Accelerating UTF-8 Decoding Using SIMD Instructions

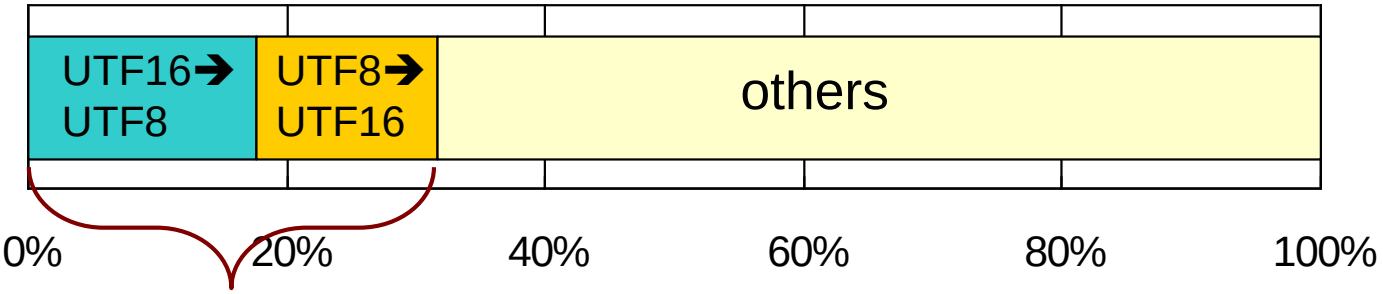
Hiroshi Inoue
IBM Research – Tokyo

Mar 17, 2008

Background and Goal

- Background
  - UTF-8 encoding is commonly used to exchange text data (e.g. HTML, XML)
  - But, Java VM uses UTF-16 as its internal representation

CPU time for SPECweb2005 on Tomcat 6



  - Conversion between UTF-8 and UTF-16 matters

➔Goal

  - Accelerate conversion from UTF-8 to UTF-16

UTF-8 and UTF-16

- What is UTF-8 and UTF-16
  - UTF-8: variable length encoding (from 1 byte to 3 bytes per character)
  - UTF-16: constant length encoding (2 bytes, other than surrogate pair)
- Mapping UTF-8 onto UTF-16

range	UTF-8	UTF-16
00-7F (ASCII char)	0xxxxxxx	00000000xxxxxxx
80-7FF (Greece char etc)	110yyyyy 10xxxxxx	00000yyyyyxxxxxx
800-FFFF (Chinese, Japanese char etc)	1110zzzz 10yyyyyy 10xxxxxx	zzzzyyyyyyxxxxxx

Decoding UTF-8 to UTF-16

Naive implementation of decoding process based on *conditional branches*

```
while (not end of UTF-8 sequence) {
    if first byte of UTF-8 starts with '0' then
        convert                                  0xxxxxxx ➔ 000000000xxxxxxx
    else if first byte of UTF-8 starts with '110' then
        convert                      110yyyyy 10xxxxxx ➔ 00000yyyyyxxxxxx
    else if first byte of UTF-8 starts with '1110' then
        convert   1110zzzz 10yyyyyy 10xxxxxx ➔ zzzzyyyyyyxxxxxx
}
```

Main source of overhead

➔Branch mispredictions caused by data-dependent conditional branches that are hard to predict (when having multiple types of characters)

simple encoding format for explanation

We use a simplified format "**simple encoding**" for ease of explanation

simple encoding

- variable length encoding for integers up to 30-bits long
- 1 value is encoding to 1 to 4 bytes
- Most significant 2 bits of the first byte show the length

length	representation in simple encoding
1 byte	00xxxxxx
2 bytes	01yyyyyy xxxxxxxx
3 bytes	10zzzzzz yyyyyyyy xxxxxxxx
4 bytes	11aaaaaa zzzzzzzz yyyyyyyy xxxxxxxx

Our Technique without conditional branches

- Steps for conversion
  - Step 1: Identify data length for next few data (by scalar)
  - Step 2: Load required constants based on the data length identified in Step 1 (by scalar)
  - Step 3: Move data using permute instruction (by SIMD)
  - Step 4: Mask off unused bits (by SIMD)

SIMD Permute (Shuffle) instruction

➔reorder input bytes based on a pattern specified by register at runtime (not compile time!)

➔many SIMD ISAs have similar instructions (e.g. VMX, SSE)

Pseudo Code for Decoding S

```
// Step 1: Identify data length for next
int gathered_prefix = 0;
int position = 0;
for (i=0; i<4; i++) {
   prefix = (p[position] >> 6);
   gathered_prefix = (gathered_prefix << 2)
   position += prefix+1;
}

// step 2: Load required constants
vector char pattern = pattern_table[gathered_prefix];
vector char mask    = mask_table[gathered_prefix];

// step 3: Move data using permute instruct
vector char vin = vector_load(p);
vin = vector_permute(vin, pattern);

// step 4: Mask off unused bits
output = vector_and(vin, mask);
```

- Gather prefix bits from 4 data
- No hard-to-predict conditional branches!

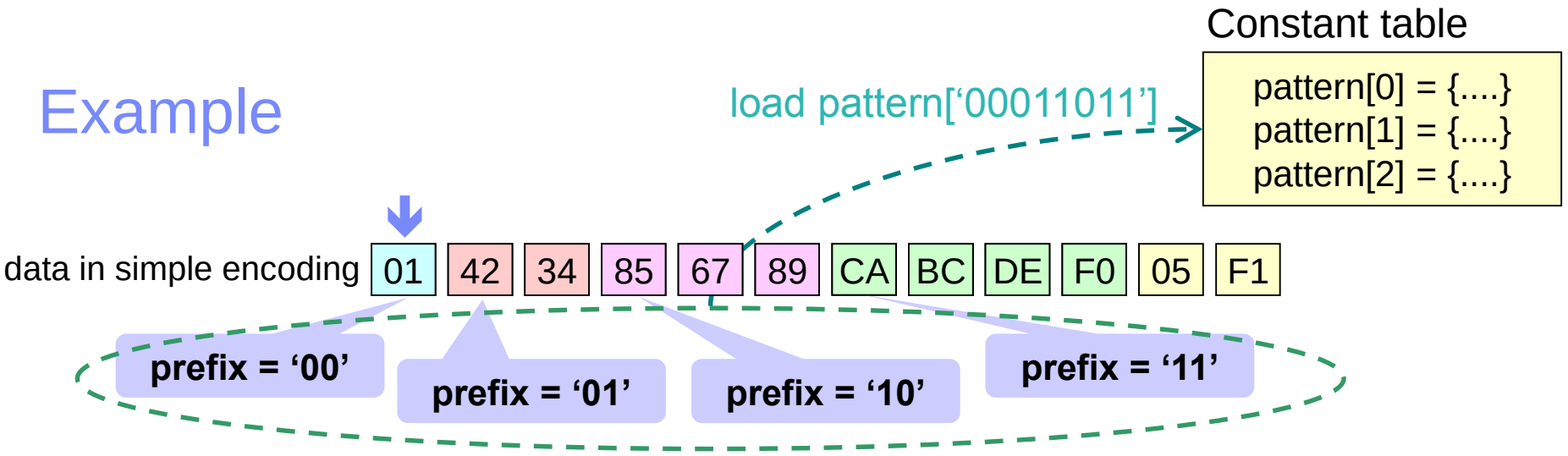
- Can be overlapped with the Step 2-4 of the previous iteration

- Load constants using prefix values gathered in Step1
- Each table uses 4 KB memory for Simple encoding

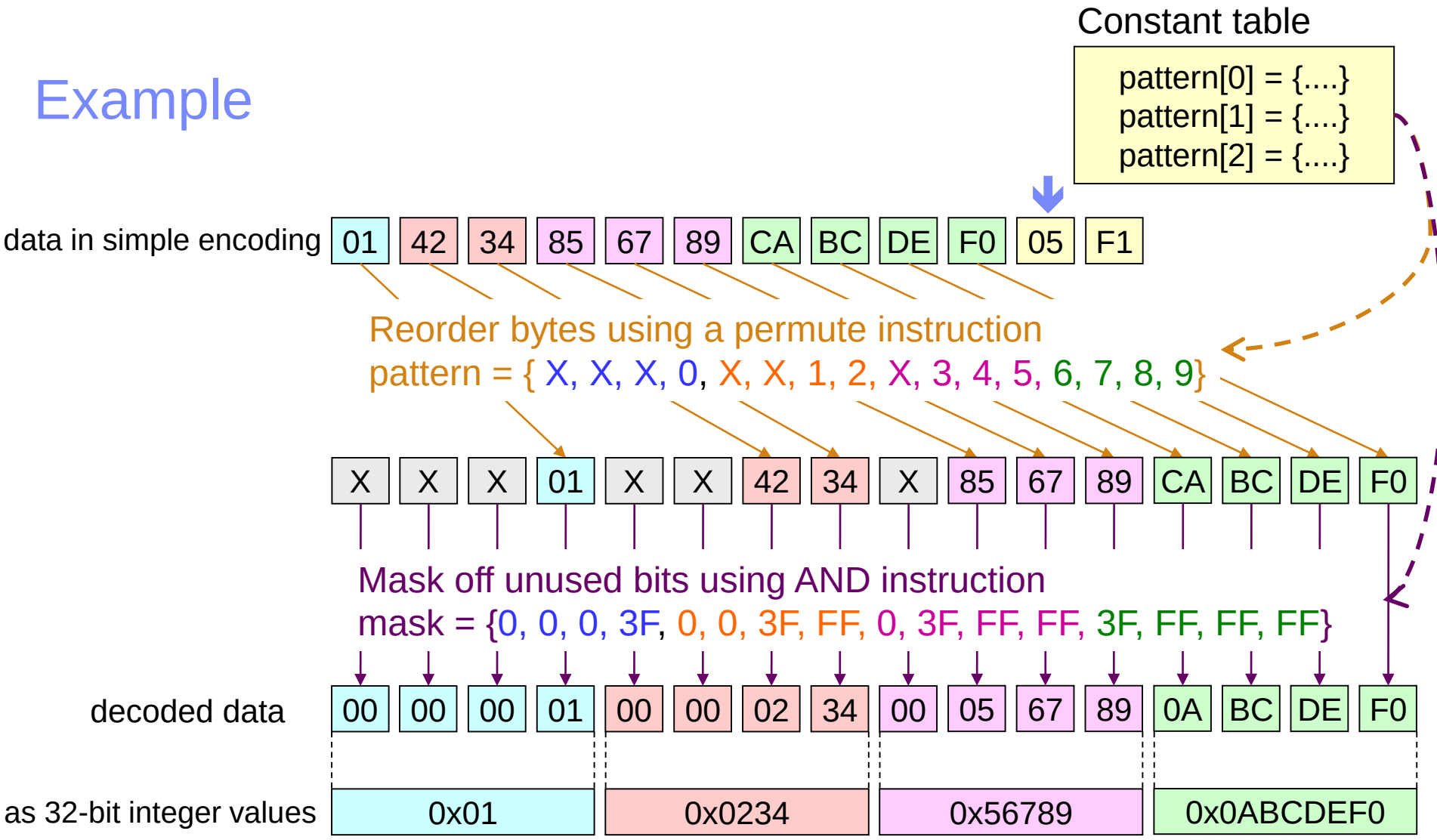
- Move data using permute instruction

- Mask off bites

Example



Example



Apply our technique for UTF-8 to UTF-16 conversion

- Not that different from the case for simple encoding...
- Some difference:
  - use two permute and two select instructions instead of just one permute instruction
  - use four constant vector values per iteration
  - convert 8 characters (2 bytes each) in each iteration

UTF-8 to UTF-16 Conversions

```
// step 1: gather prefix of 8 characters and convert them to length in by
for (i=0; i<8; i++) {
    prefix = (p[total_length] >> 3);
    length = prefix_to_length_table[prefix];
    gathered_prefix = (gathered_prefix * 3) + length;
    total_length += length;
}

// step 2: load constants from tables
vpattern1 = pattern1_table[gathered_prefix];
vpattern2 = pattern2_table[gathered_prefix];
vmask1    = mask_for_select_table[gathered_prefix];
vmask2    = mask_for_and_table[gathered_prefix];

// step 3: move data bits using constants
vin1  = vector_load(p);
vin2  = vector_load(p+16);
vtmp1 = vector_permute(vin1, vin2, vpattern1);
vtmp2 = vector_permute(vin1, vin2, vpattern2);
vtmp1 = vector_select(vector_shift_left(vtmp1, 4),
                      vector_shift_left(vtmp1, 6),
                      vconstant_0x0FC0);
vtmp3 = vector_select(vtmp1, vtmp2, vmask1);

// step 4: mask off unused bits
vout  = vector_and(vtmp3, vmask2);
```

process 8 characters in each iteration

load 4 constants

load 2 vectors as input to generate 8 characters as output

use permute, shift, select instructions for data movement

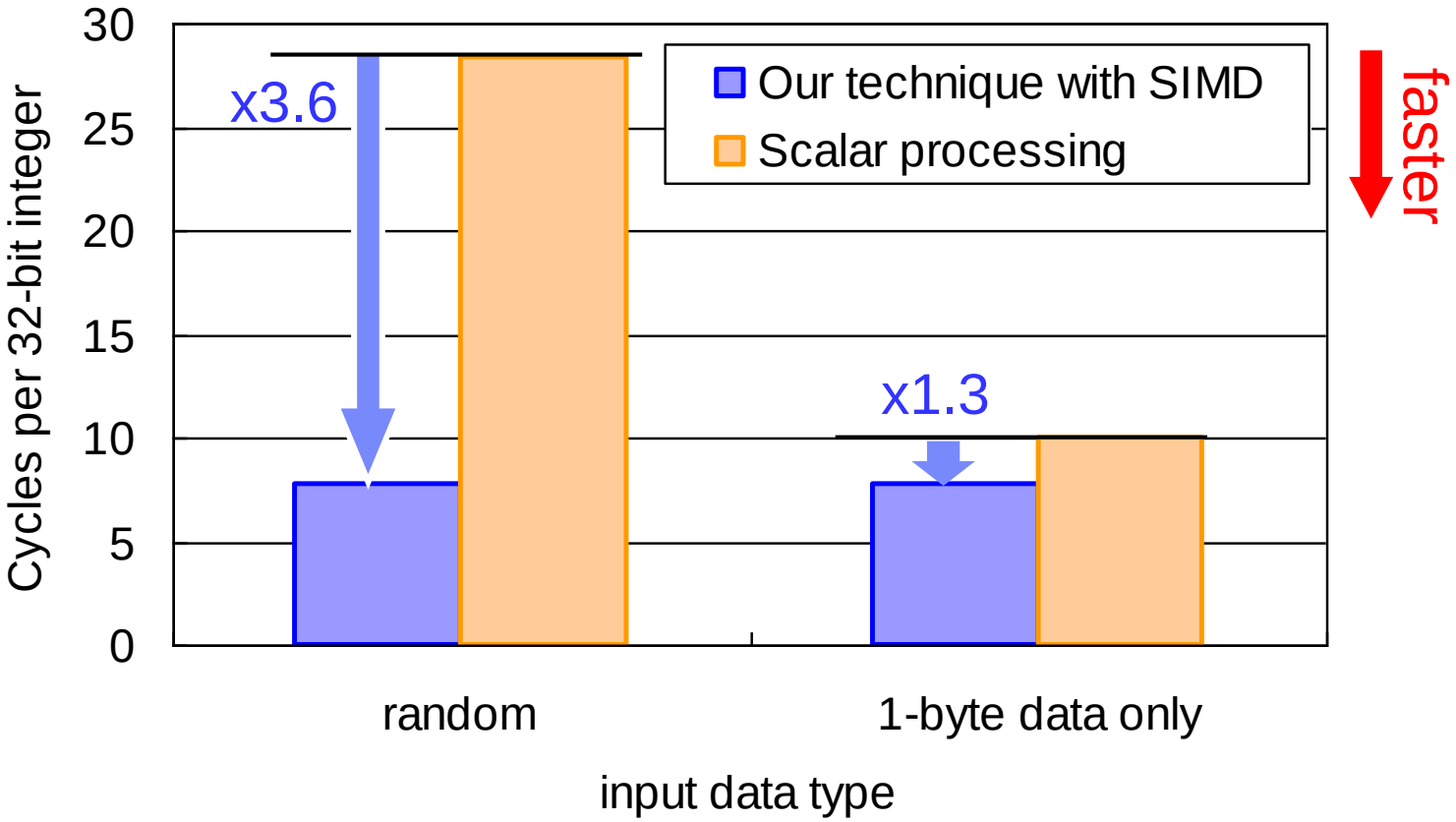
Additional optimization for real-world text

- Same type (i.e. same length in UTF-8 representation) of characters tends to appear repeatedly in real-world data
  - ➔ If all characters converted in an iteration have same type, we use specialized code for this data type
    - Specialized code for ASCII characters are especially efficient because it simply inserts 0 before each byte (0xxxxxxx ➔ 00000000 0xxxxxxx)
    - Specialized code fall backs to the default code if input values include a value of other data type
- Applied for both vectorized version and serial version

Evaluation

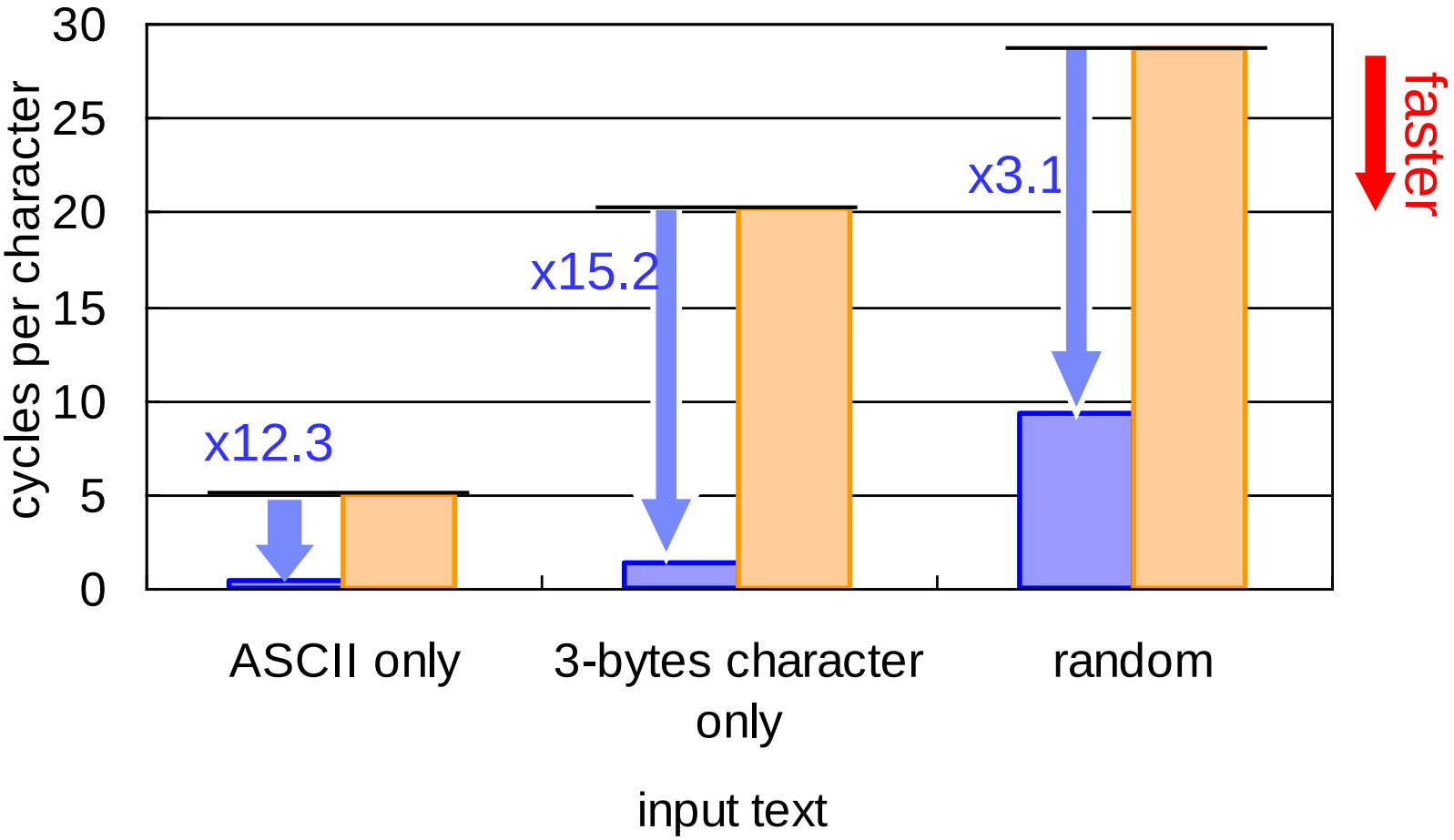
- Tested on PowerPC 970MP 2.5 GHz running Linux 2.6.18
- Implemented using VMX intrinsics
- Compiled with gcc-4.0.1

Decode performance for **simple encoding** format

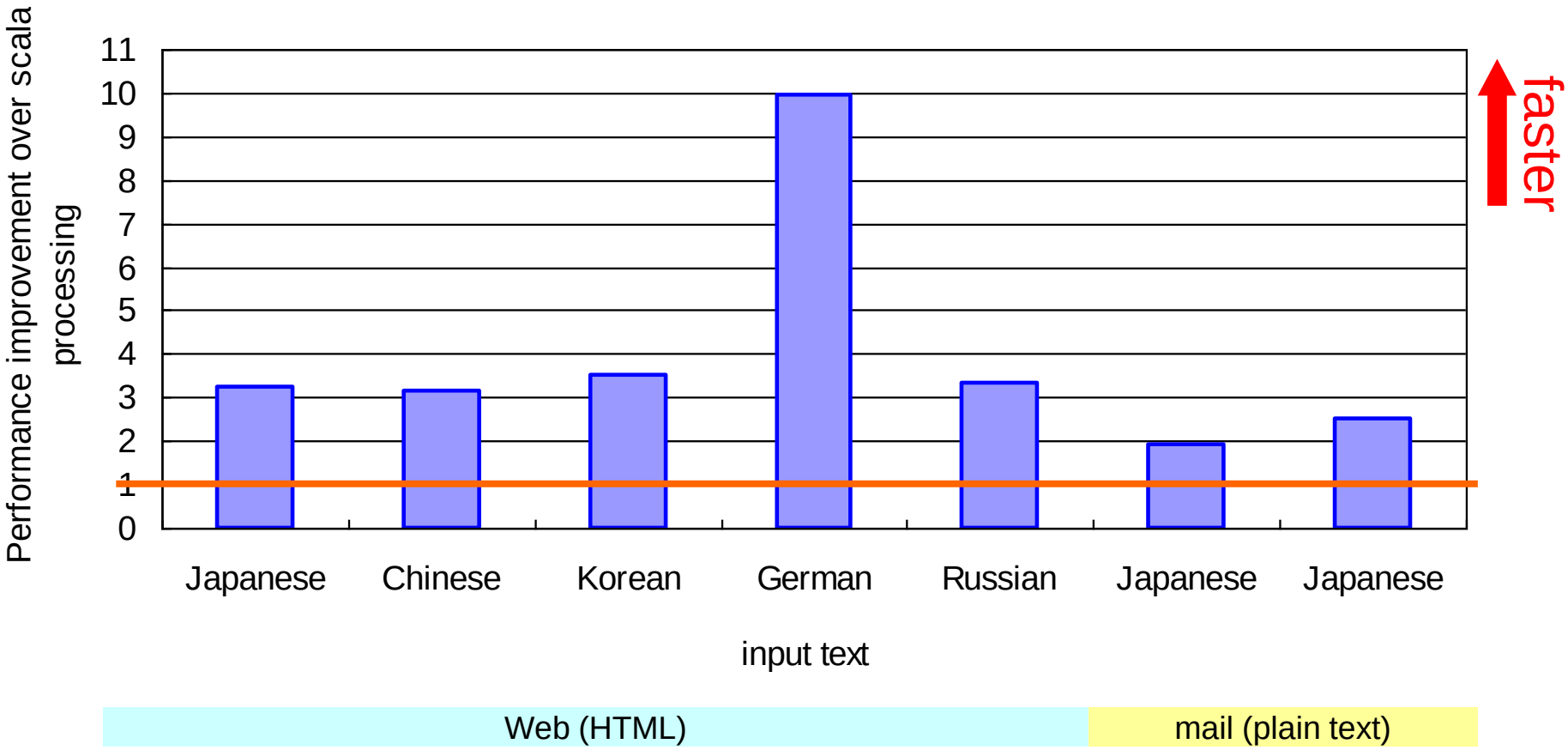


- Scalar processing caused frequent branch mispredictions for random input
- Performance of our technique does not depend on input data

Performance of UTF-8 decoding with artificial text



Performance of UTF-8 decoding with realistic text



Future work

- To integrated into Java environment
  - Conversion is implemented in Java, where programmer cannot directly write platform-dependent SIMD code
  - ➔JIT compiler must generate SIMD instructions for each platform
- UTF-16 to UTF-8 conversion
  - As important as UTF-8 to UTF-16 conversion
  - ➔Our technique is applicable. However, writing UTF-8 sequence to memory require costly unaligned memory store

Summary

- We developed decoding technique for variable-length encoding format without using hard-to-predict conditional branches
- We observed significant performance boost in UTF-8 to UTF-16 conversion with real-world text data

backup

Comparison with Cameron [PPoPP 2008]