Java 向け動的コンパイラによる冗長なインスタンス変数参照の削除

千 葉 雄 司[†1]

本論文では，Java 向け動的コンパイラにおいて，冗長なインスタンス変数参照の削除を実施する際に，固有に問題となる点を指摘し，その解決策を検討する．本論文で指摘する問題点とは，動的ロードとネイティブメソッド呼び出しである．動的ロードが問題となるのは，クラスの動的な追加によって，結果として，最適化を適用可能な箇所が変化し，追加より前に適用した最適化が不適切になりうるからである．ネイティブメソッド呼び出しが問題となるのは，Java のプログラム（バイトコード）を解析しただけでは分からないところからインスタンス変数参照を書き換えうることから，インスタンス変数参照の書き換えがいつおきるか，どのインスタンス変数参照が冗長か判定することが困難になり，最適化を適用しにくくなるからである．これらの問題を回避する手段として，最適化の適用先を保守的に選択する方法もあるが，それでは適用先が少なくなる．本論文の提案技法では，これらの問題の解決するために，実行時に動的ロードやネイティブメソッドの利用状況を監視し，監視の結果に基づいて最適化を適用する．提案技法を使って冗長なインスタンス変数参照の削除を実装し，その効果を，SPECjvm98 および SPECjbb2005 を使って評価した結果，実行速度を相乗平均で 0.6%高速化できることが分かった．

Eliminating Redundant Instance Variable References by a Dynamic Compiler for Java

Yuji Chiba[†1]

This paper shows an implementation of redundant instance variable reference elimination in a dynamic compiler for Java. The implementation solves two problems specific to Java: dynamic loading and native methods. Dynamic loading is problematic because it dynamically adds a class and this invalidates optimization applied before the addition. Native methods are problematic because analysis of Java bytecodes cannot tell instance variables they modify. We can avoid these problems by applying the optimization conservatively, but this limits optimization opportunities. This paper solves these problems by watching the use of dynamic loading and native methods at runtime to eliminate redundant instance variable references using the result. Estimation using the SPECjvm98 benchmark suite and the SPECjbb2005 benchmark showed that our optimization improves the performance by 0.6% (geometric mean).

1. はじめに

冗長なインスタンス変数参照の削除とは，インスタンス変数参照のうち，冗長なものを除去する最適化技術である．本論文では，冗長なインスタンス変数参照の削除を，Java™ ⋆1で記述したプログラムに適用するにあたって，固有に問題となる点に，動的ロードやネイティブメソッドがあることを示し，その解決策を定量的に評価しながら考察する．本論文の構成を次に示す．まず，2 章で，冗長なインスタンス変数参照の削除の基本的な実装方法を示し，過去の実装を概観する．3 章では，過去の実装の 1 つである Ghemawat らの実装[3)] について述べ，その問題点を 4 章で示す．5 章では，4 章で示した問題を改善した実装方針を示し，示した方針に従って実装した冗長なインスタンス変数参照の削除の効果を 6 章で評価する．7 章は結論である．

2. 関連研究

インスタンス変数参照を削除する方法には，大きく分けて次に示す 2 つの方法がある．

- インスタンス変数への代入と，それに後続するインスタンス変数への参照を求め，後続する参照の結果として得られる値を，先行する代入で書き込

†1 株式会社日立製作所システム開発研究所
Systems Development Laboratory, Hitachi, Ltd.

⋆1 Java および HotSpot は米国およびその他の国における米国 Sun Microsystems, Inc. の商標です．

む値で代用することで，後続する参照を削除する．

- インスタンス変数の参照と，それに後続するインスタンス変数の参照を求め，後続側で参照を行う代わりに，先行側の参照の結果を再利用することで，後続する参照を削除する．

これらの方法で後続側のインスタンス変数参照を削除できるのは次の全条件が満たされる場合に限られる．

条件 1 先行側の代入先もしくは参照先と，後続側の参照先が，同じインスタンスの同じインスタンス変数である．

条件 2 参照先のインスタンス変数が volatile 宣言されていない．

条件 3 先行側から後続側に至るまでのパスに，参照先のインスタンス変数を書き換える処理がない．

条件 4 参照先のインスタンス変数が，非同期な書き換えの対象にならない．もしくは，先行側から後続側に至るまでのパスに，次の処理がない．
- モニタの確保
- volatile 変数の参照

ここで，条件 4 は，Java の言語仕様[4] における，スレッドの挙動に関する規定を守るためのものである．Java の言語仕様は，非同期な書き換えの対象となりうるインスタンス変数について，インスタンス変数の値を，volatile 変数の参照あるいはモニタの確保を超えて再利用することを禁じている．したがって，もしインスタンス変数が非同期な書き換えの対象になりうるなら，先行する代入もしくは参照と，後続する参照の間に，volatile 変数の参照もしくはモニタの確保が存在するとき，後続側の参照を冗長と見なして削除することはできなくなる．

冗長なインスタンス変数参照の削除の手順と効果について具体的に述べるために，**図 1** (a) の 8〜11 行目にある，Java で記述したメソッド woo() を最適化する場合について考える．メソッド woo() の中には，9 行目と 10 行目にインスタンス変数参照 this.self がある．ここで後続側のインスタンス変数参照を冗長と見なして除去できるか判定するために，これらのインスタンス変数参照が条件 1〜4 を満たすか考察する．

まず，条件 1 についてだが，これは明らかに満たされる．なぜなら，局所変数 this の値が書き換えられることはなく，また，それぞれが同じインスタンス変数 self を参照するからである．次に，条件 2 についてだが，これも明らかに満たされる．なぜなら，2 行目にあるインスタンス変数 self の宣言から，インスタンス変数 self が volatile 宣言されていないと分かるからである．続いて，条件 3，4 が満たされているか

```
 1:  abstract class Brick{
 2:   private Brick self;
 3:   Brick(){
 4:    this.self = this;
 5:   }
 6:   abstract void boo();
 7:   abstract void foo();
 8:   public void woo(){
 9:    this.self.boo();
10:    this.self.foo();
11:   }
12:  }
```

(a) 最適化対象

```
1:  Brick tos = this->self;
2:  if (tos == NULL)
3:   throw new NullPointerException();
4:  tos->class.dispatch_table[BOO_INDEX]();
5:  tos = this->self;
6:  if (tos == NULL)
7:   throw new NullPointerException();
8:  tos->class.dispatch_table[FOO_INDEX]();
```

(b) 最適化前

```
1:  Brick tos = this->self;
2:  if (tos == NULL)
3:   throw new NullPointerException();
4:  tos->class.dispatch_table[BOO_INDEX]();
5:  tos->class.dispatch_table[FOO_INDEX]();
```

(c) 最適化後

図 1 冗長なインスタンス変数参照の削除
Fig. 1 Redundant instance variable reference elimination.

否か検証する．この検証を実現する方法は，これまでにいくつか提案されているが[2),3),7),8),10),11)]，最も単純な方法は，先行側のインスタンス変数参照から，後続側に至るまでに通過しうるすべてのパスを，メソッド間解析によって求め，求めたパスに，参照先のインスタンス変数の書き換えといった処理がないか調査する方法である．しかしながら，この方法では，調査に大きな手間がかりうる．たとえば 9 行目と 10 行目のインスタンス変数参照 this.self の間には，メソッド呼び出し foo() があるが，メソッド foo() の実行中に通過しうるすべてのパスを求めるために，メソッド foo() を基点として，その呼び出し先を再帰的にすべてたどるには，ときとして大きな手間がかかる．

3. Ghemawat らによる実装

調査にかかる手間を抑えつつ，条件 3，4 が満たされていることを検証する方法に，Ghemawat らが提案した手法がある．Ghemawat らの手法では，インスタンス変数のうち，参照の結果がつねに一定になるものを求め，求めたインスタンス変数への参照は，つねに条件 3，4 を満たすと判断する．ここで，参照の結

果がつねに一定になるインスタンス変数とは，次の条件を満たすものを指す．

条件 a　コンストラクト時に 1 度だけ初期化され，その後，書き換えられない．

条件 b　初期化より前に参照されない．

Ghemawat らの手法では，これらの条件を満たすインスタンス変数の探索にかかるコストを抑えるために，インスタンス変数の可視範囲を利用している．たとえば図 1 (a) の 2 行目で宣言しているインスタンス変数 `self` について考えると，この変数の可視範囲は `private` なので，原則として，インスタンス変数 `self` を書き換える処理は，宣言元のクラス `Brick` 内のみにある．そこでクラス `Brick` 内を探索すると，インスタンス変数 `self` を書き換える処理が，3〜5 行目にあるコンストラクタ `Brick()` 内のただ 1 度だけ実行される箇所にのみあることが分かり，したがって，インスタンス変数 `self` が，条件 a を満たすといえる．さらに，条件 b を満たすか調査するために，コンストラクタ `Brick()` の内部を走査し，4 行目にあるインスタンス変数 `self` の初期化より前に，インスタンス変数 `self` の参照につながる次の処理がないか調査する．

- インスタンス変数 `self` の参照
- 他スレッドから参照できる場所への参照 `this` の書き込み

調査の結果，これらの処理がないと分かったならば，インスタンス変数 `self` が，条件 b も満たすといえる．ここでインスタンス変数 `self` の参照がないことは，コンストラクタ `Brick()` の内容から明らかだが，参照 `this` の書き込みがあるか否かは，コンストラクタ `Brick()` を見ただけでは分からない．なぜなら，コンストラクタ `Brick()` は，3 行目と 4 行目の間で，暗黙に，参照 `this` を引数として親クラス `java.lang.Object` のコンストラクタ `Object()` を呼び出すからである．そこでコンストラクタ `Object()` も含めて調査すると，コンストラクタ `Object()` の中身は空であり，参照 `this` を他スレッドから参照できる場所に書き込む処理もないと分かる．したがって，インスタンス変数 `self` は条件 b も満たすといえる．なお，ここで条件 b を満たすための要件に，他スレッドから参照できる場所への参照 `this` の書き込みが入る理由は，インスタンス変数 `self` の初期化より前に，参照 `this` を他スレッドから参照できる場所に配置すると，未初期化のインスタンス変数 `self` を非同期に参照される可能性が生じてしまうからである．

さて，インスタンス変数 `self` が条件 a，b を満たすと分かったので，10 行目にあるインスタンス変数参照 `this.self` を冗長と見なして削除する．削除がもたらす効果をあきらかにするため，9，10 行目でメソッド呼び出しの前に暗黙に実施する `null` 検査を明示したコードを図 1 (b) に示す．図 1 (b) では，1 行目と 5 行目にインスタンス変数 `self` の参照があるが，5 行目のインスタンス変数参照 `this->self` を `tos` に置き換えて削除すると，6 行目の null 検査も削除可能になる．なぜなら `tos` に対する null 検査は 2 行目で実施済みであり，6 行目の時点では `tos` が null でないことは明らかだからである．最適化後のコードを図 1 (c) に示す．この例が示すように，冗長なインスタンス変数参照の削除は，単にインスタンス変数参照を削除するだけでなく，他の最適化の適用箇所に影響を与えうる．

4. Ghemawat らによる実装の問題点

Ghemawat らによる実装は，解析に要するコストを考慮した有用なものだが，次に示す問題を持つ．

- リフレクションへの非対応
- 動的ロードへの非対応
- ネイティブメソッドへの非対応
- 内部クラスへの配慮不足

これらの問題点のうち，最初の 3 つは Ghemawat らが指摘したもので，解決しないと Ghemawat らの提案を実用化できない．最後の 1 つは我々が指摘する問題で，解決しないと最適化の適用範囲が狭くなりうる．本章では，これら 4 つの問題について順次述べる．

4.1 リフレクションへの非対応

リフレクションとは，プログラムからプログラム自身を参照するための機能である．リフレクションを使うと，クラス名やメソッド名，メンバ変数名に対応するクラスやメソッド，メンバ変数の検索や，検索したメソッドの呼び出し，メンバ変数の読み書きといった操作が可能になる．

リフレクションが Ghemawat らの実装にとって問題になる理由は，リフレクションを使うと，可視範囲の外からインスタンス変数を書き換えられるからである．たとえば Ghemawat らの実装によって，図 1 (a) の 2 行目で宣言しているインスタンス変数 `self` が最適化の対象になるか判断する場合について考える．このとき Ghemawat らの実装は，インスタンス変数 `self` を書き換えるコードを探索するが，探索にあたって，探索の範囲を宣言元のクラス `Brick` 内に限定する．これはインスタンス変数 `self` の可視範囲が `private` であることから，書き換えを行うコードは宣言元のクラス内にしかないと判断した結果だが，この判断はリフ

```
1:   Class c = Class.forName("Brick");
2:   java.lang.reflect.Field f =
3:     c.getDeclaredField("self");
4:   f.setAccessible()
5:   f.set(obj, null);
```

図 2 リフレクションによるインスタンス変数の書き換え
Fig. 2 Overwriting an instance variable using reflection.

レクションの存在下では誤りになる．

リフレクションを使って，メンバ変数 `self` を，クラス `Brick` の外から書き換えるコードを**図 2** に示す．図 2 のコードでは，まず 1 行目で，クラス `Brick` への参照を取得し，次に 2〜3 行目で，クラス `Brick` が宣言するメンバ変数 `self` への参照 `f` を取得する．続く 4 行目では参照 `f` を通じた可視範囲外からのメンバ変数 `self` の読み書きを可能にし，最後に 5 行目で参照 `obj` が指示するインスタンスのメンバ変数 `self` の値を `null` に書き換える．図 2 で呼び出す `forName()`，`getDeclaredField()`，`setAccessible()`，`set()` はいずれも，Java の標準クラスライブラリが提供するリフレクション関係のメソッドである．

リフレクションの問題に対応するには，リフレクション関係のメソッドを監視すればよい．具体的な対応の手順を次に示す．

- 最適化時に，どのインスタンス変数を対象として冗長なインスタンス参照の削除を適用したか，記録を作成しておく．この記録は，動的コンパイル済みコードごとに作成する．
- 実行時に，メソッド `getDeclaredField()` を監視し，メソッドが呼び出されたら，対象となるメンバ変数がどれか求める⋆1．求めたメンバ変数が，冗長なインスタンス参照の削除の対象となっていたら，すべての動的コンパイル済みコードに作成しておいた記録を走査し，対象となるメンバ変数を含む記録に対応する全動的コンパイル済みコードに対し，脱最適化を適用する．

ここで脱最適化とは，適用済の最適化を無効化する措置のことであり，その具体的な実行手段にはコードの上書きやインタプリタへの実行の引継ぎなどがある[5),6),14)]．

⋆1 Ghemawat らは監視対象のメソッドを `setAccessible()` にするよう提案しているが，これは適切でない．なぜなら，メソッド `setAccessible()` の実行が必要になるのは，可視範囲の制約を無視して書き換えを行う場合のみであるのに対し，書き換えが最適化の妨げとなるのは可視範囲の制約を無視したか否かによらないからである．

4.2 動的ロードへの非対応

動的ロードとはクラスを実行時に読み込む機能である．Java の実行環境はプログラムを構成するすべてのクラスを動的にロードする．

動的ロードが Ghemawat らの実装にとって問題になる理由は，動的ロードの存在下では，クラス間にまたがるプログラムの解析が難しくなるからである．たとえば，Ghemawat らの実装のうち，条件 a が満たされているか否か検証する作業について考えてみる．条件 a が満されるためには，インスタンス変数を書き換えるコードが，コンストラクタの外にあってはならないが，その有無を確認するには，インスタンス変数の可視範囲をひととおり調査する必要がある．たとえば，インスタンス変数の可視範囲が宣言元のクラスと同一のパッケージに所属する全クラスであるならば，パッケージ中の全クラスを対象として調査を行う必要があるが，動的ロードの存在下では調査の時点ですべてのクラスを揃えることはできなくなる．なぜなら，動的ロードの存在下では，クラスが実行時に生成されうるからである．

この問題に対処するために，Ghemawat らは，クラスをまとめて読み込み，読み込み後のクラスの追加を禁じているが，これでは Java の言語仕様に反する．

動的なクラスの追加に対応しつつ最適化を適用する方法は，脱仮想化の実現などにおいて実用化されている[6)]．脱仮想化とは，仮想呼び出しの高速化を目的とした最適化技法であり，仮想呼び出しにおける呼び出し先の検索を省略することで高速化を実現する．具体的には，仮想呼び出しのうち，呼び出し先が唯一に定まるものを求め，求めた仮想呼び出しについては，呼び出し先の検索が冗長といえるので，検索を省略し，仮想呼び出しを，唯一の呼び出し先への直接呼び出しのみで実現する．しかしながら，仮想呼び出しの呼び出し先の候補数は動的ロードによって変化しうる．たとえば，コンパイルの時点では，メソッド $m()$ を定義するクラスが C_p だけだったので，仮想呼び出し $o.m()$ を直接呼び出し $C_p :: m(o)$ に最適化したが，最適化後に新たに動的ロードを行ったところ，動的ロードしたクラス C_s がメソッド $m()$ を再定義していたため，呼び出し先候補が 2 つに増えた，といった事態が発生しうる．このとき直接呼び出しに最適化したコードを使い続けると，仮想呼び出し $o.m()$ において，o が指示するインスタンスのクラスが C_s であるとき，$C_s :: m()$ を呼び出すべきであるにもかかわらず，$C_p :: m()$ を呼び出してしまう，といった問題が生じる．脱仮想化の実装では，この問題を解決す

```
1:   class Wolf{
2:     static native void intrude(Brick obj);
3:   }
```

(a) 宣言

```
1:   JNIEXPORT void JNICALL Java_Wolf_intrude(JNIEnv* env, jclass clazz, jobject obj){
2:     jclass klazz = (*env)->FindClass(env, "Brick");
3:     jfieldID selfID = (*env)->GetFieldID(env, klazz, "self", "LBrick;");
4:     (*env)->SetObjectField(env, obj, selfID, NULL);
5:   }
```

(b) 実装

図 3　ネイティブメソッド
Fig. 3　A native method.

るために，動的ロードの結果として過去に適用した最適化が不適切になったら，脱最適化を適用する．

この対応方法は冗長なインスタンス参照の削除にも応用できる．動的ロードへの対応方法も，前節で示したリフレクションへの対応方法も，基本的な方針は同一で，最適化の時点では，その時点の実行状況（どのインスタンス変数がリフレクション経由で書き換えられうるか，どのクラスがロード済みか）だけを考慮して最適化を適用し，後で実行状況が変化して，過去に適用した最適化が不適切になったら，最適化済みコードを脱最適化すればよい．

4.3　ネイティブメソッドへの非対応

ネイティブメソッドとは，Java のクラスが宣言するメソッドのうち，ネイティブコードで実現されているもののことである．

ネイティブメソッドが Ghemawat らの実装にとって問題になる理由は，リフレクションと同じで，ネイティブメソッド内では，可視範囲や final 宣言による制限を無視してインスタンス変数を書き換えられるからである．このような書き換えを行うプログラムの例を**図 3** に示す．図 3 は宣言と実装からなり，具体的には図 3 (a) が Java ソースコードによるクラス `Wold` の定義であり，この定義中でネイティブメソッド `intrude()` の宣言している．図 3 (b) はネイティブメソッド `intrude()` の C 言語による実装である．ネイティブメソッド `intrude()` は，図 1 (a) のクラス `Brick` のインスタンスを引数として受け取り，そのインスタンス変数 `self` の値を null に書き換える．インスタンス変数 `self` は図 1 (a) の 2 行目にある宣言から分かるように，`private` 変数なので，クラス `Wolf` からは不可視のはずだが，ネイティブメソッドからなら書き換えられる．書き換えの手順は図 3 (b) に示すとおりであり，まず 2 行目でクラス `Brick` への参照 `klazz` を取得し，次に 3 行目で `klazz` に，名前が `self` で型が `Brick` なメンバ変数を表す番号 `selfID` を問い合わせ，最後に 4 行目で引数 `obj` の `selfID` に対応するインスタンス変数の値を `NULL` に書き換える．

ネイティブメソッドの存在下で Ghemawat らの冗長なインスタンス変数参照の削除を実行するには，インスタンス変数のうち，ネイティブメソッドによる書き換えの対象になるものを求める必要が生じるが，これを静的な解析によって求めることは困難である．だからといって保守的に，ネイティブメソッドはすべてのインスタンス変数を変更しうると判断すると，最適化を適用できなくなる．なぜなら，この判断の下で，Ghemawat らの冗長なインスタンス変数参照の削除を適用できるケースは，ネイティブメソッドが 1 つもない状況に限られるが，クラス `java.lang.Object` をはじめとした，実行時に必ず読み込まれるクラスがネイティブメソッドを宣言していることから，ネイティブメソッドが 1 つもない状況などありえないからである．

この問題を解決するために，Ghemawat らは，Java の標準クラスライブラリが提供するネイティブメソッドのうち，主要なものについては，それぞれが書き換えうるインスタンス変数を，最適化系に教えておき，残りのネイティブメソッドについては，そのメソッドがネイティブでなかった場合に可視なインスタンス変数に限って書き換えうると仮定して最適化を適用している．しかしながら，このような仮定は Java の言語仕様に反する．また，ネイティブメソッドがどのインスタンス変数を書き換えうるかという情報を，手動でコンパイラに提供するには手間がかかる．

4.4　内部クラスへの配慮不足

内部クラスとは，クラスのうち，その定義が他のクラス定義の中にあるもののことである．

内部クラスが Ghemawat らの実装にとって問題になる理由は，内部クラスの実装において暗黙に使う `this` が最適化を妨げうるからである．ここではまず，**図 4** を使って内部クラスの具体例とその実装，`this`

```
 1:  class PC{
 2:    private final VGA v;
 3:    private final CPU c;
 4:    PC(){
 5:      c = new CPU();
 6:      v = new VGA();
 7:    }
 8:    class VGA{
 9:      VGA(){}
10:      void clear() { ··· }
11:    }
12:    class CPU{
13:      CPU(){}
14:      void reset() {
15:        v.clear();
16:      }
17:    }
18:  }
```

(a) ソースコード

```
 1:  class PC{
 2:    private final PC$VGA v;
 3:    private final PC$CPU c;
 4:    PC(){
 5:      c = new PC$CPU(this);
 6:      v = new PC$VGA(this);
 7:    }
 8:    static PC$VGA access$000(PC tmp){
 9:      return (tmp.v);
10:    }
11:  }
12:  class PC$VGA{
13:    private final PC this$0;
14:    PC$VGA(PC outer){
15:      this$0 = outer;
16:    }
17:    void clear() { ··· }
18:  }
19:  class PC$CPU{
20:    private final PC this$0;
21:    PC$CPU(PC outer){
22:      this$0 = outer;
23:    }
24:    void reset() {
25:      PC.access$000(this$0).clear();
26:    }
27:  }
```

(b) 実装

図 4 内部クラス
Fig. 4 An inner class.

の暗黙の使用について述べ，次に this の暗黙の使用が Ghemawat らの実装に及ぼす影響を示す．

図 4 (a) は内部クラスを使って作成したプログラムの例であり，クラス PC の定義中で，2 つの内部クラス CPU，VGA を定義している．クラス CPU，VGA はクラス PC の内部で定義されていることから，クラス PC のメンバ変数を参照できる．たとえばクラス CPU が定義するメソッド reset() の中では，15 行目でインスタンス変数 v を参照しているが，このインスタンス変数はクラス PC が 2 行目で宣言しているものである．

Java における内部クラスの実装では，ソースコードをクラスファイルにコンパイルする処理系 javac が，内部クラスを単なるクラスに変換する．したがって，クラスファイルの実行を担当する Java 実行環境や，その一部である動的コンパイラへの入力となるのは，単なるクラスのみになる．図 4 (a) のソースコードを対象として，この変換を適用した結果を図 4 (b) に示す．図 4 (b) の 12〜18 行目にあるクラス PC$VGA と，19〜27 行目にあるクラス PC$CPU は，図 4 (a) の内部クラス VGA，CPU に変換を適用した結果として得られたクラスである．

変換後のソースコード（図 4 (b)）を参照すると，5 行目のコストラクタ呼び出しにおいて，コンストラクタ PC$CPU() に実引数 this を渡していることが分かる．この実引数は変換前のソースコード（図 4 (a) の 5 行目）にはなかったもので，javac が追加したものである．実引数 this はコンストラクタ PC$CPU() 内の 22 行目でインスタンス変数 this$0 に格納される．インスタンス変数 this$0 も javac が追加したもので，内部クラスの中から，その定義を包含するクラスのインスタンスを参照する処理を実現する際に使われる．たとえば図 4 (a) の 15 行目には，内部クラス CPU の中から，その定義を包含するクラス PC のインスタンス変数 v を参照する処理があるが，この処理の実現は，変換後のソースコード（図 4 (b)）の 25 行目にあるように，メソッド呼び出し PC.access$000(this$0) であり，ここで this$0 が引数として使われている．呼び出し先のメソッド access$000() は，javac が 8〜10 行目に追加したもので，引数が参照するクラス PC のインスタンスのインスタンス変数 v の値を返戻する．

javac が図 4 (b) の 5，6 行目に追加した実引数 this は，内部クラスの実装上必要なものだが，Ghemawat らの最適化の妨げとなる．なぜなら，これらの this があると，Ghemawat らの最適化系は，インスタンス変数 c，v は条件 b を満たさないと見なし，これらのインスタンス変数に対して冗長なインスタンス変数参照の削除を適用しなくなるからである．Ghemawat らの最適化系では，インスタンス変数が条件 b を満たすか否か調べるために，コンストラクタを対象として脱出解析[1),9)] を行い，インスタンス変数の初期化より前に，this が他のスレッドから参照されうるか検証するが，脱出解析はメソッド間にまたがって解析を行うため，実行にコストがかかる．そこで Ghemawat らは，脱出解析のコストを軽減する手段として，解析の

範囲を限定する方法を提案している．この提案では，メソッド呼び出しのうち，this がレシーバになっているものに限り，呼び出し先のメソッドにまたがって解析を行うが，この提案に従うと，図 4 (b) の 5，6 行目にあるコンストラクタ呼び出しは this がレシーバでないことから解析の対象から外れる．そうなると，これらのコンストラクタ呼び出しは解析対象外のところに this を引き渡すことになり，結果として this を他スレッドから参照可能にする要因と勘違いされてしまう．勘違いを防ぐためには脱出解析における解析の範囲を見直せばよいが，解析の範囲を無闇に広げると，解析にかかるコストが大きくなってしまう．

Ghemawat らの実装にとって，内部クラスが問題になるのは，図 4 の例のように，コンストラクタ内で内部クラスのインスタンス生成を行い，そのコンストラクタに this を暗黙に実引数として引き渡す場合だが，この場合にあてはまるプログラムは必ずしも珍しくない．内部クラスを使ったプログラミングは，今では広く普及しているので，最適化の設計にあたっては，内部クラスの実装や使用例にも配慮した方がよい．

5. 提 案

我々が提案する冗長なインスタンス変数参照の削除では，基本的に，参照の結果が一定であるインスタンス変数を対象として最適化を行う．この基本方針は Ghemawat らの実装と同じである．ただし，我々の実現は，主に次の点で Ghemawat らの実装と異なる．

- 最適化候補のインスタンス変数
- 脱出解析におけるメソッド間解析の範囲
- リフレクションおよびネイティブメソッドのサポート

本章では，まず，これらの点が，なぜ，どのように異なるか示し，次に，我々が提案する冗長なインスタンス変数参照の削除の実施手順について詳述する．

5.1 最適化候補のインスタンス変数

Ghemawat らの実装では，最適化対象の候補とするインスタンス変数を，可視範囲および型によって制限している．具体的には，最適化対象の候補とするインスタンス変数の可視範囲は，任意のパッケージの任意のクラスから読み書き可能であってはならず，さらに，型は，参照型でなくてはならないとしている．

Ghemawat らの実装が型による制限を行う理由は，Ghemawat らの解析の目的が，冗長なインスタンス変数参照の削除だけでなく，参照型の値向けの最適化を幅広く促進するために必要な情報を収集することにあるからである．単純に冗長なインスタンス変数参照の削除の実装だけを考えるのならば，型による制限は無用であり，したがって我々の実装では型による制限は行わない．

一方で我々の実装では，可視範囲に関して，より厳しい制限を行うことにした．具体的には，最適化対象の候補となるインスタンス変数を，private もしくは final 宣言*1されたもの，つまり，基本的には宣言元のクラスのみが書き換え可能なものとした．

この制限の目的は，最適化の実装を容易にすることにある．宣言元のクラスのみが書き換え可能なインスタンス変数に限って最適化を適用するならば，書き換え元がコンストラクタ外にあるか否かの判定を，宣言元のクラスを解析するだけで実行できる．これに対し，たとえばパッケージ内の全クラスから書き換え可能なインスタンス変数を対象に最適化を適用するとなると，この判定のために，パッケージ内の全クラスを解析する必要が生じ，結果として次の問題に対処しなくてはならなくなる．

解析コスト パッケージ内に存在するクラスの数が多い場合に解析のコストが大きくなる．

動的ロード クラスの動的ロードにあたって，ロードしたクラスが所属するパッケージの中に，1 つでも最適化対象のインスタンス変数があったら，ロードしたクラスについても解析を行わなくてはならなくなる．また，解析の結果として，これまで最適化対象として扱ってきたインスタンス変数が，最適化対象外になったら，必要に応じて脱最適化を行わなくてはならなくなる．

これらの問題の解決に手間をかける価値があるか否か判断するために，可視範囲の制約を緩和すると，最適化の効果がどう変化するか推定した．推定にあたっては，まず，最適化候補のインスタンス変数の数と，インスタンス変数の参照回数のそれぞれについて，可視範囲との関係を調査した．それぞれの調査の方法と結果について順次述べる．

5.1.1 最適化候補のインスタンス変数の数

最適化候補のインスタンス変数の数と可視範囲の関係については，次の手順で調査を行った．調査対象は Java 実行環境（version 1.5.0_05）が提供するクラスライブラリである．

(1) クラスライブラリ内で宣言されているインスタンス変数を，書き換えるバイトコードが，コンストラクタ以外のどこにあるか求め，求めた結

*1 本論文では，final 宣言を可視範囲の規定の一種として扱う．なぜなら，final 宣言は書き換え可能なクラスを規定するからである．

表 1 インスタンス変数の分布
Table 1 A distribution of instance variables.

コンストラクタ外の書き換え用バイトコードの所在	可視範囲					合計
	final	**private**	**package**	**protected**	**public**	
なし	5486	5215	2367	1059	317	14444
private	0	8426	2413	1967	267	13073
package	0	0	642	101	77	820
public	0	0	0	19	77	96

果に応じて，インスタンス変数を，次の 4 つに分類する．

なし コンストラクタ以外の場所に，書き換えを行うバイトコードがない．ここに分類されたインスタンス変数が最適化対象の候補となる．

private 宣言元のクラス内の，コンストラクタ以外の場所に書き換えを行うバイトコードがある．

package 宣言元のクラス内にはないが，同一パッケージに所属する別のクラス内の，コンストラクタ以外の場所に書き換えを行うバイトコードがある．

public 同一パッケージのクラス内にはないが，別のパッケージに所属するクラス内の，コンストラクタ以外の場所に書き換えを行うバイトコードがある．

(2) 4 つに分類したインスタンス変数を，その可視範囲に応じて，さらに，次の 5 つに分類する．

final `final` 宣言されたもの．基本的に宣言元のクラスでしか書き換えられない．

private `private` 宣言されたが，`final` 宣言されていないもの．基本的に宣言元のクラスでしか書き換えられない．

package `final` とも `private` とも宣言されておらず，また，`public` 宣言されたクラスにおいて `protected` または `public` と宣言されてもいないもの．基本的に宣言元のクラスと同じパッケージに所属するクラスからしか書き換えられない．

protected `public` 宣言されたクラスにおいて `protected` 宣言されたが，`final` 宣言されていないもの．基本的に宣言元のクラスと同じパッケージに所属するクラスか，宣言元のクラスを継承するクラスからしか書き換えられない．

public `public` 宣言されたクラスにおいて `public` 宣言されたが，`final` 宣言されていないもの．任意のパッケージの任意のクラスから書き換えられる．

この分類によって，インスタンス変数は $4 \times 5 = 20$ 個に分割された区分のいずれかに所属することになる．どの区分にどれだけの数のインスタンス変数が所属するか求めた結果を**表 1** に示す．

表 1 から，最適化対象の候補となるインスタンス変数，つまりコンストラクタの外に書き換え用のバイトコードが存在しないインスタンス変数の総数が 14444 個あり，そのうち可視範囲が **final** もしくは **private** であるものが，全体の 74.1%にあたる 10701 個になることが分かる．ここから可視範囲を広げれば残りの 25.9%にも最適化を適用可能になりうるが，可視範囲の拡張にあたっては先述の問題を解決する必要が生じる．

5.1.2 インスタンス変数の参照回数

インスタンス変数の参照回数と可視範囲の関係については，ベンチマークをインタプリタで実行し，実行の過程で，それぞれの可視範囲のインスタンス変数をどれだけ参照するかを調査した．調査対象のベンチマークには SPECjvm98 [12] を用いた．SPECjvm98 は `javac` など 7 本の実用的 Java アプリケーションから構成されるベンチマーク集である．

調査結果を**表 2** に示す．表 2 から，参照先のインスタンス変数の可視範囲が **final** もしくは **private** である確率は，相加平均では 53.6%になることが分かるが，同時に，ベンチマークごとのばらつきが大きいことも分かる．ベンチマークごとのばらつきが大きい原因の 1 つとして，プログラマがきちんとインスタンス変数の可視範囲を指定したか否かが，ベンチマークによって異なる点を指摘できる．

たとえば表 2 から，`_213_javac` の実行時に参照先のインスタンス変数の可視範囲が **final** もしくは **private** である確率は 33.4%になることが分かるが，SPECjvm98 に含まれる `javac` の実装は古く，可視範囲の指定などが十分になされていない．現在の `javac` の実装には，より適切な可視範囲の指定がなされており，たとえば `JDK1.5.0_05` に付属の `javac` を使っ

表 2 インスタンス変数の参照回数の分布
Table 2 A distribution of instance variables.

ベンチマーク項目	参照回数の可視範囲ごとの構成比 (%)				
	final	**private**	**package**	**protected**	**public**
SPECjvm98					
_201_compress	0.0	50.2	0.0	49.8	0.0
_202_jess	4.3	24.3	37.8	33.5	0.1
_209_db	37.2	20.4	16.2	26.2	0.0
_213_javac	10.8	22.6	33.1	33.2	0.3
_222_mpegaudio	0.0	63.1	11.5	25.4	0.0
_227_mtrt	0.5	86.0	12.3	1.2	0.0
_228_jack	17.8	37.6	19.7	23.9	1.0
相加平均	10.1	43.5	18.6	27.6	0.2
その他					
JDK1.5.0_05 の javac	7.2	36.4	16.0	0.4	40.0

て_213_javac と同じ処理を行うと，同確率は 43.6%と，SPECjvm98 に含まれる_213_javac を使った場合に比べ，10.2%増加する．この例が示すように，アプリケーションプログラマが，実行性能やソフトウェアの生産性に配慮して，きちんと可視範囲の指定を行えば，参照先のインスタンス変数の可視範囲が **final** もしくは **private** になる確率は増える．したがって，実行性能が重要で，アプリケーションプログラマがきちんと可視範囲を指定する局面に限って考えれば，表 2 の相加平均値より，可視範囲が **final** もしくは **private** になる確率は大きくなりうる．

表 1 および表 2 における調査結果はともに，最適化が実行速度に与える影響と直接的には関係しない．なぜなら，最適化が実行速度に与える影響は，最適化を適用した箇所の実行回数によって定まるからである．しかしながら，もし仮に，ここで求めた最適化候補の数あるいは参照回数の分布と，最適化の効果にある程度の関連性があると想定するなら，最適化対象の可視範囲を **final** もしくは **private** に限定したとしても，効果の過半は得られると考える．この考えから，我々は，今のところ，最適化対象の候補となるインスタンス変数を，可視範囲が **final** もしくは **private** のものに限定している．

5.2 脱出解析におけるメソッド間解析の範囲

4.4 節で述べたように，Ghemawat らの実装では，脱出解析におけるメソッド間解析の範囲を限定しすぎているために，コンストラクタ内で内部クラスのインスタンスを生成するクラスに対して，最適化を適用できないという問題をかかえている．この問題の発生頻度は必ずしも小さくない．たとえば SPECjvm98 の 1 項目である_201_compress で，この問題が発生する．

この問題を解決するため，我々は，脱出解析におけるメソッド間解析の範囲を見直すことにした．解析の範囲を広くすると，解析にかかる時間が増加するが，この問題については，解析範囲の変更がもたらす実行速度の改善率と，解析時間の増加率について，定量的に評価しながら解決策を検討することにした．具体的には，解析の範囲が異なる 2 つの脱出解析を用意して，どちらを使うと，どれだけ実行速度が速くなり，どれだけ解析時間が長くなるか検討することにした．

比較対象として用意した 2 つの脱出解析について，まず，それぞれの共通部分について述べ，次に，それぞれの相違部分について述べる．

まず共通部分だが，これらの脱出解析はメソッド内解析とメソッド間解析からなる．メソッド内解析は，Choi らの実装[1] をベースに開発したもので，メソッド内で生じるインスタンス間の参照関係を求める役割を果たす．メソッド間解析は，起点となるメソッドから，その呼び出し先のメソッドに向って，メソッド内解析で得られた結果を順次つなぎあわせ，最終的な脱出解析の結果を得る役割を果す．メソッド間解析では，解析コストの爆発を防ぐために，呼び出し深さなどの観点から解析範囲を制限する．解析範囲外となったメソッドに引数として渡されるインスタンスや，起点となるメソッドが受け取る引数から参照可能になるインスタンスについては，保守的に，他スレッドから参照可能になると見なす．ただし，起点となるメソッドがコンストラクタならば，その第 1 引数，すなわち this が参照するインスタンスは，割付け直後なので，コンストラクタの起動時点では，まだ，他スレッドから参照されていないと見なす．

次に相違部分だが，用意した脱出解析の一方において，メソッド間解析のコストを軽減させることを目的として，解析範囲の制限を追加し，メソッド間解析においてたどるメソッド呼び出しを，呼び出し先が唯一に定まるもののみにした．

解析範囲の制限を追加しないことの利点は，解析範囲が広いことから，より多くのインスタンス変数を最適化対象にできることであり，欠点は解析に比較的長い時間がかかることと，動的ロードの際に，脱最適化が必要になりうることである．脱最適化が必要になる理由は，メソッド間解析において，呼び出し先が唯一に定まらないメソッド呼び出しをたどる際に，呼び出し先の候補となるメソッドを，ロード済みのクラスによって定義されているもののみと想定するのに対し，解析後の動的ロードによってメソッドを再定義すると，解析時に想定しなかった呼び出し先が増えることから，過去に実施したメソッド間解析と，その結果を使って最適化したコードが無効になるからである．

反対に，解析範囲の制限を追加することの利点は，解析にかかる時間が比較的短いことと，脱最適化が必要にならないことである．なお，我々が用意した脱出解析では，解析範囲の制限を追加する場合でも，しない場合でも，内部クラスが暗黙に使用する `this` から悪影響を受けることはない．

評価の対象は SPECjvm98 とし，実行には日立製作所製サーバ HA8500/210（CPU: Itanium® ⋆1 2（1.3 GHz × 2），Memory 2 GByte）を使用した．使用した OS は Red Hat® ⋆2 Enterprise Linux™ ⋆3 4，Java 仮想機械は日立製作所製のものである．この Java 仮想機械は Sun Microsystems が JDK1.5.0_05 向けに開発したものに対して，日立製作所が独自の改良を施したもので，日立製作所製アプリケーションサーバ製品 Cosminexus version 7.1 の一部として配布されている．本論文の評価はすべてこの環境で行った．

評価においては，メソッド間解析の範囲に関する制限の追加の有無が解析時間，コンパイル時間および実行速度に与える影響を測定した．測定結果を**表 3** に示す．表 3 において，解析時間とは，脱出解析にかかる時間を表す．解析時間の測定は，削除なし，制限の追加あり，なしの 3 つの場合について実施した．ここで削除なし，とは冗長なインスタンス変数参照の削除を実施しないことを表す．評価環境として使った Java 仮想機械は，冗長な同期の除去や静的ごみ集めのために脱出解析を行うので，冗長なインスタンス変数参照の削除を実施しない場合でも，解析時間は 0 にならない．制限あり，なしは，それぞれ，冗長なインスタンス変数参照の削除を実施した場合で，なおかつ，解析範囲の制限を追加した場合と，しなかった場合を表す．また，表 3 において，翻訳時間とは，冗長なインスタンス変数参照の削除なしでのコンパイルにかかる時間の合計を表し，速度向上率とは，解析範囲の制限を追加しないことががもたらす実行速度の向上率を表す．表 3 において，解析範囲の制限を追加する場合と，しない場合を比較すると，制限を追加しないと解析時間が最大で 29%増えるが，その一方で，制限を追加しないことで得られる速度向上率は小さく，おおむね ±0.1%未満と測定誤差の範囲に収まることが分かる．そこで我々は，脱出解析におけるメソッド間解析の範囲に制限を追加することにした．

表 3 脱出解析の範囲がもたらす影響
Table 3 Effects by escape analysis scope.

ベンチマーク項目	解析時間 (sec) 削除なし	解析時間 (sec) 制限の追加 あり	解析時間 (sec) 制限の追加 なし	翻訳時間 (sec)	速度向上率 (%)
_201_compress	0.01	0.07	0.10	0.25	0.04
_202_jess	0.39	0.59	0.64	2.14	-0.19
_209_db	0.02	0.10	0.15	0.45	-0.08
_213_javac	5.49	5.83	6.34	28.2	0.04
_222_mpegaudio	0.06	0.16	0.19	1.16	-0.05
_227_mtrt	0.21	0.41	0.53	1.58	-0.10
_228_jack	0.29	1.01	1.01	12.4	-0.08

制限の追加を行う場合について，冗長なインスタンス変数参照の削除向けの脱出解析がもたらすコンパイル時間の増加率を，表 3 から求めると，1〜23%になることが分かる．増加率が大きいベンチマーク項目は，`_201_compress` など，翻訳時間が短いものである．冗長なインスタンス変数参照の削除向けの脱出解析の実施回数は，コンパイル対象のメソッドの数に応じて定まるわけではなく，解析対象のクラスが定義するコンストラクタの数に応じて定まる．このため，コンパイル対象のメソッドが極端に少なく，翻訳時間が短い場合には，脱出解析のコストが目立ちやすくなる．翻訳時間が長いベンチマーク項目では，増加率が小さくなり，`_213_javac` では 1%になる．

5.3 リフレクションおよびネイティブメソッドのサポート

4.1 節および 4.3 節で述べたように，Ghemawat らの実装は，リフレクションおよびネイティブメソッドをサポートしていないので，そのままでは実用化できない．この問題の解決を目的として，我々はリフレクションおよびネイティブメソッドをサポートすることにした．リフレクションのサポートは 4.1 節に記述した方法で実現した．

⋆1 Itanium は，米国およびその他の国における Intel Corporation またはその子会社の商標または登録商標です．

⋆2 Red Hat は米国 Red Hat, Inc. ならびにその子会社の登録商標です．

⋆3 Linux は，Linus Torvals 氏の商標です．

ネイティブメソッドのサポートについても，リフレクションとほぼ同様の方法で実現した．具体的には，ネイティブメソッドがインスタンス変数を参照する前に呼び出す関数 `GetFieldID()` を監視することで，ネイティブメソッドから書き換えられうるインスタンス変数を求め，求めたインスタンス変数を最適化の適用対象外にすることにした．監視の処理を実現するために，関数 `GetFieldID()` の内部に，返戻値に対応するインスタンス変数 i を，実行時システムに通知する処理を追加した．実行時システムは，この通知を受けると，コンパイラに対して，インスタンス変数 i を冗長なインスタンス変数参照の削除の適用対象外にするよう求め，また，通知以前にコンパイルしたメソッドの中に，インスタンス変数 i を対象として冗長なインスタンス変数参照の削除を適用したものがあるか調べ，あったら，そのメソッドの動的コンパイル済みコードに対して脱最適化を適用する．

ネイティブメソッドのサポートにおいて，監視の対象を，関数 `GetFieldID()` にした理由は，監視のコストを隠蔽するためである．関数 `GetFieldID()` は，元々，フィールド名の比較などを行うことから，実行に長い時間がかかる処理である．したがって，関数 `GetFieldID()` に監視の処理を追加しても，その全体の実行時間に大きな影響は出ない．

監視対象の候補となる関数には，ほかにも，関数 `SetObjectField()` など，インスタンス変数の値を書き換える関数群がある．監視対象をこの関数群にすると，より多くのインスタンス変数に冗長なインスタンス変数参照の削除を適用可能になる．なぜなら，これらの関数群を監視すれば，最適化の適用対象外とするインスタンス変数を，ネイティブメソッドが実際に書き換えるものだけに限定できるからである．関数 `GetFieldID()` は，ネイティブメソッドからインスタンス変数を参照する際にも利用されるため，関数 `GetFieldID()` を監視する方法では，ネイティブメソッドから参照されるだけで，書き換えられないインスタンス変数まで最適化の対象外になってしまう．

しかしながら，関数 `SetObjectField()` など，インスタンス変数の値を書き換える関数群の処理内容は，単にインスタンス変数の値を書き換えるだけのものであり，これらの関数群に監視の処理を追加すると，監視のコストを隠蔽しきれないケースが生じうる．

関数 `GetFieldID()` あるいは `SetObjectField()` に，監視の処理を追加すると，実行時間がどれだけ増加するか，それぞれの関数を連続して呼び出すマイクロベンチマークを使って評価した．評価対象のオーバヘッドは，定常実行状態にあるプログラムにおいて生じるものとし，このため，マイクロベンチマークでは，どちらの関数で監視した場合についても，監視中に脱最適化を発生させなかった．なぜなら脱最適化はプログラムの定常的な実行状態ではあまり発生しないからである．評価の結果から，関数 `GetFieldID()` では増加幅が -0.01%と小さいのに対し，関数 `SetObjectField()` では 117%と，実行時間が 2 倍以上になってしまうことが分かった．

関数 `SetObjectField()` など，インスタンス変数を書き換える関数群を監視する場合に生じるオーバヘッドの増加が問題になるケースは，もちろん，監視対象の関数群を頻繁に実行する場合に限られる．そのようなケースは珍しいかもしれないが，6 章で述べるように，冗長なインスタンス変数参照の削除の効果が相乗平均で 0.6%と大きくないことを考えれば，珍しいケースとはいっても，関数 `SetObjectField()` を頻繁に実行するアプリケーションの性能を著しく劣化させてまで冗長なインスタンス変数参照の削除の適用範囲を広げることに大きな意義があるとは考えにくい．我々の評価によれば，監視対象を，インスタンス変数を書き換える関数群に変更し，冗長なインスタンス変数参照の削除の適用範囲を拡大しても，SPECjvm98 の実行速度に与える影響はおおむね ± 0.1%内と測定誤差の範囲に収まることが分かった．このため，我々は監視対象をインスタンス変数を書き換える関数群にはしなかった．

5.4 実 装

我々による冗長なインスタンス変数参照の削除の実装は，大きく分けて，実行時システム側の実装と，動的コンパイラ側の実装からなる．それぞれの役割について順次詳述する．

5.4.1 実行時システム側の実装

実行時システム側の実装は，リフレクションおよび JNI の利用を監視して，これらの機能を通じた書き換えの対象となるインスタンス変数を求め，求めたインスタンス変数を最適化の対象から除外する役割をはたす．また，除外の際に，除外対象のインスタンス変数を対象とした冗長なインスタンス変数参照の削除を適用済みのコンパイル済みコードを求めて脱最適化する．

インスタンス変数を最適化対象から除外する操作は，インスタンス変数を表すデータ構造に，「最適化不可」の印を付けることで実現する．インスタンス変数を表すデータ構造には，この印を付けるための欄を追加する．この欄が保持しうる値は，「最適化不可」，「最適化可」，「未定」のいずれかで，初期値は「未定」とする．

5.4.2 動的コンパイラ

動的コンパイラ側の実装は，冗長なインスタンス変数参照を削除する役割をはたす．削除にあたっては，次に示す 3 つの処理を行う．

(1) インスタンス変数が最適化対象となりうるか否かの調査
(2) 冗長なインスタンス変数参照の削除
(3) メモリバリアの挿入

個々の処理について順次，詳述する．まず，最適化対象のインスタンス変数の探索だが，この処理は，動的コンパイルの際に，コンパイル対象のメソッド中で参照するインスタンス変数について，最適化対象となるか否かの調査が済んでいるか調べ，まだであれば実施する．調査の手順を次に示す．

(1) インスタンス変数が `private` もしくは `final` 宣言されているか否か調べる．いずれにも該当しない場合には，インスタンス変数を表すデータ構造に「最適化不可」の印を付けて調査を終了する．
(2) インスタンス変数の宣言元のクラスが定義する全メソッドを走査して，インスタンス変数に代入を行うバイトコードを含むものがあるか調べる．あったならば，求めたバイトコードがコンストラクタ内で `this` のインスタンス変数に代入を行うものか調べる．調べた結果が偽ならば，インスタンス変数を表すデータ構造に「最適化不可」の印を付けて調査を終了する．
(3) インスタンス変数の宣言元のクラスが定義するコンストラクタに脱出解析を適用し，その第 1 引数，つまりコンストラクト中のインスタンスが，コンストラクタの実行が終わるまでの間に他スレッドから参照可能になるか調べる．調査の結果，参照可能にならないと分かったら，インスタンス変数を表すデータ構造に「最適化可」の印を付ける．さもなくば「最適化不可」の印を付ける．

調査が終了したら，次に，冗長なインスタンス変数参照を削除するために，次の処理を行う．

(1) インスタンス変数参照のうち，次の条件を満たすペアを求める．
  - 同じインスタンスの同じインスタンス変数を参照する．
  - 参照先のインスタンス変数を表すデータ構造に「最適化可」の印が付いている．
  - 一方が他方より必ず先に実行される．
  - 双方がインスタンス変数の宣言元のクラスのコンストラクタ内にない．インライン展開によってコンストラクタ内に展開したインスタンス変数参照については，コンストラクタ内にあると見なす．
(2) 求めたペアの後続側を，先行側の参照結果で置き換える．

なお，この実施手順では，宣言元のクラスのコンストラクタ内にあるインスタンス変数参照を，最適化対象から除外しているが，その理由は，我々の実装ではコンストラクタ内におけるインスタンス変数の書き換えを特に制限なく許しているため，コンストラクタ内ではインスタンス変数の値が一定になるとは限らないからである．Ghemawat らの実装では，最適化対象のインスタンス変数を，コンストラクタ内でただ 1 度初期化され，初期化以前に参照されないものに限定することで，コンストラクタ内にあるインスタンス変数参照にも最適化を適用しているが，我々の実装では，この限定を行わない．

最後に，メモリバリアの挿入について述べる．メモリバリアを挿入する目的は，プロセッサが次に示す 2 つのストア命令の実行順序を入れ替えないようにすることにある．

- 最適化対象のインスタンス変数を初期化するためストア命令 S_i
- ストア命令 S_i のストア先のインスタンスへの参照を他スレッドから参照可能な場所にストアする命令 S_o

ストア命令 S_i は必ずコンストラクタ内にあり，ストア命令 S_o はコンストラクタ呼び出しの後にのみ現れうるので，命令列の上では必ず S_i が S_o より先に現れるが，プロセッサの中には，実行時にその実行順序を入れ替えるものがある．入替えによって，S_o の実行が S_i の実行より前になると，他スレッドによって未初期化のインスタンス変数を参照される恐れが生じる．この問題を防ぐために，S_i を含むコンストラクタの末尾にメモリバリアを挿入し，S_i と S_o の実行順序の入替えがおきないようにする．

メモリバリアの挿入は元々，`final` 宣言されたインスタンス変数を対象として行われている処理[4] だが，我々はこの処理の適用対象に最適化対象のインスタンス変数を含めることにした．

ここで挿入するメモリバリアは実行速度の劣化要因になりうる．劣化を防ぐための手段として考えられるものに，次の 2 つがある．

(1) メモリバリアのうち，冗長なものを，最適化によって削除する．具体的には，脱出解析によっ

て単一のスレッドのみが参照可能なインスタンスの割付け元を求め，求めた割付け元の直後にあるコンストラクト処理からメモリバリアを削除する．

(2) 実行時プロファイルから実行頻度が非常に高いコンストラクタを求め，求めたコンストラクタについてはメモリバリアを挿入するのが適当でないと判断して，コンストラクタの定義元のクラスが保持するインスタンス変数を最適化対象から除外する．

我々の実現では，まだ，これらのメモリバリア対策を行っていない．その理由は，メモリバリアの挿入による実行速度の劣化がさほど大きくなかったからである．SPECjvm98 による評価では，挿入による実行速度の劣化は 0.1%に満たなかった．

6. 評 価

我々が実装した冗長なインスタンス変数参照の削除が実行速度に与える影響を，SPECjvm98 および SPECjbb2005 [13] を使って調査した．SPECjbb2005 はサーバサイドで動作するビジネスロジックを模したベンチマークである．調査結果を**図 5** に示す．図 5 の横軸はベンチマーク項目名を，縦軸は最適化の適用によって得られる実行速度の向上率を表す．なお，我々が最適化を実装した Java 向け動的コンパイラは，あらかじめ，局所フロー解析を使って冗長なインスタンス変数参照を削除する最適化を持っており，したがって，図 5 に表れる性能向上率は，既存の最適化では除去できないものを除去した結果として得られたものになる．図 5 から，実装した最適化の効果が相乗平均で 0.6%になることが分かる．

図 5 には，冗長なインスタンス変数参照の削除を，我々の提案どおりに実装した場合の効果だけでなく，我々の実装を，Ghemawat らの提案にあわせて改変した場合の効果も示してある．具体的な改変箇所を次に示す．

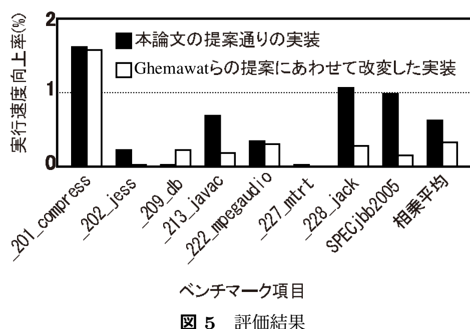

図 5 評価結果
Fig. 5 Benchmark results.

- 最適化候補のインスタンス変数を，可視範囲が `public` でなく，宣言型が参照型であるものとする．
- 脱出解析の範囲を 4.4 節で述べたとおりに変更する．

これらの改変を施した我々の冗長なインスタンス変数参照の削除と，実際の Ghemawat らの実装は同一でない．なぜなら，これらの改変を施しても，Escape 解析など最適化を構成する個々の要素の細かな実装までが同一になるとは考えられないからである．したがって図 5 に示した 2 つの実装の比較は，あくまで，我々の提案どおりの実装と，我々の実装を Ghemawat らの提案にあわせて改変したものの比較であって，Ghemawat らの実装そのものとの比較ではない点に注意する必要がある．なお，Ghemawat らの実装と，我々の実装を改変したものの間にどれほどの違いがあるかは推定し難い．推定が難しいことの原因の 1 つに，Ghemawat らの論文に，彼らの提案技術が冗長なインスタンス変数参照の削除の効果に与える影響を評価した結果が示されていないことがある．

図 5 に示した 2 つの実装の評価結果を比較すると，どちらの方法で実装しても実行速度に与える影響に大きな違いは生じないことが分かる．なお，図 5 に示した SPECjvm98 の評価結果は，動的コンパイルや脱最適化，再コンパイルの作業がすべて完了した後で測定したものである．したがって，評価結果は脱最適化や動的コンパイルの影響を含まない．結果の差異の主な原因は最適化後のコードの品質であると考える．

図 5 から分かるように，我々が実装した冗長なインスタンス変数参照の削除は必ずしも実行速度を大きく改善するわけではない．しかしながら，この最適化には，実装規模が小さくなりうるという利点があり，この利点が生きる状況では実装を検討する価値がある．実際，我々がこの最適化向けに固有に開発した処理は 200 行あまり[⋆1]にすぎない．実装の規模が小さくなる理由は，この最適化に必要な処理のほとんどが，他の最適化向けに実装済みでありうるからである．この最適化の実装の構成要素は，次に示す 4 つである．

(1) 可視範囲が `private` もしくは `final` であり，`volatile` でなく，コンストラクタ外で定義されないインスタンス変数を求める処理

(2) 脱出解析

(3) 冗長と分かったインスタンス変数参照の削除処理

⋆1 空行を含む．コメントは含まない．

(4) リフレクションおよびネイティブメソッドの利用状況を監視し，必要に応じて脱最適化を適用する処理

我々の実装では，(2) の脱出解析については，別の最適化（冗長な同期の除去や静的ごみ集め）向けに実装済みのものを，そのまま利用した．(1) の処理については，脱出解析におけるメソッド内解析の過程で実施するものとし，(3) の処理については，既存の共通部分式削除を使って実現した．(4) の処理についても，監視の機能は，実装の大部分を他の最適化と共有しており，脱最適化の機能も実装済みだった．

7. 結 論

Java 向け動的コンパイラによって，一定の値を保持するインスタンス変数への参照の繰返しを省くことで実行を高速化する技法を提案した．また，提案技法の実現にあたって問題になる動的ロードとネイティブメソッドへの対処方法を示した．さらに，最適化対象となるインスタンス変数の探索にあたって必要になる脱出解析について，解析コストと内部クラスの実装に配慮して解析範囲を設定する方法を提案した．SPECjvm98 を使って評価した結果，提案技法によれば，実行速度を相乗平均で 0.6%高速化できることが分かった．また，提案技法では，最適化対象のインスタンス変数を探すために，脱出解析を利用するが，この脱出解析にともなうコンパイル時間の増加率が 1〜23%になることが分かった．

参 考 文 献

1) Choi, J.-D., Gupta, M., Serrano, M.J., Sreedhar, V.C. and Midkiff, S.P.: Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis, *ACM Trans. Prog. Lang. Syst.*, Vol.25, No.6, pp.876–910 (2003).
2) Clausen, L.R.: A Java Bytecode Optimizer Using Side-effect Analysis, *Concurrency: Practice and Experience*, Vol.9, No.11, pp.1031–1045 (1997).
3) Ghemawat, S., Randall, K.H. and Scales, D.J.: Field Analysis: Getting Useful and Low-cost Interprocedural Information, *Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp.334–344 (2000).
4) Gosling, J., Joy, B., Steele Jr., G.L. and Bracha, G.: *The Java Language Specification*, 3rd Edition, Addison-Wesley, Reading, Mass. (2005).
5) Hölzle, U., Chambers, C. and Ungar, D.: Debugging Optimized Code with Dynamic Deoptimization, *Proc. ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pp.32–43 (1992).
6) Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H. and Nakatani, T.: A Study of Devirtualization Techniques for a Java Just-In-Time Compiler, *Proc. 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp.294–310 (2000).
7) Milanova, A., Rountev, A. and Ryder, B.G.: Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java, *Proc. 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp.1–11 (2002).
8) Nguyen, P.H. and Xue, J.: Interprocedural Side-Effect Analysis and Optimization in the Presence of Dynamic Loading, *Proc. 28th Australasian Conference on Computer Science*, Vol.38, pp.9–18 (2005).
9) Park, Y.G. and Goldberg, B.: Escape Analysis on Lists, *Proc. ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pp.116–127 (1992).
10) Rountev, A.: Precise Identification of Side-Effect-Free Methods in Java, *Proc. 20th IEEE International Conference on Software Maintenance*, pp.82–91 (2004).
11) Salcianu, A. and Rinard, M.: Purity and Side Effect Analysis for Java Programs, *Proc. 6th International Conference on Verification, Model Checking and Abstract Interpretation*, pp.119–215 (2005).
12) Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks (1998). http://www.spec.org/jvm98/
13) Standard Performance Evaluation Corporation: SPECjbb2005 (2005). http://www.spec.org/jbb2005/
14) 今城哲二，布広永示，岩澤京子，千葉雄司：コンパイラとバーチャルマシン，オーム社 (2004).

(平成 19 年 7 月 9 日受付)
(平成 19 年 10 月 9 日採録)

千葉 雄司（正会員）

1972 年生．1997 年慶應義塾大学大学院理工学研究科計算機科学専攻修士課程修了．同年（株）日立製作所入社．