Improving Deduplication Techniques by Accelerating Remainder Calculations*

M. Hirsch[a], S.T. Klein[b], Y. Toaff[a]

[a] *IBM – Diligent, Tel Aviv, Israel*
{hirschm,yair.toaff}@il.ibm.com

[b] *Department of Computer Science, Bar Ilan University, Ramat Gan, Israel*
tomi@cs.biu.ac.il

Abstract

The time efficiency of many storage systems rely critically on the ability to perform a large number of evaluations of certain hashing functions fast enough. The remainder function $B \bmod P$, generally applied with a large prime number P, is often used as a building block of such hashing functions, which leads to the need of accelerating remainder evaluations, possibly using parallel processors. We suggest several improvements exploiting the mathematical properties of the remainder function, leading to iterative or hierarchical evaluations. Experimental results show a 2 to 5-fold increase in the processing speed.

Keywords: Deduplication, Rabin-Karp, modular arithmetic, hierarchical evaluation

1. Introduction

Large storage and backup systems can be compressed by means of *deduplication*: the basic paradigm calls for locating recurrent sub-parts of the text, and replacing them by pointers to previous occurrences. One family of deduplication algorithms is known in the storage industry as CAS (Content Addressed Storage) and based on assigning a hash value to each data block [5, 1]. Such systems detect only identical blocks and are not suitable when large block sizes are used. if one relaxes this requirement and searches also for *similar* and not necessarily identical data, this may enable the use of much larger data chunks, as in the IBM ProtecTIER$^{(R)}$ product [2]. This

*This is an extended version of a paper that has been presented at the Prague Stringology Conference (PSC'11) in 2011, and appeared in its Proceedings, 173–183.

system is based on the evaluation of a hash function for a large number of strings, and most of these evaluations can be done in constant time because adjacent strings overlap.

The constant time is based on the repeated evaluation of a so-called *rolling hash* using the probabilistic pattern matching algorithm due to Karp and Rabin [4]: given is a text of length n and a pattern of length m, a hash function has to be applied on all the substrings of the text of length m. A naive implementation would thus yield a $\theta(nm)$ time complexity, which might be prohibitive. The rolling property of the hash exploits the fact that adjacent substrings are overlapping in all but their first and last characters, so that the hash of one substring can be calculated in constant time from the hash value of the preceding one, reducing the complexity to $O(n)$.

In a typical setting, a very large repository, say, of the order of 1 PB = 2^{50} bytes, will be partitioned into chunks of fixed or variable size, for example of (average) size 16 MB, to each of which one or more *signatures* are assigned. The details of the deduplication algorithm are not relevant to our current discussion and the interested reader is referred to [2]. The signature of a chunk is usually some function of the set of hash values produced for each consecutive substring of k bytes within the chunk. The length k of these substrings, which we call *seeds*, may be 512 or more, so that the evaluation might put a serious burden on the processing time.

Given a chunk $C = x_1x_2\cdots x_n$, where the x_i denote characters of an alphabet Σ, we wish to apply the hash function h on the set of substrings B_i of C of length k, $B_i = x_ix_{i+1}\cdots x_{i+k-1}$ being the substring starting at the i-th character of C. The constant time, however, for the evaluation of B_i is based on the fact that one may use the value obtained earlier for B_{i-1}, and this is obviously not true for the first value to be used. That is, B_1 needs an evaluation time proportional to k. For the example above, we would get 2^{26}, that is, about 64 million chunks, for each of which we process the first seed of size 512 bytes. This scenario is depicted in Figure 1, where the chunks are separated by vertical bars, and the first seeds, for which the signature has to be evaluated, appear as small black squares.

FIGURE 1: *First seeds for every chunk of the repository*

Moreover, in deduplication systems based on similarity rather than on identity, once a chunk of the *reference* has been identified as being similar to a chunk of the *version*, a more fine-grained comparison of the two is needed. Figure 2 is a schematic representation of the following typical scenario: given are two chunks which are already known to be similar, we need to identify

as many of their matching parts as possible. To this end, the reference is partitioned into a sequence of non-overlapping seeds, and a hash value of each of these seeds is evaluated and stored in a table H_R. As to the version, the hash value of every seed at every possible byte offset is calculated and potential matches are located in H_R. If a match is found, say, $H_V[i] = H_R[j]$, it is almost certain that the string $v_i v_{i+1} \cdots v_{i+k-1}$ is identical to $r_{(j-1)k+1} r_{(j-1)k+2} \cdots r_{jk}$, so the strings can be accessed and we shall try to extend the match to the left and right of these seeds.

Since the rolling hash property does not apply to the seed-by-seed evaluations of the reference, each substring of size k requires a $O(k)$ processing time. The techniques in this paper are aimed at speeding up the initialization and non-overlapping hashing operations using local parallelism, by means of the availability of several processors.

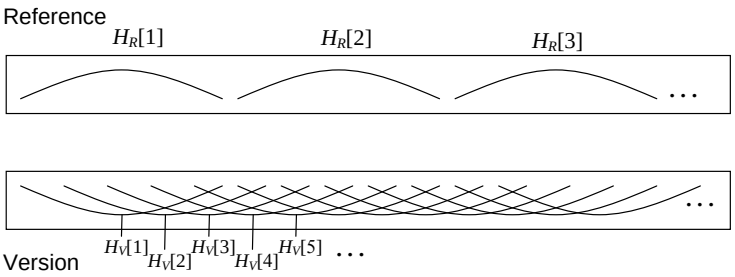

FIGURE 2: *Searching for matching parts in similar chunks*

The hash function we consider in this work is the remainder function modulo a prime number P, $h(B) = B \bmod P$, which is well known for yielding close to uniform distributions on many real-life input distributions. We interchangeably use B to denote a character string and the integer value represented by the binary string obtained by concatenating the ASCII codewords of the characters forming B. For example, the string `ABC` would be in ASCII 010000010100001001000011, so we would identify the string with the value 4,276,803. Two main improvements to the standard computation of the modulus are suggested: the first constructs a hierarchical structure enabling the use of several processors in parallel; the second exploits the fact that the computation can be performed iteratively to speed it up by calculating what we shall call *pseudo-hashes*.

2. Hierarchical evaluation of the remainder function

Consider the input string B partitioned into m subblocks of d bits each, denoted $A[0], \ldots, A[m-1]$, where $m = 2^r$ is a power of 2, and d is a small integer, so that d bits can be processed as an indivisible unit, typically

$d = 32$ or 64. Given also is a large constant number P of length up to d bits, that will serve as modulus. Typically, but not necessarily, P will be a prime number, and for our application it is convenient to choose P close to a power of 2. For example, one could use $m = 64$, $d = 64$ and $P = 2^{55} - 55$. We would like to split the evaluation of $B \bmod P$ so as to make use of the possibility to evaluate functions of the $A[i]$ in parallel on m independent processors $p_0, p_1, \ldots, p_{m-1}$, which should yield a speedup. We have

$$B \bmod P = \left(\sum_{i=0}^{m-1} A[i] \times 2^{d(m-1-i)} \right) \bmod P$$

Considering it as a polynomial (set $x = 2^d$, then $B = \sum_{j=0}^{m-1} A[m-1-j]x^j$), we can use Horner's rule to evaluate it iteratively. We first need the constant C defined by

$$C = 2^d \bmod P.$$

Note then that if we have a string D of $2d$ bits and we want to evaluate $\overline{D} = D \bmod P$, then we can write $D = D_1 \times 2^d + D_2$, where D_1 and D_2 are the leftmost, respectively rightmost d bits of D. We get that

$$\overline{D} = \overline{\overline{D_1} \times \overline{2^d} + \overline{D_2}} = \overline{\overline{D_1} \times C + D_2}.$$

Generalizing to m blocks of d bits each, we get the iterative procedure of Figure 3.

Iterative evaluation of $B \bmod P$
$R \longleftarrow 0$
for $i \longleftarrow 0$ to $m-1$ do
    $R \longleftarrow (R \times C + A[i]) \bmod P$ (1)

FIGURE 3: *Iterative evaluation of* $B \bmod P$

A further improvement can then be obtained by passing to a hierarchical tree structure and exploiting the parallelism repeatedly in $\log m$ layers, using the m available processors. In Step 0, the m processors are used to evaluate $A[i] \bmod P$, for $0 \leq i < m$, in parallel. This results in m residues, which can be stored in the original place of the m blocks $A[i]$ themselves, since P is assumed to fit into d bits. For our example values of m, d and P, only 55 of the 64 bits would be used.

In Step 1, only $\frac{m}{2}$ processors are used (it will be convenient to use those with even indices), and each of them works, in parallel, on two adjacent blocks: p_0 working on $A[0]$ and $A[1]$, p_2 working on $A[2]$ and $A[3]$, and generally p_{2k} working on $A[2k]$ and $A[2k+1]$, for $k = 0, 1, \ldots, \frac{m}{2} - 1$. The work to be performed by each of these processors is what has been described

earlier for the block D. Again, the results will be stored in-place, that is, right-justified in $2d$-bit blocks, of which only the rightmost d bits (or less, depending on P), will be affected.

Hierarchical evaluation of $B \bmod P$

for $k \longleftarrow 0$ to $m-1$ do
    $A[k] \longleftarrow A[k] \bmod P$
for $i \longleftarrow 1$ to r do
    for $k \longleftarrow 0$ to $\frac{m}{2^i}-1$ do
        use processor $p_{2^i k}$ to evaluate, in parallel,

$$A[2^i k + 2^i - 1] \longleftarrow \big(A[2^i k + 2^{i-1} - 1] \times C[i] + A[2^i k + 2^i - 1]\big) \bmod P$$

FIGURE 4: *Hierarchical parallel evaluation of* $B \bmod P$

In Step 2, the $\frac{m}{4}$ processors whose indices are multiples of 4 are used, and each of them is applied, in parallel, on two adjacent blocks of the previous stage. That is, we should have applied now p_0 on $A[0]A[1]$ and $A[2]A[3]$, etc., but in fact we know that $A[0]$ and $A[2]$ contain only zeros, so we can simplify and apply p_0 on $A[1]$ and $A[3]$, and in parallel p_4 on $A[5]$ and $A[7]$, and generally, p_{4k} working on $A[4k+1]$ and $A[4k+3]$, for $k = 0, 1, \ldots, \frac{m}{4}-1$. Again, the work to be performed by each of these processors is what has been described earlier for the block D since we are combining two blocks, with the difference that the new constant C should now be $2^{2d} \bmod P = \overline{C^2}$. The results will be stored right-justified in $4d$-bit blocks, of which, as before, only the rightmost d bits or less will be affected.

Continuing with further steps will yield a single operation after $\log m$ iterations. Note that the overall work is not reduced by this hierarchical approach, since the total number of applications of the procedure on block pairs is $\frac{m}{2} + \frac{m}{2} + \cdots = m-1$, just as for the sequential evaluation. However, if we account only once for operations that are executed in parallel, the number of evaluations is reduced to $\log m$, which should result is a significant speedup.

Summarizing, we first evaluate an array of constants

$$C[i] = \overline{C^{2^{i-1}}} = \overline{2^{d \times 2^{i-1}}}$$

to be used in step i for $i = 1, 2, \ldots, m-1$. This is easily done noticing that $C[1] = C$ and $C[i+1] = \overline{C[i]^2}$ for $i \geq 1$. The parallel procedure is then given in Figure 4, and a schematic view of the evaluation layers can be found in Figure 5.

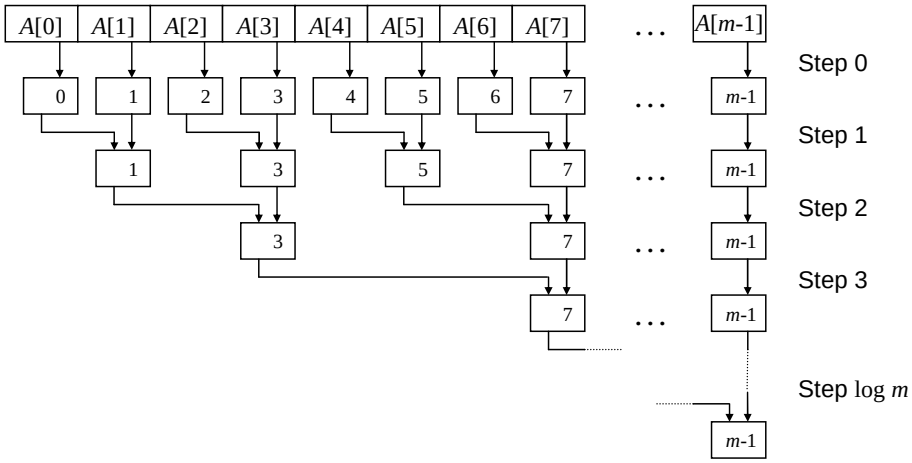

FIGURE 5: *Schematic representation of the hierarchical evaluation*

3. Avoiding overflows

The algorithm as described above dealt with integers of d bits length. We shall, for the ease of description, use the values $d = 64$ and $P = 2^{55} - 55$ in the sequel, which correspond to real-life applications, but all the ideas can easily be generalized to any other appropriate values. When two 64 bit integers are multiplied as $R \times C$ in equation (3), even though the result is sought modulo P, which is a 55-bit integer, one temporarily needs 128-bit arithmetic, which yields a serious slowdown of the performance.

One might think that to circumvent this, it suffices to work with smaller blocks, say, of $d = 32$ bits only. This will double the number of iterations, but could still result in a gain, if during multiplications the 64 bit limit is never exceeded. For the parallel implementation, the logarithmic number of parallel steps would only increase by 1. However, reducing d does not yet solve the problem, because R is a 55-bit integer, so when multiplied by the updated constant $C = 2^{32} \bmod P = 2^{32}$, we can get up to 87 bits. In order to get all the integers in this evaluation to be of length at most 64 bits (the maximum is reached when multiplying $R \times C$), so that no special 128-bit arithmetic would be needed, R has to be split and the modulus has to be applied not only at the end of each iteration.

Note that while we now assume that $d = 32$, the values of R are still stored in 64 bit integers. The way of splitting the 8 bytes representing R will be into the 23 rightmost bits and the complementing 41 leftmost bits. In fact, since the involved numbers are residues of mod$P$, where P is a 55 bit prime, the number of least significant non-zero bits in the left part is

only $55 - 23 = 32$. The representation of R is therefore

$$R = R_L \times 2^{23} + R_R,$$

where R_L are the 41 (in fact, only 32) leftmost and R_R are the 23 rightmost bits of R, so

$$R \times C = R \times 2^{32} = R_L \times 2^{55} + R_R \times 2^{32},$$

and since $2^{55} \bmod P = 2^{55} \bmod (2^{55} - 55) = 55$, we get that

$$\overline{R \times C + A[i]} = \overline{R_L \times 55 + R_R \times 2^{32} + A[i]}.$$

Revised iterative evaluation

$R \longleftarrow 0$
for $i \longleftarrow 0$ to $m - 1$ do
    $R_L \longleftarrow R \,/\, 2^{23}$
    $R_R \longleftarrow R \bmod 2^{23}$
    $R \longleftarrow R_L \times 55 + R_R \times 2^{32} + A[i]$
end-for
if $R > P$ then
    $R \longleftarrow R - P$
if $R > P$ then
    $R \longleftarrow R - P$

FIGURE 6: *Iterative evaluation without* mod

The algorithm for revised evaluation is given in Figure 6. Note that the mod P operation within the loop has been removed, and replaced by two mod operations following the loop. We thus call the intermediate values *pseudo-remainders*. The correctness of the procedure is based on the following

Theorem: The value of R is smaller than 2^{56}, that is, fits into 56 bits, at the end of each iteration.

Proof: By induction on i, the index of iteration. For $i = 0$, at the beginning of the iteration, R and thus also R_L and R_R are 0. The value of R at the end of iteration 0 is therefore $A[0]$, which has only 32 bits, less than 56.

Suppose the assumption is true at the end of iteration i, and consider the beginning of iteration $i + 1$. R_R has at most 23 bits by definition, and R_L has at most $56 - 23 = 33$ bits by the inductive assumption. Hence $R_R \times 2^{32}$ is of length at most 55 bits, and so is $R_R \times 2^{32} + A[i]$, since the 32 rightmost bits of $R_R \times 2^{32}$ are zero. The binary representation of 55 uses 6 bits, so $R_L \times 55$ is of length at most $33 + 6 = 39$ bits. At the end of the iteration, the length of R, obtained by adding a 39 bit number to a 55 bit number, must

therefore be at most 56, and this limit is achieved only if a carry propagates beyond the leftmost bit of $R_R \times 2^{32}$. ∎

It follows from the Theorem that there is no overflow if we remove the repeated application of the modulo operator, and only perform a single (and rarely, two) modulus at the end of the iteration. This is the purpose of the last four lines. Since at the end, $R < 2^{56} = 2P + 110$, the modulus can be replaced by subtraction. If $P \leq R < 2P$, then $R \bmod P = R - P$. For the rare cases in which $2P \leq R < 2P + 110$ (only 110 out of the possible almost 2^{56} values of R), a second subtraction of P will be necessary.

To understand how all the mod operations within the iteration could be saved, recall that our objective was to calculate $B \bmod P$. It would thus suffice, mathematically speaking, to apply a single mod operation after having calculated B, but in practice, such an evaluation is not feasible, because we are dealing here with a $m \times d$ bit long number, which cannot be handled. The classical solution, generally used in modular exponentiation algorithms [6], is to exploit the properties of the modulo function, to repeatedly apply the modulus to subparts of the formula, so as to never let the operands on which the modulus has to be applied grow above the limit permitted by the hardware at hand. For example, representing B as a polynomial $B = \sum_{j=1}^{m} A[m-j]x^{j-1}$, where we have set $x = 2^{32}$, using Horner's rule, we get

$$\overline{B} = \overline{\left(\cdots\left(\overline{\left(A[0]x + A[1]\right)}\, x + A[2]\right) x + \cdots\right) x + A[m-1]},$$

where after each multiplication and addition, $\bmod P$ is applied, so if we start with d bit numbers, at no stage of the evaluation do we use numbers larger than $2d$ bits. This was the approach in Section 2, and had as drawback that such a large number of modulo applications is expensive. The current suggestion reverts the process and removes again the internal modulo applications, but not entirely, since this would get us back to handling $m \times d$ bit numbers. Rather, it removes only a part of the internal operations, but leaves the cheap ones, basing ourselves on the fact that we work modulo a prime which is very close to a power of 2, namely $P = 2^{55} - 55$ in our example, but one can find such primes for any given exponent, see [3]. We thus get that $2^{55} \bmod P = 55$ in our case, an extremely small number relative to P, which can be used to decompose blocks into adjacent subblocks at a low price.

The algorithm presents a tradeoff between applying the remainder function only once (cheap but unfeasible because of the size of the numbers involved), and applying it repeatedly in every iteration (resulting in small numbers, but computationally expensive). We apply it only once (rarely

twice) at the end, but managed by an appropriate decomposition of the numbers to remove the moduli and still force all the involved numbers to be small.

Note that this technique can not be applied generally in situations where the modulus is chosen as a large random prime number, as often done in cryptographic applications, since it critically depends on the fact that $2^{55} \bmod P$ is a small number. In our case, it uses only 6 bits, and the Theorem would still hold for values needing up to 22 bits, in which case $R_L \times (2^{55} \bmod P)$ is of length at most $33 + 22 = 55$ bits. The sum of two 55 bit numbers would then still fit into the 56 bits claimed in the induction. But for 23 bits, we could already overflow into 57 bits. If P is a random prime number of 55 bits, the expected length of $2^{55} \bmod P$ is 54 bits and will only extremely rarely fit into 22 bits. The application field of the technique is thus when repeated evaluations are needed, all modulo a *constant* P, which can therefore be chosen as some convenient prime just a bit smaller than a given power of two. This is the case in rolling hashes of the Rabin-Karp type we consider here.

4. Adapting the hierarchical method

We now turn to adapting the hierarchical method, which can be used in parallel with m processors, to 64-bit arithmetic to improve processing time. The input is a sequence of $n = 2^m$ blocks of $d = 64$ bits each. The hierarchical evaluation is done in $m = \log n$ layers, with layer i processing what we shall call *superblocks*, consisting of 2^i original d-bit blocks, $i = 0, 1, \ldots, m-1$. The scenario at layer i, for the superblock indexed k, $k = 0, 1, \ldots, \frac{n}{2^i} - 1$, is given in the upper part of Figure 7.

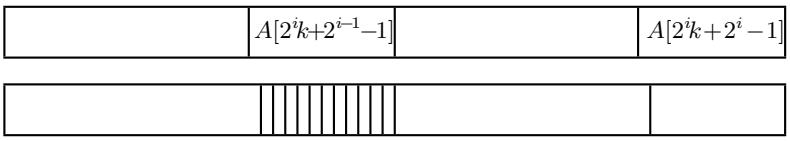

FIGURE 7: *Schematic representation of superblocks*

The superblock consists of two halves, and only the rightmost block (in fact, only its 55 rightmost bits) in each half is non-zero. The evaluation combines the two non-zero values and puts the output back into the rightmost block, using the command

$$A[2^i k + 2^i - 1] \longleftarrow \Big(A[2^i k + 2^{i-1} - 1] \times C[i] + A[2^i k + 2^i - 1]\Big) \bmod P.$$

The values $C[i] = \overline{C^{2^{i-1}}} = \overline{2^{64\times 2^{i-1}}}$ can be calculated as $C[1] = 2^{64} \bmod P$ and $C[i+1] = \overline{C[i]^2}$ for $i > 1$. For $P = 2^{55} - 55$, these values are given in Table 1.

i	$C[i]$	bits
1	28,160	15
2	792,985,600	30
3	16,336,612,484,973,479	55
4	8,143,640,278,601,598	55
5	5,745,742,201,926,802	55
6	16,594,324,020,821,548	55

TABLE 1: *Constants for hierarchical evaluation*

We thus need more than 64 bits to evaluate $A[2^i k + 2^{i-1} - 1] \times C[i]$ for $i > 1$. To fit into the 64-bit arithmetic constraint, we propose two strategies. The first is a generic one, that can be applied to any values of the parameters, and processes each layer in the same way. The second achieves some additional savings by adapting the specific values in our running example differently in each of the layers.

4.1. General uniform adaptation of the parameter values

The first iteration (layer 0), which applies the modulus on the original 64 bit blocks to produce 55 bit numbers, can be kept without change. For the higher layers, the input of which are two non-adjacent 55-bit blocks $A[2^i k + 2^{i-1} - 1]$ and $A[2^i k + 2^i - 1]$, the latter can be used as is, but the former has to be multiplied, so we split the block into 11 subblocks of length 5 bits, as depicted in the lower part of Figure 7.

Denote the 11 blocks forming $A[2^i k + 2^{i-1} - 1]$, from right to left, by $E[k, i, j]$, $j = 0, 1, \ldots, 10$, which gives

$$A[2^i k + 2^{i-1} - 1] = \sum_{j=0}^{10} E[k, i, j] \times 2^{5j}.$$

In addition, prepare a two dimensional table $CC[i, j]$ for the above values of i and j, defined by

$$CC[i, j] = \overline{C[i] \times 2^{5j}}.$$

Then

$$A[2^i k+2^{i-1}-1]\times C[i]+A[2^i k+2^i-1] = \sum_{j=0}^{10} E[k, i, j]\times CC[i, j]+A[2^i k+2^i-1].$$

Each term in the summation uses at most $5 + 55 = 60$ bits, so the sum of the 12 terms uses at most $60 + \lceil \log 12 \rceil = 64$ bits, as requested. In fact, since the elements $E[k, i, j]$ all belong to a small set $\{0, 1, \ldots, 31\}$, one can precompute a three dimensional table $CCC[i, j, p]$ defined, for $p = 0, \ldots, 31$ by

$$CCC[i, j, p] = \overline{CC[i, j] \times p} = \overline{C[i] \times 2^{5j} \times p}.$$

This reduces then the right hand side of the summation above to

$$\sum_{j=0}^{10} CCC\left[i, j, E[k, i, j]\right] + A[2^i k + 2^i - 1].$$

Table 2 brings some sample lines for $P = 2^{55} - 55$ and selected values of i, j and p. Note that for $p = 0$, all values are 0, and for $p = 1$ and $j = 0$, we get the values $C[i]$ of Table 1. Note also the small value obtained for $(i, j, p) = (1, 9, 2)$: indeed, $\overline{C[1] \cdot 2^{5\times 9} \cdot 2} = \overline{2^{64}\ 2^{46}} = \overline{2^{55}}^2 = 55^2 = 3025$.

i	j \ p	1	2	3 ⋯	30	31
	0	28160	56320	84480	844800	872960
	1	901120	1802240	2703360	27033600	27934720
	2	28835840	57671680	86507520	865075200	893911040
1	3	922746880	1845493760	2768240640	27682406400	28605153280
	⋮					
	9	18014398509483500	3025	18014398509486494	45375	18014398509528844
	10	48400	96800	145200	1452000	1500400
	0	792985600	1585971200	2378956800	23789568000	24582553600
	1	25375539200	50751078400	76126617600	761266176000	786641715200
	2	812017254400	1624034508800	2436051763200	24360517632000	25172534886400
2	3	25984552140800	51969104281600	77953656422400	779536564224000	805521116364800
	⋮					
	9	42592000	85184000	127776000	1277760000	1320352000
	10	1362944000	2725888000	4088832000	40888320000	42251264000
⋮						
	0	16594324020821548	33188648041643096	13754175043500731	29455359378115571	10020886379973206
	1	26615210400794754	17201623782625595	7788037164456436	5822777606636534	32437988007431288
	2	23024401389262129	10020005759560345	33044407148822474	6184898317549523	29209299706811652
6	3	16204904077109868	32409808154219736	12585915212365691	17772761066765171	33977665143875039
	⋮					
	9	20453807581727657	4878818144491401	25332625726219058	1124678129443189	21578485711170846
	10	6003496273934590	12006992547869180	18010488821803770	35989700142182048	5964399397152725

TABLE 2: *Sample lines of the 3-dimensional table CCC*

To take this idea of tabulating even a step further, note that the elements in the table are computed only once, so this could be done offline, and there, 128-bit operations could be permitted. Instead of partitioning $A[2^i k + 2^{i-1} - 1]$ into 11 blocks of 5 bits each, any other partition into $\lceil 55/q \rceil$ blocks of q bits each could be considered, if we were willing to extend the table $CCC[i, j, p]$ to the 2^q possible values of q-bit strings. Taking, for example, $q = 11$, we get 5 blocks of 11 bits and would have to consider 2048 possible values of p in $CCC[i, j, p]$. The number of bits needed to represent $CC[i, j] \times p$ would then be $55 + 11 = 66$, but this is evaluated only once, and what will finally

be stored (and used afterwards) is $\overline{CC[i,j] \times p}$, which again needs only 55 bits; the sum of six 55-bit numbers fits into 58 bits, so there is no overflow.

The number of elements needed in the table CCC is $m \times \left\lceil \frac{55}{q} \right\rceil \times 2^q$. Table 3 brings the size of the table for a few sample values of q, for $m = 6$ as in our example. The number of 64-bit operations for the evaluation of each new value is equal to the number of blocks b: there are $b + 1$ terms to be added, but only $x - 1$ additions are needed to add x terms.

q	# blocks	# lines	# entries	Actual size
3	19	8	912	6.2 K
4	14	16	1344	9.1 K
5	11	32	2112	14.4 K
6	10	64	3840	26.3 K
7	8	128	6144	42 K
8	7	256	10752	74 K
9	7	512	21504	147 K
10	6	1024	36864	252 K
11	5	2048	61440	420 K
12	5	4096	122880	840 K
16	4	65536	1572864	10.5 M
20	3	1048576	18874368	126 M

TABLE 3: *Size of auxiliary table for various values of q*

We can thus choose the value of q according to the required tradeoff: the lower q, the less storage is needed for the CCC tables, but the more operations have to be performed. Taking for example values of q from 5 to 7, the tables would fit into 50K, but 8 to 11 operations have to be performed.

4.2. Specific adaptation of the parameter values for $m = 6$ and $d = 64$

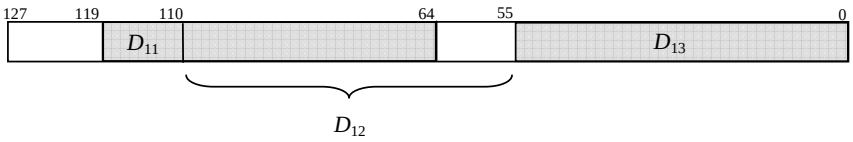

FIGURE 8: *Layer 1: two blocks of 64 bits each*

The tradeoffs in Table 3 lead to the following suggestions for the lower layers. Consider layer 1, consisting of superblocks of 128 bits. Figure 8 represents the layout after iteration 0, in which two 55-bit strings have been evaluated (in grey in the figure). We partition the superblock as indicated, which yields as value:

$$D = D_{11} \times 2^{110} + D_{12} \times 2^{55} + D_{13}.$$

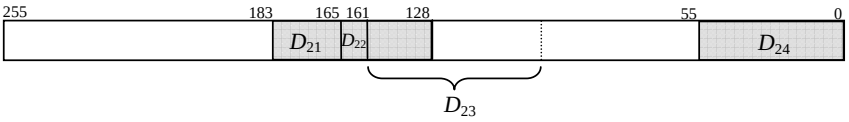

FIGURE 9: *Layer 2: two blocks of 128 bits each*

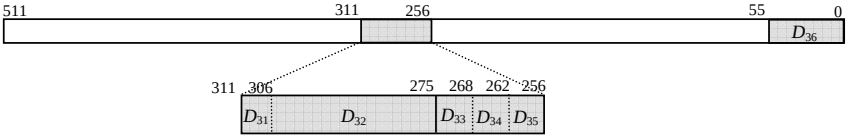

FIGURE 10: *Layer 3: two blocks of 256 bits each*

D_{13} uses only 55 bits; D_{12} also needs 55 bits, but is multiplied by 2^{55} mod $P = 55$, which needs 6 bits, so together 61 bits; D_{11} needs 9 bits, and multiplied by 2^{110} mod $P = 55^2 = 3025$, which needs 12 bits, so together 21 bits; their sum has therefore at most 62 bits, so only **two** 64-bit additions are needed.

For layer 2, we need a different layout, given in Figure 9. The superblock consists now of two subparts of 128 bits each. This partition yields the following equality:

$$D = D_{21} \times 2^{165} + D_{22} \times 2^{161} + D_{23} \times 2^{110} + D_{24}.$$

D_{24} uses only 55 bits; D_{23} is of length 51 bits, but is multiplied by 2^{110} mod $P = 3025$, which needs 12 bits, so together 63 bits; D_{22} is of length 4 bits, and is multiplied by 2^{161} mod P, which needs 55 bits, so together 59 bits; finally, D_{21} needs 18 bits, and is multiplied by 2^{165} mod $P = 55^3 = 166375$, which needs 18 bits, so together 36 bits; their sum has therefore at most 64 bits, so only **three** 64-bit additions are needed.

In Layer 3, a superblock, now consisting of two halves of 256 bits each, will be partitioned according to the layout given in Figure 10. The desired value of D is then obtained by adding the following terms:

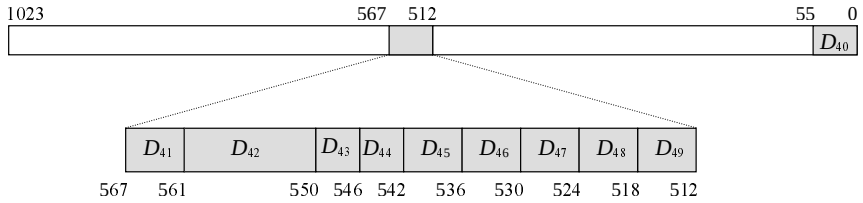

FIGURE 11: *Layer 4: two blocks of 512 bits each*

$D_{31} \times 2^{306}$,	in bits:	$5 + 55 = 60$
$D_{32} \times 2^{275}$,	in bits:	$31 + 29 = 60$
$D_{33} \times 2^{268}$,	in bits:	$7 + 55 = 62$
$D_{34} \times 2^{262}$,	in bits:	$6 + 55 = 61$
$D_{35} \times 2^{256}$,	in bits:	$6 + 55 = 61$
D_{36},	in bits:	55

Their sum has at most 64 bits, and only **five** 64-bit additions are needed.

Finally, a possible partition for layer 4 is given in Figure 11, yielding the value D as the sum of:

$D_{41} \times 2^{561}$,	in bits:	$6 + 55 = 61$
$D_{42} \times 2^{550}$,	in bits:	$11 + 50 = 61$
$D_{43} \times 2^{546}$,	in bits:	$4 + 55 = 59$
$D_{44} \times 2^{542}$,	in bits:	$4 + 55 = 59$
$D_{45}, \ldots, D_{49} \times$ (resp.) $2^{536}, 2^{530}, 2^{524}, 2^{518}, 2^{512}$,	in bits:	$6 + 55 = 61$ each
D_{40},	in bits:	55

Their sum has at most 64 bits, and only **nine** 64-bit additions are needed.

There is no sense in trying to extend this strategy also to level 5 and beyond; it would cost more than the 11 additions given in the generic solution of Section 4.1. It is a matter of tradeoff to decide how many levels should be treated by means of the special layouts given in Figures 8–11, and one could apply this only to level 1, or to levels 1 and 2, etc. We consider the *amortized* global cost for evaluating the signature, since only at the lowest level, all the n processors are involved, and for the higher levels, specifically, for level i, the number of working processors is only $n/2^i$. The amortized number of 64-bit additions if we use only the method of Section 4.1 is

$$1 \times n + 11 \times \left[\frac{n}{2} + \frac{n}{4} + \cdots\right] = n\,[1 + 11] = 12n.$$

If the special treatment is given only to level 1, the amortized cost will be

$$1 \times n + 2 \times \frac{n}{2} + 11 \times \left[\frac{n}{4} + \frac{n}{8} + \cdots\right] = n\left[1 + 1 + \frac{11}{2}\right] = 7.5n.$$

If it is given up to levels 2, 3, and 4, the cost will be, respectively

$$1 \times n + 2 \times \frac{n}{2} + 3 \times \frac{n}{4} + 11 \times \left[\frac{n}{8} + \frac{n}{16} + \cdots\right] = n\left[1 + 1 + \frac{3}{4} + \frac{11}{4}\right] = 5.5n,$$

$$1 \times n + 2 \times \frac{n}{2} + 3 \times \frac{n}{4} + 5 \times \frac{n}{8} + 11 \times \left[\frac{n}{16} + \frac{n}{32} + \cdots\right] = n\left[1 + 1 + \frac{3}{4} + \frac{5}{8} + \frac{11}{8}\right] = 4.75n,$$

$$n + 2\frac{n}{2} + 3\frac{n}{4} + 5\frac{n}{8} + 9\frac{n}{16} + 11\left[\frac{n}{32} + \cdots\right] = n\left[1 + 1 + \frac{3}{4} + \frac{5}{8} + \frac{9}{16} + \frac{11}{16}\right] = 4.625n.$$

5. Experimental results

We have compared the above methods on randomly chosen input texts, several GB of our exchange database. Actually, the exact choice of the test data is not relevant, because the number of remainder operations performed is not data dependent.

	WS	**M2**	**X5**	**GPU**
baseline	114	139	168	595
hierarchical	229	200	377	1896
pseudo remainders	582	256	1067	2327

TABLE 4: *Experimental comparison of performance*

The following methods were tested: as baseline, we took a regular iterative evaluation, processing single bytes, that is, $d = 8$. In all our tests, the size of B was $m = 2^{12} = 4096$ bits or 512 bytes. The next method was a hierarchical implementation, according to Figure 4, with blocks of size $d = 64$, and using 128-bit arithmetic where necessary. Finally, we also ran the revised iterative method of Figure 6 using pseudo remainders, with $d = 32$ and 64-bit operations only.

The tests were run on several platforms: **WS**: a 3.2GHz Intel PC Workstation, **M2**: an IBM 3850M2 server (2.93 GHz Intel Xeon X7350), **X5**: an IBM 3850X5 server (2.27 GHz Intel Xeon X7560), and **GPU**: an Nvidia GeForce GTX 465 graphics board, using copy to/from device. The results are presented in Table 4, all values giving the number of MB processed per second.

References

[1] B. ZHU, K. LI, AND H. PATTERSON: *Avoiding the disk bottleneck in the data domain deduplication file system.* Proc. FAST'08, the 6th USENIX Conference on File And Storage Technologies, 2008, pp. 279–292.

[2] L. ARONOVICH, R. ASHER, E. BACHMAT, H. BITNER, M. HIRSCH, AND S.T. KLEIN: *The design of a similarity based deduplication system.* Proc. of the SYSTOR'09 Conference, 2009, pp. 1–14.

[3] *Primes just less than a power of two*: http://primes.utm.edu/lists/2small/.

[4] R.M. KARP AND M.O. RABIN: *Efficient randomized pattern-matching algorithms.* IBM Journal of Research and Development, 1987, pp. 249–260.

[5] S. QUINLAN AND S. DORWARD: *Venti: A new approach to archival storage.* Proc. FAST'02, the 1st USENIX Conference on File And Storage Technologies, 2002, pp. 89–101.

[6] T.H. CORMEN, C.E. LEISERSON, AND R.L. RIVEST: *Introduction to Algorithms*, MIT Press, 1990.