情報処理学会論文誌　プログラミング　Vol. 1　No. 1　15–25 (June 2008)

Javaにおけるコンテナ向けキャストの除去

千　葉　雄　司[†1]

Java アプリケーションの実行性能を劣化させる要因の 1 つにキャストがある．キャストが必要となる主なケースの 1 つに，標準ライブラリが提供するクラスのうち，コンテナと呼ばれる，値を格納するためものの利用があるが，本論文では，コンテナの利用にともなうキャストを除去する最適化を提案する．この最適化では，Java アプリケーション中にあるコンテナの利用箇所を検出し，検出した箇所向けのキャストがあるならば，その除去を試みる．除去の処理は，検出箇所で利用しているコンテナのクラスを，標準ライブラリのものから，個々の検出箇所向けに最適化したものに差し替えることで実現する．キャストが必要になる原因は，標準ライブラリが提供するコンテナのクラスが，primitive 型の値を直接格納する機能や，格納する参照を特定のクラス型のものに制限する機能を提供しないことにある．したがって，個々の検出箇所向けに，これらの機能を提供するクラスを生成し，標準ライブラリのクラスの代わりに，生成したクラスを使えば，キャストを除去可能になる．最適化が実行速度に与える影響を，SPECjbb2005 を使って評価したところ，21.3% 高速化できることが分かった．

Cast Elimination for Containers in Java

Yuji Chiba[†1]

Java applications often require casting when they use classes in the standard library, such as containers, and this degrades performance. To cope with this problem, we are developping a bytecode optimizer that eliminates casting by replacing the use of a container class in the standard library with an optimized container class. While a conatiner class in the standard library can neither directly store primitive value nor narrow type of the stored reference and thus requires casting, the optimized class does both and does not require casting. The preliminary evaluation using SPECjbb2005 showed that the optimizer improves the performance by 21.3%.

1. はじめに

Java™ [⋆1]によるプログラミングでは，豊富な標準ライブラリを使って効率良くプログラムを開発できる．しかしながら，標準ライブラリは個々の利用元に固有な事情を考慮して開発されたものではないため，その利用によって実行効率が損われることもある．たとえば標準ライブラリのコンテナ（射影や可変長配列など，値を格納するもの）は，任意の参照を格納できるが，primitive 型の値を直接格納する機能や，格納する参照のクラス型を特定のものに制限する機能を提供しない．結果として，標準ライブラリのコンテナの利用時には，次のキャストが必要になる．

参照型の縮小変換　参照のクラス型を，クラス階層のより下位に位置するクラス型に変換する処理．標準ライブラリのコンテナから値を取り出すメソッドは，クラス型 `java.lang.Object`（`Object` と略記する）の参照を返戻するので，返戻値を別のクラス型の参照として扱う際に必要になる．

Box 化　primitive 型の値を，対応するラッパクラスのインスタンスに変換する処理．標準ライブラリのコンテナに primitive 型の値を格納する際に必要になる．

Unbox 化　Box 化とは反対に，ラッパクラスのインスタンスを，primitive 型の値に変換する処理．標準ライブラリのコンテナから primitive 型の値を取り出す際に必要になる．

これらのキャストのコードは，ソースコード上に明記されている場合もあれば，Java が提供する機能である Generics や Auto-boxing，Auto-unboxing によって暗黙に挿入される場合もあるが，いずれにせよ実行時オーバヘッドの発生源となる．キャストから生じる実行オーバヘッドは，必ずしも小さくなく，特に，コンテナを頻繁に使うアプリケーションでは顕著になる．

この問題を解決し，Java アプリケーションの実行を高速化することを目的として，本論文では，コンテナを利用する際のキャストを除去する最適化を提案する．提案する最適化では，Java アプリケーション中にあるコンテナの利用箇所を検出し，検出した箇所向けのキャストがあるならば，その除去を試みる．除去の処理は，検出箇所で利用しているコンテナのクラスを，標準ライブラリのものから，個々の検出箇所向けに最適化したものに差し替えることで実現する．

†1 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．

```
 1:  import java.util.ArrayList;
 2:  import java.util.Iterator;
 3:
 4:  class ResizableIntArray{
 5:    private ArrayList body;
 6:    ResizableIntArray(){
 7:      body = new ArrayList();
 8:    }
 9:    void add(int val) {
10:      // Box 化
11:      Integer iref = Integer.valueOf(val);
12:      body.add(iref);
13:    }
14:    int get(int index) {
15:      Object oref = body.get(index);
16:      // 参照型の縮小変換
17:      Integer iref = (Integer)oref;
18:      // Unbox 化
19:      int result = iref.intValue();
20:      return (result);
21:    }
22:    int sum(){
23:      Iterator itr = body.iterator();
24:      int result = 0;
25:      while(itr.hasNext()){
26:        Object oref = itr.next();
27:        // 参照型の縮小変換
28:        Integer iref = (Integer)oref;
29:        // Unbox 化
30:        int value = iref.intValue();
31:        result += value;
32:      }
33:      return (result);
34:    }
35:  }
```

(a) 最適化前

```
 1:  final class IntArray{
 2:    int[] body;
 3:    int size;
 4:    void add(int val){
 5:      ensureCapacity(size+1);
 6:      body[size++] = val;
 7:    }
 8:    int get(int index){
 9:      RangeCheck(index);
10:      return body[index];
11:    }
12:    public final class Itr{
13:      int cursor = 0;
14:      boolean hasNext(){
15:        return (cursor != size);
16:      }
17:      int next(){
18:        int result = get(cursor);
19:        cursor++;
20:        return (result);
21:      }
22:    }
23:    Itr iterator() {
24:      return (new Itr());
25:    }
26:                  ...
27:  }
28:
29:  class ResizableIntArray{
30:    private IntArray body;
31:    ResizableIntArray(){
32:      body = new IntArray();
33:    }
34:    void add(int val){
35:      body.add(val);
36:    }
37:    void get(int index){
38:      return (body.get(index));
39:    }
40:    int sum(){
41:      IntArray$Itr itr = iterator();
42:      int result = 0;
43:      while(itr.hasNext()){
44:        int value = itr.next();
45:        result += value;
46:      }
47:      return (result);
48:    }
49:  }
```

(b) 最適化後

図 1　キャストの除去
Fig. 1　Cast elimination.

たとえば，**図 1** (a) のクラス ResizableIntArray は，標準ライブラリの可変長配列 java.util.ArrayList（ArrayList と略記する）を使って整数の可変長配列を実現するもので，次のキャストを含む．

参照型の縮小変換　17，28 行目．ArrayList から取り出した参照を Integer 型に変換する．

Box 化　11 行目．ArrayList に格納する値を int 型の値から，クラス Integer のインスタンスへの参照に変換する．

Unbox 化　19，30 行目．ArrayList から取り出した参照の指示先にあるクラス Integer のインスタンスから，対応する int 型の値を取り出す．

これらのキャストが必要になる理由は，ArrayList が格納する値の宣言型が Object 型であり，int 型でないためである．本論文で提案する最適化では，これらのキャストを除去するために，次の処理を行う．

(1)　クラス ArrayList と同様の機能を提供するが，格納する値の型が int 型のクラス IntArray を作成する．ここで作成するクラス IntArray のコードを図 1 (b) の 1〜27 行目に示す．

(2)　図 1 (a) のコードを書き換え，クラス ArrayList の代わりにクラス IntArray を使わせる．書き換え後のコードを図 1 (b) の 29〜49 行目に示す．

書き換え前後のクラス ResizableIntArray を比較すると，書き換え前のメソッド add() や get() の中にあったキャストが，書き換え後のメソッドからなくなっていることが分かる．また，書き換え前後のメソッド sum() を比較すると，書き換え後に，キャストがなくなるだけでなく，イテレータを参照する変数 itr の宣言型が java.util.Iterator から IntArray$Itr に書き換わっていることが分かる．このことが示すように，本論文で提案する最適化は，コンテナのクラスだけでなく，イテレータなど，コンテナの内部クラスも変更する．

本論文で提案するキャストの除去は，我々が開発中の Java アプリケーション最適化支援系向けに実装している最適化である．この最適化支援系は，Java アプリケーションの開発者がアプリケーションを最適化するのを支援するもので，Java アプリケーション構成するクラスファイルを入力として受け取って，最適化を適用し，最適化済みのクラスファイルを出力する．最適化にあたっては，リフレクションやネイティブメソッドの挙動を静的に解析できないことが問題になるが，この問題については，挙動を利用者に問い合わせることで解決する．

たとえば図 1 (a) から図 1 (b) への最適化について考えると，最適化にあたって，クラス

ResizableIntArray のインスタンス変数 body の型を ArrayList から IntArray に変更しているが，この変更を実施すると，リフレクションやネイティブメソッドを通じてメンバ変数 body を参照するアプリケーションの挙動がおかしくなりうる．この問題を回避する手段に，リフレクションやネイティブメソッドを通じたインスタンス変数 body の参照が発生しえない場合に限って最適化を適用する方法があるが，リフレクションやネイティブメソッドの挙動を静的に予測することは一般には困難である．保守的に参照が発生しうると判断することは可能だが，そうすると図 1 (a) から図 1 (b) への最適化を適用できなくなる．そこで，我々の最適化支援系は，利用者に，リフレクションやネイティブメソッドを通じたインスタンス変数 body の参照が発生しうるか問い合わせ，その結果に応じて最適化の可否を判断する．

本論文の構成は次のとおりである．まず 2 章でキャストの除去の実装について詳述し，3 章で最適化が実行速度に与える影響を評価した結果を示す．4 章で関連研究について述べる．5 章は結論である．

2. 実　　装

本論文で提案する最適化は，単純にキャストを除去する処理のみからなるのではなく，除去の対象となる箇所を増やすための準備の処理を含む．本論文で提案する最適化の実施手順は次に示すとおりである．

(1)　準備：キャストの除去を阻害する要因を除去するために，次の処理を適用する．
　(a)　委譲コンテナの新規生成処理の検出：標準ライブラリのコンテナに委譲を行うインスタンス（委譲コンテナと略記）を新規生成する処理を検出する．
　(b)　インライン展開：標準ライブラリのコンテナあるいは委譲コンテナを新規生成して返戻するメソッドをインライン展開し，コンテナのクラスを差替可能なケースを増やす．
　(c)　キャストの移動：委譲コンテナのメソッドの呼び出し元にあるキャストを呼び出し先に移動する．
(2)　キャストの除去：標準ライブラリのコンテナのメソッドの呼び出し元にあるキャストを除去する．

これらの処理のうち，(2) のキャストの除去については，その概要を 1 章で示した．本章ではまず，(1) の準備の処理について，その概要を示し，次に，個々の最適化の手順について詳述する．

2.1 準　　備

準備の処理の目的は，キャストの除去を阻害する，次の 2 つの要因を除去することにある．

(1)　間接的なキャスト
(2)　格納する値の型が異なるコンテナを生成するインスタンス生成処理

これらの阻害要因とその除去方法について，**図 2** (a) のプログラムを使って詳述する．

2.1.1 間接的なキャスト

まず，間接的なキャストについて述べる．本論文において，間接的なキャストとは，クラス C が標準ライブラリのコンテナに委譲を行うメソッドを定義するとき，そのメソッドの呼び出し元で実引数（コンテナに格納する値）や返戻値（コンテナから取り出した値）に適用するキャストを指す．また，直接的なキャストとは，標準ライブラリのコンテナが定義するメソッドについて，メソッドの呼び出し元で実引数や返戻値に適用するキャストを指す．

たとえば図 2 (a) のプログラムでは，5〜19 行目で定義しているクラス MyMap が，メソッド put() や get() の実行を，標準ライブラリのコンテナ java.util.HashMap（HashMap と略記）などに委譲している．これらのメソッドの呼び出し元は，21〜45 行目で定義しているクラス ListOfNames のメソッド put() や match() の中にあり，それぞれのメソッド内にはクラス MyMap のメソッド put() や get() の実引数や返戻値をキャストする処理があるが（26，29，41 行目），これらのキャストが間接的なキャストになる．

本論文で提案する最適化では，間接的なキャストを除去するために，準備の段階で，間接的なキャストを直接的なキャストに変換する．この変換を適用すれば，キャストを除去する段階では，直接的なキャストのみを対象に除去の処理を適用すればよいことになる．変換は，間接的なキャストを呼び出し元から呼び出し先に移動することで実現する．移動の処理も含め，準備の処理をすべて適用した後のプログラムを図 2 (b) に示す．図 2 (b) のプログラムでは，キャストの処理がクラス ListOfNames のメソッド put() や match()（35〜48 行目）の中からなくなり，クラス MyMap2 や MyMap3（4〜30 行目）のメソッド put() や get() の中に移動していることが分かる．クラス MyMap2 および MyMap3 はキャストの移動にあたって，最適化系がクラス MyMap の定義から合成したものである．移動後のキャストは，いずれも HashMap のメソッド put() や get() の引数もしくは返戻値に対する直接的なキャストになっている．

2.1.2 格納する値の型が異なるコンテナを生成するインスタンス生成処理

図 2 (a) のプログラムでは，クラス ListOfNames のインスタンス変数 i2s が参照するクラス MyMap のインスタンスを，int 型の値から String 型の値への射影として使い，s2i が

```
 1:  import java.util.HashMap;
 2:  import java.util.Iterator;
 3:  import java.util.Map;
 4:
 5:  class MyMap{
 6:    private Map body;
 7:    MyMap(){ body = new HashMap(); }
 8:    MyMap(Map m){ body = m; }
 9:    void put(Object key, Object value){
10:      body.put(key, value);
11:    }
12:    Object get(Object key){
13:      return (body.get(key));
14:    }
15:    Iterator elements(){
16:      return (body.values().iterator());
17:    }
18:    MyMap create(){ return new MyMap(); }
19:  }
20:
21:  class ListOfNames{
22:    private MyMap i2s = MyMap.create();
23:    private MyMap s2i = MyMap.create();
24:    void put(int id, String name){
25:      // Box 化
26:      Integer iref = Integer.valueOf(id);
27:      i2s.put(iref, name);
28:      // Box 化
29:      iref = Integer.valueOf(id);
30:      s2i.put(name, iref);
31:    }
32:    int match(String regexp){
33:      Iterator itr = i2s.elements();
34:      while(itr.hasNext()){
35:        Object oref = itr.next();
36:        // 参照型の縮小変換
37:        String name = (String)oref;
38:        if (name.match(regexp)){
39:          oref = s2i.get(name);
40:          // 参照型の縮小変換&Unbox 化
41:          return ((Integer)oref).intValue();
42:        }
43:      }
44:    }
45:  }
```

(a) 準備前

```
 1:  import java.util.HashMap;
 2:  import java.util.Iterator;
 3:
 4:  final class MyMap2{
 5:    private Map body;
 6:    MyMap2(){ body = new HashMap(); }
 7:    void put(int key, String value){
 8:      // Box 化
 9:      Integer iref = Integer.valueOf(key);
10:      body.put(iref, value);
11:    }
12:    Iterator elements(){
13:      return (body.values().iterator());
14:    }
15:  }
16:
17:  final class MyMap3{
18:    private Map body;
19:    MyMap3(){ body = new HashMap(); }
20:    void put(String key, int value){
21:      // Box 化
22:      Integer iref = Integer.valueOf(value);
23:      body.put(key, iref);
24:    }
25:    int get(String key){
26:      Object oref = body.get(key);
27:      // 参照型の縮小変換&Unbox 化
28:      return ((Integer)oref).intValue();
29:    }
30:  }
31:
32:  class ListOfNames{
33:    private MyMap2 i2s = new MyMap2();
34:    private MyMap3 s2i = new MyMap3();
35:    void put(int id, String name){
36:      i2s.put(id, name);
37:      s2i.put(name, id);
38:    }
39:    int match(String regexp){
40:      Iterator itr = i2s.elements();
41:      while(itr.hasNext()){
42:        Object oref = Itr.next();
43:        // 参照型の縮小変換
44:        String name = (String)oref;
45:        if (name.match(regexp))
46:          return (s2i.get(name));
47:      }
48:    }
49:  }
```

(b) 準備後

図 2　キャストの除去の準備
Fig. 2　Preparations for the cast elimination.

```
1:  class ListOfNames{
2:    private MyMap i2s = new MyMap();
3:    private MyMap s2i = new MyMap();
```

図 3　インライン展開
Fig. 3　Inlining.

参照するクラス MyMap のインスタンスを，String 型の値から int 型の値への射影として使う．これらのクラス MyMap のインスタンスの生成元はいずれも 18 行目のインスタンス生成処理 new MyMap() だが，このように，1 つのインスタンス生成処理で，格納する値の型が異なるコンテナを生成するケースには，キャストの除去を適用しにくい．なぜなら，キャストの除去では，コンテナを新規生成する処理を書き換えて，キャストを必要としないコンテナを生成させるが，1 つのインスタンス生成処理が，格納する値の型の異なるコンテナを生成する場合，インスタンス生成処理で生成すべきコンテナのクラスが定かでなくなるからである．

この問題の解決を目的として，我々の最適化系は，コンテナを新規生成するメソッドに対しインライン展開を適用する．たとえば図 2 (a) の 22〜23 行目にあるインスタンス変数 i2s および s2i の初期化処理の右辺にあるメソッド呼び出し MyMap.create() にインライン展開を適用すると，**図 3** のコードになる．インライン展開後の図 3 のコードでは，2 行目で新規生成するクラス MyMap のインスタンスは，必ず int 型の値から String 型の値への射影として使われ，3 行目で新規生成するクラス MyMap のインスタンスは，必ず String 型の値から int 型の値への射影として使われ，それぞれが異なる型の値を格納するコンテナを生成することがなくなる．インライン展開の適用後に，キャストの移動を適用すると，それぞれのインスタンス生成処理で生成するインスタンスのクラスが MyMap2 と MyMap3 に書き換わる（図 2 の 33〜34 行目）．

2.2　最適化の手順

ここでは，本論文で提案する最適化を構成する個々の処理について順次，詳述する．

2.2.1　委譲コンテナの新規生成処理の検出

まず最初に，インライン展開やキャストの移動といった処理を実施可能にするために，委譲コンテナを新規生成する処理を特定する．我々の最適化系は次の条件を満たすインスタンス生成処理を委譲コンテナの新規生成処理と見なし，それ以外の新規生成処理が生成するインスタンスを委譲コンテナと見なさない．

- 生成するインスタンス o が保持するインスタンス変数 v が，標準ライブラリのコンテ

ナ c を指示する．また，インスタンス変数 v を宣言するクラスが，最適化対象である．

- インスタンス o のクラスが，コンテナ c に，値を格納もしくは取り出す処理を委譲するメソッドを持つ．また，そのメソッドを定義するクラスが，最適化対象である．

2.2.2　インライン展開

委譲コンテナの新規生成処理を検出する処理が終わったら，次に，メソッド呼び出しのうち，次の条件を満たすものにインライン展開を適用する．

- 呼び出し先のメソッドが，次の条件を満たす．
  - バイトコード `areturn` を唯 1 個保持する．
  - バイトコード `areturn` によって返戻する値の定義元が唯一，標準ライブラリのコンテナの新規生成処理，もしくは委譲コンテナの新規生成処理のみである．
  - バイトコード長が閾値（たとえば 20 バイト）を下回る．
- インライン展開の結果として，可視範囲の制限にかかわる挙動が変化しない．

ここで可視範囲の制限とは，Java が提供する可視範囲の規定に従って Java 仮想機械がクラスやメンバ変数，メソッドの可視範囲を制限することを表す．インライン展開を適用すると，展開元のコードが，クラスやパッケージを越えて展開先のメソッド内に移動する．この結果，可視範囲の制限によって，インライン展開前のコードでは発生しなかったエラーが，展開後のコードでは発生するといった問題が生じることがあるが，このような問題が発生しうる場合にはインライン展開を適用しない．

2.2.3　キャストの移動

インライン展開が終わったら，次に，キャストの移動を適用する．移動の実施手順は次に示すとおり．

(1)　委譲コンテナを新規生成するバイトコード n について，委譲コンテナのクラス C_n を新規作成し，n が生成する委譲コンテナ o のクラスを C_n に差し替える．また，委譲コンテナ o の使用点となる全バイトコードについて，委譲コンテナ o のクラスの差替えに必要な処理を適用する．ここで適用する処理の詳細は後述する．

(2)　クラス C_n が定義する，コンテナに値の格納もしくは取り出しを委譲するメソッドの全呼び出し元を調査する．
全呼び出し元において，返戻値（コンテナから取り出した値）に同じ型 T へのキャストをかけ，実引数（コンテナに格納する値）も型 T の値と見なせるならば，呼び出し元にあるキャストを呼び出し先に移動する．
さもなくば，最適化を諦め，手順 (1) で適用した全変更をキャンセルして元に戻す．呼び出し元によって，実引数が，Box 化を経由して受け取る値になったり，Box 化を経由せずに受け取る値になったりする場合には，Box 化を経由せずに受け取る呼び出し元に対して，実引数をいったん Unbox 化する処理と，再度 Box 化する処理を挿入する．この処理により，全呼び出し元で実引数を Box 化経由で受け取る形を作ったうえで，Box 化を含めたキャストの移動を行い，キャストの除去に進む．ここでキャストの除去の結果として，挿入した Box 化が除去されなかったならば，Unbox 化と Box 化を挿入する前の時点に戻り，挿入を行わずに最適化をやりなおす．具体的には，実引数が primitive 型の値ではなくラッパクラスのインスタンスだと見なしてキャストの移動を行ってから，再度，キャストの除去に進む．

手順 (1) で適用する処理について具体的に述べる．まず最初に，次の略称を定義する．

C_o　委譲コンテナ o の差替前のクラス．

C_p　クラス C_o が持つ，委譲先のコンテナを指示するインスタンス変数 v と，コンテナへの値の格納もしくは取り出しを委譲するメソッドの宣言元にあたるクラスの中で，最もクラス階層の根に近いクラス．

S_c　クラス C_o を起点として C_p を終点とするクラス階層に属するクラスの集合．

S_i　クラス C_o が実現し，かつ，C_p の親クラスが実現しないインタフェースの集合と，S_c の和集合．

このとき，手順 (1) の実施手順は次のとおりとなる．

(1)　クラス C_n を作成する．作成の時点で，クラス C_n は次の性質を持つ．
- クラス C_p の親クラスを継承する．
- メンバ変数やメソッドを持たない．
- 修飾子 `final` を持つ．

(2)　集合 S_n および集合 S_u を作成する．これらの集合は，委譲コンテナ o のクラスを変更する過程で書き換える処理を記憶するためのもので，集合 S_n には，インスタンスを新規生成するバイトコードを，集合 S_u には，集合 S_n に属するバイトコードが新規生成する参照の使用点を格納する．作成の時点で，集合 S_n は唯一の要素 n を持ち，集合 S_u は，何も要素を持たない．

(3)　集合 S_n に属するバイトコードのうち，新規生成するインスタンスのクラスが C_n でないものについて，新規生成するインスタンスのクラスを C_n に変更する．さらに，それらのバイトコードを出発点として，データフロー解析を行い，バイトコードが新規生成するインスタンス i の使用点を求める．

使用点は，インスタンス i を使用するバイトコードと，バイトコードが，i を何番目の入力値として使うかを表す値のペアとして表現する．
使用点が集合 S_u に属していないならば，使用点の種類に応じて，次の処理を行う．処理の過程でプログラムに何らかの変更が必要になった場合，使用点を集合 S_u に追加する．ただし，変更すべき箇所が最適化対象のクラスの外にある場合には，委譲コンテナ o のクラスの差替えを諦める．諦める際には，それまで差替えのために適用した全変更をキャンセルして元に戻す．

- 使用点がメソッド呼び出しを行うバイトコードで，インスタンス i をレシーバとして使う場合，バイトコードが参照する，呼び出し先のメソッドの名前やシグネチャ，宣言元のクラスを記録したエントリを調査する．調査の結果，宣言元のクラスが，集合 S_i に属していると分かった場合，エントリを変更し，宣言元のクラスを C_n にする．
  また，メソッド呼び出しを行うバイトコードが，インタフェース呼び出しを行うバイトコード `invokeinterface` である場合には，仮想呼び出しを行うバイトコード `invokevirtual` に書き換え，クラス C_n に呼び出し先のメソッドがなければ追加する．
- 使用点がメソッド呼び出しを行うバイトコードで，インスタンス i を実引数として使う場合，バイトコードが参照する，呼び出し先のメソッドの名前やシグネチャ，宣言元のクラスを記録したエントリを調査する．調査の結果，シグネチャ中に記載されている引数の宣言型が，集合 S_i に属していると分かった場合，次の処理を適用する．
  - 呼び出し先のメソッドの引数の宣言型を C_n に変更する．
  - 呼び出し先のメソッドの全呼び出し元について，呼び出し元のバイトコードが参照する，呼び出し先のメソッドの名前やシグネチャ，宣言元のクラスを記録したエントリを変更して，シグネチャ中の引数の宣言型を C_n にする．また，呼び出し元で引数を受け取る使用点が，集合 S_u に属していないならば，集合 S_u に追加する．
- 使用点がインスタンス i のインスタンス変数を参照するバイトコードである場合，クラス C_n が参照先のインスタンス変数を持たないなら追加する．
- 使用点がインスタンス i への参照をメンバ変数に代入するバイトコードである場合，メンバ変数の宣言型が集合 S_i に属しているならば，次の処理を適用する．
  - 代入先のメンバ変数の宣言型を C_n に変更する．
  - 代入先のメンバ変数に全代入元について，代入元のバイトコードが参照する，代入先のメンバ変数の名前やシグネチャ，宣言元のクラスを記録したエントリを変更し，シグネチャ中のメンバ変数の宣言型を C_n にする．また，代入元で代入する値を受け取る使用点が，集合 S_u に属していないならば，集合 S_u に追加する．
- 使用点がインスタンス i への参照を要素型が集合 S_i の配列に代入しうるバイトコードである場合，委譲コンテナ o のクラスの差替えを諦める．
- 使用点がインスタンス i の型を比較するバイトコード（`checkcast` もしくは `instanceof`）であり，なおかつ，比較対象のクラスが集合 S_i に属しているならば，比較対象のクラスを C_n に修正する．

(4)　新たに追加すべき使用点がなくなるまで，(3) の処理を行ったら，次に，(3) で集合 S_u に追加した使用点に到達する参照を生成しうる全バイトコードを求める．求めたバイトコードの中に，集合 S_n に属さないものがあったら，それらを集合 S_n に追加したうえで (3) に戻る．
ただし，追加にあたっては，バイトコードが新規生成するインスタンスのクラスをチェックし，クラスが C_o でないならば，委譲コンテナ o のクラスの差替えを諦める．また，バイトコードが最適化対象のクラスの外にある場合にも，委譲コンテナ o のクラスの差替えを諦める．

2.2.4　キャストの除去

キャストの移動が終わったら，次に，キャストの除去を適用する．除去の実施手順は次に示すとおり．

(1)　標準ライブラリのコンテナを新規生成するバイトコード n について，コンテナのクラス C_n を新規作成し，n が生成するコンテナ o のクラスを C_n に差し替える．また，コンテナ o の使用点となる全バイトコードについて，コンテナ o のクラスの差替えに必要な処理を適用する．

(2)　クラス C_n が定義するメソッドのうち，コンテナに値を格納するものと，コンテナから値を取り出すものについて，その全呼び出し元を調査する．
全呼び出し元において，返戻値（コンテナから取り出した値）に同じ型 T へのキャストをかけ，実引数（コンテナに格納する値）も型 T の値と見なせるならば，呼び出し元にあるキャストを除去し，呼び出し先のメソッドを型 T 向けのものに差し替

える．

さもなくば，最適化を諦め，手順 (1) でコンテナ o のクラスを差し替えるために適用した全変更をキャンセルして元に戻す．

呼び出し元によって，実引数が，Box 化を経由して受け取る値になったり，Box 化を経由せずに受け取る値になったりする場合には，Box 化を経由せずに受け取る呼び出し元に対して，実引数をいったん Unbox 化する処理と，再度 Box 化する処理を挿入する．この処理により，全呼び出し元で実引数を Box 化経由で受け取る形を作ったうえで，Box 化を含めたキャストを除去する．

手順 (1) で行う処理は，キャストの移動における処理とほぼ同様なので，ここでは，その詳細を省略する．ただし，キャストの除去における手順 (1) の実施にあたっては，イテレータなど，コンテナが生成する，コンテナのクラスの内部クラスのインスタンスについても，クラスの差替えが必要となる点に配慮する必要がある．

手順 (2) において，クラス C_n のメソッドを型 T 向けのものに差し替える処理は，C++ のテンプレートと同様の技法を使って実現できる．具体的な実現の手順を次に示す．

(1)　標準ライブラリが提供する個々のコンテナのクラスに対応するテンプレートを定義する．このテンプレートは，格納する値の型を引数として受け取り，キャストなしで値の格納および取り出しが可能なコンテナのクラスを生成する．

(2)　テンプレートに引数として型 T を与えることでクラスを新規生成し，クラス C_n のメンバ（メンバ変数やメソッド，内部クラス）を，新規生成したクラス中にあるメンバで差し替える．

2.2.5　その他の最適化

また，我々の最適化系は，キャストの除去の際に，別途，次の最適化を適用する．

2.2.5.1　冗長なインスタンス生成の除去

標準ライブラリの射影（`HashMap` など）は，格納する値を順次取り出すイテレータを返戻するメソッドを直接的には提供せず，代わりに，格納する値の集合を返戻するメソッド `values()` を提供する．集合はイテレータを返戻するメソッド `iterator()` を持つので，射影 `m` が格納する値のイテレータ `i` を求めるには，現状では，集合を取得してから，集合のイテレータを求める処理 `i = m.values().iterator()` を実行すればよいが，これではイテレータのみを必要とする場合にまで集合を生成することになり，効率が悪い．

そこで我々の最適化系では，射影のクラスに，集合を介することなく，直接イテレータを取得可能にするメソッド `value_iterator()` を実装し，さらに，コード `values().iterator()` を，`value_iterator()` に書き換えることで，冗長な集合の生成を不要にする．

また，我々の最適化系が使う `ArrayList` の実装は，コンストラクト時に初期サイズが 0 と指定された場合に，`ArrayList` の本体にあたる配列の生成を省略する機能を提供する．

2.2.5.2　呼び出されうるメソッドに応じた最適化

標準ライブラリのコンテナのクラスや，その内部クラスは，利用元によっては冗長な処理を行うことがある．たとえば `HashMap` の内部クラスであるイテレータ `HashIterator` は，エントリを削除するメソッド `remove()` を実現するために，削除の対象となるエントリを指示するインスタンス変数 `current` を定義し，さらに，その内容を維持管理するための処理を，次のエントリを取得するメソッド `nextEntry()` の内部などに挿入しているが，この維持管理の処理は，メソッド `remove()` が呼び出されないなら冗長になる．

そこで我々の最適化系は，コンテナやその内部クラスの実装を，適用されうるメソッドに応じて複数用意し，個々のインスタンスを新規生成する処理ごとに，どの実装を使うか定める．具体的には，まず，個々の処理が生成するインスタンスの使用点を求め，求めた使用点から，インスタンスをレシーバとして呼び出されうるメソッドの集合を求め，この集合に対応するクラスの実装を定める．

たとえば，イテレータ `HashIterator` のメソッド `remove()` が呼び出されることがないならば，イテレータのクラスとして，インスタンス変数 `current` および，その維持管理に関する処理を省略したものを使う．

また，`HashMap` のメソッド `hash()` の実装を，`HashMap` の新規生成元ごとに最適化する．メソッド `hash()` は，`HashMap` が鍵に対応するハッシュ値を計算する際に呼び出すメソッドである．このメソッドの JDK1.5 までの実装は，命令レベル並列性の活用を考慮したものではなく，効率的な計算が困難であった．そこで JDK1.6 からは，より効率的な実装が採用されているが，この新たな実装を JDK1.5 に移植することは単純にはできない．なぜならメソッド `hash()` は `HashMap` 内の値の格納順序に影響を与え，格納順序はイテレータを使った `HashMap` 内の値の走査の順序に影響を与えるが，JDK1.5 向けに開発された Java アプリケーションの中には，この順序が古い `hash()` の実装に従って定まると仮定しているものがあるからである．しかしながら，Java アプリケーションが利用する `HashMap` の中には，イテレート処理の対象となりえないものもあり，そういったものには新たな `hash()` の利用を許しても問題は生じない．そこで我々の最適化系は，JDK1.5 向けに開発された Java アプリケーションを最適化する際には，`HashMap` の新規生成元を最適化する際に，生成する `HashMap` がイテレート処理の対象となりうるか静的に解析し，なりえない場合にはメソッド

hash() の実装として新しいものを使う．

2.2.5.3　イテレータにおける障害検知機能の除去

標準ライブラリのコンテナは，イテレート処理の過程で要素の読み飛ばしといった障害が発生することを防ぐことを目的として，イテレータとコンテナに，障害を検知する機能を組み込んでいる．この機能の実現は次のとおりとなっている．まず，個々のコンテナに，カウンタを設けて，要素の追加や削除といった処理を何回行ったか記録する．イテレータは生成時に，このカウンタの値を記憶し，イテレート処理の途中でカウンタの値が変化したら例外を投げる．

この機能の実現には，イテレータを利用するか否かによらず，実行時オーバヘッドを発生するという問題がある．また，この機能は，プログラム開発の際には有効だが，運用段階でも必ず使うべきものか否か定かでない．特に，極端に実行性能が重要になるケースでは，この機能の除去によって実行性能を改善したいという要求もありうる．この機能の実現は，コンテナの仕様的には必須でないので，実現しなくてもプログラムの実行結果に問題が起きるとは考えにくい．そこで我々の最適化系は，新規生成するコンテナおよびそのイテレータに，障害検知機能の実現を含めない．

なお，障害検知機能の実現を含めないことが問題になる場合には，障害検知機能を実現しつつ実行性能を改善することもできる．具体的には，インスタンスがイテレート処理の対象となりうるか，静的解析によって求め，求めた結果に応じて障害検知機能を除去するか否か定めればよい．

2.2.5.4　実行プロファイルを使った最適化

コンテナの中には，複数のアルゴリズムによって実現可能なものがあるが，どのアルゴリズムを利用するのが妥当か検討する際に，実行プロファイルを利用するとよいケースがある．

たとえば，赤黒木のアルゴリズムによって射影を実現する java.util.TreeMap（TreeMap と略記する）の実装について考える．TreeMap の実装では，個々のエントリを，**図 4** (a) にあるように，左右の子と親への参照を持つノードとして実現することもできるが，図 4 (b) にあるように，左右の子と親に加え，イテレート処理における前後のノードへの参照 next，previous を持つノードとして実現することもできる．

これらの実装については，どちらが必ず優れているということはできない．図 4 (a) の実装には，参照 next，previous を保持しないことから，メモリ消費量が小さく済み，また，それらの参照を維持管理する手間がない分，ノードの追加や削除を効率的に実行できるという利点がある．一方，図 4 (b) の実装には，参照 next を使って，イテレート処理を高速化できるという利点がある．これらのどちらを使うべきか定める手段として，実行プロファイルを使うことができる．

```
1:   class Entry<K,V>{
2:     K key;
3:     V value;
4:     boolean color;
5:     Entry<K,V> left;
6:     Entry<K,V> right;
7:     Entry<K,V> parent;
8:           ...
9:   }
```

(a) 追加削除の高速化

```
1:   class Entry<K,V> {
2:     K key;
3:     V value;
4:     boolean color;
5:     Entry<K,V> left;
6:     Entry<K,V> right;
7:     Entry<K,V> parent;
8:     Entry<K,V> previous;
9:     Entry<K,V> next;
10:          ...
11:  }
```

(b) イテレータの高速化

図 4　赤黒木のノードの実装
Fig. 4　Red-black tree node implementations.

そこで，我々の最適化系は，実行プロファイルを使って，コンテナの実装を最適化する機能を提供する．たとえば，TreeMap については，実行プロファイルから，イテレート処理と，ノードを追加削除する処理の実施頻度を求め，イテレート処理の実施頻度が相対的にも絶対的にも大きい場合には図 4 (b) の実装を，さもなくば図 4 (a) の実装を利用するよう定める．

また，我々の最適化系は，実行プロファイルを，HashMap におけるハッシュ表の初期サイズの調整や，ArrayList におけるごみ集め支援機能の削除にも利用する．HashMap の実装は格納する値の数に応じてハッシュ表の大きさを調整する機能を持つが，調整にはコストがかかるので，HashMap の生成時にハッシュ表の大きさを適切に定めておく方が良い．そこで我々の最適化系は，実行プロファイルを利用して，HashMap の生成元ごとに適切な大きさを求め，生成元を最適化する際に，ハッシュ表を求めた大きさにするようにする．

ArrayList はごみ集めの支援を目的として，内部の要素を除去するメソッドの実行時に，ArrayList の本体にあたる配列に null を代入する処理を行うが，この処理は ArrayList の寿命が短い場合には冗長になる．そこで我々の最適化系は，実行プロファイルから寿命の短い ArrayList を生成すると分かる生成元を最適化する際には，要素を除去するメソッドから null を代入する処理を除去する．

2.2.5.5　オブジェクトインライニング

オブジェクトインライニング[5)–7),22)] とは，あるクラスのインスタンスが，別のクラスのインスタンスを参照するとき，参照先のインスタンスのクラス定義を，参照元のクラスに展開する最適化である．我々の最適化系は，オブジェクトインライニングによって，委譲コン

テナから委譲先のコンテナへの委譲を除去する．

3. 評　　価

本論文で提案した最適化がプログラムの実行速度に与える影響を，SPECjbb2005 [16] を使って評価した．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．評価に使った計算機は PC（Xeon3210 2.13 GHz，RAM 4 GByte），OS は CentOS 4.4，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 製の Java 仮想機械（Java Runtime Environment 1.5.0_05 のもの）をベースに，日立製作所が独自の機能強化を施したものである．Java 仮想機械には，実行時オプションを通じて，ヒープの初期サイズ，最大サイズを 1.5 GByte とするよう指示し，また，全最適化を適用するよう指示した．この指示によって適用される最適化の中には冗長な Box 化の削除[23] が含まれる．

冗長な Box 化の削除と，本論文で提案したキャストの除去は，ともに Box 化を除去しうる．これらの最適化は，それぞれが適用可能になる条件が異なるので，一般に一方が優れていると述べることはできない．しかしながら，SPECjbb2005 への適用に限ると，冗長な Box 化の削除によって除去できる Box 化は，すべて，キャストの除去でも除去できる．

評価は SPECjbb2005 のプログラムを手動で解析して，本論文で述べた最適化を適用し，個々の最適化が実行速度に与える影響を測定することで行った．評価結果を**図 5** に示す．図 5 の縦軸は，本論文で提案した全最適化を適用しない場合に対する実行速度の向上率を表し，横軸は適用した最適化を表す．適用した最適化は，最も左の項目ではキャストの除去のみであり，そこから右に進むに従って順次追加される．最も右の項目は本論文で述べた全最適化を適用した場合に対応する．

図 5 から，最適化が実行速度に与える影響は，キャストの除去が 14.8%，呼び出されうるメソッドに応じた最適化が 5.1%，実行プロファイルに応じた最適化が 1.4%であり，その他の最適化も含めトータルで 21.3%になることが分かる．

キャストの除去の効果が比較的大きい原因としては，大きな実行時オーバヘッドの発生源である Box 化の除去に加え，動的コンパイラによる最適化機会の増加を指摘できる．我々の最適化系は，キャストを除去するために，コンテナのクラスを生成する際に，コンテナに格納する値の型を明確にするが，この型情報は，動的コンパイラにとって，最適化を適用するための根拠となりうる．たとえば，`HashMap` では，鍵に対応する値を検索する際に，まず，鍵をレシーバとしてメソッド `hash()` を呼び出し，鍵のハッシュ値を計算する．このメソッド呼び出し `hash()` の最適化なしでの実現は，たとえば間接呼び出しなど，効率的でないものになる．しかしながら，ここでもし，鍵の型が `String` 型だと分かれば，動的コンパイラは，この型情報を使ってメソッド呼び出しを最適化し，メソッド呼び出しの実現をより効率的なもの（クラス `String` が定義するメソッド `hash()` の直接呼び出し）に変換可能になる．

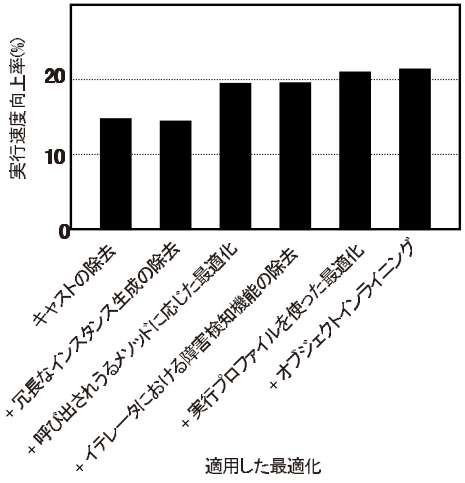

図 5　評価結果
Fig. 5　Benchmark results.

なお，Java 仮想機械で冗長な Box 化の削除を適用しないと，キャストの除去によって得られる実行速度の向上率が 3.7%増加して 18.5%になり，トータルでの向上率も同様に増加する．なぜなら，SPECjbb2005 では，冗長な Box 化の削除を適用しなくても，キャストの除去から同様の効果を得られるので，冗長な Box 化の削除を適用しない場合，その効果がキャストの除去の効果に上乗せされるからである．

4. 関 連 研 究

Java の標準ライブラリのコンテナは Generics [2),3),9)] を使って記述されており，その利用にあたって，プログラマがキャストの処理を書く必要はない．ただ，Java の Generics はキャストを自動挿入するためのものなので，Java の Generics を使ってもキャストのオーバヘッドはなくならない．Java の Generics と同様の型パラメータを使ったクラス定義でも，その実装が，C#の Generics [11] や C++のテンプレート [17] のように型パラメータの値に応

じてクラス定義を生成するものならば，キャストを不要にできる．Java でも Generics の実装を変更すれば，C#や C++と同様に，キャストの不要なプログラミングを可能にできるが，Java の Generics 自体の変更は難しい．Java の Generics の実装は数多くの設計が考えられる中から[1),4),8),12),13),18),21)]，後方互換性などを考慮して定まったもので，一度定まった言語仕様は容易に変更できない．

我々の技法は Generics の実装を変更するものでなく，既存の Java アプリケーションに適用できる．我々の技法では，標準ライブラリのコンテナを，C++ のテンプレートにあたる技術を使って実装し，個々のコンテナの生成元ごとに，キャストや冗長な処理を必要としない，最適なクラスを生成することで，キャストのオーバヘッドを軽減する．テンプレートを使ったコンテナの定義は，C++ の標準テンプレートライブラリで提供されており，新規でない．ただし，C++ のコンテナでは，格納する値の型情報をプログラマが提供するのに対し，我々の最適化系は，バイトコードを走査してコンテナに格納する値の型を収集する．我々の最適化系と同様に，個々のインスタンスの生成元ごとに，インスタンスが実行するメソッドが受け取る引数の情報を収集し，収集した情報を使ってインスタンスのクラスを最適化する技法は，Schultz らによって提案されている[14),15)]．ただし，Schultz らの技法は，我々の技法のように，コンテナに特化したものではなく，コンテナ向けのキャストを除去できるわけではない．

Java においてキャストを除去する技法としては，これまでに，冗長なキャスト[10)]や Box 化[23)]を除去する方法が提案されている．これらは，本論文で提案した技術のように，コンテナ向けに特化した技術ではなく，コンテナに値を格納するメソッドの全呼び出し元をチェックして，格納する値の型を求め，求めた型情報を使ってキャストを除去するといった処理は行わない．

本論文で提案する最適化系は，コンテナのクラスの作成にあたり，冗長なメソッドが最適化やコードサイズに悪影響を与えることを防ぐために，クラスに追加するメソッドを，呼び出し元があるものに限定しているが，この技法は既存のものである[19),20)]．また，コンテナのクラスの作成にあたって適用する最適化のうち，オブジェクトインライニング[5)–7),22)]や動的プロファイルを使ったクラスの最適化[24)]も既存の技法である．ただし，呼び出されうるメソッドに応じた最適化に関する文献は，我々の調査の限りでは見つかっていない．

5. 結　　論

本論文では，キャストのうち，特に標準ライブラリのコンテナの利用にあたって必要になるものが，実行速度の劣化要因となりうることを指摘し，対策としてキャストを除去する最適化技術を提案した．また，コンテナを個々の使用箇所向けに最適化する技術を提案した．提案した技術が実行速度に与える影響を，SPECjbb2005 を使って計測したところ，実行速度を 21.3%改善することが分かった．

参 考 文 献

1) Allen, E., Bannet, J. and Cartwright, R.: A First-class Approach to Genericity, *Proc. 18th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications*, pp.96–114 (2003).
2) Arnold, K., Gosling, J. and Holmes, D.: *The Java Programming Language*, 4th Edition, Addison-Wesley, Reading, Mass. (2005).
3) Bracha, G., Odersky, M., Stoutamire, D. and Wadler, P.: Making the Future Safe for the Past: Adding Genericity to the Java Programming Language, *Proc. 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications*, pp.183–200 (1998).
4) Cartwright, R. and Steele Jr., G.L.: Compatible Genericity with Run-time Types for the Java Programming Language, *Proc. 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications*, pp.201–215 (1998).
5) Dolby, J.: Automatic Inlining Allocation of Objects, *Proc. ACM SIGPLAN 1997 conference on Programming Language Design and Implementation*, pp.7–17 (1997).
6) Dolby, J. and Chien, A.: An Evaluation of Automatic Object Inline Allocation Techniques, *Proc. 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications*, pp.1–20 (1998).
7) Dolby, J. and Chien, A.: An Automatic Object Inlining Optimization and its Evaluation, *Proc. ACM SIGPLAN 2000 conference on Programming Language Design and Implementation*, pp.345–357 (2000).
8) Garcia, R., Jarvi, J., Lumsdaine, A., Siek, J.G. and Willcock, J.: A Comparative Study of Language Support for Generic Programming, *Proc. 18th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications*, pp.115–134 (2003).
9) Gosling, J., Joy, B., Steele Jr., G.L. and Bracha, G.: *The Java Language Specification*, 3rd Edition, Addison-Wesley, Reading, Mass. (2005).
10) 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 Optimization for a Java Just-In-Time Compiler, *Proc. 18th ACM SIGPLAN Conference on Object-*

Oriented Programming, Systems, Languages and Applications, pp.187–204 (2003).
11) Kennedy, A. and Syme, D.: Design and Implementation of Generics for the .NET Common Language Runtime, *Proc. ACM SIGPLAN 2001 conference on Programming Language Design and Implementation*, pp.1–12 (2001).
12) Bonk, J.A., Myers, A.C. and Liskov, B.: Parameterized Types for Java, *Proc. 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pp.132–145 (1997).
13) Sasitorn, J. and Cartwright, R.: Deriving Components from Genericity, *Proc. 2007 ACM symposium on Applied Computing*, pp.1109–1116 (2007).
14) Schultz, U.P., Lawall, J.L. and Consel, C.: Automatic Program Specialization for Java, *ACM Trans. Programming Languages and Systems*, Vol.25, No.4, pp.452–499 (2003).
15) Schultz, U.P., Lawall, J.L., Consel, C. and Muller, G.: Towards Automatic Specialization of Java Programs, *Proc. 13th European Conference on Object-Oriented Programming*, pp.367–390 (1999).
16) Standard Performance Evaluation Corporation: SPECjbb2005 (2005). http://www.spec.org/jbb2005/
17) Stroustrup, B.: *The C++ Programming Language*, Addison-Wesley, Reading, Mass. (2000).
18) Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R. and Lee, P.: TIL: A Type-Directed Optimizing Compiler for ML, *Proc. ACM SIGPLAN 1996 conference on Programming Language Design and Implementation*, pp.181–192 (1996).
19) Tip, F., Laffra, C., Sweeney, P.F. and Streeter, D.: Practical Experience with an Application Extractor for Java, *Proc. 14th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications*, pp.292–305 (1999).
20) Tip, F., Sweeney, P.F., Laffra, C., Eisma, A. and Streeter, D.: Practical Extraction Techniques for Java, *ACM Trans. Programming Languages and Systems*, Vol.24, No.6, pp.625–666 (2002).
21) Viroli, M.: Parametric Polymorphism in Java: An Approach to Translation Based on Reflective Features, *Proc. 15th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications*, pp.146–165 (2000).
22) Wimmer, C. and Mössenböck, H.: Automatic Feedback-Directed Object Inlining in the Java HotSpot Virtual Machine, *Proc. 3rd International Conference on Virtual Exection Environments*, pp.12–21 (2007).
23) 千葉雄司：Java 向け動的コンパイラによる冗長な Box 化の削除，情報処理学会論文誌：プログラミング，Vol.48, No.SIG 10(PRO 33), pp.165–175 (2007).
24) 神戸隆行：テンプレート・メタ・プログラミングによる FFT の適応的最適化，情報処理学会論文誌：プログラミング，Vol.46, No.SIG 1(PRO 24), pp.78–96 (2005).

(平成 19 年 11 月 26 日受付)
(平成 20 年 3 月 15 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．