日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

文字列処理高速化のための Java 処理系の改良

河内谷 清久仁　緒方 一則　小野寺 民也

String は Java の基本データ構造の一つであるが，国際化などを標準でサポートするために，汎用かつ高機能なものとなっている．しかし，String が扱うデータは多くの場合単純な ASCII 文字列であり，その汎用性が処理性能の低下をまねいている可能性がある．本発表では，String の汎用性が引き起こす性能上の問題について調査・分析し，それを解決するための手法を提案する．また，商用 Java 処理系へのプロトタイプ実装と，それを用いた性能評価についても示す．提案する手法は Java 処理系とそれに付随するクラスライブラリ内で実現されているため，既存の Java アプリケーションを変更することなく文字列処理の高速化が実現できる．

1　はじめに

Java [10] は，いまや最もよく利用されているプログラミング言語の一つ [27] で，大規模 Web アプリケーション [25] のほか，Eclipse [8] などの統合開発環境（IDE）や，最近では他の言語処理系 [6] [12] [15] [16] [22] の実行環境としても用いられている．これらの Java アプリケーションでは，文字列処理が頻繁に行われる．文字列データの処理のために，Java では String，StringBuffer，StringBuilder などのクラスを基本データタイプとして提供している [7]．これらのクラスは文字コードとして Unicode [28] を採用しているため，国際化などを行いやすいという利点があるが，すべての処理が Unicode を前提とした汎用かつ高機能なものとなっており，処理が重いという問題がある．

たとえば，String.toLowerCase は文字列を「小文字」に変換するメソッドであるが，Java の String は任意の Unicode 文字を含むことができるため，ロケール [30] に従った複雑な条件文や変換表を用いて，1 文字ずつ処理を行わなければならない．

しかし，Java アプリケーションで実際に使用される String の多くは単純な ASCII 文字列であると考えられる†1．そのような「大多数のケース」について，汎用的で複雑な処理を行わず，ASCII 文字列であることを前提とした処理を行えれば，文字列処理を高速化できる可能性がある．本論文では，String などの基本文字列クラスに，このような文字列の「特性」を示すフラグを追加することで文字列処理を高速化する「StringFlags」手法を提案し，商用 Java 処理系への実装と評価についても述べる．

本論文の主な貢献点は，以下の 3 点である．

- **実 Java アプリケーションでの String の使われ方の調査．**複数の Java アプリケーションについて，String がどれくらい使われているか，またその中身にどのような性質が見られるかの調査を行った．特に，英語以外のロケールにおいても，生成される String の 97%以上が ASCII 文字列であることを確認した．
- **Java の文字列処理の高速化手法の提案．**上記調査に基づき，String データにその特性を示すフラグを付加し処理を高速化する StringFlags 手法

Java VM Improvement for Faster String Processing.
Kiyokuni Kawachiya, Kazunori Ogata, and Tamiya Onodera, 日本アイ・ビー・エム（株）東京基礎研究所, IBM Research - Tokyo.

†1 本論文では，Unicode 値が 0 から 127 の文字を「ASCII 文字」[29]，ASCII 文字のみから構成されている文字列を「ASCII 文字列」と呼ぶ．

```
1 public final class String ... { // only has private fields
2   private char[] value;         // char array to hold the value
3   private int offset;           // start offset in the char array
4   private int count;            // length of the string value
5   private int hashCode;         // hash code of this object, or 0
6       :
7 }
```

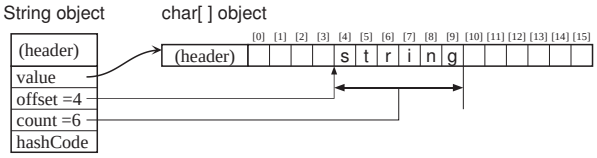


図 1　典型的な実装における `String` オブジェクトの構造．

のデザインを示した．提案手法は Java 処理系とそれに付随するクラスライブラリ内で実現されるため，既存の Java アプリケーションを変更することなく文字列処理を高速化できる．

- **商用 Java 処理系へのプロトタイプ実装と評価．** `String` の内容が ASCII 文字列である場合について上記フラグを付加し，処理の高速化を行うプロトタイプを商用 Java 処理系に実装し，評価を行った．ASCII 文字列の 87%にフラグがセットされ，文字列処理は最大 4.4 倍高速化された．

2　Java の文字列処理

本節では，Java が提供する文字列処理について概観する．まず，文字列に関するデータ構造を説明し，次に，文字列処理の例と問題点について示す．さらに，予備実験を通じて解決の可能性を示す．

2.1　文字列関連クラス

Java において最も基本となる文字列データ構造は，「`String`」である．`String` クラスは，文字列を操作するための様々なメソッドを提供している．これには，文字列の一部を取り出し新しい `String` を返す `substring` や，小文字に変換する `toLowerCase`，バイト列に変換する `getBytes` などがある．`String` はまた，多くのメソッドで引数や返り値の型として用いられている．一例として，データベースにアクセスするための JDBC メソッド群 [9] があげられる．

Java ソースコード中に文字列を定数（リテラル）として記述した場合，クラスファイルには UTF-8 エンコードされた情報として保持される [19]．`ldc` (load constant) バイトコードによってその情報を読み出すと，Java 処理系がそれを Unicode（UTF-16）エンコードに変換し，`String` オブジェクトを生成する．

Java の言語仕様 [10] では，`String` をどのように実装するかまでは規定されていない．しかし，`String` の典型的な実装は図 1 のようになっている．これは，Harmony クラスライブラリ [2] での実装を簡略化したものであるが，我々が知る限りほとんどの Java 処理系で同様の実装が用いられている．一つの文字列は 2 つのオブジェクト（`String` と `char` 配列）によって表される．`String` オブジェクトは 4 つのフィールドを持ち，このうち `value` フィールドが，文字データの実体を保持する `char` 配列オブジェクトを指している．この `char` 配列の，`offset` と `count` フィールドで指定される範囲が，`String` オブジェクトの表す文字列となる．Java では，`String` オブジェクトのハッシュ値はその内容から計算するように規定されており [26]，一旦計算したハッシュ値を保持しておくために `hashCode` フィールドが用いられる．なお，図の各オブジェクト中の「header」は，オブジェクトのクラス情報などを管理するために Java 処理系が内部的に使用する領域である．

Java では `String` で表された文字列は変更不能（immutable）で，変更するには新たな `String` オブ

```
 1 class StrOpsTest {
 2   public static void main(String[] args) {
 3     String s = "ABab\uFF21\uFF22\uFF41\uFF42\u3042\u3044"; // 10 chars
 4     String l = s.toLowerCase();
 5     byte[] b = l.getBytes();
 6         :
 7     /* Dump the contents of s, l, b (code omitted) */
 8         :
 9   }
10 }
```

```
s: "ABabＡＢａｂあい"
l: "ababａｂａｂあい"
```

```
Results: (for LANG=ja_JP)
 s: "\u0041\u0042\u0061\u0062\uFF21\uFF22\uFF41\uFF42\u3042\u3044"
 l: "\u0061\u0062\u0061\u0062\uFF41\uFF42\uFF41\uFF42\u3042\u3044"
 b: { 0x61, 0x62, 0x61, 0x62, 0xA3,0xE1, 0xA3,0xE2, 0xA3,0xE1, 0xA3,0xE2,
      0xA4,0xA2, 0xA4,0xA4 } (16 bytes)
```

図 2　単純な文字列操作プログラムでも，内部では複雑な処理が行われている例．

ジェクトを生成する必要がある．一方，変更可能な文字列データ構造として，StringBuffer と StringBuilder が提供されている．後者は JDK 1.5 から導入された新しいクラスで，単一スレッドのみの使用を前提に同期処理を省き高速化したものである．これらのクラスのオブジェクトに対しては，append メソッドで文字列を追加するなどの変更操作も可能だが，操作は各クラスに予め用意されているメソッド経由のものに限られる．

これら 3 つの「文字列関連クラス」の各文字は 16 ビットの char 型で，Unicode（UTF-16）[28] でエンコードされている．これにより，Java では国際化対応が比較的容易に行える．しかし一方で，すべての文字列処理が，Unicode を前提とした汎用かつ高機能なものとなっており，処理が重いという問題が起こりえる．

図 2 はその一例で，文字列を小文字にし，バイト列として取り出すという単純な文字列操作プログラムと，それを LANG=ja_JP ロケールで実行した場合の結果である．この結果は，toLowerCase や getBytes が，Java においては必ずしも単純な処理ではないことを示唆している．たとえば，Unicode において'\uFF21' は全角文字の「A」を表しているため，toLowerCase では対応する小文字の「ａ」（'\uFF41'）に変換されている．また，getBytes によるバイト列への変換も，単純な数式による処理が行えず，1 文字ずつ変換表をチェックしなければならない．さらに，文字によって 1 文字が何バイトになるかが異なるために，得られる byte 配列のサイズを事前に確定できず，余計なオブジェクト生成やデータコピーが行われてしまう．

2.2　高速化の可能性

図 2 では，Unicode 文字列の処理はそれほど単純ではないということを示した．しかし一方，Java プログラムで HTML や XML のタグ情報を操作したり，JDBC インタフェース [9] を用いてデータベース処理を行ったり，言語処理系を実装 [6] [12] [15] [16] [22] したりする場合，これらの処理の対象となる文字列の多くは単純な ASCII 文字列であると考えられる．もしそうなら，多くの場合で，Unicode を考慮した複雑な文字列処理は不要だということになる．

この仮説を検証するため，Java 処理系のオブジェクト生成部を修正し，表 1 に示した Java プログラムについて，実行中に生成された String の数と，そのうち全文字が ASCII 文字であったものの割合などを調査した．用いた Java 処理系は，IBM の商用 Java 処理系である J9 Java VM [4] [11] 6.0 for Linux である．この実験では，new や newarray バイトコードを通さず Java 処理系が暗黙のうちに生成するオブジェクト（ldc による文字列リテラルからの String オブジェクト生成や，リフレクションによるもの）も集計している．

表 1　実験に用いた Java プログラム．

プログラム名	計測方法
DaCapo (`-n 1`)	DaCapo ベンチマーク [5]（バージョン 2006-10-MR2）の 11 個のプログラムを個別に実行．問題サイズはデフォルト，繰り返し回数はそれぞれ 1 回（`-n 1`）とし，ベンチマークの起動から終了までの全体を計測.
WAS start+warmup および WAS processing	IBM の商用 Java EE [25] サーバ WebSphere Application Server (WAS) [14] 7.0 上で，DayTrader 2.0 アプリケーション [1] を実行．10 のクライアントからランダムリクエストを各 1,000 回（計 10,000 回）行うシナリオを用意し，WAS 起動から 1 回目のシナリオ処理までを「start+warmup」，2 回目のシナリオ処理を「processing」として個別に計測．前者には，WAS の初期化，メッセージの読み込み，クラスロードや JIT コンパイルなどの処理が含まれる．WAS は国際化されたアプリケーションなので，`en_US`，`fr_FR`，`ja_JP` の 3 種類のロケールについて調査.
SPECjvm2008 Installer	SPECjvm2008 ベンチマーク [23]（バージョン 1.01）のインストーラ．このプログラムは国際化されており，指定したロケールに応じて表示されるメニュー文字列などが変化する．`en_US`，`fr_FR`，`ja_JP` の 3 種類のロケールについて，インストーラの起動から終了までの全体を計測.

表 2　様々な Java アプリケーションでの `String` オブジェクトの数と，ASCII 文字列の割合．
国際化されたアプリケーションでも，生成される `String` の 97%以上が単純な ASCII 文字列である．

プログラム名	生成された オブジェクト数 (x1,000)	生成された `String` の数 (x1,000)	ASCII 文字列 の割合
DaCapo (`-n 1`)			
antlr	3,903	546 (14.0%)	100.0%
bloat	22,347	4,897 (21.9%)	100.0%
chart	26,779	10,446 (39.0%)	100.0%
eclipse	47,913	1,549 (3.2%)	100.0%
fop	1,253	315 (25.1%)	99.5%
hsqldb	4,524	178 (3.9%)	100.0%
jython	20,296	2,503 (12.3%)	99.3%
luindex	12,486	2,421 (19.4%)	100.0%
lusearch	18,345	1,790 (9.8%)	100.0%
pmd	29,394	152 (0.5%)	100.0%
xalan	5,550	425 (7.7%)	100.0%
(geometric mean)	11,535	1,049 (9.1%)	99.9%
WAS start+warmup			
`LANG=en_US`	43,689	6,880 (15.7%)	100.0%
`LANG=fr_FR`	42,679	6,505 (15.2%)	99.9%
`LANG=ja_JP`	44,109	6,874 (15.6%)	99.9%
WAS processing			
`LANG=en_US`	32,410	4,340 (13.4%)	100.0%
`LANG=fr_FR`	32,390	4,327 (13.4%)	100.0%
`LANG=ja_JP`	32,425	4,338 (13.4%)	100.0%
SPECjvm2008 Installer			
`LANG=en_US`	1,033	223 (21.6%)	99.7%
`LANG=fr_FR`	1,019	224 (22.0%)	97.7%
`LANG=ja_JP`	1,034	229 (22.2%)	97.5%

調査結果を表 2 に示す．調査した Java プログラムの半数以上で，生成されるオブジェクトの 10%以上を `String` オブジェクトが占めていることがわかる．この集計には，文字列の内容を保持する `char` 配列や，`StringBuffer` および `StringBuilder` オブジェクトは含めていないので，文字列を保持するために作られるオブジェクトの割合はさらに多いはずである．そして，国際化されたアプリケーションを英語以外のロケールで実行しても，生成された全 `String` オブジェクトの 97%以上が ASCII 文字列であった．

```
 1 public final class String ... {
 2   private char[] value; // char array to hold the value
 3   private int offset;   // start offset in the char array
 4   private int count;    // length of the string value
 5       :
 6   String toLowerCaseForASCII() {
 7     char[] chars = new char[count];
 8     boolean modified = false;
 9     for (int i = 0; i < count; i++) {
10       char c = value[offset + i];
11       if ('A' <= c && c <= 'Z') {
12         c += ('a' - 'A'); modified = true;
13       }
14       chars[i] = c;
15     }
16     if (!modified) return this;
17     return new String(0, count, chars); // chars is directly used
18   }
19       :
20   byte[] getBytesForASCII() {
21     byte[] bytes = new byte[count];
22     for (int i = 0; i < count; i++)
23       bytes[i] = (byte)value[offset+i]; // just use low 8 bits
24     return bytes;
25   }
26       :
27 }
```

図 3 `String.toLowerCase` と `getBytes` の，ASCII 文字列特化版．元のコードより単純で高速である．

前述したとおり，Java の文字列操作は Unicode を前提として作成されているため，処理が複雑である．しかし，文字列が ASCII のみであることが事前にわかっていれば，高速な処理を行える．図 3 は，そのような処理コードの一例である．`toLowerCaseForASCII` は，文字が`'A'` から`'Z'` の範囲の場合に `0x20` を加えればよい（11–12 行目）．また，`getBytesForASCII` は，各文字の下位バイトを単純に集めてバイト列を作ればよい（23 行目）．ここでは，複雑な条件分岐や変換表の参照，余計なオブジェクト生成は不要である．また，このように単純なメソッドに対しては，JIT コンパイラによるインラインなどの最適化がより有効に機能することも期待できる．

3 文字列処理の高速化

前節では，Java の文字列処理は汎用的である分オーバーヘッドがあるが，ほとんどの `String` データは単純な ASCII 文字列であり，それに特化したコードを用いれば処理を高速化できる可能性があることを示した．本節では，実際に高速化を行うための Java 処理系の修正について述べる．

3.1 「フラグ」の導入

ASCII 文字列であることを前提とした高速処理を行うナイーブな方法としては，図 3 で示した高速版メソッドを明示的に呼ぶことが考えられる．しかし，この方法では，既存のアプリケーションの書き直しが必要となる上，文字列が ASCII 以外の文字を含まないことをアプリケーション側で保証しなければならない．また，API の仕様拡張が必要となるため，特に Java では現実的とはいえない．

我々が採った方法は，文字列関連クラスのオブジェクトに内部的に「フラグ」を追加し，保持している文字列の特性（たとえば「ASCII 文字のみからできている」）を示せるようにすることである．このフラグがセットされている `String` に対して文字列処理を行

表 3 String に付加するフラグの例と，高速化できる処理．

フラグ名	意味する特性	高速化できる処理
IS_ASCII	すべての文字が ASCII（値が 0–127）	toLowerCase, toUpperCase, getBytes, ...
ISNT_REGEX	正規表現の特殊文字を含まない	split, replaceAll, replaceFirst, ...
IS_LOWER	大文字を含まない	toLowerCase
IS_UPPER	小文字を含まない	toUpperCase
IS_INTERNED	インターン済みである	intern

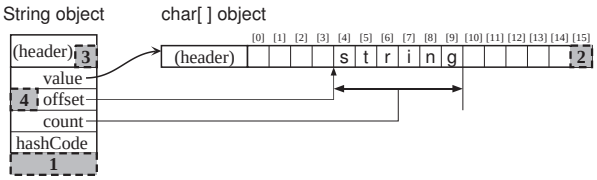


図 4 フラグを追加する場所の候補．プロトタイプは方式 4 を採用しているが，コード例では方式 1 を用いている．

う場合，従来の汎用版に代えてその特性を利用した高速版（たとえば図 3）を実行することで，処理を高速化できる．我々はこの手法を「StringFlags」アプローチと名づけた．

なお，2.1 節でも述べたように，各 String オブジェクトの値は生成された後は変更されることがない．StringBuffer や StringBuilder の保持する文字列は変更可能であるが，そのためには，各クラスが提供するメソッドを経由する必要がある．そのため，フラグの示す特性が知らぬ間に正しくなくなるということは起こらない．

String の特性を示すフラグの種類として，ここまでは「ASCII 文字列であること（IS_ASCII）」を例として説明してきた．しかし実際には，これ以外にも高速化に役立ちそうな特性が考えられる．表 3 はそのうちいくつかについて，使用したい特性と，高速化が期待できる処理について示したものである．たとえば，ISNT_REGEX は正規表現の特殊文字（\^$.?*+|[]{}()）が含まれないことを示すフラグで，これを利用することで String.split("/") のような処理が高速化できる．なぜなら，引数の String（この例では"/"）にこのフラグがセットされていた場合，正規表現のパターンマッチャーを生成しなくてよいからである．

3.2 フラグの追加場所

提案する StringFlags 手法では，各文字列オブジェクトごとに，それが保持している文字列の特性を示す複数のフラグを用意する必要がある．しかし，予備実験の結果（表 2）からもわかるように，文字列オブジェクトは非常に多く使用されるため，フラグ保持のためにオブジェクトのサイズを大きくすることはなるべく避けたい．そこでまず，フラグをどこに置くかについて考える．

図 4 は，図 1 に示した構造をもつ String オブジェクトのどこにフラグを追加するかについて検討したものである．まず，String オブジェクトにフィールドを追加するというナイーブな方式（方式 1）では，オブジェクトのサイズが増加してしまう．char 配列の未使用スロットを使用する方式（方式 2）も考えられるが，方式 1 と同様オブジェクトのサイズを増加させる危険性がある．ヘッダ領域に空きがあればそこを利用する（方式 3）ことも可能だが，使用できるフラグの数が制限されてしまうのと，Java プログラムからフラグに直接アクセスできないという点が問題となる．また，String 以外のオブジェクトにもフラグ領域が用意されてしまうことになり，効率が悪い．これらのことから我々は，フラグを String オブジェクトの既存フィールドに埋め込む方式（方式 4）を採ることにした．

String オブジェクトのフィールドはいずれも private であり，String クラスの外からは直接参照・操作できない．そのため，ここにフラグを埋め込むことは可能である．ただし，value フィールドは char 配列へのオブジェクト参照なので，ここに別のデータを埋め込むと GC に修正が必要となってしまう．また，String のハッシュ値は計算法が規定されており [26]，それを保持する hashCode フィールドには空きがない．一方，Java では配列の最大長は $2^{31}-1$ (Integer.MAX_VALUE) であり，32 ビット Java 処理系の場合でも，offset と count フィールドの最上位

ビットは常にゼロである．特に，offset フィールドは，大きな値になることは稀である[†2]と考えられるので，プロトタイプでは offset フィールドの上位ビットをフラグとして用いることにした[†3]．

3.3 フラグのセットと伝播

次に，このフラグをいつどのようにセットするかについて議論する．ここで注意すべき点は，フラグをセットし管理するために多くのコードが実行されてしまっては，かえって処理が遅くなる危険性があるということである．そのためフラグは，従来の文字列処理の「ついで」に低コストで条件が調査できる場合に限ってセットすることとした．これはつまり，フラグがセットされていれば条件が満たされていることが保証されるが，条件が満たされていてもフラグがセットされていない場合があるということである．

具体的には，String 中の文字を全部チェックするような処理に，フラグの条件を調査し，セットするコードを追加する．また，生成される文字の種類が事前にわかる処理でもフラグをセットしている．これらの例としては，以下の処理があげられる．

- リテラルから String を生成する処理.
  Java 処理系がバイトコード「ldc」を処理する際に，コンスタントプール中の UTF-8 文字列を走査し，Unicode（UTF-16）に変換する処理が行われる．この時同時にフラグの条件を調査し，生成された String オブジェクトに結果をセットする．
- 文字列全体を走査するメソッド.
  例としては，String.toLowerCase，toUpperCase，getBytes，hashCode（新たに計算する場合），regionMatches（全部調べた場合）などがあげられる．
- 生成文字列の特性が予めわかっているメソッド.
  たとえば，String.valueOf(float) によって生成される String で使用される文字はすべて ASCII 文字であるので，IS_ASCII フラグをセット可能である．

図 5 は，ハッシュ値を求める String.hashCode および，float 値を文字列に変換する String.valueOf(float) 処理でフラグをセットする例で，傍線のある行がフラグ処理のために追加されたコードである．

さらにユニークな工夫として，文字列処理中にフラグを「伝播」することで，フラグが新しい文字列オブジェクトになるべく引き継がれていくようにしている．文字列の操作は StringBuffer や StringBuilder を経由して行われることも多いので，これらのオブジェクトにも同様のフラグを追加し，append などの処理の際にはフラグを伝播するようにする．

図 6 はこのフラグ伝播の例を示したものである．フラグが引き継げるかどうかは，フラグが表す特性と，文字列操作の種類によって決まる．たとえば，String.substring は，部分文字列をとる操作なので，表 3 にあげたフラグのうち，IS_INTERNED 以外の 4 つについてはそのまま引き継ぐことができる（5 行目）．また，StringBuilder.append では，元の文字列と追加する文字列の両方が満たしているフラグだけが引き継がれる（17 行目）．

3.4 フラグの使用

最後に，セットしたフラグの使用法であるが，基本的には表 3 に示した各フラグによって高速化できる処理について，フラグがセットされていた場合に用いられる高速版処理コードを用意すればよい．図 7 は，IS_ASCII フラグによって toLowerCase 処理を高速版（図 3）に切り換える例である．なお，トルコ語などの特殊なロケールでは，ASCII 文字列であっても図 3 の高速版が使用できないことがある [32]．6 行目の追加チェックは，そのような特殊ロケールを排除するためのものである．

以上，本節では，文字列オブジェクトにフラグを追加することで，典型的なケースを高速化する手法

†2 実際，2.2 節で行った予備実験において，生成された全 String オブジェクトのうち最大の offset は 7,662 であった．つまり，19 ビット程度をフラグに割り当てても問題ないといえる．理論上は offset がさらに大きな値になることも考えられるが，その場合は char 配列を作り直すことで対処できる．

†3 わかりやすさを優先するため，コード例（図 5 など）では，フラグ保持用に flags という新しいフィールドを追加して説明している．

```
 1 public final class String ... {
 2   private char[] value; // char array to hold the value
 3   private int offset;   // start offset in the char array
 4   private int count;    // length of the string value
 5   private int hashCode; // hash code of this object, or 0
|6   private int flags;    // separately added for easy explanation
 7       :
 8   public int hashCode() {
 9     if (hashCode == 0) {
10       int hash = 0;
|11      char check = 0;
12       for (int i = 0; i < count; i++) {
13         char c = value[offset+i];
14         hash = hash * 31 + c;
|15        check |= c;     // check the condition
16       }
17       hashCode = hash;
|18      if (check < 128) flags |= IS_ASCII; // set the flag
19     }
20     return hashCode;
21   }
22       :
23   public static String valueOf(float f) {
24     String newStr = Float.toString(f);
|25     newStr.flags |= IS_ASCII; // set the flag
26     return newStr;
27   }
28       :
29 }
```

図 5　通常の文字列処理の「ついで」にフラグをセットする例.

```
 1 public final class String ... {
 2       :
 3   public String substring(int begin, int end) {
 4     String newStr = ...; // substring processing
|5     newStr.flags = flags & ~IS_INTERNED; // propagate flags
 6     return newStr;
 7   }
 8       :
 9 }
10
11 public final class StringBuilder ... {
12       :
13   public StringBuilder append(String str) {
14       :
15     /* append processing */
16       :
|17    this.flags &= str.flags; // merge flags
18     return this;
19   }
20       :
21 }
```

図 6　フラグを「伝播」する例.

```
 1 public final class String ... {
 2       :
 3   public String toLowerCase() {
 4     String lang = Locale.getDefault.getLanguage();
 5     if ((flags & IS_ASCII) != 0 &&            // if the flag is set...
 6     !lang.equals("tr") && !lang.equals("az") && !lang.equals("lt")) {
 7       String lowerStr = toLowerCaseForASCII(); // switch to the fast path
 8       lowerStr.flags |= IS_ASCII;              // and propagate the flag
 9       return lowerStr;
10     }
11     // Original (generic but slow) toLowerCase processing
12     return toLowerCaseOriginal();
13   }
14       :
15 }
```

図 7　フラグの使用例．フラグがセットされていたら高速版のコードに分岐している．

```
1  String s1 = "ABC";                  // flag is set
2  String s2 = String.valueOf(1.2f);   // flag is set
3  String s3 = s1 + s2;                // flag is propagated
       // Compiled as: new StringBuilder().append(s1).append(s2).toString();
4  String s4 = s3.substring(1, 4);     // flag is propagated
5  String s5 = s4.toLowerCase();       // flag is used, and propagated
```

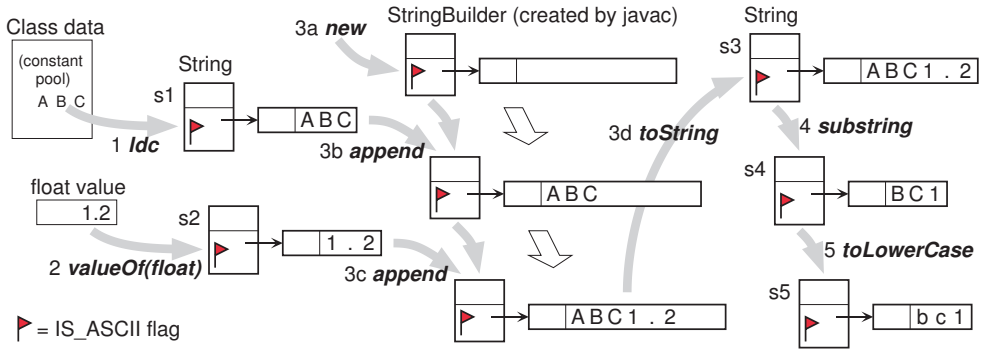


図 8　フラグのセット・伝播・使用の流れ．すべての `String` に `IS_ASCII` フラグがセットできている．

「StringFlags」について提案した．図 8 は，フラグのセット・伝播・使用の流れを `IS_ASCII` フラグを例にまとめたものである．文字列 `s1` については，リテラルから `String` オブジェクトを生成する際に内容がチェックされ，フラグがセットされる．`s2` については，`String.valueOf(float)` の結果は必ず ASCII 文字列となるので，フラグがセットされる．これらの文字列を連結するには，`StringBuilder.append` メソッドが使われるが，いずれの文字列にもフラグがセットされているので，生成された文字列 `s3` にもフラグが伝播される（図 6）．同様に `s4` にもフラグが伝播され，`String.toLowerCase` の処理が高速化される（図 7）．最後に生成された文字列 `s5` にも，フラグは伝播されている．

提案する高速化手法の最大の特長は，Java 処理系および標準クラスライブラリ内の修正だけで実現可能だという点である．フラグは Java 処理系内部で自動的にセット・伝播・使用されるため，従来の Java アプリケーションは一切変更することなく高速化できる．

```
1 class StrOpsBench {
2   public static void main(String[] args) {
3     String s1 = "ABCDEabcde";                    // 10 chars
4     String s2 = "\uFF21\uFF22\uFF23\uFF24\uFF25
                   \uFF41\uFF42\uFF43\uFF44\uFF45"; // 10 chars
5     // Try multiple times for JIT optimization
6     doTest("ASCII", s1); doTest("non-ASCII", s2);
7     doTest("ASCII", s1); doTest("non-ASCII", s2);
8       :
9   }
10  doTest(String name, String s) {
11    long c = 0; System.gc();
12    long t0 = System.currentTimeMillis();
13    for (int i = 0; i < 1000000; i++) {
14      String l = s.toLowerCase();  // Test1: toLowerCase
15      c += l.length(); // to avoid code elimination by JIT
16    }
17    long t1 = System.currentTimeMillis();
18    for (int i = 0; i < 1000000; i++) {
19      byte[] b = s.getBytes();     // Test2: getBytes
20      c += b.length;   // to avoid code elimination by JIT
21    }
22    long t2 = System.currentTimeMillis();
23    System.out.printf("%s: toLowerCase=%dms, getBytes=%dms, c=%d\n",
24                      name, t1-t0, t2-t1, c);
25  }
26 }
```

図 9　調査に用いたマイクロベンチマーク．ASCII 文字列と非 ASCII 文字列について同じ処理を行い実行時間を計測．

4　実装と評価

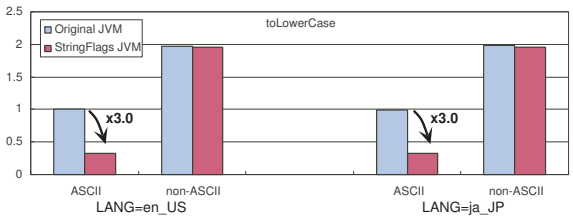
前節で提案した StringFlags 手法の有効性を評価するため，2.2 節の予備実験でも用いた IBM J9 Java VM (JVM) 6.0 for Linux 上にプロトタイプを実装した．このプロトタイプでは,「文字列が ASCII のみで構成されている」ことを示すフラグ（表 3 の `IS_ASCII` に相当）だけを用意し，`offset` フィールドの最上位ビットをそのために割り当てている．

フラグのセットは，3.3 節でも議論したように，リテラルから `String` オブジェクトを生成する部分と文字列全体を走査している部分，生成される文字列が ASCII であることがわかっている部分で行っている．ただし，`toLowerCase`，`toUpperCase`，`getBytes` のオリジナル版は処理が複雑すぎるため，プロトタイプではフラグセットのコードを追加していない．フラグの伝播は，`String`，`StringBuffer`，`StringBuilder` クラスの各メソッドで可能な限り行っている．そして，フラグを使用した高速化は，`String.toLowerCase`, `toUpperCase`, `getBytes` で行っている．以上の変更の総量は，13 ファイル，1,300 行程度であった．

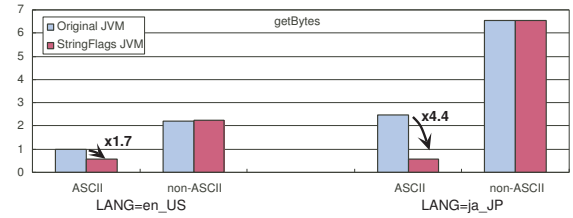
本節の残りの部分では，このプロトタイプを用いた評価結果を示す．測定はすべて，3.40 GHz の Pentium4 プロセッサと 3 GB のメモリを搭載し，Red Hat Enterprise Linux 5.3 オペレーティングシステムが動作する PC 上で行ったものである．

4.1　マイクロベンチマーク

まず，フラグを使用することでどれくらい文字列処理が高速化されるかをマイクロベンチマークで調査した．図 9 が調査に用いたプログラムで，同じ長さの ASCII 文字列と非 ASCII 文字列について，`toLowerCase` と `getBytes` を繰り返し行い，処理時間を測定している．このプログラムを，オリジナル JVM と StringFlags 修正を行った JVM の上で実行した．ヒープサイズは 512MB に設定し，`en_US` と `ja_JP` の



(a) toLowerCase の相対実行時間



(b) getBytes の相対実行時間

図 10　マイクロベンチマークの結果（短い方が高速）．ASCII 文字列の処理が 1.7–4.4 倍高速化された．

2 つのロケールで計測した†4．テストは複数回行い，最も速いものを採用している．なお，JIT コンパイラはデフォルト設定のままオンにして用いている．

結果を図 10 に示す．上のグラフ (a) は toLowerCase，下のグラフ (b) は getBytes の結果を示しており，それぞれ左の 4 本が en_US，右の 4 本が ja_JP ロケールの場合である．4 本の棒は，左のペアが ASCII 文字列，右のペアが非 ASCII 文字列に対する結果で，ペアのうち左がオリジナル JVM での結果，右が提案する StringFlags 手法による結果である．2 つのグラフはそれぞれ，オリジナル JVM で ASCII 文字列を処理した場合の時間を 1 として正規化されており，短い方が高速であることを示している．

これらのグラフから，以下のことが読み取れる．

- 提案手法により，ASCII 文字列に対する toLowerCase 処理は 3.0 倍高速化できている．
- 同様に，ASCII 文字列に対する getBytes 処理は 1.7–4.4 倍程度高速化できている．
- これらの性能向上はまた，Java 処理系内部でのフラグのセットがうまく機能していることも示している．
- JIT コンパイラによる高レベルの最適化が期待できるマイクロベンチマークであっても，提案手法による高速化が可能である．
- IS_ASCII フラグのみをサポートした今回のプロトタイプでは，非 ASCII 文字列に対する処理は高速化されないが，性能の低下も見られない．
- getBytes の処理はもともと，ja_JP ロケールでは en_US よりも 3 倍近く低速であるが，提案手法を用いれば，ASCII 文字列に対しては同等のコストで処理が行えるようになる．

つまり，StringFlags 手法により，非 ASCII 文字列の処理性能を犠牲にせず ASCII 文字列の処理が 1.7–4.4 倍程度高速化できたということである．オリジナル JVM でも，ASCII 文字列に対する処理は非 ASCII 文字列に対する処理よりも高速であった．これは，ASCII 文字の方が処理が簡単に済む場合が多いからだと考えられる．しかし，提案手法を使えば，ASCII 部分の処理だけを分離できるので，更なる最適化と高速化が達成できている．

4.2　マクロベンチマーク

次に，より現実的な評価として，2.2 節の予備実験で用いた Java プログラム群（表 1）での実験を行った．

まず，表 4 は，フラグがどれくらいうまくセットできているかの調査結果である．第 3 カラムに再掲したとおり，各プログラムで生成された String の大部分が ASCII 文字列であったが，このうち，プロトタイプで正しく IS_ASCII フラグがセットされたものの割合が一番右のカラムに示されている．幾何平均では，生成された ASCII String オブジェクトの 87%にフラグをセットできた．

一部のプログラム（bloat や jython など）では，ASCII 文字列であるにもかかわらず IS_ASCII フラグがうまくセットできない状況が比較的多く発生している．これらの String オブジェクトは，現在のプロトタイプではフラグ条件をチェックできていない部分で生成されているか，そこでフラグがセットされなかった String から派生して生成されているためだと考えられる．

†4　fr_FR ロケールについても計測したが，en_US と同様の結果であったため省略している．

表 4 フラグのセット状況の調査．ASCII 文字列の平均 87%に IS_ASCII フラグがセットできた．

プログラム名	生成された String の数 (x1,000)	ASCII 文字列の数と割合 (x1,000)	IS_ASCII フラグのセット率
DaCapo (-n 1)			
antlr	546	545 (100.0%)	87.7%
bloat	4,897	4,897 (100.0%)	56.2%
chart	10,446	10,446 (100.0%)	100.0%
eclipse	1,549	1,549 (100.0%)	95.4%
fop	315	313 (99.5%)	94.2%
hsqldb	178	178 (100.0%)	99.3%
jython	2,503	2,487 (99.3%)	78.4%
luindex	2,421	2,421 (100.0%)	99.6%
lusearch	1,790	1,790 (100.0%)	99.9%
pmd	152	152 (100.0%)	97.7%
xalan	425	425 (100.0%)	99.7%
(geometric mean)	1,049	1,048 (99.9%)	90.5%
WAS start+warmup			
LANG=en_US	6,880	6,880 (100.0%)	89.8%
LANG=fr_FR	6,505	6,499 (99.9%)	89.4%
LANG=ja_JP	6,874	6,866 (99.9%)	88.2%
WAS processing			
LANG=en_US	4,340	4,340 (100.0%)	72.7%
LANG=fr_FR	4,327	4,327 (100.0%)	72.4%
LANG=ja_JP	4,338	4,338 (100.0%)	72.8%
SPECjvm2008 Installer			
LANG=en_US	223	223 (99.7%)	98.3%
LANG=fr_FR	224	219 (97.7%)	98.5%
LANG=ja_JP	229	223 (97.5%)	98.6%

さらに，toLowerCase，toUpperCase，getBytes のメソッド呼び出しで，どれくらい高速版が使用できたかの調査結果が表 5 である．3 つのメソッドについて，使用された回数と，そのうち高速化できた（対象 String に IS_ASCII フラグがついていた）割合が示されている．幾何平均すると，約 98%の処理が IS_ASCII フラグにより高速化できている．表 4 で示したフラグの平均セット率 87%よりも大きいのは，うまくフラグがセットできなかった文字列がこれらの処理に使われる率が低かったためである．

しかしそれでも，一部のケースで高速化成功率が低いものが見られるが，これには 2 つの理由がある．一つは，ASCII 文字列であるにもかかわらずフラグがうまくセットできていない場合で，luindex の getBytes（成功率 81.5%）はこのケースであった．これについては今後，フラグがセットできない原因を調査し，可能ならフラグをセットできるようプロトタイプ実装を改良していく必要がある．一方，jython の toLowerCase（成功率 89.7%）と toUpperCase（成功率 89.6%）の場合，高速化されなかったものはすべて非 ASCII 文字列に対する処理であった．SPECjvm2008 Installer の，英語以外のロケールでの toLowerCase の高速化率が低い（85–87%）のも，非 ASCII 文字列が渡されているためであった．しかし，図 10 のマイクロベンチマーク結果からわかるように，これらのケースは高速化されないだけで，オリジナル JVM に比べて処理が遅くなるわけではない．

まとめると，提案する StringFlags 手法のプロトタイプ実装では，ASCII String の 87%に IS_ASCII フラグがセットでき，toLowerCase，toUpperCase，getBytes の呼び出しの 98%で高速版を使用することができた．

5 関連研究

本節では，いくつかの視点から関連する研究を紹介し，提案した StringFlags 手法との比較を行う．

表 5　高速版を用意した 3 つのメソッドの使用回数と高速化できた割合．平均 98%が高速化できた．

プログラム名	toLowerCase		toUpperCase		getBytes	
	使用回数	高速化成功	使用回数	高速化成功	使用回数	高速化成功
DaCapo (`-n 1`)						
antlr	132	100.0%	100	100.0%	406	100.0%
bloat	390,697	100.0%	100	100.0%	1	100.0%
chart	2,660	90.1%	119	100.0%	49	100.0%
eclipse	60,263	100.0%	78	100.0%	293	99.7%
fop	182	100.0%	101	100.0%	5	80.0%
hsqldb	311	100.0%	56,672	100.0%	41	100.0%
jython	48,687	89.7%	48,100	89.6%	78	100.0%
luindex	1,382,313	100.0%	101	100.0%	1,517	81.5%
lusearch	131,302	100.0%	103	100.0%	296,913	100.0%
pmd	130	100.0%	101	100.0%	401	100.0%
xalan	3,563	100.0%	305	100.0%	1	100.0%
(geometric mean)	6,980	98.1%	345	99.0%	105	96.2%
WAS start+warmup						
`LANG=en_US`	37,799	97.7%	1,690	96.3%	453	100.0%
`LANG=fr_FR`	37,509	92.2%	1,688	96.3%	452	100.0%
`LANG=ja_JP`	37,776	97.4%	1,695	95.8%	453	100.0%
WAS processing						
`LANG=en_US`	20,025	100.0%	0	—	0	—
`LANG=fr_FR`	20,000	100.0%	0	—	0	—
`LANG=ja_JP`	20,077	100.0%	0	—	0	—
SPECjvm2008 Installer						
`LANG=en_US`	6,590	95.1%	118	100.0%	5,447	99.3%
`LANG=fr_FR`	5,578	85.0%	118	100.0%	5,448	96.4%
`LANG=ja_JP`	6,100	87.0%	119	100.0%	5,444	95.3%

汎用の処理を,「よく現れる特別なケース」について高速化するというアイデアは，プログラミングの各レイヤで行われる方法である．最もナイーブなものは，汎用の低速版と専用の高速版を用意し，アプリケーションが明示的にそれらを使い分けるというものである．JDK 1.5 から導入された `StringBuilder` クラスは，マルチスレッドに対応した（スレッドセーフな）`StringBuffer` の単一スレッド専用版 [31] で，このような高速化の例といえる．しかし，この手法ではアプリケーション自身が高速版を明示的に使用しなければならない†5ほか，高速化できる前提（たとえば，1 つのスレッドからしか同時アクセスしない）を間違えていると動作がおかしくなる危険性がある．

JIT コンパイラなどの動的コンパイラでは，メソッドの引数などについて,「よく現れるケース」をプロファイリングによって発見し，それ専用のコードを生成する「特殊化（specialization）」が行われる [24]．StringFlags 手法は,「よく現れるケース（たとえば，ASCII 文字列である）」が事前に設定されている点で柔軟性が低いともいえるが．`String` という複雑なデータ構造に適用できることや，処理中にフラグを伝播させることで適用可能性を高めているという特長がある．実際，コンパイラによる特殊化のみで，汎用コードを図 3 のような特殊コードに変形することは困難であろう．

Java 処理系の内部に目を移すと,「よく現れるケース」を高速化している例として，ロック（スレッド間同期）処理の最適化があげられる．Java は標準でマルチスレッドをサポートしているため，多くのライブラリはスレッドセーフに記述されている．そのため，Java のロック処理は実際には衝突しないというケースが非常に多い．このことに着目し,「shape bit」というフラグを導入して行った最適化が Bacon らの「Thin ロック [3]」である．また，Java ではそれぞれのロックは特定のスレッドからのみ行われることが非

†5　ただし Java の場合は，`String` を “`+`” 演算子で連結する処理について，Java コンパイラ（`javac`）が `StringBuilder` を使用してくれる．

常に多いため，そのケースを高速化する「予約ロック [17] [21]」という手法も提案されている．StringFlags 手法はこれらと同様のアプローチを文字列処理に対して適用したものと考えられるが，ロックが Java 処理系内部の機構に関するものであるのに対し，文字列処理はアプリケーションから明示的に呼ばれるものであるという点で若干異なっている．

Java アプリケーションで `String` が大量に使用されていることに着目した最適化として，`String` のメモリ消費の無駄を調べ削減している研究がある [18]．彼らの研究の本題は `String` によるメモリ消費の削減であるが，同じ内容の文字列をまとめることで，`String.equals` 処理が高速化されることや，Java の国際化に対応したメッセージ処理機構がメモリ消費を増やしていることなども示されている．

また，Häubl らは，別のアプローチとして，`String` を実装する 2 つのオブジェクト（図 1）を一つにまとめることでメモリ利用の無駄を削減する手法を提案している [13]．この研究では，保持している文字列の内容については手を加えていないが，オブジェクトをまとめることで性能も 8%程度向上することが報告されている．

我々の StringFlags 手法では各 `String` にフラグを追加する必要があるが，それによるメモリ消費の増加を抑える方法については，3.2 節で説明した．ちなみに今回の調査では，`String` のほとんどは ASCII 文字列であることがわかった．これはつまり，String の内容を保持する `char` 配列は半分が無駄であるということを示している．メモリ節約のためには，ASCII 文字列の場合は内容を `byte` 配列で保持する改良なども考えられる．

Mitchell らは，`String` をはじめとするデータが，大規模 Java アプリケーション中でどのように変換されていくかについての調査を行っている [20]．我々の StringFlags では，3 つの文字列関連クラスおよび文字列リテラルからのデータ変換について考慮し，フラグの伝播処理などを行っている（図 8）が，この研究のようにもっと他のクラスへの変換についても調査をすすめることで，より効率的なフラグのセットと使用が可能になるかもしれない．

6 おわりに

本論文では，Java アプリケーションで大量に使用される文字列の処理を高速化する手法について提案した．この「StringFlags」手法では，`String` オブジェクトに，保持している文字列の特性を示す「フラグ」を追加する．フラグは，文字列の全文字が走査される処理中で副次的にセットされ，文字列の連結処理などを通じて「伝播」される．各文字列処理メソッドでは，フラグをチェックし，可能なら高速化バージョンを実行することで，処理性能を向上させる．これにより，Java の文字列がもつ汎用性を損なわずに，Java 処理系内部の変更だけで大多数の文字列処理を高速化できる．商用 Java 処理系に実装したプロトタイプでは，ASCII 文字のみから構成される `String` の 87%にそのことを示すフラグを付加することができ，文字列処理を最大 4.4 倍高速化できた．

Java VM のように仮想化された環境では，実マシンの詳細を隠蔽し共通化した実行環境を提供することが可能である．しかしその一方で，一見単純に見える処理に意外とコストがかかっていることがある．本論文ではそのような例として文字列処理に着目し，その汎用性から性能が低下していることを示した．一方で，仮想化されていることは，アプリケーションから独立して性能を改善することができるというメリットでもある．本論文では，Java 処理系内部で文字列の特性を追跡するフラグを設けることで，大多数の文字列について高速化した処理を行えることを実証した．これによって，Java アプリケーションを変更することなく，実行性能を改善することができた．

参 考 文 献

[1] The Apache Software Foundation. Apache Geronimo v2.0 Documentation, DayTrader. `http://cwiki.apache.org/GMOxDOC20/daytrader.html`

[2] The Apache Software Foundation. Apache Harmony. `http://harmony.apache.org/`

[3] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio J. Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the SIGPLAN 98 Conference on Programming Language*

Design and Implementation (PLDI '98), pp. 258–268, 1998,

[4] Chris Bailey. Java Technology, IBM Style: Introduction to the IBM Developer Kit: An overview of the new functions and features in the IBM implementation of Java 5.0, 2006.
`http://www.ibm.com/developerworks/java/library/j-ibmjava1.html`

[5] Stephen M. Blackburn, et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '06)*, pp. 169–190, 2006.

[6] Caucho Technology, Inc. Welcome to the home of Quercus.
`http://quercus.caucho.com/`

[7] Patrick Chan, Rosanna Lee, and Douglas Kramer. *The Java Class Libraries, Second Edition*, Addison Wesley, 1998.

[8] The Eclipse Foundation. Eclipse.org home.
`http://www.eclipse.org/`

[9] Maydene Fisher, Jon Ellis, and Jonathan Bruce. *JDBC API Tutorial and Reference, Third Edition*, Addison Wesley, 2003.

[10] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*, Addison Wesley, 2005.

[11] Nikola Grcevski, Allan Kielstra, Kevin Stoodley, Mark Stoodley, and Vijay Sundaresan. Java Just-In-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. In *Proceedings of the 3rd USENIX Virtual Machine Research and Technology Symposium (VM '04)*, pp. 151–162, 2004.

[12] Groovy: An Agile Dynamic Language for the Java Platform.
`http://groovy.codehaus.org/`

[13] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Optimized Strings for the Java HotSpot Virtual Machine. In *Proceedings of the ACM International Conference on Principles and Practice of Programming in Java (PPPJ '08)*, pp. 105–114, 2008.

[14] IBM Corporation. WebSphere Application Server.
`http://www.ibm.com/software/webservers/appserv/was/`

[15] JRuby.org. JRuby: 100% Pure-Java Implementation of the Ruby Programming Language.
`http://www.jruby.org/`

[16] The Jython Project. Jython: Python for the Java Platform.
`http://www.jython.org/`

[17] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, pp. 130–141, 2002.

[18] Kiyokuni Kawachiya, Kazunori Ogata, and Tamiya Onodera. Analysis and Reduction of Memory Inefficiencies in Java Strings. In *Proceedings of the 23rd ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '08)*, pp. 385–401, 2008.

[19] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*, Addison Wesley, 1999.

[20] Nick Mitchell, Gary Sevitsky, and Harini Srinivasan. The Diary of a Datum: An Approach to Analyzing Runtime Complexity in Framework-Based Applications. In *Proceedings of the 1st International Workshop on Library-Centric Software Design (LCSD '05)*, pp. 85–90, 2005.

[21] Tamiya Onodera, Kiyokuni Kawachiya, and Akira Koseki. Lock Reservation for Java Reconsidered. In *Lecture Notes in Computer Science, LNCS 3086 (ECOOP '04)*, Springer-Verlag, pp. 559–583, 2004.

[22] Project Zero. Looking for PHP on Java? It's here!
`http://www.projectzero.org/php/`

[23] Standard Performance Evaluation Corporation. SPECjvm2008.
`http://www.spec.org/jvm2008/`

[24] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A Dynamic Optimization Framework for a Java Just-In-Time Compiler. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*, pp. 180–194, 2001.

[25] Sun Developer Network. Java EE at a Glance.
`http://java.sun.com/javaee/`

[26] Sun Microsystems. Java Platform, Standard Edition 6, API Specification: java.lang.String.
`http://java.sun.com/javase/6/docs/api/java/lang/String.html`

[27] TIOBE Software. TIOBE Programming Community Index for August 2010.
`http://www.tiobe.com/index.php/content/paperinfo/tpci/`

[28] The Unicode Consortium. *The Unicode Standard, Version 5.0*, Addison Wesley, 2006.

[29] Wikipedia. ASCII.
`http://en.wikipedia.org/wiki/ASCII`

[30] Wikipedia. Locale.
`http://en.wikipedia.org/wiki/Locale`

[31] Wikipedia. StringBuffer and StringBuilder.
`http://en.wikipedia.org/wiki/StringBuffer_and_StringBuilder`

[32] Joe Winchester. Turkish Java Needs Special Brewing.
`http://java.sys-con.com/node/46241`