Optimizing Rolling Hash computation using SIMD Vector registers

Described is a technique for speeding up the rolling hash computation used in content-aware chunking for data deduplication. This technique can be conditionally used in certain environments depending on the nature of the rolling hash and the CPU capabilities available. Typically rolling hashes computed for variable chunk data deduplication use a sliding window that keeps track of the bytes in the current hash window. Newest bytes are added to the window while the oldest bytes are popped out. Both the newest byte and the oldest byte values are used in the rolling hash computation.

There are two typical ways of keeping track of the bytes:

1. The simplest is to use two indexes to have a logical sliding window:

   Index1 = current buffer scan position
   Index2 = Index1 – window-size

   Where Index1 starts from buffer start + window-size position after initializing the rolling hash with the first window-size bytes. Now byte at Index1 and byte at Index2 are using in updating the rolling hash. Then Index1 is updated and the process repeated.

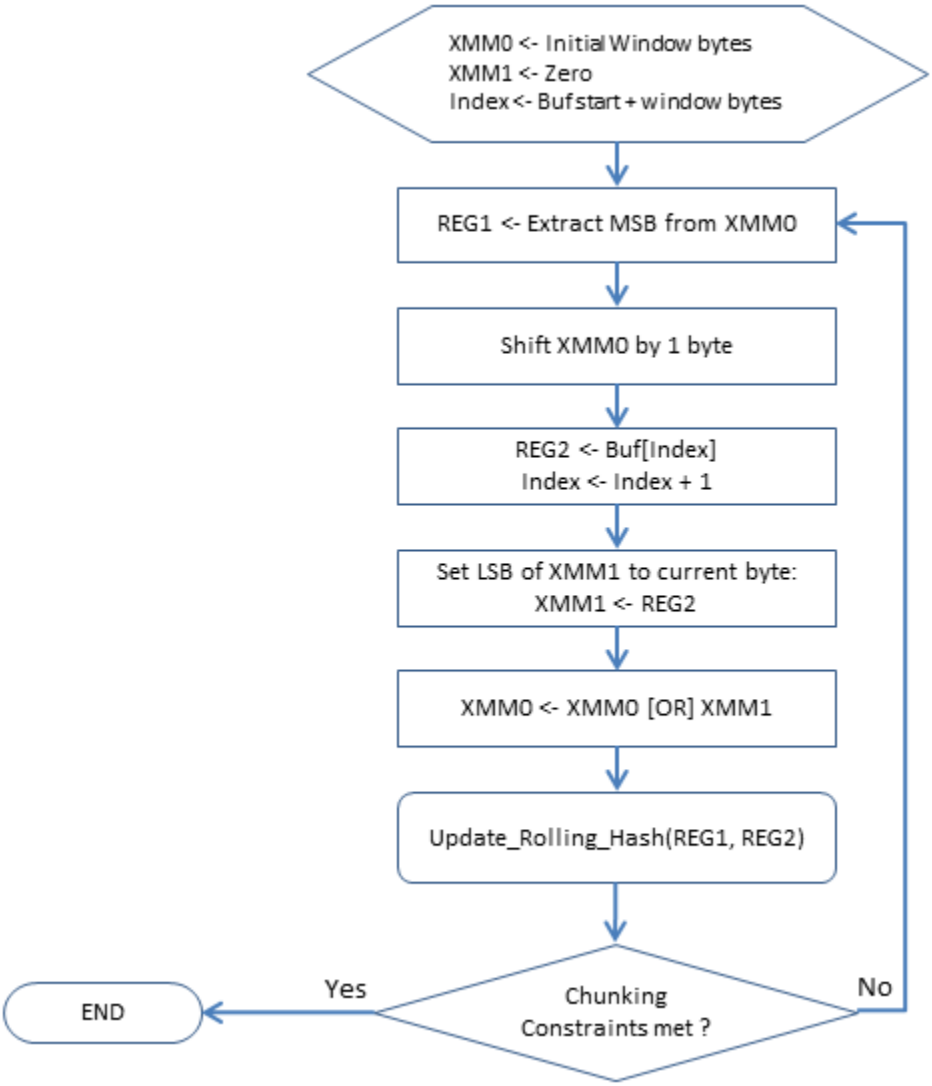
2. Some implementations maintain a separate window-size buffer and keep a window-index that circles thru the buffer round and round. In addition a buffer-index is also present that scans through the given data buffer.
   The byte at the current window-index position and the byte at the current buffer-index position are then used to update the rolling hash.

Both these approaches have performance limiting characteristics. In the first approach we are reading every memory location twice. First read happens when the leading edge of the sliding window reaches the byte and the second read happens when the trailing edge of the window reaches it. It is not a huge issue on modern processors which are endowed with caches. Unless we are using an unusual window size the window data will already be in the L1 cache. However in byte by byte scanning it can happen that the trailing edge byte is from the previous cacheline which may have to be fetched. The second approach requires a memory write for every byte being accessed. Memory writes even to the same locations repeatedly are expensive so this is worse than the previous approach.

The idea proposed here is to leverage the large SIMD vector registers on modern processors to store the window bytes and slide the window forward. For this to be practical a few requirements must be met:

a) The vector register should be large enough.
b) The rolling hash window size should be selected to fit into one vector register.
c) The CPU should support byte-wise shift operations on vector registers.
d) Ideally the CPU should also support instructions to fetch a specific byte or a word from the vector register cheaply into a regular register.

If we take the example of a modern x86 processor with SSE4 capability and 128-bit or 16-byte SSE registers we need to consider a rolling hash that uses at most a 16-byte window. If the processor supports AVX2 then we get 32-byte AVX registers. Assuming a window size of 16 bytes and SSE4 capability, the following flowchart illustrates the algorithm. XMM0 and XMM1 are notations for SSE4 vector registers 0 and 1 respectively and REG1 and REG2 are notations for regular 32-bit registers.



This approach avoids spurious memory reads and writes and accesses the bytes of the buffer only once. There is a good performance gain to be had as we are doing sliding window handling purely within registers. The same approach can also be easily implemented on the ARM NEON vector instruction set as well. The following pseudocode and SSE4 intrinsics show the core implementation logic for x86 processors:

```
# Initial conditions
XMM0 <= Initial Window bytes
XMM1 <= Zero
Index <= Buffer start + window bytes
# Core logic
Do {
  REG1 = _mm_extract_epi8(XMM0, 15);
  XMM0 = _mm_slli_si128(XMM0, 1); # Slide window bytes left
  REG2 = buffer[Index];
  Index = Index + 1;
  Move(REG2, XMM1); # XMM1 LSB <= REG2 (which is current byte)

  XMM0 = _mm_or_si128(XMM0, XMM1); # Current byte added to window
  Update_Rolling_Hash(REG1, REG2);
} While(Chunking_Constraints() == False);
```

Relevant Tags

Single Instruction Multiple Data
Vectorized hash function
X86 Processor Vector Instructions
Streaming SIMD Extensions
Advanced Vector Extensions
Sliding Window Rolling Hash
Content-Aware Chunking
Data De-Duplication