情報処理学会論文誌　プログラミング　Vol. 1　No. 1　26–35 (June 2008)

Java 標準ライブラリを対象とした配列参照の最適化

柳　　優[†1]　田中俊之[†1]
千葉雄司[†2]　土居範久[†1]

本論文では，Java における配列参照の高速化を目的として，標準ライブラリ中の冗長な null 検査および配列添字検査を除去する技法を提案する．提案技法では，データフロー解析を使う従来の最適化では除去が困難な検査を除去するために，標準ライブラリ中の冗長な検査の所在をあらかじめ動的コンパイラに教えておく．検査の中にはネイティブメソッドやリフレクションの動的な振舞い次第で冗長か否かが変化するものもあるが，そうした検査も除去可能にするために，実行時にリフレクションやネイティブメソッドの振舞いを監視し，検査が冗長か否か判断可能にする．SPECjvm98 および SPECjbb2005 を使った評価から，本論文で提案した最適化によって実行速度を平均で 0.87%向上させることができることが分かった．

Optimizing Array References in the Java Standard Library

Yu Yanagi,[†1] Toshiyuki Tanaka,[†1] Yuji Chiba[†2]
and Norihisa Doi[†1]

This paper presents an optimization technique for array references eliminating redundant null tests and array index tests in the standard library. Our optimization technique eliminates redundant tests using manually collected information that tells where the redundant tests are, and thus can eliminate redundant tests that traditional optimization techniques using data flow analysis can hardly do. Some tests in the standard library are redundant if some related fields are not overwritten by reflections or native methods. To eliminate such tests, our optimization dynamically watches the behavior of the reflection or native methods, and eliminates the tests if possible. Estimation using the SPECjvm98 and the SPECjbb2005 benchmark suites showed that our optimization techniques improve performance by 0.87%.

```
1:  value = byte_array[index];
```

(a) Java による配列参照の記述

```
1:  // null 検査
2:  if (byte_array == NULL)
3:   throw new NullPointerException();
4:  // 配列添字検査:((index < 0) || (byte_array->length <= index))
5:  // を符号無し整数向け比較 1 つで実現
6:  if (((unsigned int)byte_aray->length) <= ((unsigned int)index))
7:   throw new ArrayIndexOutOfBoundsException();
8:  // 配列参照
9:  value = byte_array[index];
```

(b) 配列参照の実現

図 1　Java における配列参照の実現
Fig. 1　An implementation of an array reference in Java.

1. はじめに

Java™ ⋆1 における実行時オーバヘッドの発生原因の 1 つに，配列参照の際に実施する null 検査や配列添字検査がある．Java で記述した配列参照の例を**図 1** (a) に，その実現の例を図 1 (b) に示す．図 1 (b) から，Java の配列参照においては，まず最初に null 検査を行い，続いて配列添字検査を行うことが分かる．

null 検査や配列添字検査はセキュリティの確保やソフトウェアの開発効率を改善するために必要な処理だが，必ずしも配列参照のたびに必要になる処理ではなく，冗長と見なせる場合には除去できる．そこで，本論文では，これらの処理を除去する動的コンパイラ向けの最適化技法を提案する．

我々が提案する最適化技法は，標準ライブラリ中にある null 検査および配列添字検査を対象としたもので，どの null 検査と配列添字検査が冗長か，あらかじめ動的コンパイラに教えておくことで除去を可能にする．最適化対象となる null 検査や配列添字検査の中には，

†1 中央大学大学院理工学研究科情報工学専攻
Information and System Engineering Course, Graduate School of Science and Engineering, Chuo University
†2 中央大学研究開発機構
Research and Development Initiative, Chuo University
⋆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．

リフレクションやネイティブメソッドの振舞いによっては除去できなくなるものもあるが，提案技法では，これらを除去可能にするために，リフレクションやネイティブメソッドの振舞いを監視する．提案技法には，適用対象のクラスが標準ライブラリ内のものに限られるという欠点はあるが，次の利点もある．

- 従来のデータフロー解析などを使った最適化では除去が困難な検査を除去できる．
- データフロー解析を必要としないため，最適化がコンパイル時間にもたらす影響が小さい．

本論文の構成を次に示す．まず，2 章において null 検査および配列添字検査を除去する技法を提案する．次に，3 章で最適化の効果を評価する．4 章では，これまでに実装された null 検査や配列添字検査向けの最適化技法を概観し，本論文で提案した技法と比較する．5 章はまとめである．

2. null 検査および配列添字検査の除去

本章では，まず，冗長な null 検査および配列添字検査の除去にあたって問題となる点を明らかにするために，標準ライブラリの次のメソッドをとりあげ，それぞれの内部にある検査を困難にする要因を示す．

- `java.lang.String.charAt()`
- `java.lang.Long.getChars()`

次に，これらのメソッドが含む検査を除去可能にする，我々の最適化の実装について詳述する．

2.1 `java.lang.String.charAt()`

標準ライブラリのクラス `java.lang.String` は文字列を表し，そのメソッド `charAt(int i)` は文字列の `i` 番目の文字を返戻する．メソッド `charAt()` の実装を**図 2** に示す．

図 2 の 8 行目にある配列参照では，`null` 検査や配列添字検査が冗長になることが多い．`null` 検査が冗長になることが多い理由は，インスタンス変数 `value` の値が，文字列のコンストラクト時に，文字列の本体を表す配列を参照するよう，非 null の値に初期化され，その後，リフレクションやネイティブメソッドを使わない限り上書きされえないからである．配列添字検査が冗長になることが多い理由は，8 行目の配列参照に先行して，9 行目において，変数 `i` の値が文字列の長さの範囲内にあるか否かの検査しており，なおかつ，リフレクションやネイティブメソッドを使って上書きしない限り，インスタンス変数 `value` や `count`，`offset` が例外の原因となるような値を保持することはないからである．

```
1:  public class String{
2:    private final char[] value;              // 文字列の本体
3:    private final int offset;                // 先頭文字のオフセット
4:    private final int count;                 // 文字数
5:                                  ...
6:    public char charAt(int i) {
7:      if ((i < 0) || (i >= count)) 例外発生;
8:      return value[i + offset];
9:    }
```

図 2　`charAt()` の実装
Fig. 2　Implementation of `charAt()`.

8 行目の `null` 検査および配列添字検査のうち，`null` 検査に関しては，従来の最適化技法で除去できる．Ghemawat らの提案技法[4] に，非 `null` の値に初期化され，その後，上書きされえないメンバ変数を求め，求めたメンバ変数向けの null 検査を除去するというものがある．Ghemawat らの提案技法はリフレクションやネイティブメソッドがもたらす影響を無視しているので，そのままでは実用化できないが，この問題を解決する方法はすでにある[6),7),22)]．

しかしながら，配列添字検査に関しては，従来の最適化技法では除去できない．8 行目の配列添字検査を除去するためには，3 つのインスタンス変数 `value`，`count`，`offset` の間に成り立つ関係を求める必要がある．これに類する場合向けの最適化技法には Aggarwal らが提案したものがある[1)]．Aggarwal らの技法は 1 つの配列の長さと 1 つのインスタンス変数の値の関係を解析し，解析結果を使って配列添字検査を除去する．ただし，この最適化で 8 行目の配列添字検査を除去することはできない．なぜなら，8 行目の配列添字検査を除去するためには，1 つの配列の長さと 2 つのインスタンス変数 `count`，`offset` の値の関係を解析する必要があるからである．Aggarwal らの技法を応用し，配列の長さと複数のインスタンス変数の値に成り立つ関係を求めることを可能にすることは不可能ではないかもしれないが，めったに有用にならない解析機能を追加しても，コンパイル時間の増加を招くばかりで実行速度を改善できないという結果になりかねない．また，Aggarwal らの技法には，リフレクションやネイティブメソッドがもたらす影響を無視しているという問題があり，そのままでは実用化できない．

なお，null 検査や配列添字検査にともなうオーバヘッドを回避するための手段として，標準ライブラリのメソッドをネイティブメソッドに書き換えることは，必ずしも適切でない．

その理由を次に示す．

- ネイティブメソッドへの書き換えは，少なくとも，charAt() 内の検査のように，冗長か否かが動的な文脈によって定まる検査を除去する手段として適切でない．なぜなら，ネイティブメソッドの挙動を，動的な文脈に応じて変更することが困難だからである．
- ネイティブメソッド呼び出しには大きな実行時オーバヘッドがかかる．実行時オーバヘッドの発生原因は，正確なごみ集めを支援するためのレジスタの待避や復帰など様々で，発生する実行時オーバヘッドの大きさは Java 仮想機械の実装に依存するが，charAt() のように実行コストの小さなメソッドについては，ネイティブメソッドに書き換えても，実行速度を改善できないことが多い．
- ネイティブメソッドは，動的コンパイラの最適化を阻害することがある．たとえば，動的コンパイラはネイティブメソッドをインライン展開の対象とすることはなく，また，ネイティブメソッドから生じる副作用を解析できない場合が多い．charAt() のように小さなメソッドは，インライン展開を促進するためにも，Java で記述する方が適当である．

実際，charAt() を呼び出す処理を 1,000 万回行うループを Java で記述し，その実行時間を測定したところ，charAt() をネイティブメソッド化しない方が 125 倍速いことが分かった．このように大きな差が生じた原因には，ネイティブメソッド呼び出しのオーバヘッドと，インライン展開の適用の有無がある．

なお，この測定にあたり，我々が使用した計算機は PC（CPU: Intel Xeon E5320（1.86 GHz Quad Core）× 2，RAM: 4 GByte）であり，OS は Intel64 向けの CentOS 5，Java 仮想機械は，Sun Microsystems 製 Java Development Kit 1.5 に付属のものである．この Java 仮想機械は，我々が最適化の実装に用いたものでもある．本論文で示す測定結果は，特に明記しない限り，この環境で測定した．

2.2 java.lang.Long.getChars()

標準ライブラリのクラス java.lang.Long は 64 bit 長の整数を表し，そのメソッド getChars(long i, int index, char[] buf) は整数 i を表す文字の列を配列 buf に格納する．仮引数 index は文字の列を配列 buf のどこに格納するか指定するためのもので，具体的には，最下桁の数字を格納する位置を指定する．メソッド getChars() の実装の一部を**図 3** に示す．

図 3 の 30，31 行目にはクラス Intger のクラス変数 DigitOnes，DigitTens が指示する配列に対する配列参照がある．それぞれのクラス変数は 2〜11 行目の定義により，長さ

```
 1:  public class Integer{
 2:   static final char[] DigitOnes = {
 3:    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
 4:    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
 5:                              ...
 6:    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
 7:   static final char[] DigitTens = {
 8:    '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 9:    '1', '1', '1', '1', '1', '1', '1', '1', '1', '1',
10:                              ...
11:    '9', '9', '9', '9', '9', '9', '9', '9', '9', '9'};
12:                              ...
13:  }
14:
15:  public class Long{
16:   static void getChars(long l, int pos, char[] buf){
17:    char sign = 0;
18:
19:    if (l < 0) {
20:     sign = '-';
21:     l = -l;
22:    }
23:
24:    while (l > Integer.MAX_VALUE) {
25:     //l を 100 で除して商 q，剰余 r を算出
26:     long q = l / 100L;
27:     int r = (int)(l - ((q << 6) + (q << 5) + (q << 2)));
28:     l = q;
29:     // 下 2 桁を数字に変換
30:     buf[--pos] = Integer.DigitOnes[r];
31:     buf[--pos] = Integer.DigitTens[r];
32:    }
33:                              ...
```

図 3　getChars() の実装
Fig. 3　Implementation of getChars().

100 の配列を指示するよう初期化される．これらの配列は整数を 2 桁ずつ数字に変換する際に使うもので，DigitOnes の指示する配列が 1 桁目に対応する数字を，DigitTens の指示する配列が 2 桁目の数字を格納する．一方で配列参照の添字 r は 0〜99 の範囲に収まる（r

は整数 l を 100 で除した際の剰余であり，かつ，整数 l の値が 19〜22 行目の処理で非負になっているため）．したがって，30，31 行目の配列参照は null 検査も添字検査も必要としないことが多いが，これらの検査を従来の最適化によって除去するのは必ずしも容易でない．

固定長の配列に対する配列参照を最適化する技法は Ghemawat らによって提案されている[4]．Ghemawat らの技法では，固定長の配列を指示するよう初期化され，上書きされないメンバ変数を求め，求めたメンバ変数の指示する配列を参照する際の `null` 検査を除去する．また，添字のとりうる値が 0 以上配列長未満であると分かる場合には，配列添字検査も除去する．この最適化を使えば，30，31 行目の配列参照にともなう `null` 検査や配列添字検査を除去できそうだが，除去にあたっては次の問題を解決しなくてはならない．

- 添字 `r` の値域を解析する．添字 `r` は非負の整数 `l` を 100 で除した際の剰余なので 0〜99 の範囲に収まるが，この `r` の値を算出するための式は `l%100` といった単純なものでなく，26〜27 行目にある複雑な式である．
  コンパイラの解析機能を強化して，この式を解析し，値域を求められるようにすることは可能だが，必ずしも有益とは限らない．なぜなら機能強化の結果として，解析にかかる時間が長くなりうるからである．動的コンパイラでは，26〜27 行目の式のようにめったに現れないパターンのために解析機能を強化すると，コンパイル時間が長くなり，かえって実行性能の劣化を招くことがある．
- ネイティブメソッドへの対策を用意する．クラス変数 `DigitOnes` や `DigitTens` は `final` 修飾子つきで宣言されているが，`final` 修飾子は必ずしも変数の上書きを防ぐものではない．たとえば，ネイティブメソッドを使えば，`final` 修飾子をつけて宣言した変数についても上書きできる．上書きが発生しうるということは，クラス変数 `DigitOnes` や `DigitTens` が指示する配列の長さは必ず 100 とはいえなくなる．このとき，たとえ添字 `r` の値域が 0〜99 だと分かったとしても，配列参照の際の `null` 検査や配列添字検査を除去できなくなる．

2.3 実　　装

我々が実装した null 検査および配列添字検査の除去では，標準ライブラリ中にある検査のうち，冗長なものの所在を，あらかじめ動的コンパイラに教えておき，動的コンパイルの際に検査を生成しないようにする．リフレクションやネイティブメソッドの動的な振舞いが，冗長か否かを定める検査については，リフレクションやネイティブメソッドの動的な振舞いを監視し，検査が冗長である間に限って除去する．除去を適用したあとで，検査が冗長でなくなった場合には，脱最適化を行う．

この最適化の実現にあたり，我々は，標準ライブラリを構成するメソッドのうち，実行頻度が高いものの内部にある冗長な検査の所在を，目視で求め，動的コンパイラに教えた．メソッドの実行頻度は，実用的 Java アプリケーションを構成要素とするベンチマーク SPECjvm98[18] および SPECjbb2005[19] の実行プロファイルから求めた．具体的には，個々のベンチマーク項目について，実行プロファイルから，実行にかかる CPU 時間全体の 0.1%以上を消費しているメソッドを求め，求めたメソッドを調査した．調査を目視で行った理由は，我々の最適化の目的が，既存の技術では冗長と解析できない検査を除去することにあるためである．我々が調査対象のメソッドを使用頻度が高いものに限定した理由は，最適化による高速化の実現と，最適化の実装の手間（目視による調査の手間）を小さく抑えることを両立させるためである．目視による調査は標準ライブラリの更新のたびに必要になるので，調査対象のメソッドを限定することは，最適化の維持管理にかかる手間を軽減するうえでも重要になる．

我々が SPECjvm98 および SPECjbb2005 を対象に，個々のベンチマーク項目の実行中に高頻度で実行されるメソッドについて，それぞれのメソッド内にある冗長な配列添字検査の実行頻度（1 コアが 1 秒間に実行する回数）を求めた結果を**表 1** に示す．

表 1 から，ベンチマークの実行中に冗長な配列添字検査を実行するメソッドは，全部で 16 個であり，その定義元のクラスは，全部で 10 個になることが分かる．我々は，これらのメソッド中の配列参照から冗長な null 検査および配列添字検査を除去する最適化を実現した．実現の概要を次に示す．

(1)　最適化対象のメソッドごとに，`TargetOffsets` 型のデータ構造を用意する．`TargetOffsets` 型の宣言は**図 4** の 3〜6 行目にあり，メソッド `charAt()` に対応するデータ構造の定義は 9 行目にある．このデータ構造は，メソッド内にある配列参照のバイトコードで，null 検査および配列添字検査が冗長なものの所在を記憶する役割を果たす．
　なお，図 4 のプログラムは，最適化の実装の概要を示すために，実装の一部を簡略化して記述したものである．

(2)　最適化対象のメソッドのクラスごとに，手順 (1) で用意したデータ構造へのポインタを格納する配列を用意する．この配列の長さはクラス内で宣言されたメソッドの数と等しく，配列の n 番目の要素は，n 番目に宣言されたメソッドに対応する `TargetOffsets` 型のデータ構造を指示する．クラス `java.lang.String` 向けの配列の定義を 11〜16 行目に示す．

表 1　冗長な配列添字検査の実行頻度
Table 1　Redundant array index test execution frequency.

メソッド名	ベンチマーク項目名							
	_201_ compress	_202_ jess	_209_ db	_213_ javac	_222_ mpegaudio	_227_ mtrt	_228_ jack	jbb 2005
java.lang.String.charAt()	662	1491	103	1842662	47	222287	481539	8
java.lang.String.equals()	14	117113	61476	4558110	23	658	60547	3345
java.lang.String.hashCode()	355	3766	17	2916746	76	179	716333	0
java.lang.String.init([BIII)V	0	0	85662	0	0	0	0	0
java.lang.String.compareTo()	0	0	73	0	0	0	0	0
java.lang.Integer.getChars()	104	261	10	17	13	870	22707	5011
java.lang.Integer.valueOf()	0	0	0	0	0	0	0	1969
java.lang.Long.getChars()	11	29	3	14	19	33	33624	205087
java.util.Hashtable.get()	1	2548662	0	1048538	1	2	148363	0
java.util.Vector.indexOf()	0	140925	122264	632	0	0	1307022	0
java.util.Vector.addElement()	7	12106	11791	11423	0	2	713426	0
java.util.Vector$1.nextElement()	0	0	690176	50444	0	0	29414	0
java.io.DataOutputStream.writeUTF()	0	0	0	112924	0	0	0	0
java.io.DataInputStream.readLine()	0	0	1349	0	0	597849	0	0
java.math.BigDecimal.compareTo()	0	0	0	0	0	0	0	232
java.util.GregorianCalendar.computeFields()	0	0	0	0	0	0	0	34387
合計	1154	2824354	972923	10541510	180	821880	3512975	250039

(3)　コンパイル時には，メソッドのバイトコードをパースして中間表現に変換する際に，まず，パースに先立ち，パース対象のメソッドに対応する TargetOffsets 型のデータ構造があるか検索する．検索を行う関数 targetOffsets() の実装を図 4 の 19〜27 行目に示す．

検索の結果，データ構造が見つかった場合には，パースの過程で配列参照のバイトコードを発見するたびに，検査が冗長かデータ構造に問い合わせる．問合せを行う関数 isRedundantTest() の実装を図 4 の 30〜38 行目に示す．問合せの結果，冗長であると分かった場合には，検査に対応する中間表現の生成を省略する．

手順 (1)〜(3) のうち，手順 (3) は，動的コンパイルの際に実施する処理なので，効率的に実現する方がよい．ただ，手順 (3) で使う関数 targetOffsets()，isRedundantTest() の，図 4 に示した実装は，必ずしも効率的でないが，この実装でも，現状では，次の理由から大きな問題にはなっていない．

- 関数 targetOffsets() の実現の 21，23 行目にある比較の処理は，パース対象のメソッド m のクラス holder が，最適化対象のメソッドを定義するクラスのいずれかにあたるか調査するためのものである．この調査を，21，23 行目にあるように，逐次比較する処理として実現するのは必ずしも効率的でない．しかしながら現状では，この逐次比較から大きな実行時オーバヘッドは生じない．なぜなら，我々の実装では，比較対象のクラス，つまり最適化対象のメソッドを定義するクラスが 10 個しかないからである．
  より多くのクラスを最適化対象とする場合には，もちろん，ハッシュ表などを使うべきである．我々の実験結果によれば，ハッシュ表の検索にかかる時間は，ハッシュ表に，標準ライブラリが含むクラスのうち，配列参照を行うメソッドを定義するものを全部 (3,930 個) 登録した場合でも，26 nsec だった（ハッシュ表のエントリ数は 4,096 個とした）．この時間は，最も小さなメソッド（仮引数をとらない空メソッド）のコンパイルにかかる時間の 0.017%にすぎず，コンパイル時間に悪影響を与えるようなものでない．
- 関数 isRedundantTest() では，33〜35 行目のループで線型探索を行っている．線型探索は，状況によっては，大きな実行時オーバヘッドの発生源となるが，現状では，次

```
 1:  /* データ構造の型宣言:メソッド内にある配列参照のバイトコードで，冗長な検査を含むものの所在を格納 **/
 2:  #define ANY -1
 3:  typedef struct {
 4:    int length; // 配列参照のバイトコードのうち，冗長な検査を含むものの個数．全部冗長な場合は ANY.
 5:    unsigned short* offsets; // 配列参照のバイトコードのうち，冗長な検査を含むもののオフセット群
 6:  } TargetOffsets;
 7:
 8:  /* データ構造の実体の定義 ************************************************************************/
 9:  TargetOffsets charAtOffsets = { ANY, NULL }; // メソッド charAt() 内の所在を格納するデータ構造
10:                                                ...
11:  TargetOffsets* StringOffsetsArray[] = { // クラス java.lang.String 内の所在を格納する配列
12:    NULL,
13:    ...
14:    &charAtOffsets,
15:    ...
16:  };
17:                                                ...
18:  /* メソッド m に対応する TargetOffsets のアドレスを返戻する関数の定義 ***************************/
19:  TargetOffsets* targetOffsets(methodOop m){
20:    klass holder = m->holder_klass(); // m の定義元のクラス holder を取得
21:    if (holder == _string_klass){ // holder が java.lang.String ならば
22:      return (StringOffsetsArray[m->index()]); // StringOffsetsArray から情報を取得
23:    } else if (holder == _int_klass){ // holder が java.lang.Integer ならば
24:       ...
25:    }
26:    return (NULL); // holder が最適化対象のメソッドを含むクラスでないならば NULL を返戻
27:  }
28:                                                ...
29:  /* オフセット bci にある配列参照のバイトコードの検査が冗長か TargetOffsets に問い合せる関数の定義 */
30:  inline bool isRedundantTest(TargetOffsets* to, unsigned short bci) {
31:    if (to != NULL) {
32:      if (to->length == ANY) return (true); /* length が ANY なら真．あるいは */
33:      for(int i=0; i<to->length; i++) /* 配列 to->offsets に bci が記録されていれば真 */
34:        if (bci == to->offsets[i])
35:          return (true);
36:    }
37:    return (false);
38:  }
```

図 4　冗長な検査の所在の記録に関わる定義
Fig. 4　Definitions for recording redundant test locations.

の理由から，大きな問題になっていない．

– 33〜35 行目の線型探索は，そもそも，めったに実行されない．最適化対象のメソッドは全部で 16 個あるが，そのうち 12 個において，メソッド中の配列参照にともなう検査は，すべて冗長だった．このような場合，処理が 31〜32 行目で終わってしまい，33〜35 行目の線型探索が必要にならない．
– 探索が必要になるケースについても，比較対象の要素数 `to->length` が小さく（最大 5，平均 3），どのようなアルゴリズムで探索しても，探索にかかるコストは，コンパイル時間全体に対して十分小さなものになった．

実際，我々が，これらの関数から生じる実行時オーバヘッドによって動的コンパイルの時間がどれだけ延びるか評価したところ，SPECjvm98 および SPECjbb2005 の全ベンチマーク項目を通じて，コンパイル時間の延長率は最大でも 1%，相乗平均で 0.1%だった．

なお，手順 (3) では，冗長な検査に対応する中間表現の生成を省略するが，省略にあたっては，その結果として生じる副作用に配慮する必要がある．具体的には，検査に対応する中間表現がなくなると，コンパイラが添字の値域が非負になることを解析できなくなり，結果として値域を利用する最適化（符号拡張の除去[10] など）が適用できなくなることがある．我々の実装では，この問題を回避するために，添字を表す中間表現に，値が非負であるという情報を付与した．

3. 評　　価

我々が実装した最適化が実行速度にもたらす効果を，SPECjvm98 および SPECjbb2005 を使って評価した．評価は次の 2 つの環境で実施した．

(1) 計算機：PC（CPU: Intel Xeon E5320（1.86 GHz Quad Core）× 2，RAM: 4 GByte），OS: CentOS 5（Intel64 向け），Java 仮想機械：Sun Microsystems 製 Java Development Kit 1.5 に付属のもの（Intel64 向け）
(2) 計算機：PC（CPU: Intel Xeon 3070（2.66 GHz Dual-Core），RAM: 2 GByte），OS: Windows XP（IA32 向け），Java 仮想機械：Sun Microsystems 製 Java Development Kit 1.5 に付属のもの（IA32 向け）

評価結果を**図 5** に示す．図 5 から，Intel64 では相乗平均で 0.65%，IA32 では相乗平均で 1.1%，それぞれ実行速度が向上していることが分かる．2 つの評価環境の実行速度向上率の平均値は 0.87%になる．各ベンチマーク項目ごとに実行速度向上率を比較すると，いずれの環境においても `_228_jack` で特に最適化の効果が現れていることが分かる．これは

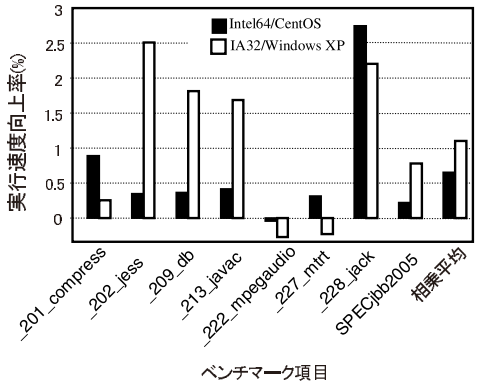

図 5　最適化が実行速度に及ぼす影響
Fig. 5　Optimization effects on performance.

表 1 から分かるように `_228_jack` における冗長な検査の実行頻度が高いことが原因と考える．表 1 から，`_228_jack` のほかに，`_202_jess` や `_209_db`，`_213_javac` でも高い頻度で冗長な検査を実行していることが分かるが，IA32 ではこれらのベンチマーク項目でも他より大きく実行速度が向上している．

4. 関 連 研 究

null 検査や配列添字検査は古くから実行時オーバヘッドの発生源として問題視されており，その除去に向けて数多くの最適化技法が提案されてきた．null 検査や配列添字検査を除去する代表的な方法では，検査が除去可能か否か判定するためにデータフロー解析を使う[1),3)–5),8),9),11),13)–15),17),20),21)]．この方法では，プログラムを静的に調査して，参照の値が null になりうるか，添字の値が例外をおこさない範囲に収まるか解析し，解析結果から冗長と判断できる null 検査や配列添字検査を除去する．データフロー解析を使った最適化は，ループのピーリングやバージョニングといった技法と併用すると，適用機会が増え，有効性が高まることがある．

データフロー解析を使った null 検査や配列添字検査の除去は，多くの Java 向けコンパイラに実装されている最適化だが，必ずしもすべての冗長な検査を除去できるわけではない．たとえば，2 章でとりあげた，メソッド `charAt()` の例では，配列添字検査を除去するために，複数のインスタンス変数の間に成立する関係を見抜く必要があり，`getChars()` の例で

は，複雑な計算式を対象に，非負の整数を 100 で除した剰余が 0〜99 の範囲に収まることを見抜く必要があった．こういったことを見抜くことが可能な解析機能を実現することは，可能かもしれないが，コンパイル時間への配慮が必要な動的コンパイラでは利用しにくい．動的コンパイラでの利用がしにくい場合でも，プログラムの実行を始めるより前に，静的な解析系などで解析を行っておき，解析結果を動的コンパイラに入力して最適化に役立てることはできる．我々の研究では，この静的な解析を手動で行った．解析を手動で行う方法の利点は，解析系の実装をせずに済むことであり，自動的な解析系を使う方法の利点は，大きな規模のプログラムの解析や，標準ライブラリの更新のたびの解析が容易になることである．どちらの方法を使うべきかは，解析対象のプログラムの規模や，解析系の実装規模に依存する．

静的な解析の結果を動的コンパイラに入力する方法には，我々の実装のように，解析結果をコンパイラにハードコーディングする方法のほかに，クラスファイルに指示文を付与する方法がある[2),12),16)]．指示文を付与する方法では，動的コンパイラがクラスファイルに付与された指示文を解釈して最適化を適用する．クラスファイルに指示文を付与する方法の利点としては，解析対象のプログラム（クラスファイル中のメソッド）と，解析結果が乖離しないことを指摘できる．解析結果をコンパイラにハードコーディングする方法では，たとえば，ユーザが標準ライブラリを差し替えると，コンパイラ中の解析結果と，解析対象のクラスファイルが乖離して，解析結果が無効になる．無効な解析結果を使って最適化を行うと，実行結果が不正になるといった問題がおきるので，この問題を回避するために，ハードコーディングを行う場合，標準ライブラリの差替が発生していないか検査する手段が必要になる．我々の実装では，差替の有無をチェックするために，標準ライブラリを構成する jar ファイルの署名を確認している．

一方，クラスファイルに指示文を付与する方法には，次の欠点がある．

- 実行時に指示文を解釈する機能が必要になる．
- ユーザプログラム中のクラスファイルは，どのような内容になっているかが不明であり，コンパイラがその内容を指示文と勘違いして，間違った最適化を適用する恐れがある．

2 つ目の欠点は，指示文の取得対象となるクラスファイルを標準ライブラリ中のものに限定すれば回避できる．ただし，この限定を行うために必要になる機能は，我々が実装した署名を確認する機能に類するものになる．我々の実装では，どちらの方法を採用しても何らかの確認の機能が必要になることと，実行時に指示文を解釈する機能の実装にかかる手間を考慮して，ハードコーディングする方法を採用した．

指示文を使った配列添字検査の除去は Pominville らによって試みられている[16)]．彼らの実装では，`private` もしくは `final` と宣言された参照型のインスタンス変数について，その値が変化しないものと仮定して最適化を適用する．Aggarwal らの技法も同様の仮定を行うが，この仮定は誤りである．なぜならリフレクションやネイティブメソッドを使えば，`private` や `final` といった宣言による制約を無視してインスタンス変数の値を変更できるからである．我々が実装した null 検査や配列添字検査の除去も，リフレクションやネイティブメソッドの影響を受ける点では Pominville らの提案技法と同様である．ただし，我々の実装では，実行時にリフレクションやネイティブメソッドを監視する技法[6),7),22)] を使うことで，最適化を可能にしている．

なお，null 検査については，暗黙の null 検査という技法によって実行時オーバヘッドを軽減させることもできる[9)]．この技法では，実行時オーバヘッドを軽減させるために，null 検査のための比較，分岐命令を省略し，代わりに，ページトラップを使って例外を検出する．この技法を使って図 1 (a) の配列参照を実現すると，その実現にあたる図 1 (b) のコードから，2〜3 行目にある比較と分岐による null 検査がなくなる．2〜3 行目の比較と分岐がなくなった状態で，`byte_array` の値を null にして図 1 (b) の処理を実行すると，6 行目で配列の長さを参照する際にページトラップがおきる[⋆1]．暗黙の null 検査では，このページトラップの発生をきっかけに，`NullPointerException` を生成し，投げる処理を行う．

暗黙の null 検査は，比較と分岐の省略によって，null 検査の実行時オーバヘッドを軽減させる技術ではあるが，皆無にする技術ではない．なぜなら，暗黙の null 検査はメモリ参照命令を分岐として使う技術であって，分岐を除去する技術ではないからである．分岐はコードスケジューリングなどの最適化の適用範囲を制限し，実行時オーバヘッドの発生源となる．このため，暗黙の null 検査を利用している Java 実行環境向けの動的コンパイラでも，最適化によって冗長な null 検査の除去を試みる場合が多い[9),14)]．実際，我々が評価に利用した Java 実行環境は，データフロー解析や我々が提案した機能による null 検査の除去と，暗黙の null 検査の双方を利用する[14)]．

⋆1 配列中にある長さを格納するフィールド `length` のオフセットと，null を実現する値の和がページサイズより小さく，なおかつ，論理アドレス空間の先頭のページを読み込み禁止に保護している場合．いくつかのオペレーティングシステムは，デバッグの支援などを目的として，論理アドレス空間の先頭のページを読み込み禁止に保護する．

5. ま と め

5.1 結　　論

Java における配列参照の高速化を目的として，標準ライブラリ中の配列参照にともなう null 検査および添字検査を除去する技法を提案した．提案技法では，標準ライブラリ中にある冗長な null 検査および添字検査を目視によって静的に求め，動的コンパイルの際に除去する．検査の中には冗長になるか否かがリフレクションやネイティブメソッドの挙動によって変化するものもあるが，提案技法では，それらを除去可能にするために，リフレクションやネイティブメソッドの動的な振舞いを監視する．SPECjvm98 および SPECjbb2005 を使った評価の結果から，提案技法により，実行速度を平均で 0.87%高速化できることが分かった．

5.2 今後の課題

今後の課題としては，次の事項がある．

- 本論文では SPECjvm98 および SPECjbb2005 の実行上，重要なメソッドを最適化対象とし，その効果を SPECjvm98 および SPECjbb2005 を使って評価したが，他のプログラムでも同じ効果が得られるとは限らない．今後，より多くの実用的なアプリケーションを用いて，本最適化が実行速度に及ぼす影響を評価する必要がある．
- 提案技法では最適化対象の検査を目視で求めたが，この作業を自動化できれば，標準ライブラリの更新の際に目視による調査を行う必要がなくなる．自動化によってどこまで最適化対象の検査を求められるか検討する必要がある．

謝辞　本研究は，文部科学省の科学振興調整費振興分野人材養成による委託業務「情報セキュリティ・情報保証人材育成拠点」，および，21 世紀 COE プログラム「電子社会の信頼性向上と情報セキュリティ」の一環として行われた．

参 考 文 献

1) Aggarwal, A. and Randall, K.H.: Related Field Analysis, *Proc. ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pp.214–220 (2001).
2) Azevedo, A., Nicolau, A. and Hummel, J.: Java Annotation-Aware Just-In-Time (AJIT) Complilation System, *Proc. ACM 1999 conference on Java Grande*, pp.142–151 (1999).
3) Bodík, R., Gupta, G. and Sarkar, V.: ABCD: Eliminating Array Bounds Check on Demand, *Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp.321–333 (2000).
4) 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).
5) Harrison, W.H.: Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis, *IEEE Trans. Softw. Eng.*, Vol.3, No.3, pp.243–250 (1977).
6) Hirzel, M., Dincklage, D.V., Diwan, A. and Hind, M.: Fast Online Pointer Analysis, *ACM Trans. Programming Languages and Systems*, Vol.29, No.2, Article11 (2007).
7) Hirzel, M., Diwan, A. and Hind, M.: Pointer Analysis in the Pressence of Dynamic Class Loading, *Proc. 18th European Conference on Object-Oriented Programming*, pp.96–122 (2004).
8) Juján, M., Gurd, J.R., Freeman, T.L. and Miguel, J.: Elimination of Java Array Bounds Checks in the Presence of Indirection, *Proc. 2002 Joint ACM-ISCOPE Conference on Java Grande*, pp.86–95 (2002).
9) Kawahito, M., Komatsu, H. and Nakatani, T.: Effective Null Pointer Check Elimination Utilizing Hardware Trap, *ACM SIGPLAN Notices*, Vol.35, No.11, pp.139–149 (2000).
10) Kawahito, M., Komatsu, H. and Nakatani, T.: Effective Sign Extension Elimination for Java, *ACM Trans. Programming Languages and Systems*, Vol.28, No.1, pp.106–133 (2006).
11) Kolte, P. and Wolfe, M.: Elimination of Redundant Array Subscript Range Checks, *Proc. ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pp.270–278 (1995).
12) Krintz, C. and Calder, B.: Using Annotations to Reduce Dynamic Optimization Time, *Proc. ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pp.156–167 (2001).
13) Markstein, V., Cocke, J. and Markstein, P.: Optimization of Range Checking, *Proc. 1982 SIGPLAN Symposium on Compiler Construction*, pp.114–119 (1982).
14) Paleczny, M., Vick, C. and Click, C.: The Java HotSpot Server Compiler, *Proc. Java Virtual Machine Research and Technology symposium*, pp.1–12 (2001).
15) Patterson, J.R.C.: Accurate Static Branch Prediction by Value Range Propagation, *Proc. ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pp.67–78 (1995).
16) Pominville, P., Qian, F., Vallée-Rai, R., Hendren, L. and Verbrugge, C.: A Framework for Optimizing Java using Attributes, *Proc. 2000 Conference of the Centre for Advanced Studies on Collaborative Research*, pp.152–169 (2000).

17) Rugina, R. and Rinard, M.: Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions, *Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp.182–195 (2000).
18) Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks (1998). http://www.spec.org/jvm98/
19) Standard Performance Evaluation Corporation: SPECjbb2005 (2005). http://www.spec.org/jbb2005/
20) Suzuki, N. and Ishihata, K.: Implementation of an Array Bound Checker, *Proc. 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp.132–143 (1977).
21) Xu, Z., Miller, B.P. and Reps, T.: Safety Checking of Machine Code, *Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp.70–82 (2000).
22) 千葉雄司：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，情報処理学会論文誌：プログラミング，Vol.49, No.SIG1(PRO35), pp.103–116 (2008).

(平成 19 年 12 月 21 日受付)
(平成 20 年 3 月 24 日採録)

柳　　優
2008 年中央大学大学院理工学研究科情報工学専攻修士課程修了．同年日本アイ・ビー・エム（株）入社．

田中　俊之
2008 年中央大学大学院理工学研究科情報工学専攻修士課程修了．同年（株）ACCESS 入社．

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．

土居　範久（フェロー）
中央大学理工学部教授，慶應義塾大学名誉教授．現在，日本学術会議副会長，文部科学省科学技術・学術審議会委員，総務省情報通信審議会委員・座長代理，文部科学省「次世代 IT 基盤構築のための研究開発」プログラムディレクター，科学技術振興機構（JST）社会技術研究開発センター（RISTEX）「情報と社会」領域総括，科学技術振興機構（JST）戦略的創造研究推進事業研究総括，NPO 法人日本セキュリティ監査協会会長，国際計算機学会（ACM）日本支部長．専門はソフトウェアを中心とした計算機科学および情報セキュリティ．情報処理学会名誉会員・フェロー，日本ソフトウェア科学会フェロー．