情報処理学会論文誌　プログラミング　Vol. 1　No. 2　1–8 (Sep. 2008)

# SIMD 命令を用いる UTF-8 文字列デコード処理の高速化

井 上　拓[†1]　小 松 秀 昭[†1]　中谷 登志男[†1]

近年，XML など多くの用途において，テキストデータの標準的な表現形式として，1 文字を 1〜3 バイトの可変長で表現する UTF-8 エンコーディングが用いられている．一方，Java 仮想マシンなど多くの処理系においては，文字列の内部表現として 1 文字が 2 バイトの固定長である UTF-16 エンコーディングが用いられている．そのため，Java で記述された Web アプリケーションサーバなどの多量のテキストデータを取り扱うワークロードにおいては，テキストデータを UTF-8 と UTF-16 との間で相互に変換する処理が大きな処理時間を占める場合があり，このテキストデータ変換処理の高速化はシステム全体の性能向上において重要な意味を持つ．本研究では，SIMD 命令を用いて UTF-8 から UTF-16 への変換をはじめとする可変長符号化データのデコード処理を高速に行う手法を提案する．この手法では複数のデータを並列に処理することに加えて，条件分岐での分岐予測ミスによるオーバヘッドを減少させることで，大きな性能向上が得られる．本手法を PowerPC アーキテクチャの SIMD 命令セットである VMX 命令を用いて実装し，様々なテキストデータを入力として UTF-8 文字列デコード処理の性能を計測した結果，SIMD 命令を用いない既存の方法と比較して単純な例で 10 倍以上，実際のテキストデータを用いたケースでも 2 倍から 10 倍の性能向上が得られた．

# Accelerating UTF-8 Decoding Using SIMD Instructions

Hiroshi Inoue,[†1] Hideaki Komatsu[†1]
and Toshio Nakatani[†1]

Recently UTF-8 encoding is widely used as a standard format for text data exchange. The Java programming language, however, uses UTF-16 encoding as its internal representation format for text data. As a result, data conversions between UTF-8 and UTF-16 consume considerable amount of CPU time in workloads that process large amount of text data, such as web application servers. Hence accelerating these conversions are important to improve the performance of many applications. In this paper, we present our new technique to accelerate decoding of variable-length formats, such as conversion from UTF-8 to UTF-16, by using SIMD instructions. The new technique can achieve higher performance by reducing overhead of branch mispredictions in addition to exploiting data parallelism of SIMD instructions. We implemented the technique using VMX instructions of the PowerPC architecture and evaluated its performance to decode various UTF-8 sequences on a PowerPC 970 MP processor. As a result, we showed that our technique significantly accelerated the UTF-8 decoding compared to the existing method.

## 1. はじめに

近年，XML などの多く用途において，テキストデータの標準的な形式として，1 文字（UNICODE の補助文字を除く）を 1〜3 バイトの可変長で表現する UTF-8 エンコーディング[1] が用いられている．一方，テキストデータを処理するシステムにおいては多くの場合，1 文字（同様に補助文字を除く）が 2 バイトの固定長で処理の行いやすい UTF-16 エンコーディング[1] が用いられている．たとえば，Java 仮想マシンにおいては，文字列の内部表現として UTF-16 が用いられており，そのため Java で記述された Web アプリケーションサーバなどの多量のテキストデータを取り扱うワークロードにおいては，テキストデータを読み込む際に UTF-8 から UTF-16 に変換し，逆に出力する際に UTF-16 から UTF-8 に変換する必要があり，この変換処理が大きな処理時間を必要とする．一例として，JSP による動的ページを含む WEB サーバのパフォーマンスを計測するベンチマークである SPECweb2005[2] を Apache Tomcat 6.0[3] を用いて実行した際の実行時間の調査を行ったところ，Java プログラムの実行時間の 30%あまりをこれらの文字コード変換処理が占めていた．また，このベンチマークは入出力データが UTF-8 で 1 バイトで表現される ASCII 文字列であり処理が単純であるが，複数バイトで表現される日本語などの文字を含む場合には，さらに大きな処理時間を必要とする．このように Java 環境をはじめとして，多くのシステムにおいて文字コード変換処理の高速化はシステム全体の性能向上において重要な意味を持つ．

そこで本研究では，プロセッサの SIMD 命令を用いて UTF-8 などの可変長符号化されたデータを高速に固定長表現にデコードする手法を提案する．この手法では可変長符号化データの要素ごとのサイズを調べ，それを条件分岐を用いて処理する代わりに，バイトごとに任意の順でデータの並べ替えを行う permute 命令を用いて並列に演算処理を行う．これによ

†1 日本 IBM 東京基礎研究所
IBM Tokyo Research Laboratory



© 2008 Information Processing Society of Japan

り，SIMD 命令の並列度を活用することに加えて，条件分岐での分岐予測ミスによるペナルティを減少させ，大きな性能向上が得られる．

本手法を PowerPC 970 MP プロセッサ上で PowerPC アーキテクチャの SIMD 命令セットである VMX 命令を用いて実装し，様々なテキストデータの UTF-8 から UTF-16 へのデコード処理性能を計測した結果，SIMD 命令を用いない既存の方法と比較して，単一の文字種のみからなるような単純な例で 10 倍以上，実際のテキストデータを用いたケースでも 2 倍から 10 倍の大きな性能向上が得られた．

## 2. SIMD 命令セット

現在，多くの高性能なプロセッサが 1 つの命令で複数のデータを処理する SIMD（Single Instruction Multiple Data）命令セットを提供している．これらの SIMD 命令では命令自身の持つデータ並列度により性能向上が得られる利点に加えて，条件分岐を用いないプログラミングを可能にすることで，分岐予測ミスによるオーバヘッドを削減することを可能にしている．これらの特長を生かして，科学技術計算や画像処理などに加えて，たとえばデータベース操作[4] や，ソートおよびマージ[5),6)] などの条件分岐が多い処理を SIMD 命令を用いて高速に処理する方法がこれまでに提案されている．

本研究では PowerPC アークテクチャの SIMD 命令セットである Vector Multimedia eXtension（VMX，AltiVec とも呼ばれる）[7] を主な対象とする．VMX では 16 バイト長のベクタレジスタを用いて，1 バイト × 16 スロット，2 バイト × 8 スロット，4 バイト × 4 スロットのいずれかの形式での並列処理が可能である．以下に VMX 命令セットのうち本研究で多く使用する，permute 命令および select 命令について挙動を説明する．

permute 命令（vec_perm）は入力データをバイト単位で任意の順に並べ替える命令である．この命令は 3 つのベクタレジスタを入力，1 つのベクタレジスタを出力とし，ベクタレジスタのデータを 16 個の 1 バイト値の列として扱う．処理においては，まず第 1 引数と第 2 引数をこの順で結合し 32 個の 1 バイト値の配列とする．そのうえで第 3 引数の各スロットの下位 5 ビットの値をインデックスとして配列からデータを取り出し，出力レジスタの対応するスロットに出力する．これによりバイト単位で任意の順にデータの並べ替えを行うことができる．

select 命令（vec_sel）はマスクに応じて 2 つのベクタレジスタのデータを混合する命令である．この命令も 3 つのベクタレジスタを入力，1 つのベクタレジスタを出力とし，出力レジスタの各ビットについて第 3 引数の対応するビットが 0 だった場合には第 1 引数の対応するビットを，1 だった場合には第 2 引数の対応するビットを出力する．

このようなデータの並べ替えをレジスタの値に応じて行う permute 命令については VMX 命令セットのほかに，Cell Broadband Engine プロセッサ[8] の SPU コアの SIMD 命令セット[9] も同等の命令（shuffle 命令）を持っている．また，予定されている AMD64 命令セットの SSE5 [10] も同等の命令をサポートする．

## 3. 可変長符号化データの変換手法の提案

本研究で提案する SIMD 命令を用いる可変長符号化されたデータのデコード手法の基本的なアイデアについて，まず**表 1** に示す簡単な符号化フォーマットの例を用いて説明する．その後，この手法をより複雑な UTF-8 文字列のデコード処理に適用する方法について述べる．

### 3.1 基本アイデア

表 1 のフォーマットでは，30 ビット長までのデータを 2 ビットの prefix を付与し 1〜4 バイトの可変長のデータで表現する．各データの最初のバイトの先頭 2 ビットが，そのデータの符号化データ長を表す prefix であり，残りのビットが実際のデータを表す．SIMD 命令を用いずにこのフォーマットにより符号化されたデータの復元を行い，32 ビット整数の配列として出力するプログラムの例を**図 1** に示す．この例から分かるように，一般的に可変長符号化データのデコード処理は多くの条件分岐を含み，特に複雑なデータのデコード処理においては分岐予測ミスによるペナルティが大きくなる．

本研究で提案する SIMD による命令を用いるデコード手法は，以下の 3 つのステップからなる．

**ステップ 1**　処理する複数の入力データ要素の符号化データ長を調べる．

**ステップ 2**　ステップ 1 の結果から，事前に準備された定数のテーブルにアクセスしデコード処理で使用するパラメータを決定する．

**ステップ 3**　得られたパラメータを用いて演算を行うことで，複数のデータを 1 度に処理

**表 1**　単純な可変長符号化規則の例
Table 1　An example of simplified variable-length encoding rule.

| 入力<br>データ長 | 符号化<br>データ長 | 2-bit<br>prefix | 形式<br>[1 バイト目] [2 バイト目] ‥　x : データビット |
|---|---|---|---|
| 1〜6 ビット | 1 バイト | 0 (00b) | [00xxxxxx] |
| 7〜14 ビット | 2 バイト | 1 (01b) | [01xxxxxx] [xxxxxxxx] |
| 15〜22 ビット | 3 バイト | 2 (10b) | [10xxxxxx] [xxxxxxxx] [xxxxxxxx] |
| 23〜30 ビット | 4 バイト | 3 (11b) | [11xxxxxx] [xxxxxxxx] [xxxxxxxx] [xxxxxxxx] |


© 2008 Information Processing Society of Japan

```
1  while (! end of encoded_data) {
2     byte *p = encoded_data[src_offset];
3     prefix = p[0] >> 6;                          // 2 bitのprefixを取得
4     length = prefix + 1;                         // 符号化データ長は(prefix + 1)
5     switch (prefix) {                            // prefixの値によって条件分岐
6        case 0:
7           output = p[0];
8           break;
9        case 1:
10          output = (p[0] & 0x3F) << 8  | p[1];
11          break;
12       case 2:
13          output = (p[0] & 0x3F) << 16 | p[1] << 8  | p[2];
14          break;
15       case 3:
16          output = (p[0] & 0x3F) << 24 | p[1] << 16 | p[2] << 8 | p[3];
17          break;
18    }
19    decoded_data[dst_offset++] = output;   // デコードした結果をメモリに書き出す
20    src_offset += length;
21 }
```

**図 1**　本研究の手法を用いない可変長符号化のデコード処理プログラム
Fig. 1　Pseudocode of decoding process for the simplified encoding rule.

```
1   while (!end of encoded_data) {
2      byte *p = encoded_data[src_offset];

3      // ステップ 1: 4データ分のprefixを調べ, 各データの符号化データ長を調べる
4      gathered_prefix = 0, total_length = 0;
5      for (i=0; i<4; i++) {
6         prefix = (p[total_length] >> 6);                    // 2 bitのprefixを取得
7         gathered_prefix = (gathered_prefix << 2) | prefix;  // テーブルアクセスのindexに使用
8         total_length += prefix+1;                           // 符号化データ長は(prefix + 1)
9      }

10     // ステップ 2: 一度に変換する4データの組み合わせに応じて, ベクタ定数をテーブルから読み込む
11     vpattern = pattern_table[gathered_prefix];      // petmute命令用ベクタ定数
12     vmask    = mask_table[gathered_prefix];         // and命令用ベクタ定数

13     // ステップ 3: 定数を使用して実際のデータ移動を行う
14     vin  = vector_load(p);                          // 符号化データをベクタレジスタに読み込む
15     vtmp = vector_permute(vin, vin, vpattern);      // 定数に応じてバイト単位で並べ替えを行う
16     vout = vector_and(vtmp, vmask);                 // prefixなどの不要なビットを0にマスクする
17     vector_store(vout, &decoded_data[dst_offset]); // 変換結果をメモリに書き出す

18     src_offset += total_length;
19     dst_offset += 4;
20  }
```

**図 2**　本研究で提案する SIMD 命令を用いた可変長符号化のデコード処理プログラム
Fig. 2　Pseudocode of our algorithm for decoding the simplified encoding rule.

する．

本手法ではそれぞれの入力データを条件分岐を用いて個々に処理するのではなく，複数の入力データを読み込み，それらに対する適切な処理を事前に計算された定数値を使用して行うことで，条件分岐を用いずに可変長符号化データの処理を行うことを可能にする．

この処理を表 1 のフォーマットに適用する場合のプログラム例を**図 2** に示す．ここでは 16 バイト長のレジスタを用いて，4 データを 1 度に処理するものとして説明するが，本手法自体はこの並列度に限定されるものではない．図 2 のステップ 1 ではスカラ命令を用いて 4 つの入力データの prefix のみを集める．この段階では条件分岐なしに単純に prefix を集める演算を行うだけなので，大きな計算負荷にはならない．ステップ 2 では集めた prefix をインデックスとして定数テーブルにアクセスすることで 4 つのデータの符号化データ長の組合せに応じたベクタ定数を読み込む．ここでは，permute 命令によるデータの並べ替えに用いる `vpattern` と，and 命令によりデコード後のデータとしては不要な prefix ビットをマスクするために用いる `vmask` の 2 つの定数をロードする．ステップ 3 では，符号化されたデータをベクタレジスタに読み込んだ後，ステップ 2 で準備した 2 つのベクタ定数を用いて実際に並べ替えおよびマスクの処理を行う．ステップ 2 でアクセスする定数テーブルは，4 つのデータそれぞれの符号化データ長に応じて，各バイト値をどのように移動した後で，どのビットをマスクすればよいかを事前に計算して作成しておく．4 つのデータそれぞれが 4 種類の符号化データ長をとりうるので，2 つのテーブルのそれぞれに 256 通りの定数が必要であり，16 バイト × 256 エントリで各 4 KB となる．

**図 3** に，4 つの入力データの符号化データ長が { 1 バイト，2 バイト，3 バイト，4 バイト } であるような場合の 2 つのベクタ定数の値およびデコード処理の例を示す．このような入力データの組の場合，符号化されたデータの先頭バイトが 1 つ目の出力値，2〜3 バイト目が 2 つ目の出力値，4〜6 バイト目が 3 つ目の出力値，7〜10 バイト目が最後の出力値に対応する．パラメータ `vpattern` はそれぞれのバイト値を対応する出力の場所に移動するように作られている．ここで `vpattern` 中で “`*`” となっている要素は，次に 0 にマスクされるため任意の値でよい．この時点では移動したデータ中に prefix ビットが残っており，最終的な結果を得るためにはこれらを除去する必要がある．そのため，パラメータ `vmask` は prefix ビットの部分を 0 にマスクするように決められている．`vmask` で prefix ビットをマスクすることで，デコード処理が終了し，4 つの 32 ビット値が得られる．

図 2 のプログラムでは図 1 のプログラムと比較して，4 データを 1 度に処理できることに加えて，条件分岐を用いていないため分岐予測ミスによるペナルティがないという利点がある．加えて，ステップ 1 の処理は，前のイタレーションでのステップ 2，3 の処理に依存


© 2008 Information Processing Society of Japan

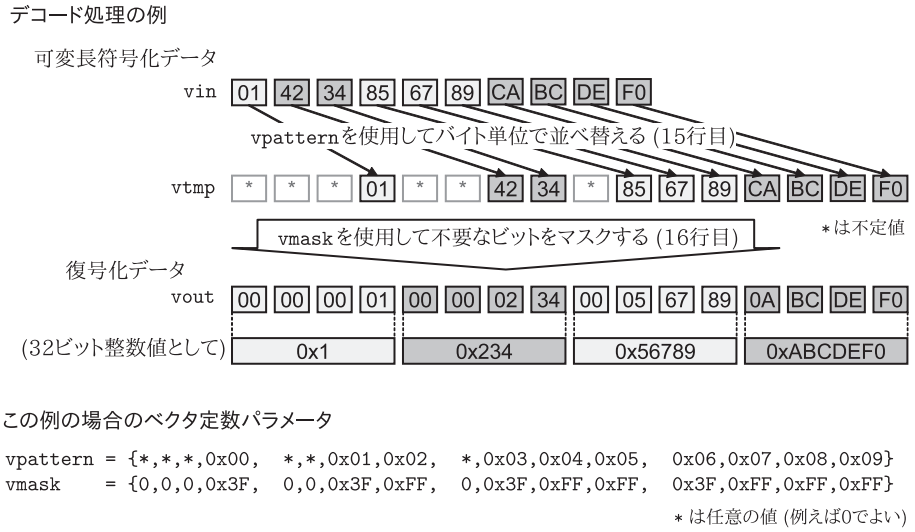


**図 3**　SIMD 命令を用いたデコードでの定数パラメータおよび処理の例
Fig. 3　An example of vector constants and decoding process.

しないためこれと並列に実行でき，高い命令レベルの並列性を得られる．一方，問題点としてはステップ 2 での定数テーブルへのアクセスがキャッシュミスを起こす場合には，性能低下を引き起こすことが考えられるが，今回の評価の結果では顕著な L1 キャッシュミスの増加は観測されなかった．

### 3.2　UTF-8 文字列のデコード処理への適用

本研究の手法を UTF-8 文字列のデコード処理に適用する手法について述べる．前節の例に使ったフォーマットは先頭のバイト以外に prefix を持たないためにデータの移動がバイト単位のみであった．一方，UTF-8 エンコーディングでは**表 2** に示すように各バイトに prefix を持つため permute 命令によるバイト単位での並べ替えだけでは処理できず，各バイトの prefix を取り除くためにより複雑な処理が必要となる．

**図 4** に本手法を UTF-8 文字列のデコード処理に適用した場合のプログラム例を示す．基本的な流れは前述の単純なフォーマットの例と同じであるが，以下のような点で異なる．1) 前の例ではステップ 3 の処理がバイト単位でのデータの移動を行うための permute 命令と不要なビットをマスクするための and 命令のみだったが，UTF-8 のデコード処理では

**表 2**　UTF-8 フォーマットの符号化規則
Table 2　UTF-8 encoding rule.

| 符号化タイプ | 文字コード範囲 | UTF-8 ビット列 [1 バイト目] [2 バイト目] ‥　x:データビット |
|---|---|---|
| 1 バイトコード | 0x00〜0x7F | [0xxxxxxx] |
| 2 バイトコード | 0x80〜0x7FF | [110xxxxx] [10xxxxxx] |
| 3 バイトコード | 0x800〜0xFFFF | [1110xxxx] [10xxxxxx] [10xxxxxx] |
| 4 バイトコード | 拡張文字 | [11110xxx] [10xxxxxx] [10xxxxxx] [10xxxxxx] |

```
1  while (!end of UTF8_data) {
2     byte *p = UTF8_data[src_offset];

3     // ステップ 1: 8文字分のprefixを調べ, 各文字のUTF-8表現での長さを調べる
4     gathered_prefix = 0, total_length = 0;
5     for (i=0; i<8; i++) {
6        prefix = (p[total_length] >> 3);                    // 5 bitのprefixを取得
7        length = prefix_to_length_table[prefix];            // prefixからその文字の長さを調べる
8        gathered_prefix = (gathered_prefix * 3) + length;  // テーブルを引くためのindexを計算
9        total_length += length;
10    }

11    // ステップ 2: 一度に変換する8文字の組み合わせに応じて, ベクタ定数をテーブルから読み込む
12    vpattern1 = pattern1_table[gathered_prefix];          // petmute命令用ベクタ定数
13    vpattern2 = pattern2_table[gathered_prefix];
14    vmask1    = mask_for_select_table[gathered_prefix]; // select命令用ベクタ定数
15    vmask2    = mask_for_and_table[gathered_prefix];    // and命令用ベクタ定数

16    // ステップ 3: 定数を使用して実際のデータ移動を行う
17    vin1  = vector_load(p);                 // UTF-8文字列を16バイト長のベクタレジスタに読み込む
18    vin2  = vector_load(p+16);              // 変換により短くなる場合があるため,次の16バイトも必要
19    vtmp1 = vector_permute(vin1, vin2, vpattern1);      // ステップ2で読み込んだ定数に応じて
20    vtmp2 = vector_permute(vin1, vin2, vpattern2);      // バイト単位で並べ替えを行う
21    vtmp1 = vector_select(vector_shift_left(vtmp1, 4), // vtmp1とvtmp2をマージするため
                            vector_shift_left(vtmp1, 6), // ビット単位の位置をあわせる
                            vconstant_0x0FC0);
22    vtmp3 = vector_select(vtmp1, vtmp2, vmask1);        // 読み込んだ定数に応じてマージする
23    vout  = vector_and(vtmp3, vmask2);                  // prefixなど不要なビットをマスクする
24    vector_store(vout, &UTF16_data[dst_offset]);        // 変換結果をメモリに書き出す

25    src_offset += total_length;
26    dst_offset += 8;
27 }

ただし, vconstant_0x0FC0 = {0x0F, 0xC0, 0x0F, 0xC0, 0x0F, 0xC0,.. };
```

**図 4**　SIMD 命令を用いた UTF-8 文字列のデコード処理プログラム
Fig. 4　Pseudocode of decoding process for an UTF-8 sequence.

ビット単位のデータの移動を行うために，これに加えてビット単位でのシフト命令やそれらをまとめるための select 命令を使用する．2) これにともない，ステップ 2 で読み込む定数パラメータが 2 つから 4 つに増加する．3) 出力が 2 バイト固定長の UTF-16 データであ


© 2008 Information Processing Society of Japan

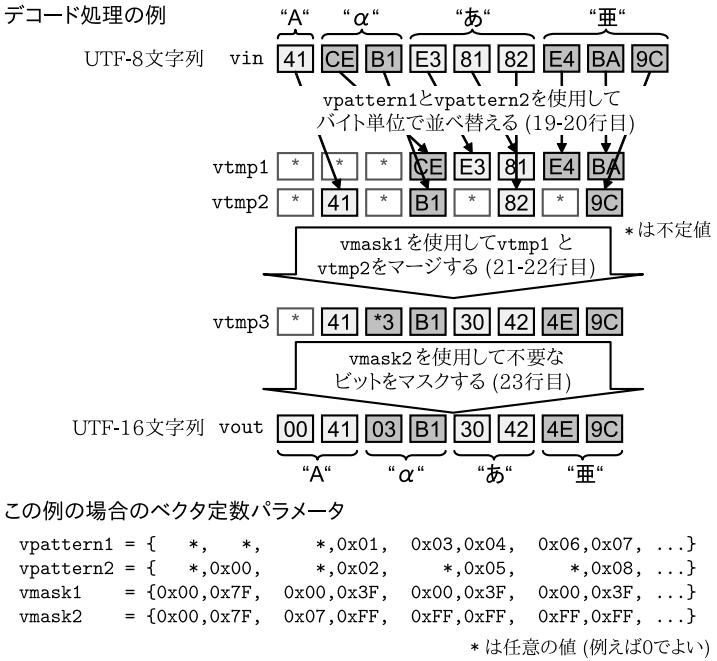


**図 5**　SIMD 命令を用いた UTF-8 文字列のデコードでの定数パラメータおよび処理の例
Fig. 5　An example of vector constants and decoding process for an UTF-8 sequence.

るために，16 バイトのベクタレジスタを使用して 1 度に 8 データ（8 文字）ずつ処理を行う．図 4 のステップ 1 では，8 文字のそれぞれについて先頭 5 ビットを prefix として読み込む．UTF-8 エンコーディングでは，単純な演算で prefix から符号化データ長を得ることができないため，事前に各 prefix ごとの符号化データ長を計算しておきテーブル（コード中の `prefix_to_length_table`）に保持しておく．ステップ 2 では 8 文字の符号化データ長の組合せに応じて 4 つの定数パラメータを読み込む．ステップ 3 では定数パラメータを使用して，実際のデータ移動を行う．

**図 5** に，処理する文字列が “a α あ亜” である場合の定数とデータ移動の例を示す．本来は 1 度に 8 文字処理するが，簡単のために図では前半の 4 文字分のデータ移動についてのみ示している．プログラムの 17〜18 行目では UTF-8 文字列をベクタレジスタに読み込む．UTF-8 から UTF-16 への変換では，前の例と異なり変換によりデータサイズが減少する場合があるため，8 文字（1 ベクタレジスタ）分の処理を 1 度に行うために，2 ベクタレジスタ分のデータを読み込む必要がある．19〜20 行目では，UTF-8 文字列データを permute 命令でバイト単位で 2 つのベクタレジスタに並べ替える．最初のベクタレジスタ（`vtmp1`）には上位側のバイト，次のベクタレジスタには最下位バイトを配置する形でデータを移動する．このうち `vtmp1` の内容は，バイト内でのビットの位置が最終的に出力される位置からずれているため，21 行目でシフト命令を用いてビット単位での位置をあわせたうえで，22 行目でステップ 2 で読み込んだ定数パラメータを使用し select 命令で `vtmp2` とマージする．23 行目では前の単純な例と同様に不要なビットを and 命令でマスクすることで最終的な UTF-16 文字列を得ている．

本手法では UTF-8 で 4 バイトにエンコードされ，UTF-16 ではサロゲートペアで表される UNICODE の補助文字のデコードには対応していない．入力データ中に補助文字が現れた場合を判定するには，図 4 の `prefix_to_length_table` で，非対応文字種の prefix に対応する符号化データ長として異常値を入れておき，これをチェックすることで簡単に行うことができる．定数テーブルのサイズは 8 文字それぞれに 3 通りの組合せを考えると，それぞれが約 100 KB と大きくなる．これは 4 文字ずつについてのテーブルを作成しておき，これを 1 イタレーション（8 文字の処理）につき 2 回アクセスするようにすることでそれぞれ約 1 KB のサイズに削減することが可能である．

## 4. 実装と性能評価

本研究で提案する SIMD 命令を用いる変換手法および SIMD 命令を用いない従来のスカラ処理手法について，2.5 GHz の PowerPC 970 MP プロセッサ[7] を搭載したマシン上で実装を行い，その性能評価を行った．OS には Linux-2.6.18，コンパイラには gcc-4.0.1 を用いた．どちらのプログラムも C 言語で記述しており，32 ビットプロセスとしてコンパイルした．本手法のプログラムはコンパイラの提供するイントリンシック[11] を用いて VMX 命令を直接記述する形で実装を行った．

### 4.1 実装方法

UTF-8 デコード処理の実装においては，より現実的な性能評価を行うために，多くの高性能な文字コード変換プログラムにおいて行われるように，テキスト文書の中では同じ文字種が連続する場合が多いという知見に基づき最適化を行った．スカラ処理の場合にはたとえば UTF-8 で 1 バイトにエンコードされる ASCII 文字が現れた場合には，その後も ASCII 文字が続くと仮定して 4 回アンロールしたループで他の文字種が現れるまで連続して処理


© 2008 Information Processing Society of Japan

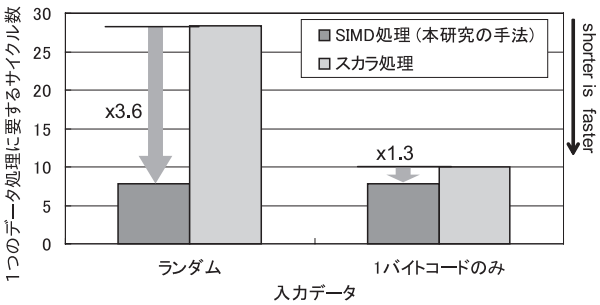


**図 6**　表 1 のフォーマットのデコード処理性能
Fig. 6　Performance of decoding the simplified encoding rule.

を行う．本研究の SIMD 命令を用いる手法の場合には，1 度に処理する 8 文字すべてが同じ文字種だった場合に，次の文字以降を専用のショートカットループで処理を行う．たとえば，8 文字すべてが ASCII 文字だった場合には次の 8 文字もすべて ASCII 文字であると仮定したチェックを行った後，単純に各バイトの間に 0 をパディングする処理を行い UTF-16 文字列を得る．

また本手法のステップ 2 でアクセスする定数テーブルについては，プログラム例に示したようにそれぞれを独立したテーブルにするのではなく，1 つのテーブルにまとめることで，キャッシュミスの削減を図った．

### 4.2　表 1 のフォーマットのデコード性能

本手法の基本的な特性を理解するために，表 1 のフォーマットをデコードする際の性能を評価する．**図 6** に，1〜4 バイトの符号化データ長のデータがランダムに現れるようにした場合（図中のランダム）と，1 バイトの符号化データ長だけの場合を入力データとして，本手法と従来のスカラ処理の手法のデコード処理性能の比較を示す．縦軸は 1 データのデコードにかかった CPU サイクル数であり，小さい値ほど高い性能を示す．ランダムデータの場合には，本手法によりスカラ処理と比較して 3.6 倍の高い性能向上が得られた．これは SIMD 命令の並列性による性能向上に加えて，条件分岐での分岐予測ミスのオーバヘッドがないことによる．プロセッサのパフォーマンスカウンタで計測を行うと，本手法の場合には分岐予測ミスは無視可能な頻度であった反面，スカラ処理では 1 データに 1 回程度の高い頻度で分岐予測ミスが生じていた．一方，つねに同じデータ長の単純な処理の場合には，スカラ処理でも分岐予測ミスが生じないため，その性能が大きく向上する反面，本手法での性能は条件分岐を使用していないため乱数データの場合とほぼ等しい．そのため，本手法による向上率は約 1.3 倍と低下している．この場合には，どちらも分岐予測ミスは無視可能な頻度であった．

**表 3**　評価に使用したテキスト文書の詳細
Table 3　Properties of text files used in our evaluation.

| 文書名 | 言語 | 総文字数 | 1 バイトコード文字数 | 2 バイトコード文字数 | 3 バイトコード文字数 |
|---|---|---|---|---|---|
| ASCII | - | 10,000 | 10,000 (100%) | 0 (0.0%) | 0 (0.0%) |
| Japanese | - | 10,000 | 0 (0.0%) | 0 (0.0%) | 10,000 (100%) |
| random | - | 10,000 | 3,361 (33.6%) | 3,315 (33.2%) | 3,324 (33.2%) |
| Web-JP | 日本語 | 27,529 | 26,460 (96.1%) | 2 (0.0%) | 1,067 (3.9%) |
| Web-CN | 中国語 | 29,778 | 28,904 (97.1%) | 2 (0.0%) | 872 (2.9%) |
| Web-KR | 韓国語 | 19,050 | 18,357 (96.4%) | 1 (0.0%) | 692 (3.6%) |
| Web-DE | ドイツ語 | 25,140 | 25,125 (99.9%) | 15 (0.0%) | 0 (0.0%) |
| Web-RU | ロシア語 | 18,305 | 17,174 (93.8%) | 1,131 (6.2%) | 0 (0.0%) |
| Mail1-JP | 日本語 | 5,488 | 2,485 (45.3%) | 0 (0.0%) | 3,003 (54.7%) |
| Mail2-JP | 日本語 | 6,617 | 2,301 (34.8%) | 0 (0.0%) | 4,316 (65.2%) |

### 4.3　UTF-8 のデコード性能

本手法を用いて UTF-8 文字列のデコードを行う場合の性能について性能測定を行った．本評価で入力に用いたテキスト文書は，基本的なテキストとして ASCII 文字（UTF-8 表現で 1 バイト）のみのテキスト，日本語文字（同 3 バイト）のみのテキスト，ASCII 文字，日本語文字，ギリシア文字（同 2 バイト）をランダムに並べたもの，様々な言語の Web ページ，日本語のメール文書を用いた．これらの特性を**表 3** にまとめる．ただし，性能評価に用いたテキスト文書には補助文字は含まれなかった．

**図 7** に本手法での変換処理の従来のスカラ処理手法と比較した相対性能を示す．ASCII 文字のみ，日本語文字のみの単純なテキストの場合には，どちらも 10 倍を超える性能向上が得られている．これは同じ文字種が続く場合には，どちらの手法でも同じ文字種の繰返しに最適化された処理を行うが，本手法の場合には SIMD 命令による並列度を活かして，より効率良く行うことができるためである．一方，符号化データ長がランダムなテキストの場合にも本研究の手法により 3 倍以上の性能向上が得られている．この場合には同じ文字の繰返しに対する最適化は性能向上に貢献しないが，本手法はスカラでの変換と比較して分岐予測ミスによるオーバヘッドがないという利点により，やはり性能向上を得ることができる．

より現実的なテキスト文書においても，図 7 に見られるように，2.0 倍から 9.9 倍の性能


© 2008 Information Processing Society of Japan

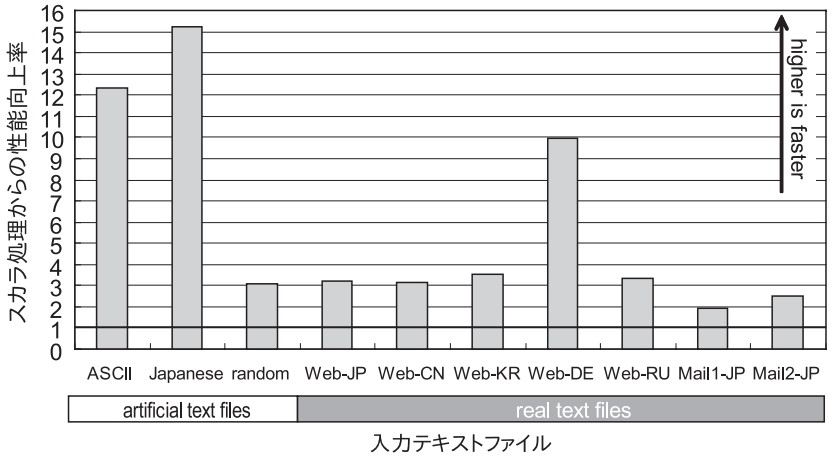


**図 7**　UTF-8 文字列デコード処理の性能
Fig. 7　Performance of deconging UTF-8 sequences.

向上が得られた．現実的なテキスト文書では，複数の文字種が混在するが，ある程度の同種文字の繰返しが存在することになる．しかし，上に示したように本研究の手法は，同種文字の繰返しがある場合，ない場合のどちらにおいても性能向上を得ることができるため，現実的なテキスト文書においても高い効果を示した．

図 7 の結果では 10 KB 以上の比較的長い文字列の変換性能を示したが，本手法はベクタレジスタ長の数倍程度の文字列長があれば有意な高速化が得られた．変換文字列がより短い場合には複数の変換をまとめて行うように処理手順を変えることでより有効に本手法の効果を得ることができる．

## 5. 今後の課題

本論文では UTF-8 から UTF-16 への変換処理のみを対象とした．Java 環境をはじめ多くの場合に UTF-16 から UTF-8 にエンコードする処理も同様に多くの実行時間を占め，この処理の高速化も重要な課題である．今回提案した手法は，ほぼそのままで可変長符号化のエンコード処理にも適用可能であるが，この場合には出力するデータ長がデータに応じて変化してしまうことが問題になる．多くの SIMD 命令セットではベクタ長にアラインしないメモリアクセスは大きなオーバヘッドを生じ，これは特にロードよりもストアで顕著になる．本手法でエンコードを行う場合にはこのアラインしないメモリストアのオーバヘッドにより性能向上が得られなかった．そのためアラインしないメモリストアを高速化するソフトウェアもしくはハードウェア技術が確立されれば，本手法の適用範囲はさらに拡大することが期待できる．

また，今回の実装では，SIMD 命令を C 言語上でイントリンシックを用いて直接記述した．しかし，Java 環境では文字コード変換は一般に Java 言語で記述された標準ライブラリによって行われる．Java 言語では SIMD 命令などの環境依存の命令を直接記述することが困難である．そのため，本論文の手法を Java 処理系に適用するためには，たとえば JIT コンパイラでのイディオム認識技術[12] などを使用して，Java 言語で記述された文字コード変換処理を SIMD 命令を使用した処理に置き換える必要がある．

## 6. ま と め

本論文では可変長符号化されたデータのデコード処理を SIMD 命令を使用して効率良く行う手法を提案し，UTF-8 から UTF-16 への文字コード変換への適用・実装・評価を行った．この手法では，データごとに条件分岐を用いて変換を行う代わりに，複数のデータについて符号化データ長を調べ，その組合せに応じた定数パラメータを使用して permute 命令などによる演算でデコード処理を行う．そのため，SIMD 命令のデータ並列性による性能向上に加えて，分岐予測ミスのオーバヘッドの削減効果が得られる．様々なテキストデータを用いた評価の結果，SIMD 命令を用いない従来の手法と比較して，単純なケースでは 10 倍以上，実際のテキスト文書を用いた場合でも 2 倍以上の性能向上が得られることを示した．

## 参 考 文 献

1) The Unicode Consortium: The Unicode Standard, Version 5.0 (2007).
2) Standard Performance Evaluation Corporation: SPECweb2005. http://www.spec.org/web2005/
3) The Apache Software Foundation: Apache Tomcat. http://tomcat.apache.org/
4) Zhou, J. and Ross, K.A.: Implementing database operations using SIMD instructions, *Proc. ACM SIGMOD International Conference on Management of Data*, pp.145–156, ACM (2002).
5) Govindaraju, N.K., Gray, J., Kumar, R. and Manocha, D.: GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management, *Proc. ACM SIGMOD international conference on Management of data*, pp.325–336, ACM (2006).


© 2008 Information Processing Society of Japan

6) Inoue, H., Moriyama, T., Komatsu, H. and Nakatani, T.: AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors, *Proc. IEEE International Conference on Parallel Architectures and Compilation Techniques*, pp.189–198, IEEE (2007).
7) IBM Corp.: *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*.
8) Pham, D., Asano, S., Bolliger, M., Day, M.N., Hofstee, H.P., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang, M., Warnock, J., Weitzel, S., Wendel, D., Yamazaki, T. and Yazawa, K.: The Design and Implementation of a First-Generation CELL Processor, *International Solid-State Circuits Conference*, pp.184–185, IEEE (2005).
9) IBM Corp., Sony Computer Entertainment Inc. and Toshiba Corp.: *SPU Instruction Set Architecture*.
10) Advanced Micro Devices, Inc.: 128-Bit SSE5 Instruction Set. http://developer.amd.com/SSE5
11) Freescale Semiconductor Inc.: *AltiVec Technology Programming Interface Manual*.
12) Kawahito, M., Komatsu, H., Moriyama, T., Inoue, H. and Nakatani, T.: A new idiom recognition framework for exploiting hardware-assist instructions, *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.382–393, ACM (2006).

(平成 20 年 2 月 17 日受付)
(平成 20 年 3 月 28 日採録)

**井上　　拓**（正会員）
1977 年生．2000 年慶應義塾大学理工学部システムデザイン工学科卒業．2002 年同大学院理工学研究科総合デザイン工学専攻修士課程修了．同年日本アイ・ビー・エム（株）に入社．現在，同社東京基礎研究所に勤務．プログラム最適化技術の研究に従事．

**小松　秀昭**（正会員）
1960 年生．1985 年早稲田大学大学院理工学研究科電気工学専攻修了．同年日本 IBM 東京基礎研究所入社．コンパイラ，アーキテクチャ，並列処理の研究に従事．博士（情報科学）．

**中谷登志男**（正会員）
1975 年早稲田大学理工学部数学科卒業．同年日本アイ・ビー・エム（株）入社．1985 年米国プリンストン大学大学院コンピュータ・サイエンス学科より M.S.E. および M.A.，1987 年 Ph.D. を各取得．同年より同社東京基礎研究所においてプログラム最適化技術の設計・実装・評価の研究に従事．2000 年よりディスティングイッシュト・エンジニア．現在同研究所システムズ担当．基礎研究部門におけるコンパイラ・ファームウエア技術のストラテジストを兼任．ACM，IEEE 各会員．ACM ディスティングイッシュト・エンジニア．


© 2008 Information Processing Society of Japan