Java向け動的コンパイラによる冗長なBox化の削除

千葉雄司†

Auto-Boxingを利用したプログラミングでは，primitive型からwrapper型への変換を暗黙に実施するため，ソースコード上に変換の処理を記述する必要はなくなるが，変換の処理自体がなくなるわけではない．変換の処理は冗長である場合もあり，このとき最適化によって変換処理を削除することで実行を高速化できる．本論文ではJavaのAuto-Boxingを利用したプログラムを対象とした最適化として，Auto-Boxingで暗黙に挿入される変換処理が冗長か判断し，冗長ならば除去する手法を提案する．提案技法の特長は，変換処理を冗長にするために，null検査の除去や冗長な比較の除去，メンバ変数参照の除去，インタプリタによる実行再開地点の移動といった技法を用いる点にある．SPECjbb2005を使った評価の結果，提案技法によって3.6%の高速化が可能であることが分かった．

Eliminating Redundant Boxing by a Dynamic Compiler for Java

Yuji Chiba†

Programming using auto-boxing makes it unnecessary to write code that converts a primitive value into an instance of the wrapper class because auto-boxing inserts the code implicitly. The implicit code is sometimes redundant, and we propose an optimization that eliminates the redundant code to improve performance. Our optimization also makes the implicit code redundant by null test elimination, redundant reference comparision elimination, member variable reference elimination, and unwinding execution when an interpreter takes over the execution. Evaluation using SPECjbb2005 showed that our optimization improved the performance by 3.6%.

1. はじめに

Java™☆言語はversion 5.0からAuto-Boxingという機能を提供している[5]．Auto-Boxingとはprimitive型からwrapper型への変換（たとえばint型の値からクラスjava.lang.Integerのインスタンスへの変換）を暗黙に実施する機能である．Auto-Boxingを利用したプログラミングでは，型変換に関する記述を省略し，ソースコードの可読性を改善することができる．

図1にAuto-Boxingを使って記述したソースコードの例を示す．図1の14行目ではメソッド`get()`を呼び出している．このメソッド`get()`は**図2**のインタフェース`Map`が宣言しているもので，引数としてクラス`Object`のインスタンスへの参照を受け取るが，図1の14行目で与えている引数はint型の値`i`である．このように異なる型の値を引数に与えることができるのは，Auto-Boxingを利用しているからである．Auto-Boxingでは型変換のコードを暗黙に挿入することで型の不整合を解消する．JavaにおけるAuto-Boxingの実現では，型変換のコードの挿入を，ソースコードからバイトコードへの変換の過程で実施する．Javaによるプログラムの開発では，開発したプログラムのソースコードを，`javac`などのツールを使ってバイトコード形式に変換したうえで配布するが，`javac`は，この変換の過程で，図1の14行目を，**図3**の処理に相当するバイトコード列に書き換える．書き換え前後の処理を比較すると，`javac`がメソッド呼び出し`Integer.valueOf()`を挿入したことが分かる．メソッド`valueOf()`は型変換を行う役割を果たすもので，その実装は**図4**の31〜36行目に示すとおりである．この`valueOf()`の実装では，頻繁に使用されると思われる整数値（−128〜127）について，実行ごとにクラス`Integer`のインスタンスを新規生成する手間を省くために，−128〜127の範囲の整数に対応するクラス`Integer`のインスタンスをあらかじめ作成，キャッシュしておき，実行時に受け取った引

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

☆ 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．

```
 1:  import java.util.Map;
 2:  import java.util.HashMap;
 3:
 4:  class MapStorage{
 5:    Map map;
 6:
 7:    MapStorage(){
 8:      map = new HashMap();
 9:    }
10:
11:    ···
12:
13:    Object get(int i){
14:      return (map.get(i));
15:    }
16:  }
```

図 1 Auto-Boxing を利用するソースコードの例
Fig. 1 Sample source code using auto-boxing.

```
 1:  package java.util;
 2:
 3:  public interface Map{
 4:    ···
 5:    Object get(Object key);
 6:  }
 7:
 8:  public class HashMap implements Map{
 9:    static class Entry{
10:      final Object key;
11:      final int hash;
12:      Object value;
13:      Entry next;
14:      ···
15:    }
16:    transient Entry[] table;
17:    static final Object NULL_KEY =
18:      new Object();
19:    ···
20:    public Object get(Object key){
21:      Object k =
22:        (key == null) ?  NULL_KEY : key;
23:      int hash = k.hashCode();
24:      Entry e =
25:        table[hash & (table.length-1)];
26:      while(true){
27:        if (e == null)
28:          return null;
29:        if ((e.hash == hash) &&
30:          ((k == e.key) || k.equals(e.key)))
31:          return e.value;
32:        e = e.next;
33:      }
34:    }
35:  }
```

図 2 インタフェース Map，クラス HashMap の実装
Fig. 2 Implementation of interface Map and class HashMap.

数 i が −128〜127 の範囲に収まる場合には，キャッシュしたインスタンスへの参照を返戻する．

Auto-Boxing を使ったプログラミングでは，ソースコード上に型変換の処理を記述する必要はなくなるが，型変換の処理自体がなくなるわけではない．したがって実行時には型変換のオーバヘッドが生じるが，プログラムによっては，型変換の処理が冗長な場合もあり，実際，図 1 の 14 行目の処理では 3 章で述べる理由から冗長になりうる．そこで本論文では，Java 向け動的コンパイラによって，Auto-Boxing における型変換が冗長か判断し，冗長ならば除去する最適化を提案する．本論文の 2 章では関連研究を示し，3 章では最適化の提案を行い，4 章でその代案を示す．5 章では，SPECjbb2005 [18] を用いて，提案した最適化の効果を評価する．6 章は結論である．

```
1:  Integer tmp = Integer.valueOf(i);
2:  return (map.get(tmp));
```

図 3 javac による valueOf() の挿入
Fig. 3 Insertion of valueOf() by javac.

```
 1:  package java.lang;
 2:
 3:  public class Integer{
 4:    private final int value;
 5:
 6:    public Integer(int v){
 7:      this.value = v;
 8:    }
 9:
10:    public int hashCode(){
11:      return value;
12:    }
13:
14:    public boolean equals(Object obj){
15:      if (obj instanceof Integer)
16:        return value==((Integer)obj).value;
17:      return false;
18:    }
19:
20:    ···
21:
22:    private static class IntegerCache {
23:      static final Integer cache[];
24:      static {
25:        cache = new Integer[256];
26:        for(int i = 0; i < 256; i++)
27:          cache[i] = new Integer(i - 128);
28:      }
29:    }
30:
31:    public static Integer valueOf(int i) {
32:      if (i >= -128 && i < 128) {
33:        return IntegerCache.cache[i + 128];
34:      }
35:      return new Integer(i);
36:    }
37:  }
```

図 4 クラス Integer の実装
Fig. 4 An implementation of class Integer.

2. 関連研究

Java における冗長なインスタンス生成を回避する

最適化は，これまで，次の 2 種類のインスタンスを対象に行われてきた．

(1) **例外**：制御を例外ハンドラに移すためだけに使われる例外の生成を抑止する[4),12)]．

(2) **メンバ変数のみ参照されるインスタンス**：インスタンス生成文の返戻する参照の使用点をフロー解析によって求め，そのすべてがメンバ変数参照である場合に，メンバ変数の値を保持する領域をレジスタや，実行時スタックに確保する一方で，インスタンス生成文を削除し，インスタンス生成にともなうヒープ割付けやヘッダの初期化を省略する[7),8)]．

本論文における提案手法も，冗長なインスタンス生成を削除する最適化の一種だが，対象とするインスタンスが例外でない．提案する最適化は，インスタンスへの参照の使用点がなくなった場合に適用できる点で，メンバ変数のみ参照されるインスタンス向けの最適化に類似するが，提案する最適化では，参照の使用点を除去するために，Auto-Boxing 向けに固有な方法で，null 検査の除去や参照比較の除去を実施する．

データの表現を最適化し，冗長な Box 化やタグ付けを回避する技法は関数型プログラミング言語などの実現において古くから提案されている[2),9),11),14),15),19),20)]．本論文における提案手法と，これら過去の手法の相違点は，本論文の提案手法において，インタプリタへの実行の引継ぎなど，動的最適化に固有な事情を考慮している点にある．

3. 実　　装

本論文で提案する最適化では，次の手順によって型変換の処理を除去する．

(1) Auto-Boxing において挿入する型変換の処理（図 3 のメソッド呼び出し `valueOf()`）の返戻値の使用点を除去するために，次に示す最適化を適用する．
- null 検査の除去
- 冗長な比較の除去
- メンバ変数参照の除去
- インタプリタによる実行再開地点の移動

(2) すべての使用点を除去できたら，型変換の処理を冗長と見なして削除する．

最適化の具体例を示すために，図 3 の 1 行目に挿入したメソッド呼び出し `valueOf()` を除去するまでの手順について述べる．図 3 のコードでは，2 行目で `valueOf()` の返戻値 `tmp` を引数としてメソッド `get()` に与える，つまり `tmp` の使用点があるので，このままでは `valueOf()` を除去できないことが分かる．そこで，2 行目のメソッド呼び出し `get()` にインライン展開を適用する．メソッド呼び出し `get()` の呼び出し先は，レシーバ `map` のクラスが不定であることから不定だが，実行時プロファイルから `map` のクラスが `HashMap` になる確率が非常に高いと分かれば，メソッド呼び出し `map.get(tmp)` を**図 5** のガードつきコール[3)]に変換し，ガードつきコール内にある直接呼び出し `HashMap::get()` にインライン展開を適用できる．

```
1:  if (map!=NULL && map->klass==HashMap)
2:    HashMap::get(map, tmp);
3:  else{
4:    インタプリタへ実行引継:
5:      図 3 の 2 行目から実行再開
6:  }
```

図 5 ガードつきコール
Fig. 5 A guarded call.

図 5 のコードは，Java 向け動的コンパイラが出力するガードつきコールの一例を，C++ 風に記述したものである．Java の実行環境の中には，バイトコード形式のプログラムを実行する手段として，インタプリタと動的コンパイラを併用するものがあるが，その中には，インタプリタによるメソッドの実行を，途中から動的コンパイル済みコードに引き継いだり，逆に，動的コンパイル済みコードからインタプリタに引き継いだりできるものもある．図 5 のコードでは，メンバ変数 `map` の値が `NULL` である，あるいは `map` が参照するインスタンスのクラスが `HashMap` でないといった例外的な事態に陥った際には，実行をインタプリタに引き継ぐことにしており（4〜5 行目），たとえば `map` の値が `NULL` だからといって，自身で `NullPointerException` を生成して投げたりはしない．

図 3 のコードに対し，2 行目のメソッド呼び出し `get()` を図 5 のガードつきコールに変換し，さらにガードつきコール内にある直接呼び出し `HashMap::get()` にインライン展開を適用した結果を**図 6** に示す．インライン展開対象のメソッド `HashMap::get()` の内容は図 2 の 20〜34 行目に示すとおりである．インライン展開対象のコード（図 2 の 20〜34 行目）は Java で記述したものであるのに対し，インライン展開後のコード（図 6）は C++ 風に記述したものであり，両者を比較すると，インライン展開後のコードに次の処理が明示されていることが分かるが，これは Java における暗黙の処理を明示したものである．

null 検査　図 6 の 6〜9 行目にある null 検査は，図 2 の 25 行目における配列長 `table.length` の取得

```
 1:  Integer tmp = Integer::valueOf(i);
 2:  if(map!=NULL && map->klass==HashMap){
 3:    Object k =
 4:      (tmp == NULL) ? NULL_KEY : tmp;
 5:    int hash = k->hashCode();
 6:    if (map->table == NULL){
 7:      インタプリタへ実行引継:
 8:        図 2 の 25 行目から実行再開
 9:    }
10:    unsigned index = (unsigned)
11:      (hash & (map->table->length-1));
12:    if (index >= map->table->length){
13:      インタプリタへ実行引継:
14:        図 2 の 25 行目から実行再開
15:    }
16:    Entry e = map->table[index];
17:    while(true){
18:      if (e == NULL)
19:        return NULL;
20:      if ((e->hash == hash) &&
21:          ((k == e->key) ||
22:           (k->equals(e->key))))
23:        return e->value;
24:      e = e->next;
25:      安全点:
26:        脱最適化時にインタプリタへ実行引継
27:    }
28:  }
29:  else{
30:    インタプリタへ実行引継:
31:      図 3 の 2 行目から実行再開
32:  }
```

図 6 HashMap::get() のインライン展開済コード
Fig. 6 Code after inlining HashMap::get().

にともなう null 検査を明示化したものである．

配列添字検査 図 6 の 12〜15 行目にある配列添字検査は，図 2 の 25 行目における配列参照にともなう配列添字検査を明示化したものである．

安全点 図 6 の 25〜26 行目にある安全点は，ごみ集めや脱最適化[6),21)] などの際にスレッドを停止させる位置を明示化したものである．脱最適化とは，コンパイラによって適用した最適化を解除することを意味する．脱最適化の具体的な実現手段には，最適化済みコードを上書きする方法や，最適化済みコードによる実行をやめ，インタプリタなど，別の実行手段を使う方法がある．図 6 のコードでは，脱最適化の際に，安全点からインタプリタに実行を引き継ぐことにしており，たとえば図 6 の 25〜26 行目にある安全点でスレッドが停止した際に脱最適化がおきると，インタプリタで図 2 の 26 行目から実行を再開する．安全点はループのバックエッジやメソッド呼び出しからの返戻地点などに挿入される．図 6 では簡単のため，ループのバックエッジに挿入した安全点のみ明示した．

3.1 null 検査の除去

図 6 のコードには，4 行目に valueOf() の返戻値 tmp の使用点があり，したがって，まだ valueOf() の除去を適用できない．そこで最適化によって使用点を削除する．4 行目には tmp の使用点が 2 つあり，その一方は tmp に対して null 検査を適用しているが，これは冗長である．なぜならメソッド valueOf() の実装は図 4 の 31〜36 行目に示すとおりであり，その 33 行目ではクラス変数 cache が指示する配列に収められた参照を，35 行目では新規生成したインスタンスへの参照を返戻するが，新規生成したインスタンスへの参照が null になることはありえず，クラス変数 cache が指示する配列の内容も，22〜29 行目における配列の初期化処理が例外をおこすことなく完了していれば，null になりえないからである☆．リフレクションやネイティブメソッドによる配列の破壊は発生しうるが，そのようなケースが頻発するとは考えにくい．そこで本論文では valueOf() の返戻値に対する null 検査に特化した次の最適化技法を提案する．

- クラス変数 cache の指示する配列の全要素が非 null になっていることを，動的コンパイラによって確認する．
- 確認がとれたならば，valueOf() の返戻値に対する null 検査を除去する．

この最適化を適用した動的コンパイル済みコードには，リフレクションやネイティブメソッドによって，クラス変数 cache や，その指示先にある配列の内容が破壊される恐れが生じたら☆☆，脱最適化を適用する．

この最適化によって図 6 の 4 行目にある null 検査を除去すると，3〜4 行目の処理は変数 tmp を変数 k に複写するだけの文になり，コピー伝播[1)] によって変数 k を除去可能になる．コピー伝播まで適用した後のコードを**図 7** に示す．

3.2 冗長な比較の除去

図 7 のコードには，3 行目と 19 行目，20 行目に valueOf() の返戻値 tmp の使用点があり，したがって，まだ valueOf() の除去を適用できない．そこで最適化によって使用点を削除する．

3 行目および 20 行目の使用点では tmp がメソッド呼び出しのレシーバになっているが，ここで tmp のクラスが Integer であることから，それぞれの呼び出

☆ 図 4 の省略部分（20 行目）には，クラス変数 cache が指示する配列の内容を更新する処理が含まれないものとする．

☆☆ Java アプリケーションがリフレクションを通じてクラス IntegerCache への参照を取得した場合など．

```
 1:  Integer tmp = Integer::valueOf(i);
 2:  if(map!=NULL && map->klass==HashMap){
 3:    int hash = tmp->hashCode();
 4:    if (map->table == NULL){
 5:      インタプリタへ実行引継:
 6:        図 2 の 25 行目から実行再開
 7:    }
 8:    unsigned index = (unsigned)
 9:      (hash & (map->table->length-1));
10:    if (index >= map->table->length){
11:      インタプリタへ実行引継:
12:        図 2 の 25 行目から実行再開
13:    }
14:    Entry e = map->table[index];
15:    while(true){
16:      if (e == NULL)
17:        return NULL;
18:      if ((e->hash == hash) &&
19:          ((tmp == e->key) ||
20:           (tmp->equals(e->key))))
21:        return e->value;
22:      e = e->next;
23:      安全点:
24:        脱最適化時にインタプリタへ実行引継
25:    }
26:  }
27:  else{
28:    インタプリタへ実行引継:
29:      図 3 の 2 行目から実行再開
30:  }
```

図 7 null 検査の除去後のコード
Fig. 7 Code after null test elimination.

```
 1:  Integer tmp = Integer::valueOf(i);
 2:  if(map!=NULL && map->klass==HashMap){
 3:    int hash = tmp->value;
 4:    if (map->table == NULL){
 5:      インタプリタへ実行引継:
 6:        図 2 の 25 行目から実行再開
 7:    }
 8:    unsigned index = (unsigned)
 9:      (hash & (map->table->length-1));
10:    if (index >= map->table->length){
11:      インタプリタへ実行引継:
12:        図 2 の 25 行目から実行再開
13:    }
14:    Entry e = map->table[index];
15:    while(true){
16:      if (e == NULL)
17:        return NULL;
18:      if ((e->hash == hash) &&
19:          ((e->key instanceof Integer) &&
20:           (tmp->value==e->key->value)))
21:        return e->value;
22:      e = e->next;
23:      安全点:
24:        脱最適化時にインタプリタへ実行引継
25:    }
26:  }
27:  else{
28:    インタプリタへ実行引継:
29:      図 3 の 2 行目から実行再開
30:  }
```

図 8 冗長な比較の除去後のコード
Fig. 8 Code after redundant comparison elimination.

し先はクラス Integer の定める hashCode() および equals()（図 4 の 10〜12 および 14〜18 行目）に定まり，それぞれにインライン展開を適用可能になる．

19 行目の使用点では 2 つの参照 tmp と e->key を比較しているが，この比較は冗長である．なぜなら後続する 20 行目でメソッド equals() によって tmp と e->key の比較を行っているので，19 行目の比較がなくても同じ計算結果が得られるからである．19 行目で参照による比較を行うと，2 つの参照 tmp と e->key が等しいときに 20 行目のメソッド呼び出しを省略し，実行を高速化できるが，動的プロファイルから 19 行目の比較結果が真になる確率が十分低いと判明した場合には，19 行目の処理はむしろ性能劣化の原因となっているので除去した方がよい．なお，メソッド equals() の実行結果が参照による比較の代替になるか否かはレシーバのクラスに依存するが，ここではレシーバ tmp のクラスが Integer であり，クラス Integer が定義する equals() は 2 つの参照が等しいとき必ず真を返戻するので，この場合は代替になる．

ここでは動的プロファイルによって 19 行目の比較結果が真になる確率が十分に低いことが分かったものと仮定し，19 行目の比較を除去する．また，3 行目と 20 行目のメソッド呼び出しにインライン展開を適用する．適用後のコードを**図 8** に示す．

3.3 メンバ変数参照の除去

図 8 のコードには，3 行目と 20 行目に valueOf() の返戻値 tmp の使用点があり，したがって，まだ valueOf() の除去を適用できない．そこで，最適化によって，これらの使用点を除去する．

3 行目と 20 行目では tmp が指示するインスタンスのメンバ変数 value を参照しているが，その結果として得られる値は，valueOf() が受け取る引数 i と同じはずである．そこで 3 行目と 20 行目にあるメンバ変数参照 tmp->value を i で差し替える．

もちろん，valueOf() の定義（図 4 の 31〜36 行目）に従えば，valueOf() が受け取る仮引数 i が -128〜127 の範囲にあるとき，valueOf() が対応する Integer のインスタンスへの参照を返戻するためには，クラス変数 cache が参照する配列に，整数 -128〜127 に対応するクラス Integer のインスタンスへの参照が順に入っている必要がある．そこで，図 8 の 3 行目と 20 行目にあるメンバ変数参照 tmp->value の，

i による差替えを，次の手順により実施する．

(1) クラス変数 cache の指示する配列に，整数 -128〜127 に対応するクラス Integer のインスタンスへの参照が順に入っているか否かを，動的コンパイラによって確認する．

(2) 確認がとれたならば，valueOf() の返戻値のメンバ変数 value への参照を，valueOf() が受け取る実引数への参照で差し替える．

この最適化を適用した動的コンパイル済みコードには，リフレクションやネイティブメソッドによって，クラス変数 cache や，その指示先にある配列の内容が破壊される恐れが生じたら，脱最適化を適用する．

3.4 インタプリタによる実行再開地点の移動

メンバ変数参照の除去によって，図 8 のコードから，3 行目と 20 行目にある tmp の使用点を除去すると，使用点の除去が完了したように見えるが，まだすべての使用点がなくなったわけではない．なぜなら，図 8 のコードでは 5，11，23，28 行目からインタプリタに実行を引き継ぐことがあるが，引継ぎにあたってインタプリタ向けスタックフレームを構築する際に，tmp の値を参照するので，これらインタプリタへの実行引継処理が実質的に tmp の使用点となっているからである．

インタプリタへの実行引継処理を tmp の使用点でなくす方法は，次に示すとおり，いくつかある．

- valueOf() の実行を遅延する．具体的には，インタプリタへ実際に実行を引き継ぐ時点になってから valueOf() を実行する．ただし，後から valueOf() を実行する場合には，実行中にヒープメモリが不足すると，Java 仮想機械の仕様とは異なるタイミングで OutOfMemoryError が発生し，問題となる．この問題の発生を防ぐためには，後から valueOf() を実行する際に必要になる記憶領域をスタックフレームに確保しておけばよい．
- インタプリタによる実行再開地点を移動する．図 8 の 5，11，23，28 行目にある引継処理で tmp への参照が必要になる理由は，インタプリタによる実行再開地点が，valueOf() の実行後にあり，なおかつ，実行再開後に valueOf() の返戻値を参照しうるからである．もし，インタプリタによる実行再開地点を，これらの条件を満たさない箇所に移動できれば，インタプリタへの実行引継処理は使用点でなくなる．

  たとえば valueOf() の実行終了後から，その返戻値の実質的な使用点となっている，インタプリタへの引継処理に至るすべての制御パスをたどり，たどった制御パスに，実行引継ぎ後に参照するメモリ領域へ副作用をおよぼす処理が含まれていなければ，インタプリタによる実行再開地点を，valueOf() の実行前まで移動できる☆．移動を行うと，インタプリタに実行を引き継ぐ際には，valueOf() から計算をやりなおすことになるが，引継ぎまでの過程で副作用が生じていなければ，再度計算しても結果がおかしくなることはない．図 8 のコードについては，実行再開地点を移動によって，5，11，23，28 行目にある引継処理を tmp の使用点でなくすことができる．

これら 2 つの方法を比較すると，後者の方が適用範囲が狭いことが分かる．なぜなら後者は valueOf() からインタプリタへの実行引継地点までの間に副作用が生じうる場合に適用できないからである．しかしながら，我々が HotSpot™VM 向けに実装したのは後者である．なぜなら，HotSpotVM 向け実装では後者の方が実装に必要な作業量が小さかったからである．前者を実装するには実行時スタック上に（クラス Integer の）インスタンスを配置可能にする必要があるが，HotSpotVM のごみ集めはすべてのインスタンスがヒープ上にあることを前提とした作りになっており，インスタンスを実行時スタック上に配置可能にするためには大きな作業量が必要になると判断した．

図 8 のコードにメンバ変数参照の除去と，インタプリタによる実行再開地点の移動を適用すると，tmp の使用点がなくなり，valueOf() の除去が可能になる．valueOf() を除去したコードを**図 9** に示す．

4. 他の最適化手段

3 章では冗長な Box 化を除去する最適化を提案したが，本章では，既存の最適化を使って同等の効果を得る方法がないか考察する．もし既存の最適化によって同等の効果を得られるのであれば，本論文で提案した最適化を新規に実装する必要はなくなる．結論から述べると，本論文で提案した最適化技法のうち，valueOf() の返戻値に特化した null 検査の除去およびメンバ変数参照の除去については，部分冗長性の除去[10] という既存の最適化によって，ある程度代用できるが，その他のものについては，既存の最適化では

☆ 厳密には，副作用をおよぼす処理が含まれる場合でも，引継ぎ時に副作用を解除できるなら移動できる．たとえば引継ぎ後に参照しうる局所変数に対する副作用に関しては，valueOf() を実行する前の時点における局所変数の値を保存しておけば，引継ぎ時に値を回復できる．

```
 1:  if(map!=NULL && map->klass==HashMap){
 2:    int hash = i;
 3:    if (map->table == NULL){
 4:      インタプリタへ実行引継:
 5:      図 3 の 1 行目から実行再開
 6:    }
 7:    unsigned index = (unsigned)
 8:      (hash & (map->table->length-1));
 9:    if (index >= map->table->length){
10:      インタプリタへ実行引継:
11:      図 3 の 1 行目から実行再開
12:    }
13:    Entry e = map->table[index];
14:    while(true){
15:      if (e == NULL)
16:        return NULL;
17:      if ((e->hash == hash) &&
18:          ((e->key instanceof Integer) &&
19:           (i==e->key->value)))
20:        return e->value;
21:      e = e->next;
22:      安全点:
23:        脱最適化時にインタプリタへ実行引継
24:    }
25:  }
26:  else{
27:    インタプリタへ実行引継:
28:      図 3 の 1 行目から実行再開
29:  }
```

図 9 valueOf() の除去後のコード
Fig. 9 Code after eliminating valueOf().

必ずしも代用できない．

本章でも 3 章と同様に，図 3 の 1 行目に挿入したメソッド呼び出し valueOf() によるオーバヘッドを軽減させる方法を考える．まず，図 3 の 1 行目にあるメソッド呼び出し valueOf() にインライン展開を適用すると，**図 10** (a) のコードになる．図 10 (a) のコードにおいて，Box 化にともなうオーバヘッドの主な発生源は，6 行目にあるクラス Integer のインスタンス生成なので，その削除に向けて最適化を進める．インスタンス生成を削除するには，まず，インスタンス生成が生成したインスタンスへの参照を使用する処理を，すべて除去する必要がある．図 10 (a) の 6 行目にあるインスタンス生成を削除するには，6 行目のインスタンス生成が生成したインスタンスへの参照 tmp を使用する処理を削除する必要が生じるが，削除の対象となる処理は 8 行目のメソッド呼び出し get() の中にある．そこで最適化を進めるためには，まず，get() にインライン展開を適用する必要が生じるが，get() を 8 行目においたままインライン展開するのは得策でない．なぜなら 6 行目のインスタンス生成から得られる最適化に有用な情報が，8 行目には届かないからである．6 行目のインスタンス生成については，実行中

```
1:  Integer tmp;
2:  if (i >= -128 && i < 128) {
3:    tmp = IntegerCache.cache[i + 128];
4:  }
5:  else{
6:    tmp = new Integer(i);
7:  }
8:  return (map.get(tmp));
```

(a) インライン展開の適用

```
1:  Integer tmp;
2:  if (i >= -128 && i < 128) {
3:    tmp = IntegerCache.cache[i + 128];
4:    return (map.get(tmp));
5:  }
6:  else{
7:    tmp = new Integer(i);
8:    return (map.get(tmp));
9:  }
```

(b) 部分冗長性除去の適用

図 10 部分冗長性除去
Fig. 10 Partial redundancy elimination.

に例外が発生しなければ，実行終了時点で tmp に整数 i をラップしたクラス Integer のインスタンスへの参照を代入し，その時点で tmp の値は null でなくなるといえる．これらの情報は null 検査の除去といった最適化に必要だが，8 行目には届かない．なぜなら 8 行目の tmp を定義する処理は 6 行目のインスタンス生成のほかに，3 行目の配列参照もあるが，配列参照の結果は一般には不定であり，したがって 8 行目の tmp について，たとえば必ず非 null になるとはいえないからである．

4.1 部分冗長性の除去

この問題を解決する手段の 1 つは，本論文の 3.1 節で提案した方法☆を使うことだが，ここでは既存の技法である部分冗長性除去を使うことにする．部分冗長性の除去とは，図 10 (a) のように，制御の合流から最適化の機会が失われることを防ぐために，コードを複製する技法である．図 10 (a) に部分冗長性の除去を適用し，8 行目のメソッド呼び出し get() を複製して直前の if 節および else 節の末尾に移動した結果を図 10 (b) に示す．図 10 (b) のコードでは，8 行目にあるメソッド呼び出し get() が受け取る引数 tmp の定義点が 7 行目のインスタンス生成のみになっていることから，get() 内にある tmp を参照する処理の最適化に際して，インスタンス生成から得られる情報を活用可能になる．

☆ メソッド Integer.valueOf() が引数をラップしたクラス Integer のインスタンスへの参照を返戻すると仮定してコンパイルする方法．

```
 1:  Integer tmp;
 2:  if (i >= -128 && i < 128) {
 3:    tmp = IntegerCache.cache[i + 128];
 4:    return (map.get(tmp));
 5:  }
 6:  else{
 7:    tmp = new Integer(i);
 8:    if(map!=NULL && map->klass==HashMap){
 9:      Object k =
10:        (tmp == NULL) ? NULL_KEY : tmp;
11:      int hash = k->hashCode();
12:      if (map->table == NULL){
13:        インタプリタへ実行引継:
14:          図 2 の 25 行目から実行再開
15:      }
16:      unsigned index = (unsigned)
17:        (hash & (map->table->length-1));
18:      if (index >= map->table->length){
19:        インタプリタへ実行引継:
20:          図 2 の 25 行目から実行再開
21:      }
22:      Entry e = map->table[index];
23:      while(true){
24:        if (e == NULL)
25:          return NULL;
26:      if ((e->hash == hash) &&
27:            ((k == e->key) ||
28:             (k->equals(e->key))))
29:          return e->value;
30:        e = e->next;
31:        安全点:
32:          脱最適化時にインタプリタへ実行引継
33:      }
34:    }
35:    else{
36:      インタプリタへ実行引継:
37:        図 3 の 2 行目から実行再開
38:    }
39:  }
```

図 11 map.get(tmp) のインライン展開後のコード
Fig. 11 Code after inlining map.get(tmp).

4.2 null 検査の除去

最適化を進めるために，図 10 (b) の 8 行目にあるメソッド呼び出し map.get(tmp) を図 5 のガードつきコールに変換し，さらに，ガードつきコール内の HashMap::get() にインライン展開を適用した結果を**図 11** に示す．図 11 のコードをみると，7 行目で定義した tmp を 10 行目で使用していることが分かるが，ここで 10 行目の使用のうち，null 検査については冗長と判断して除去できる．なぜなら 7 行目で定義する tmp の値が null になりえないからである．なお，ここで null 検査を除去できるのは，部分冗長性の除去を適用したからである．ここで null 検査を除去すると，さらなる最適化として，変数 k にコピー伝播を適用可能になる．コピー伝播適用後のコードを**図 12** に示す．

```
 1:  Integer tmp;
 2:  if (i >= -128 && i < 128) {
 3:    tmp = IntegerCache.cache[i + 128];
 4:    return (map.get(tmp));
 5:  }
 6:  else{
 7:    tmp = new Integer(i);
 8:    if(map!=NULL && map->klass==HashMap){
 9:      int hash = tmp->hashCode();
10:      if (map->table == NULL){
11:        インタプリタへ実行引継:
12:          図 2 の 25 行目から実行再開
13:      }
14:      unsigned index = (unsigned)
15:        (hash & (map->table->length-1));
16:      if (index >= map->table->length){
17:        インタプリタへ実行引継:
18:          図 2 の 25 行目から実行再開
19:      }
20:      Entry e = map->table[index];
21:      while(true){
22:        if (e == NULL)
23:          return NULL;
24:      if ((e->hash == hash) &&
25:            ((tmp == e->key) ||
26:             (tmp->equals(e->key))))
27:          return e->value;
28:        e = e->next;
29:        安全点:
30:          脱最適化時にインタプリタへ実行引継
31:      }
32:    }
33:    else{
34:      インタプリタへ実行引継:
35:        図 3 の 2 行目から実行再開
36:    }
37:  }
```

図 12 コピー伝播適用後のコード
Fig. 12 Code after copy propagation.

4.3 メンバ変数参照の削除

図 12 を参照すると，7 行目における tmp の定義を明示的に使用する箇所が，9 行目と 25，26 行目にあることが分かる．ここで 9 行目の tmp->hashCode() については，インライン展開を適用すると tmp->value に変換できるが，ここで tmp->value はさらに i に置き換えることができる．なぜなら部分冗長性を除去した結果，tmp は 7 行目で生成する i をラップしたクラス Integer のインスタンスを参照することがあきらかになっているからである．同様の最適化は，26 行目にあるメソッド呼び出し equals() にインライン展開を適用した結果として現れる tmp->value にも適用できる．これらの最適化を適用し，さらに 9 行目で定義する変数 hash にコピー伝播を適用した結果を**図 13** に示す．

```
 1:  Integer tmp;
 2:  if (i >= -128 && i < 128) {
 3:    tmp = IntegerCache.cache[i + 128];
 4:    return (map.get(tmp));
 5:  }
 6:  else{
 7:    tmp = new Integer(i);
 8:    if(map!=NULL && map->klass==HashMap){
 9:      if (map->table == NULL){
10:        インタプリタへ実行引継:
11:          図 2 の 25 行目から実行再開
12:      }
13:      unsigned index = (unsigned)
14:        (i & (map->table->length-1));
15:      if (index >= map->table->length){
16:        インタプリタへ実行引継:
17:          図 2 の 25 行目から実行再開
18:      }
19:      Entry e = map->table[index];
20:      while(true){
21:        if (e == NULL)
22:          return NULL;
23:      if ((e->hash == i) &&
24:            ((tmp == e->key) ||
25:              ((e->key instanceof Integer)
26:                && (i == e->key->value))))
27:          return e->value;
28:        e = e->next;
29:        安全点:
30:          脱最適化時にインタプリタへ実行引継
31:      }
32:    }
33:    else{
34:      インタプリタへ実行引継:
35:        図 3 の 2 行目から実行再開
36:    }
37:  }
```

図 13 メンバ変数参照除去後のコード
Fig. 13 Code after member variable reference elimination.

4.4 インスタンス生成の削除

図 13 のコードを参照すると，7 行目のインスタンス生成で定義する tmp が，まだ次の箇所で使われていることが分かる．

- 参照の比較 tmp == e->key（24 行目）
- インタプリタへの実行引継ぎ（10，16，29，34 行目）

インスタンス生成を削除するには，これらの使用箇所を除去しなければならない．これらを除去する方法で，3 章での提案とは異なる手段としては，次の方法が考えられる．

新旧参照比較の除去 24 行目にある参照の比較は冗長と見なして除去できる．なぜなら比較対象の一方が新規に割り付けたインスタンスを参照するのに対し，もう一方が割付け前から存在するインスタンスを参照するので，比較結果は必ず偽といえるからである．

```
 1:  Integer tmp;
 2:  if (i >= -128 && i < 128) {
 3:    tmp = IntegerCache.cache[i + 128];
 4:    return (map.get(tmp));
 5:  }
 6:  else{
 7:    if(map!=NULL && map->klass==HashMap){
 8:      if (map->table == NULL){
 9:        コンパイル済みコードへ実行引継:
10:          図 2 の 25 行目から実行再開
11:      }
12:      unsigned index = (unsigned)
13:        (i & (map->table->length-1));
14:      if (index >= map->table->length){
15:        コンパイル済みコードへ実行引継:
16:          図 2 の 25 行目から実行再開
17:      }
18:      Entry e = map->table[index];
19:      while(true){
20:        if (e == NULL)
21:          return NULL;
22:      if ((e->hash == i) &&
23:            ((e->key instanceof Integer)
24:              && (i == e->key->value)))
25:          return e->value;
26:        e = e->next;
27:        安全点:
28:          脱最適化時にコンパイル済みコードへ
29:      }
30:    }
31:    else{
32:      コンパイル済みコードへ実行引継:
33:        図 3 の 2 行目から実行再開
34:    }
35:  }
```

図 14 インスタンス生成削除後のコード
Fig. 14 Code after instance allocation elimination.

コンパイル済みコードへの引継ぎ 実行の引継ぎ点を tmp の使用点でなくす手段として，引継ぎ先をコンパイル済みコードにする方法が考えられる．インタプリタへの実行の引継ぎに，tmp が必要になる理由は，バイトコード実行の際に tmp の値が必要になりうるからであり，もし，引継ぎ先が tmp を必要としないなら，引継ぎ点は tmp の使用点でなくなる．そこでバイトコードをコンパイルし，その際に最適化を適用して tmp を参照する処理を除去したうえで，生成したコードを実行の引継ぎ先とすれば，引継ぎ点を tmp の使用点でなくすことができる．

これらの最適化を適用して tmp の使用箇所を除去すれば，7 行目のインスタンス生成を削除可能になる．最適化を適用し，インスタンス生成を削除した結果として得られるコードを**図 14** に示す．

4.5 提案技法との比較

ここまでに述べたことから分かるように，本論文での提案とは異なる方法でも，Box 化にともなうオーバヘッドの主な発生源（インスタンス生成）を削除できる．ただし，本章で述べた最適化方法は，本論文の提案技法に比べ，次の点で劣る．

Box 化処理の除去能力 本章で述べた最適化では，Box 化の処理を完全には除去できない．本章で述べた最適化を適用して得られたコード（図 14）を参照すると，2〜3 行目に，Box 化の処理が一部残っていることが分かる．これに対し，本論文の提案技法は Box 化の処理を完全に除去できる．

コード量の増加 部分冗長性の除去を適用するとコード量が増えてしまう．また，本章で述べた最適化がかかるようにするには，部分冗長性の除去の適用先に，Box 化に関係する箇所を追加する必要が生じるが，無駄にコード量を増やさないよう追加するには相応の工夫が必要になる．これに対し，本論文の提案技法はコード量を増やさない．

引継処理の実装コスト 脱最適化などの際の実行の引継ぎ先をコンパイル済みコードにするには，引継ぎ先として利用可能なコードを生成するコンパイラが必要になるが，その実装にかかる手間は大きい．

5. 評　　価

提案した最適化技法を，HotSpot サーバコンパイラ[13] に実装し，その効果を評価した．実装対象とした HotSpot サーバコンパイラは，Sun Microsystems が配布している JDK（version 1.5.0_05）に付属のものを，日立製作所が独自に改良したもので，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．評価に使用したハードウェアは Hewlett-Packard 社製のワークステーション ZX6000（CPU：Itanium® ☆2 900 MHz×2，主記憶：2 GByte）であり，OS は CentOS 4.3 である．評価対象のソフトウェアは次の 2 つである．

マイクロベンチマーク 図 1 のメソッド `get()` を 100 万回呼び出すのにかかる時間を計測する．計測は動的コンパイルの処理が終了してから行う．

SPECjbb2005 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．

SPECjbb2005 の実行にあたっては，ヒープの初期サイズと最大サイズを 1.5 GByte に指定した．SPECjbb2005 は Auto-Boxing を使って作成されており，提案技法により，ほぼ図 3〜図 9 に示したとおりに最適化できる．評価の結果，提案技法により，マイクロベンチマークの実行性能を 17%，SPECjbb2005 の実行性能を 3.6%改善できることが分かった．

なお，本論文で提案した最適化は Auto-Boxing 向けだが，提案技法を応用すれば，ユーザアプリケーションによる明示的な Box 化も除去できる．Java 言語が Auto-Boxing を導入したのは version 5.0 からで，それ以前は Box 化用の API である `valueOf()` も提供していなかったので，ユーザアプリケーションは，それぞれ自前のコードによって Box 化を行ってきた．たとえば Java 向けベンチマークの SPECjbb2000[17] や `_202_jess`（SPECjvm98[16] というベンチマーク集の 1 構成項目）では，単純にクラス `Integer` のインスタンスを新規生成する処理によって Box 化を行っている．これらのうち，`_202_jess` における Box 化の中には，結果をハッシュ表の検索鍵としてのみ使う冗長なものがあり，提案技法を応用して除去したところ，実行性能を 3.5%改善することができた．

ただし，ユーザアプリケーションによる自前の Box 化の除去は必ずしも容易でない．なぜなら，ユーザアプリケーションごとに Box 化の実現が変化しうることから，Box 化の処理がどこにあるか，コンパイラを使って検知するのに手間がかかるからである．Box 化には様々な実現がありえ，たとえば `_202_jess` のように毎回必ずインスタンス生成を行う実現もあれば，Auto-Boxing 用 API の `Integer.valueOf()` のように，ラップする値に応じてインスタンス生成を省略する実現もありうる．あらゆる Box 化の実現を検知可能にするのは容易でないが，`_202_jess` における Box 化のように単純な実現に限って検知可能にするだけでも，ある程度の有用性を確保できる可能性はある．

6. 結　　論

Box 化にともなう実行時オーバヘッドの軽減を目的として，冗長な Box 化を削除する最適化技法を提案した．提案技法では冗長な Box 化を除去するために，Box 化処理の返戻値に特化した null 検査の除去やメンバ変数参照の除去を適用し，さらに，冗長な参照比較の除去やインタプリタによる実行再開地点の移動を行う．評価の結果，提案技法により，SPECjbb2005 の実行性能を 3.6%向上できることが分かった．

☆ 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登録商標です．

参 考 文 献

1) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers Principles, Techniques and Tools*, Addison-Wesley, Reading, Mass. (1986).
2) Brooks, R.A., Gabriel, R.P. and Steele Jr., G.L.: An Optimizing Compiler for Lexically Scoped LISP, *Proc. 1982 SIGPLAN Symposium on Compiler Construction*, pp.261–275 (1982).
3) Calder, B. and Grunwald, D.: Reducing Indirect Function Call Overhead in C++ Programs, *Proc. 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.397–408 (1994).
4) Cierniak, M., Lueh, G.-Y. and Stichnoth, J.M.: Practicing JUDO: Java Under Dynamic Optimizations, *Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp.13–26 (2000).
5) Gosling, J., Joy, B., Steele Jr., G.L. and Bracha, G.: *The Java Language Specification*, 3rd edition, Addison-Wesley, Reading, Mass. (2005).
6) 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).
7) Ishizaki, K., Takeuchi, M., Kawachiya, K., Suganuma, T., Gohda, O., Inagaki, T., Koseki, A., Ogata, K., Kawahito, M., Yasue, T., Ogasawara, T., Onodera, T., Komatsu, H. and Nakatani, T.: Effectiveness of Cross-Platform Optimizations for a Java Just-In-Time Compiler, *Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Language and Applications*, pp.187–204 (2003).
8) Kotzmann, T. and Mössenböck, H.: Escape Analysis in the Context of Dynamic Compilation and Deoptimization, *Proc. 1st ACM/USENIX International Conference on Virtual Execution Environments*, pp.111–120 (2005).
9) Leroy, X.: Unboxed Objects and Polymorphic Typing, *Proc. 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.177–188 (1992).
10) Morel, E. and Renvoise, C.: Global Optimization by Supression of Partial Redundancies, *Comm. ACM*, Vol.22, No.2, pp.96–103 (1979).
11) Morrison, R., Dearle, A., Connor, R.C.H. and Brown, A.L.: An Ad Hoc Approach to the Implementation of Polymorphism, *ACM Trans. Progr. Lang. Syst.*, Vol.13, No.3, pp.342–371 (1991).
12) Ogasawara, T., Komatsu, H. and Nakatani, T.: EDO: Exception-Directed Optimization in Java, *ACM Trans. Prog. Lang. Syst.*, Vol.28, No.1, pp.70–105 (2006).
13) Paleczny, M., Vick, C. and Click, C.: The Java HotSpot Server Compiler, *Proc. Java Virtual Machine Research and Technology symposium*, pp.1–12 (2001).
14) Peterson, J.: Untagged Data in Tagged Environments: Choosing Optimal Representations at Compile Time, *Proc. 4th International Conference on Functional Programming Language and Computer Architecture*, pp.89–99 (1989).
15) Peyton Jones, S.L. and Launchbury, J.: Unboxed Values as First Class Citizens in a Non-strict Functional Language, *Proc. 5th ACM Conference on Functional Programming Language and Computer Architecture*, pp.636–666 (1991).
16) Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks (1998). http://www.spec.org/jvm98/
17) Standard Performance Evaluation Corporation: SPEC JBB2000 (2000). http://www.spec.org/jbb2000/
18) Standard Performance Evaluation Corporation: SPEC jbb2005 (2005). http://www.spec.org/jbb2005/
19) Steele Jr., G.L.: Fast Arithmetic in MacLISP, *Proc. 1977 MACSYMA User's Conference*, pp.215–224 (1977).
20) Thiemann, P.J.: Unboxed Values and Polymorphic Typing Revisited, *Proc. 7th International Conference on Functional Programming Language and Computer Architecture*, pp.24–35 (1995).
21) 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 18 年 11 月 15 日受付)
(平成 19 年 3 月 5 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．システム開発研究所にてコンパイラの研究開発に従事．