A Fast SIMD-Based Chunking Algorithm

Yehonatan Dude, Michael Hirsch, and Yair Toaff

Toga Networks, a Huawei Company
4 Ha'Harash St, Hod Hasharon, Israel
{johnny.david, michael.hirsch, yair.toaff}@toganetworks.com

Abstract. Deduplication is a special case of data compression where repeated chunks of data are stored only once. The input data is divided into chunks using a chunking algorithm and a cryptographically strong hash is calculated on each chunk and used as its unique identifier for further searching and duplicate elimination. As the input stream is processed, a chunk boundary is declared at a byte address in the input stream if some weak hash of a fixed number of preceding bytes (the "hash window") satisfies some criterion. Commonly, a rolling hash like Karp-Rabin [6] or some cyclic polynomial [7] is used for the weak hash since these cheaply support moving the hash window forward one byte in the input stream.
This work presents a way to calculate n weak rolling hashes at a time using single instruction multiple data (SIMD) instructions available on today's processors. Furthermore, it shows how to calculate chunk boundaries cheaply using other instructions also available on these processors. Empirical results show that the proposed algorithm is four times as fast as previous algorithms, and that these optimizations save up to 25% of the computation required for deduplication.

Keywords: chunking algorithm, deduplication, rolling hash

1 Introduction

In todays world, growing quantities of data need to be stored and/or transported. These enormous quantities of data present major costs and complexity challenges with respect to storage space and network bandwidth. Often, the data contains duplicates of data that is available elsewhere in some broader system. Data deduplication is a technique for reducing the data volume by eliminating this repeated data, reducing the storage and transportation requirements.

In the deduplication process of an input data stream, the input stream is divided into chunks which are compressed and stored uniquely in storage. The chunks may be entire files or objects, blocks of a block device, or content-aware variable length chunks. In deduplication, only one unique chunk of a data stream is actually retained while redundant data chunks (which are identical to an already retained data chunk) are replaced with pointers to their respective retained data chunks.

Figure 1 shows an example of an input data stream with two versions of a text file. A small insert or delete in the content of the file shifts the remainder of the file. In the case of file deduplication, this results in the need to store the entire file. In the case of block deduplication, all the blocks including the change and after it need to be retained. By contrast, in the case of content aware deduplication, only the changed chunk needs to be retained.

Content-aware deduplication usually provides the best results with respect to data reduction. The problem with content-aware variable length deduplication is that it is computationally expensive and adds latency to the process. This paper focuses on

Yehonatan Dude, Michael Hirsch, Yair Toaff: A Fast SIMD-Based Chunking Algorithm, pp. 84–91.
Proceedings of PSC 2019, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06618-8

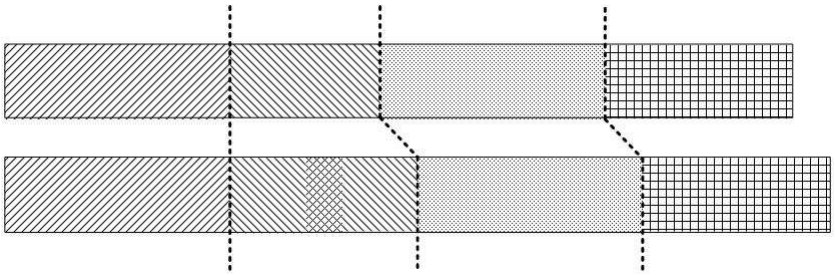

Figure 1. In this example we can see two nearly identical data streams. The second contains a small portion of inserted data. By splitting the stream into chunks using a content aware chunking algorithm, only the chunk containing the inserted data will be retained. The other chunks will be replaced by pointers to the chunks of the first stream.

one of the steps in content-aware variable length deduplication, showing how to chunk data approximately 4 times faster than accepted practice.

The process of content aware deduplication is composed of four CPU intensive parts: chunking, hashing, searching, and compressing. First, the input data stream is divided into chunks using a content aware chunking algorithm. Then a probabilistically unique hash value is generated for each chunk using a hash algorithm. Usually, a cryptographical strength hash algorithm is used for this purpose, for example SHA-2, mainly for its probabilistically unique properties and optimizations that CPU vendors have implemented. Some kind of index data structure maps existing hashes to their locations. This data structure is then searched using this hash value. If the hash is found, the chunk is not retained but rather replaced by a pointer to the existing chunk. If the hash is not found, then it is retained: the chunk is compressed to further reduce storage efficiency, stored, and the hash and its location added to the index data structure.

In this paper we present a chunking algorithm that exploits single instruction multiple data (SIMD) technology in order to process vectors of bytes rather than single bytes, and other instructions to cheaply calculate the criteria for chunk boundaries.

The methods discussed here complements previous work [2,3,4,5] in the field by one of the authors.

This paper is organized as follows. In Section 2, we survey previous algorithms. In Section 3, we present our rolling hash function and an algorithm to calculate it efficiently. In Section 4, we compare its performance with previous algorithms. Conclusions are in Section 5.

2 Related Work

Throughout the paper we will make use of the following notation and terminology. A string x of length $|x|$ is represented as a finite array $x_1x_2\cdots x_n$ of characters from a finite alphabet Σ of size σ. We refer to the i-th element in x as x_i and use the notation $x_i\cdots x_j$ to denote the subsequence of x from the element at position i to the element at position j including both, where $1 \leq i \leq j \leq |x|$.

"Chunking" is a way to split a data stream on context sensitive boundaries. The main goal of effectively chunking the data stream is to ensure that the chunk boundaries are affected as little as possible by changes to the chunks' data contents. A chunk boundary is designated when a (weak) hash of some n consecutive bytes complies with

one or more predefined chunking criteria. In this way, only the last n consecutive bytes affect the boundary, not changes within the chunk.

Calculating even a weak hash at every byte offset in a data stream is expensive. In general, in the industry, rolling hash techniques are used for chunking data streams. A rolling hash is a hash that can be calculated either as a function of a sequence of n bytes $x_i \cdots x_{i+n-1}$ or as a function of rolling hash of $x_{i-1} \cdots x_{i+n-2}$, and the values x_{i-1} and x_{i+n-1}. The latter is usually significantly cheaper to compute. The term "rolling" is used to illustrate how the hash "rolls" from one byte window to the next by removing the effect of the byte leaving the sequence and adding the effect of the byte entering the sequence. As before, the calculated hash value is checked for compliance with some predefined one or more chunking criteria and in case the compliance is identified, the end of the respective rolling sequence is designated as a chunk boundary.

Input: A string $x = x_1 x_2 \cdots x_l$
Result: Chunk Boundary Positions[]
for $i \leftarrow w$ **to** l **do**
  $h \leftarrow$ rolling hash of $x_{i-w+1} \cdots x_i$;
  if *criterion(h)* **then**
    Append i to Chunk Boundary Positions;
  end
end

Algorithm 1: Content Aware Chunking

Algorithm 1 provides a brief implementation of a chunking algorithm using a rolling hash [8], such as cyclic polynomial [7] or Karp-Rabin [6]. As an example, the criterion function here could be true if $h = 0 \bmod N$ and false otherwise, where N is the average chunk size. An alternative criterion function could be accomplished by comparing a bit masked version of the hash to a number, which will give 2^n average chunk size for mask with n bits.

The problem is that chunking in algorithm 1 is CPU-intensive because it iterates over all the bytes. Also, at each byte, there is a certain amount of computation that depends on results computed in the previous iteration. This forces the iteration to rely on sequential execution of the algorithm. In fact, it is so expensive that it takes significantly more time to chunk the data than to hash it using a cryptographic strength hash algorithm (see Section 4.4). On paper, cryptographic strength hashes do more work, but they were designed for parallel execution and modern CPUs have further optimized hardware for them.

This problem exists in many of the proposed algorithms and optimizations, including ones exploiting SIMD, SSE, and other modern CPU architectures.

3 A Fast Chunking Algorithm

This section is divided into two parts. The first is the description of the rolling hash function, and the second is how to calculate it efficiently.

3.1 The Rolling Hash and Boundary Criteria

The hash function we are about to define is designed as a chunking algorithm. Let x be the input data string over an alphabet Σ of size σ. Let k be the size of a vector we use for calculations, l be the number of bits in each element, such that $\sigma \leq 2^l$, and

$w = kl$ be the window size of the rolling hash. We will use $\oplus$ to denote bitwise xor, | to denote bitwise or, & for bitwise and, $\ll$ for bitwise shift left, and $\gg$ for bitwise shift right.

The rotate n bits left of a l bits charecter c function could be written this way:

$$rol(c, n) = \big((c \ll n) \mid (c \gg (l - n))\big) \,\&\, (2^l - 1)$$

We define an intermediate hash h_i at a position $i > w$ in the string x as following:

$$h_i = \bigoplus_{j=1}^{l} rol(x_{i-k(l-j)}, l - j)$$

In the algorithm we discuss here, calculating the criterion function of a vector of intermediate hashes is a two stage process. First, we calculate an intermediate criterion vector of the intermediate hashes. Then the final criterion holds only when a sequence of k intermediate criteria hold. The probability of a boundary to be declared is then the probability of k intermediate criteria holding.

Some criterion function c operates on one intermediate hash value. We define a intermediate criterion result c_i at position i in string x as following:

$$c_i = \begin{cases} 0 & \text{if } c(h_i) \text{ holds} \\ 1 & \text{otherwise} \end{cases}$$

The final criterion C_i for a chunk boundary at position i is a combination of all k preceding intermediate criteria $c_{i-k+1}, c_{i-k+2}, \ldots, c_i$.

$$C_i = \begin{cases} \text{true} & \text{if } \left(\sum_{j=i-k+1}^{i} c_i\right) = 0 \\ \text{false} & \text{otherwise} \end{cases}$$

If C_i is true, a boundary is designated at position i.

Let's show how C_i is calculated with an example. Suppose we have $l = 4$ bits, and a vector of size $k = 4$, and window size of $w = 16$, and let's use the criterion function $c = (b >= h)$, and hence $c_i = (b < h_i)$ (using 0 as "false"). The following four conditions show how to calculate $c_{13} \cdots c_{16}$:

$$\begin{aligned} b < h_{13} &= rol(x_1, 3) + rol(x_5, 2) + rol(x_9, 1) + rol(x_{13}, 0) \\ b < h_{14} &= rol(x_2, 3) + rol(x_6, 2) + rol(x_{10}, 1) + rol(x_{14}, 0) \\ b < h_{15} &= rol(x_3, 3) + rol(x_7, 2) + rol(x_{11}, 1) + rol(x_{15}, 0) \\ b < h_{16} &= rol(x_4, 3) + rol(x_8, 2) + rol(x_{12}, 1) + rol(x_{16}, 0) \end{aligned}$$

Since k is 4, if all 4 $c_{13} \cdots c_{16}$ evaluate to 0, then C_{16} holds, designating a boundary at position 16.

There are a couple of "tricks" here. Firstly, we use a single SIMD instruction to convert a vector of hash values to bits in a computer word that represent whether or not they meet the criterion function (for example _mm256_cmpgt_epi8[1]). Secondly, we use instructions that count sequences of zero bits (for example _tzcnt_u32[1]). In this way, we can count the number of consecutively satisfied conditions in $O(1)$ computer instructions. This combination makes this algorithm efficient.

Below, we discuss how to calculate k intermediate hashes at a time.

3.2 Boundary Criteria Algorithm

Our algorithm is divided into three parts, the first initializes the intermediate hashes, the second checks if there is a boundary at any position among the last intermediate hashes calculated, and third calculates the next intermediate hashes by inserting and omitting new and old vectors (the rolling part). The algorithm operates as following:

Input: A string $x_1x_2\cdots x_n$
Result: Chunk Boundaries (Positions)

```
// Part 1 - Initialize hashes
H ← 0, 0, ..., 0;
for i ← 0 to l − 1 do
    j ← ik;
    V ← x_j x_{j+1} ··· x_{j+k−1};
    H ← rol(H, 1) ⊕ V;
end
i ← lk;
p ← 0;
// Iterate over the string
while i + k ≤ n do
    // Part 2 - Check for boundaries
    B ← b, b, ..., b;
    C ← H ≤ B;
    c ← mask(C);
    s ← p + count_leading_zeros(c);
    if s ≥ k then
        mark a boundary at i − p;
    end
    p ← count_trailing_zeros(c);
    // Part 3 - Calculate the next k hashes
    N ← x_i x_{i+1} ··· x_{i+k−1};
    O ← x_{i−kl} x_{i−kl+1} ··· x_{i−kl+k−1};
    H ← rol(H, 1) ⊕ N ⊕ O;
    i ← i + k;
end
```

Algorithm 2: Generate Chunk Boundaries on an input String

The first part calculates the rolling hash value at the start.

The second part checks for boundaries. In the implemented version of the algorithm we use 0 to denote passing of the criterion c_i in order to count the leading and trailing zeros of the resulting bitmask register. By counting two integers, k bits of length, representing sequential $2k$ intermediate criteria results, we ensure finding any k sequences of pass results. The second integer is saved to be used in the next iteration after we calculate the next k criteria.

And in the third part, we move forward with rolling calculation of the intermediate hashes of the next vector (of k bytes).

4 Results

This section focuses mainly on empirical results we obtained from benchmark tests executed with an AVX version of the code. Our main goal was to produce an algorithm comparable to that of Karp-Rabin in terms of its quality and usefulness for chunking for deduplication, namely keeping the distribution of chunk size similar, but speeding up the calculations.

All tests were performed using AVX instructions which are present in Intel CPUs that are commonly found in today's data center servers. If we had used the newer AVX3 instructions and vector length, we would expect to get a further 3 to 4 factor performance increase.

In this section, we present results from several aspects. The first is a table showing the amortized cost of the different instructions when calculating Karp-Rabin and SIMD chunking. Secondly, we show a simple speed comparison. Thirdly, we show that the chunk sizes resulting from both algorithms on the same data sets are similar. Lastly, we show the effect of this speedup on the overall system performance.

4.1 Workloads Comparison

Command	Cyclic Polynomial	SIMD Optimized
Shift	4	2/16
And	1	0
Or	2	1/16
Xor	2	2/16
Other	2	4/16
Total	11	9/16

Table 1. Number of CPU operations per byte comparison

Different arithmetic operations require different amounts of time. Division for example takes more time than xor. However, for similar operations, the execution time of regular arithmetic operations and that of SIMD arithmetic operations are similar.

In table 1, we can see a comparison between Cyclic Polynomial and SIMD Optimized Chunking written using AVX instructions with a vector of 16 bytes, and a window size of 64 bytes. Theoretically, the rolling part should be 20 times faster.

4.2 Chunk Size Distribution Results

As we noted before, one of the goals was to keep a similar chunk size distribution, because it implicitly affects many aspects of the deduplication system.

For this test, we used a generated corpus of data that emulates the data we expect to work with in a real storage system. We processed the data both with Karp-Rabin and SIMD Optimized Chunking, and counted the number of chunks in each range of sizes, consisting of 32 buckets of size ranges, each containing about 447 discrete sizes.

In figure 2, we see visually that the chunk size distribution is very similar. In fact, they differ by less than 2%. The end result, that is the quality of the deduplication, is unchanged.

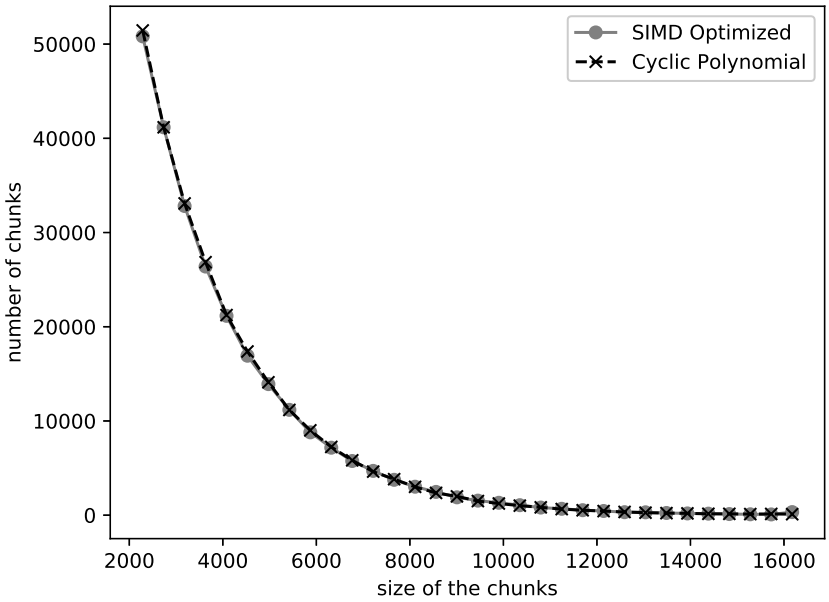

Figure 2. Chunk Size Distribution Comparison

Algorithm	Random Data	Mixed Data
Karp-Rabin	975 MB/s	927 MB/s
Cyclic Polynomial	1675 MB/s	1676 MB/s
SIMD optimized	6715 MB/s	7136 MB/s

Table 2. Benchmark of the chunking algorithms

4.3 Chunking Performance Results

In table 2, we see the performance results from executing the chunking algorithms alone. Karp-Rabin, Cyclic Polynomial, or SIMD Optimized Chunking are compared. Each algorithm was run on two sets of data: random data, and the same corpus data used previously to test the chunk size distribution.

4.4 Deduplication Performance Results

We also ran an emulation of an entire deduplication system, including chunking, hashing, and compressing data. The process was limited to a single thread on a single CPU core.

Chunking Algorithm	MB/s	CPU Usage			
		LZ-4	Sha-1	Other	Chunking
Karp-Rabin	262	63.9%	4.1%	3.8%	28.1%
SIMD Optimized	345	84.5%	5.4%	5.1%	4.7%

Table 3. Benchmark of deduplication

In table 3, we see that the throughput of the deduplication system increased by 32% from 262MB/s to 345MB/s. We also show how the CPU usage is split between the most CPU intensive parts.

5 Conclusions

This paper has shown a new way to calculate a rolling hash. It shows a way to "roll" the hash an entire vector at a time. The process of rolling a whole vector at a time maps cleanly onto SIMD instructions available in today's CPUs, making for fast implementations. In this way, only a fraction of the time is needed to calculate the hashes compared to previous methods. In a further step, this paper also shows how to evaluate a criterion function a whole vector at a time, and in so doing, yet more saving is made to the overall process.

References

1. *Intel intrinsics guide*: `http://software.intel.com/sites/landingpage/IntrinsicsGuide`.
2. L. Aronovich, R. Asher, D. Harnik, M. Hirsch, S. T. Klein, and Y. Toaff: *Similarity based deduplication with small data chunks*, in Proceedings of the Prague Stringology Conference 2012, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2012, pp. 3–17.
3. M. Hirsch, A. Ish-Shalom, and S. T. Klein: *Optimal partitioning of data chunks in deduplication systems*, in Proceedings of the Prague Stringology Conference 2013, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2013, pp. 128–141.
4. M. Hirsch, S. T. Klein, D. Shapira, and Y. Toaff: *Controlling the chunk-size in deduplication systems*, in Proceedings of the Prague Stringology Conference 2015, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2015, pp. 78–89.
5. M. Hirsch, S. T. Klein, and Y. Toaff: *Improving deduplication techniques by accelerating remainder calculations*, in Proceedings of the Prague Stringology Conference 2011, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2011, pp. 173–183.
6. R. M. Karp and M. O. Rabin: *Efficient randomized pattern-matching algorithms*, 1987.
7. D. Lemire and O. Kaser: *Recursive hashing and one-pass, one-hash n-gram count estimation.* CoRR, abs/0705.4676 2007.
8. M. O. Rabin: *Fingerprint by random polynomials*, 1981.