 SPECjvm98 の測定結果

SPECjvm98 は，コマンドラインから 7 つのベンチマークを個別に実行させた．また，1 回の独立な実行につき，問題の大きさを 100 にして連続で 10 回ずつ実行させて測定を行った☆．

図 16 に，SPECjvm98 の 1 回目の実行時間に対するそれぞれの版の相対的なパフォーマンスを示す．幾何平均を見ると，前の 2 つのベンチマークと傾向が異なり，再コンパイルしない場合・する場合ともに，JIT だけで実行する場合と比べてパフォーマンスが向上している．さらに再コンパイルを行う方法は再コンパイルを行わないものよりもパフォーマンスの向上率が高い．なかでも，jess と javac については大きく改善している．再コンパイルを行う方法は，多くのメソッドを低い最適化レベルでコンパイルし，実効頻度が高い一部のメソッドだけを時間がかかる高い最適化レベルでコンパイルする．そのため，ある程度大きなプログラムでは，再コンパイルを行う方法は他の版と比べて，全体のコンパイル時間が短くなることが多い．このことから推測すると，この 2 つのベンチマークは他のベンチマークと比べて JIT Only のコンパイル時間が全体の実行時間に大きく影響していると考えられる．

図 17 に，SPECjvm98 の各ベンチマークを連続で 10 回実行した中のベストタイムに対するそれぞれの版の相対的なパフォーマンスを示す．幾何平均を見ると，1 回目の実行時間での傾向とは異なり，再コンパイルを行う方法は再コンパイルを行わないものよりも若干パフォーマンスが悪くなっている．なかでも，jack ベンチマークは再コンパイルを行う方法によって大きくパフォーマンスを悪くしている．この原因を調べた結果，再コンパイルの決定を行うために必要なプロファイルの取得のオーバヘッドが，パフォーマンスの悪化に大きく影響していることが分かった．

図 18 に，各ベンチマークを連続で 10 回ずつ実行したときの 7 つのベンチマークの幾何平均の変化の様子を示す．再コンパイルする版は，初回のパフォーマンスを大きく改善させているが，2 回目の実行は JIT のみの版と比べて相対的に大きくパフォーマンスが悪くなる．さらに，定常状態にはなかなかならず，10 回目の実行を終えてもまだ収束する気配がないことが分かる．また，本稿で述べた我々の手法は再コンパイルする版ではほとんど改善が見られなかった．これは，実行頻度が高いメソッドの多くが再コンパイルされたためと思われる．

次に，再コンパイルしない版だが，本稿で述べた我々の手法は 1 回目から 10 回目の実行を通じて，つねに 5%程度パフォーマンスを改善することが分かる．ま

☆ java -ms96m -mx96m -noclassgc SpecApplication -m10 -M10 -g -a -s100 各ベンチマーク

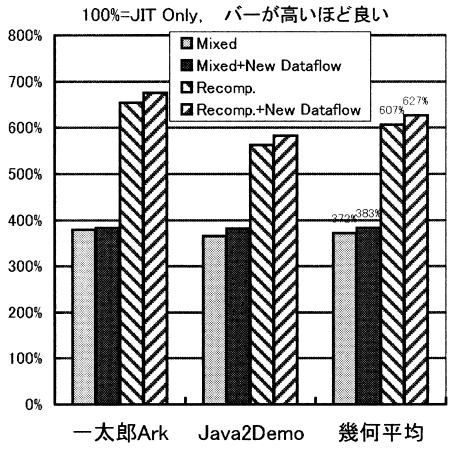

図 19 2つのアプリケーションの起動時間に対する相対的なパフォーマンス（JIT Only=100%）
Fig. 19 Relative performance for starting up two applications (JIT Only=100%).

た，我々の実装では再コンパイルしない版は，このベンチマークでは 4 回目の実行でほぼ定常状態になることが分かった．

4.4 アプリケーションの起動時間

ベンチマークの測定結果を見ると，JIT Only の版はかなり良い結果を出していることがわかる．しかし，JIT Only はすべてのメソッドを高い最適化レベルでコンパイルするため，コンパイル時間がかかり，通常のアプリケーションでは遅くなるケースが多い．この節では，アプリケーションの起動時間を測定することにより，JIT Only が他の版に比べてどの程度遅いかを検証する．

一太郎 Ark と Java2Demo の測定方法はともに，コマンドを実行してから初期画面が表示され，入力待ちとなるまでの起動時間を測定した．**図 19** に，2 つのアプリケーションの起動時間に対するそれぞれの版の相対的なパフォーマンスを示す．

再コンパイルしない版・する版ともに，JIT だけを使った方法と比べて大きくパフォーマンスが向上した．特に再コンパイルする版では，起動時間が約 6 分の 1 にまで短縮している．このことは，これらのアプリケーションの起動時間が，多くのコンパイル時間で費やされていたことを意味している．

本稿で述べた最適化手法は再コンパイルしない版・する版ともに 3%程度パフォーマンスを向上させた．ベンチマークの測定結果を見れば分かるように，再コンパイルが頻繁に起きているとすれば向上の度合いが減ると考えられる．そのような現象は起きていないことから，アプリケーションの起動中にはあまり再コンパイルが行われていないと考えられる．

4.5 実験結果の考察

3 つのベンチマークと 2 つのアプリケーションを測定した結果，再コンパイルしない版・する版について次のような考察を得られた．

再コンパイルしない版：

- **本稿で述べた最適化の手法は，多くのプログラムについてパフォーマンスを改善するため，必須の処理と思われる．**
- 数値計算のように，1 つのメソッドを長く実行するようなプログラムに対して有効．

再コンパイルする版：

- **本稿で述べた最適化の手法は，再コンパイルしない版に対して適用した場合と比べると，幾分限定的ではあるがやはり有効．**
- コンパイル時間を大きく削減するため，クライアント向けのプログラムに対して非常に有効．
- プログラムによっては，プロファイルの取得のオーバヘッドによるパフォーマンスの悪化が見られる．
- パフォーマンスが定常状態となるまでには，長時間の実行が必要となる．

本稿で述べた最適化の手法の効果は，再コンパイルする版では幾分限定的であった．しかし，ベンチマーク上での最大性能を比較すると（図 11，図 15，図 17），全体的に再コンパイルしない版のほうがする版よりも勝っており，再コンパイルする版が必ずしも優れているとはいえない．そのため，再コンパイルしない版も，なお使う必要性はあると考えられる．たとえば，それぞれの版の長所を生かし，再コンパイルする版は最大パフォーマンスよりもコンパイル時間の削減を重視するクライアントプログラム向け，再コンパイルしない版はそれ以外のプログラム向けに使用するといった利用法が良いと考えられる．

5. おわりに

本稿ではインタプリタと動的コンパイラを組み合わせたときに生じる問題点とその解決法について述べた．式を実行とは逆向きに移動させる際に移動が阻害される問題は，ループ本体に合流する直前に補償コード用のブロックを挿入することにより回避できる．また，変数の live range を実行方向に伸ばす最適化に対しては，コンパイラが作成した変数の live range が最適化後に合流点をまたぐ場合に，その変数に対して正しい値を代入する補償コードを生成することにより，最適化を抑制する問題を解決することができる．それ以外の前進データフロー解析を使った最適化は，単純にイ

ンタプリタからのパスを無視して最適化すれば，最適化を抑制する問題を解決することができる．本稿で述べた遷移による最適化が抑制される問題は，インタプリタと動的コンパイラの組合せだけの問題ではなく，実行環境が異なる2つのコード間の遷移を行う際には一般的に起こる問題である．我々は，本稿で述べた手法の重要性がますます高まることを期待している．

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IBM東京基礎研究所JIT compilerグループの皆様に深く感謝します．また，有益なコメントをいただきました査読者の方に心より感謝申し上げます．

参 考 文 献

1) Hölzle, U., Chambers, C. and Ungar, D.: Debugging optimized code with dynamic deoptimization, *PLDI'92*, pp.32–43 (1992).
2) Ishizaki, K., Kawahito, M., Yasue, T., Takeuchi, M., Ogasawara, T., Suganuma, T., Onodera, T., Komatsu, H. and Nakatani, T.: Optimizations to reduce overheads of the Java language in a Just-in-Time Java compiler, *ACM SIGPLAN Java Grande Conference* (1999).
3) Justsystem Corp.: Ark Site. http://www.justsystem.co.jp/ark/
4) Kawahito, M., Komatsu, H. and Nakatani, T.: Effective Null Pointer Check Elimination Utilizing Hardware Trap, *ASPLOS'00*, Cambridge, MA, pp.139–149 (2000).
5) Kawahito, M., Komatsu, H. and Nakatani, T.: Effective Sign Extension Elimination, *PLDI'02*, Berlin, Germany, pp.187–198 (2002).
6) Knoop, J., Rüthing, O. and Steffen, B.: Optimal code motion: Theory and practice, *TOPLAS'94*, Vol.16, No.4, pp.1117–1155 (1994).
7) Paleczny, M., Vick, C. and Click, C.: The Java HotSpot Server Compiler, *Java Virtual Machine Research and Technolozy Symposium*, pp.1–12 (2001).
8) Pendragon Software Corp.: CaffeineMark 3.0. http://www-sor.inria.fr/ java/tools/ cmkit/embed.zip
9) Standard Performance Evaluation Corp.: SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/
10) 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 Journal*, Vol.39, No.1, pp.175–193 (2000).
11) Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H. and Nakatani, T.: A Dynamic Optimization Framework for a Java Just-In-Time Compiler, *OOPSLA'01* (2001).
12) 川人基弘，小松秀昭，中谷登志男：Java言語に対する投機的なメモリアクセスの最適化手法，情報処理学会論文誌，Vol.44, No.3, pp.883–896 (2003).

(平成15年5月20日受付)
(平成15年8月15日採録)

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）．