SFVInt: Simple, Fast and Generic Variable-Length Integer Decoding using Bit Manipulation Instructions

Gang Liao, Ye Liu, Yonghua Ding, Le Cai, Jianjun Chen

DAMON

DATA MANAGEMENT ON NEW HARDWARE

ByteDance 字节跳动

Introduction & Motivation

- Variable-Length Integers (Varints) are ubiquitous in data systems and frameworks

- Decoding **LEB128 varints** is a performance bottleneck
  - Unpredictable lengths lead to branch mispredictions
  - Limited vectorization opportunities

- Goal: Leverage **BMI2** instructions to accelerate LEB128 decoding

Background - Varints

Varints: encode integers using variable number of bytes

- **Smaller numbers use fewer bytes**
- Space-efficient for data with predominantly small integers

LEB128: widely adopted varint format

- Encodes integers as sequence of **bytes**
- **Continuation Bit: Most significant bit** of each byte indicates continuation
- Decoding requires byte-by-byte processing

```
MSB ------------------ LSB  Using 624486 as an example
      10011000011101100110  In raw binary
     010011000011101100110  Padded to a multiple of 7 bits
 0100110  0001110  1100110  Split into 7-bit groups
00100110 10001110 11100110  Add high 1 bits on all but last
(most significant) group to form bytes
    0x26     0x8E     0xE6  In hexadecimal

→ 0xE6 0x8E 0x26            Output stream (LSB to MSB)
```

ByteDance字节跳动

Background - BMI2 Instructions

BMI2: Bit Manipulation Instruction Set 2

- Available in modern Intel and AMD CPUs
- Enables fast bit-level operations

Key instructions for varint decoding:

- PDEP: Parallel Bit Deposit
  - Deposit bits from src to dest based on mask
- PEXT: Parallel Bit Extract
  - Extract bits from src to dest based on mask

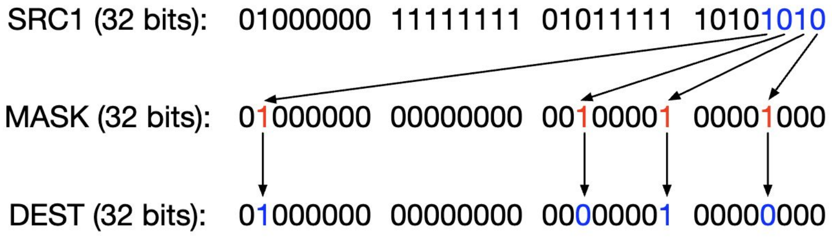


Figure 1. PDEP Example.

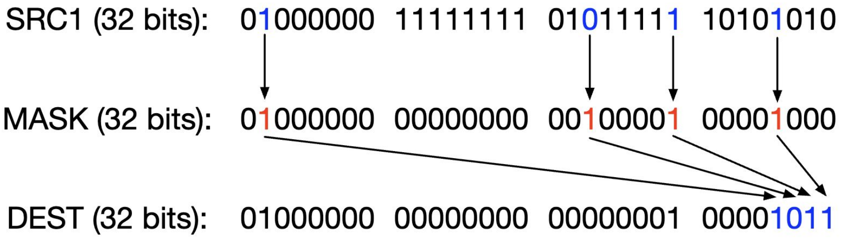


Figure 2. PEXT Example.

ByteDance 字节跳动

SFVInt Approach Overview

- Leverage BMI2 for fast and efficient varint decoding
- Simple: ~500 lines of code
- Fast: Up to 2x decoding speed compared to existing systems
- Generic: Unified C++ template for 32-bit and 64-bit integers
- **Key ideas**:
  - **Use PEXT to extract integer counts and positions**
  - **Tailored masks for efficient integer extraction**
  - **Handle cross-boundary cases**

Basic Varint Operations

Varint Encoding **(referring to slide 3)**:

- Divide integer into bytes with the lower 7 bits actual store data
- Set continuation bit for all but last byte

Varint Decoding **(focused)**:

- Read bytes and extract 7-bit groups

  (with PEXT)

- Reconstruct original integer

ByteDance字节跳动

https://en.wikipedia.org/wiki/Variable-length_quantity

BMI2-Enhanced Bulk Varint Decoding

Mask configuration for parallel processing

- Opt1: 6-byte mask (0x000080808080808080) process 6 bytes of encoded data
- Opt2: 8-byte mask (0x8080808080808080) process 8 bytes of encoded data
- Opt3: …
- Balance between decoding efficiency and instruction cache usage

Tailored masks for efficient integer extraction :

- Empirically, a 6 byte-mask configuration provides the best performance
- Here, for simplicity, we demonstrate with an 8 byte-mask

Extracting integer counts and positions:

- Use PEXT with mask to get varint structure (the continuation bits)
- Switch statement to handle different cases based on the varint structure

ByteDance 字节跳动

BMI2-Enhanced Bulk Varint Decoding I

1A. Mask configuration for parallel processing using an 8-byte mask (0x8080808080808080), “word” is 8-byte segment in the encoded data for processing

value = _pext_u64(word, 0x8080808080808080)

If yields 0 (00000000), it indicates 8 complete integers in the segment

```
int1 = _pext_u64(word, 0x000000000000007f);
int2 = _pext_u64(word, 0x0000000000007f00);
int3 = _pext_u64(word, 0x00000000007f0000);
int4 = _pext_u64(word, 0x000000007f000000);
int5 = _pext_u64(word, 0x0000007f00000000);
int6 = _pext_u64(word, 0x00007f0000000000);
int7 = _pext_u64(word, 0x007f000000000000);
int8 = _pext_u64(word, 0x7f00000000000000);
```

ByteDance字节跳动

BMI2-Enhanced Bulk Varint Decoding II

1B. Mask configuration for parallel processing using an 8-byte mask (0x8080808080808080), “word” is 8-byte segment in the encoded data for processing

value = _pext_u64(word, 0x8080808080808080)

If yields 63 (00011111), it indicates 3 complete integers in the segment

```
int1 = _pext_u64(word, 0x00007f7f7f7f7f7f);
int2 = _pext_u64(word, 0x007f000000000000);
int3 = _pext_u64(word, 0x7f00000000000000);
```

BMI2-Enhanced Bulk Varint Decoding III

2. Cross-Boundary Cases: integers can span multiple 8-byte segments, maintain state using shift_bits and partial_value

- shift_bits: tracks bit displacement for cross-boundary integers
- partial_value: stores previously decoded partial integer

First 8-byte segment (word1)

If value = _pext_u64(word1, 0x8080808080808080)

yields 223 (11011111)

MSB is 1, meaning that the second integer spans to the second 8-byte segment

Second 8-byte segment (word)

```
int1 = _pext_u64(word, 0x00007f7f7f7f7f7f);
int2 = _pext_u64(word, 0x007f000000000000);
int3 = _pext_u64(word, 0x7f00000000000000);
```

```
int1 = (_pext_u64(word, 0x00007f7f7f7f7f7f)
        << shift_bits) | partial_value;
int2 =  _pext_u64(word, 0x007f000000000000);
int3 =  _pext_u64(word, 0x7f00000000000000);
```

Experimental Setup

AWS EC2 instances with diverse CPU architectures

- Intel: Ice Lake, Skylake, Cascade Lake, Haswell
- AMD: EPYC Milan, EPYC 7571, EPYC 7R32

Dataset distributions:

- Uniform: Balanced across value range
- Skewed: Mirror real-world LEB128 patterns

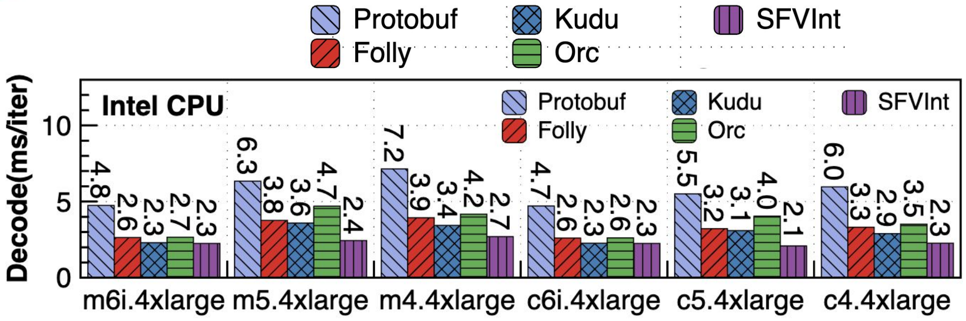
Workloads:

- W1: Uniform 32-bit integers
- W2-W4: Skewed distributions from real-world data

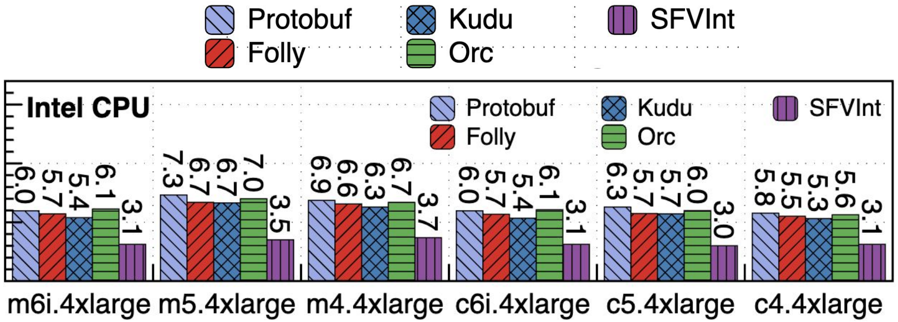
Comparison against Protobuf, Folly, Kudu, ORC

ByteDance字节跳动

Performance Evaluation - Intel CPUs



Workload 1 (W1): uniform distribution.



W4: varint byte distribution — **1 byte: 72.13%**

SFVInt consistently outperforms other systems

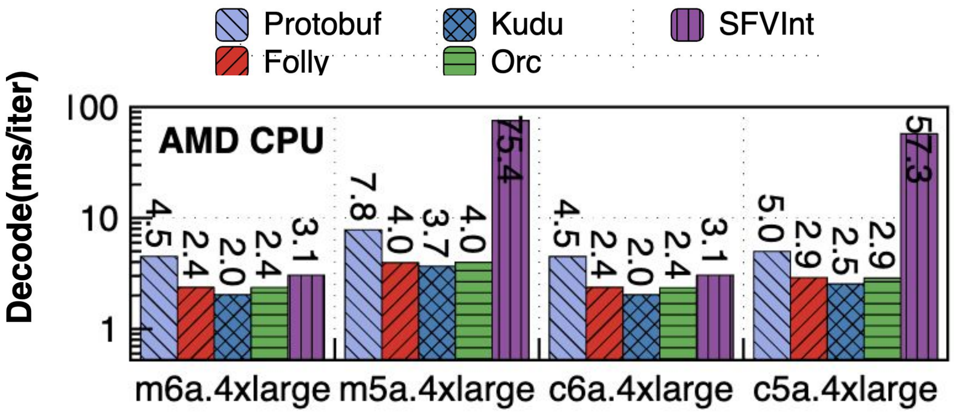
- Up to 2x faster than Protobuf

Performance gap widens with increasing varint lengths

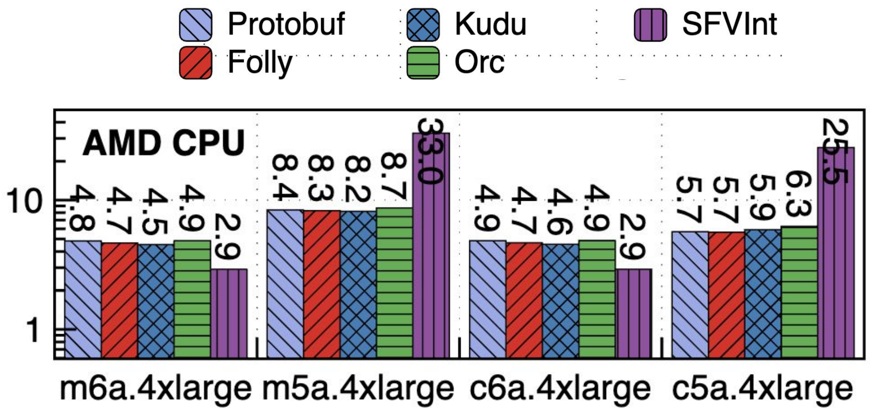
- SFVInt's BMI2 usage excels for multi-byte varints

ByteDance字节跳动

Performance Evaluation - AMD CPUs



Workload 1 (W1): uniform distribution.



W4: varint byte distribution — **1 byte: 72.13%**

SFVInt on **3rd gen EPYC (Milan)**: Up to 40% faster

Slower on 2nd gen EPYC due to BMI2 emulation overhead

- Latency of PEXT/PDEP higher on older AMD CPUs

Future work: Dynamic selection or AMD-specific optimizations

ByteDance字节跳动

Conclusions

SFVInt: A simple, fast, and generic varint decoding approach

- Leverages BMI2 instructions for efficiency
- Achieves up to 2x decoding speedup over existing methods

Future considerations:

- Improving performance consistency on AMD CPUs
- Exploring integration into data processing systems

Thank You

Thank you for your attention!

Contact Info:

- Gang Liao: gangliao@umd.edu
- Ye Liu: ye.liu@bytedance.com