Prediction-Capable Data Compression Algorithms for Improving Transmission Efficiency on Distributed Systems

Hann-Huei Chiou, Alexander I-Chi Lai, and Chin-Laung Lei
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan, R.O.C. 10617

Abstract

Network bandwidth is a kind of limited and precious resources in modern distributed computing environments. Insufficient bandwidth will severely degrade the performance of a distributed computing task in exchanging massive data among the networked hosts. A feasible solution to save bandwidth is to incorporate data compression during transmission. However, blind, or unconditional, compression may only result in waste of CPU power and even slow down the overall network transfer rate, if the data to be transmitted are hard to compress. In this paper, we present a prediction-capable lossless data compression algorithm to address this problem. By adapting to the compression speed of a host CPU, current system load, and network speed, our algorithm can accurately estimate the compression time of each data block given, and decide whether it should be compressed or not. Experimental results indicate that our prediction mechanism is both efficient and effective, achieving 93% *of prediction accuracy at the cost of only* 3.2% *of the execution time of unconditional compression.*

1. Introduction

Due to the fast advances in the microprocessor technology and computer networks, researches in the field of distributed computing get more and more attention in recent years. A distributed system [3, 13, 16] consists of a number of CPUs connected by a high-speed network.

In contrast to centralized ones, distributed systems have the merit of being more flexible, scalable, and reliable. Since the cost-to-performance ratio of personal computers tends toward lower and lower each year, it now becomes much easier and more feasible to construct distributed systems than before.

Still, one must first address several important issues to build an efficient distributed system. Among those issues, minimizing communication cost is amid the most vital and challenging ones, because most distributed computing tasks involve massive data exchange.

There are several possible solutions to this problem, such as caching [10, 14], prefetching [8, 17], and data compression [2]. Amongst these approaches, data compression makes use of relatively abundant CPU computation power to minimize the amount of data to transmit, equivalently reducing the bandwidth requirement. Since this approach is virtually suitable to all types of platforms and communication media, it has been long existent and widely put into practice. For example, some Internet archives sites are capable of compressing and decompressing files on the fly. Some communication protocols and devices also have incorporated compression facilities and options, such as the Internet Point-to-Point Protocol (PPP), the ITU-T V.42bis recommendation (V.42bis), and the Microcom Networking Protocol Level 5 (MNP5) standard used in most modems.

Unfortunately, many kinds of data are not suitable for compression because the resulting compression ratio is poor. Attempting to compress them is a waste of time and may even increase the data size at the end. Even if the resulting data size does shrink, the gain of communication speed may still be negative due to excessive compressing/decompressing overhead. More precisely, applying compression in communication is effective or "good," if the time of transmitting compressed data, plus the compressing/decompressing overhead, is less than the time of transmitting uncompressed data (see Figure 1).

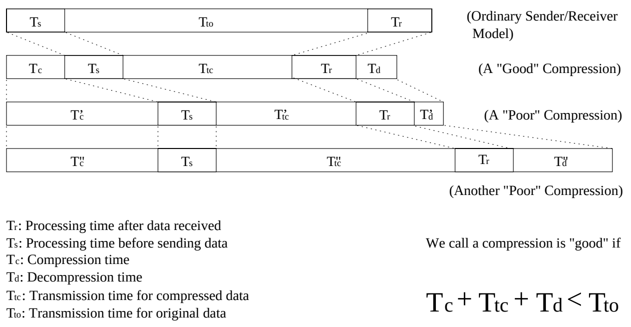
Besides, there are other run-time factors affecting the effectiveness of on-the-fly compression in distributed systems, such as the compression speed of a host CPU, current system load, and network speed. More specifically, when the network is slow or overcrowded, the CPUs can spend more time in compression; on the other hand, if the network is fast and its utilization is low, it may be better to transmit data without any compression, saving the CPU time for other processes. In a word, a good data compression scheme should dynamically trade network bandwidth with CPU computation power or vice versa, depending on circumstances. Most existing data compression schemes are, however, "blind" (i.e. unconditional) and static ones. That is, they all suffer from the problem of potential performance loss when they are applied to distributed computing environments (the "poor" scenarios in Figure 1).

In this paper we present an on-line lossless data compression algorithm to address this problem. There are two key techniques in our algorithm: one is a compression-speed measurement mechanism and the other is a compression-ratio prediction mechanism. By combining both mechanisms, our algorithm can accurately estimate the compression time of each data block without actually compressing it, and decide whether it should be compressed or not by adapting to the current system status. Since the target to be compressed is the intermediate data during distributed computation, our algorithm adopts the dictionary-based compression technique because it is lossless and efficient.

The rest of this paper is organized as follows. In Section 2, we review researches related to this topic. Next, in Section 3, terminologies, notations, methodologies as well as design issues are given. Section 4 describes our approach to estimate compression time from ratio. In Section 5 we present the main result of this paper, our compression ratio prediction mechanism for a given data chunk. Finally, we conclude this paper with a summary of results in Section 6.

2. Related Works

Data compression techniques can be divided into two categories: lossless and lossy compression. Lossless compression,can be further divided into three sub-categories: *statistical*, *dictionary-based*, and *prediction-based coding*. The compression ratios of the statistical coding schemes, such as Huffman coding [5] and arithmetic coding [6, 9, 18], are affected by the degree of skewness in the message sequences. Most data however only exhibit small skewness, thus most statistical coding schemes do not perform well [1]. The concept of dictionary-based coding was introduced by Ziv and Lempel in [19, 20]. The major part of the computation steps in dictionary-based coding schemes is string matching, whereas the major part in statistical coding schemes are probability addition and sorting. Hence, the dictionary-based coding schemes run faster and achieve better compression ratio than statistical coding ones in general. Prediction-based techniques are represented by Markov models. Markov algorithms yield the best compression ratio, but are much slower and need more storage space than dictionary-based algorithms. Therefore dictionary-based techniques are preferred under most circumstances. On the other hand, there is a duality between Markov compression and prediction; that is, if a character sequence is highly compressible, it is also highly predictable. Phalke [11] and Vitter [17] take advantages of this duality in their researches.



T_s
T_{to}
T_r
(Ordinary Sender/Receiver Model)
T_c
T_s
T_{tc}
T_r
T_d
(A "Good" Compression)
T_c'
T_s
T_{tc}'
T_r
T_d'
(A "Poor" Compression)
T_c''
T_s
T_{tc}''
T_r
T_d''
(Another "Poor" Compression)
T_r: Processing time after data received
T_s: Processing time before sending data
T_c: Compression time
T_d: Decompression time
T_{tc}: Transmission time for compressed data
T_{to}: Transmission time for original data
We call a compression is "good" if
$T_c + T_{tc} + T_d < T_{to}$

Figure 1. Good and poor scenarios of applying compression in distributed environments.

Another category in data compression is lossy compression.The compression ratio is generally higher than lossless ones since some information are lost during compression. Lossy compression is often used in the compression of graphics (e.g. JPEG) speeches, and videos (e.g. MPEG) Lossy compression schemes are inappropriate here since our focus is on the distributed computation tasks.

Several researches, including hardware and software ones, are aiming for improving the performance of on-line compression. The first idea is to implement the compression routines in hardware, for example, a lossless data compression and decompression chip [12]. There are also compression protocols in the modem technology, such as V.42bis and MNP5. V.42bis uses a dictionary-based coding scheme and it can yield up to a 4:1 compression ratio in general. MNP5 uses an adaptive statistical coding scheme which is similar to Huffman coding, plus run-length encoding if a single symbol occurs more than three times in sequence. It can yield up to a 2:1 compression ratio in general. Many compression extensions to PPP are proposed, such as RFC1962, RFC1967, RFC1974, RFC1977, RFC1978, and RFC1993. Cheung [2] presented two controlled compression methods during data transfer. His first algorithm is based on sampling past performance with and without compression. And the other is to exploit feedback from the network using a water-and-funnel model.

3. The Scheme of Our Proposed Algorithm

3.1. Notations and Methodology

There are several ways to describe the term "compression ratio." In this paper, we define compression ratio to be *the ratio of the reduced size to the original size*; that is, compression ratio, $Ratio_c$, is defined as

$$Ratio_c = \frac{Size_o - Size_c}{Size_o}, \tag{1}$$

where $Size_o$ is the size of the original data chunk and $Size_c$ is the size of the data after compression. This definition complies with the convention of GNU's compression utility `gzip(1)`, where the value of 1 is the best case and 0 is the worst case. A negative value means the compression result is useless because the data chunk is not "compressed." Therefore we define the compression ratio to be 0 if $Size_c$ is greater than $Size_o$.

The primary goal of this paper is to design a scheme to save the transmission time by compressing data on the fly, as well as to avoid the risk of potential system slowdown brought by blind compression. Recall that a "good" compression scheme satisfies the criterion

$$T_c + T_{tc} + T_d < T_{to} \tag{2}$$

shown in Figure 1, where T_c is the compression time, T_{tc} is the transmission time for compressed data, T_d is the decompression time, and T_{to} is the transmission time for original data, respectively.

Observe that although T_{to} can be easily obtained from the raw data size $Size_o$ and the network speed $Speed_{net}$ directly, we still need the following key components to achieve the goal of this paper:

- A lossless algorithm to perform the core compression/decompression functions, which should be fast enough to minimize $T_c + T_d$;
- A compression ratio prediction mechanism to estimate T_{tc} without actually compressing the data, and
- A compression/decompression speed estimation mechanism to measure T_c and T_d in advance.

Once the ratio prediction and speed estimation mechanisms are impersonated, we can replace in Equation 2 the parameters T_c, T_{tc}, T_d, and T_{to} with their estimation peers T_c^{est}, T_{tc}^{est}, T_d^{est}, and T_{to}^{est}, respectively, so that we can tell if a data block is suitable for compression by examining

$$T_c^{est} + T_{tc}^{est} + T_d^{est} < T_{to}^{est}. \tag{3}$$

The technical details of how to impersonate these essential components will be discussed later.

Still, there are several extra issues that should be clarified here. First, the prediction cost T_p (if not zero) must be much smaller than the actual compression overhead T_c, otherwise we should directly go to blind compression without any hesitation. Also, the prediction overhead must not counterbalance the gain from compression; that is, Equation 3 should be augmented to incorporate T_p as follows:

$$T_p + T_c^{est} + T_{tc}^{est} + T_d^{est} < T_{to}^{est}. \tag{4}$$

Another important issue is whether there is any relation between compression ratio and compression/decompression speed. The answer to this question is yes. Observe that a compression/decompression algorithm always needs to produce/process more data when hardly compressible data are encountered; that is, good compression ratio usually means high compression speed. Existence of such a relationship also suggests that we may only have to devise a compression ratio prediction mechanism, and derive estimated compression speed (and the time required for compression) from the predicted ratio. In the following sections we will formulate the ratio/speed relationship specific to the core compression algorithm adopted in our scheme. Once the relationship is successfully formulated, we can get both T_c^{est} and T_{tc}^{est} simultaneously; therefore T_p can also be substantially lowered.

```
Compress(InBuf[],OutBuf[],Size)
1 i←0;
2 while i<Size do begin
3  find the longest and nearest match-
    ing substring in the sliding win-
    dow of InBuf[i];
4  encode ($maxmatchlen$, $maxmatchpos$),
    and append the output to OutBuf[];
5  i←i+$maxmatchlen$;
6 end
```

Figure 2. Our compression procedure.

3.2. The Proposed Scheme

There are several existing algorithms that can be adopted as our core compression mechanism. The LZ77 algorithm is an appropriate candidate for its simplicity, speed, high compression ratio, and ease of implementation. In fact, what we actually implemented is the LZSS [15] algorithm, a variant of LZ77, with enhancements by utilizing Huffman coding to further squeeze the output data size. This incarnation is called the LZHUF algorithm. One famous feature of all LZ-algorithms is that its decompression speed is much faster than its compression speed in a homogeneous environment. Consequently, the decompression cost can be safely dropped from Equation 4 in the analyses of our scheme hereafter; i.e. Equation 4 can be simplified as:

$$T_p + T_c^{est} + T_{tc}^{est} < T_{to}^{est}. \tag{5}$$

The pseudo-code of our compression routine is shown in Figure 2, while the details of our implementation are omitted for the sake of conciseness.

The whole execution flow of our proposed scheme can be described as follows. When one chunk of raw data is to be transmitted. we first use our prediction mechanism (detailed later in Section 5) to estimate the compression ratio of that data chunk. Next, we use the relationships between the compression ratio and speed/time of our core LZHUF algorithm (described in depth in Section 4), as well as run-time factors such as system load and network speed, to estimate the compression time T_c^{est} and the two transmission times T_{tc}^{est} and T_{to}^{est} respectively. Note that we also get T_p, which is exactly the total elapsed execution time of these two estimation phases. Finally, with all parameters fixed we can decide whether compression should be initiated by examining if Equation 5 is satisfied.

4. Compression Speed Estimation

In this section, we formulate the relationship between compression ratio and compression speed. Clearly, such a relationship is algorithm-specific; nevertheless, since we have fixed our choice of compression mechanism (LZHUF), we can focus ourselves on finding the ratio/speed relationship of the LZHUF algorithm only. Also note that such a relationship is obtained off-line from real-world experimental results, therefore we can efficiently estimate the compression and transmission time in our prediction mechanism by directly applying the pre-computed results.

4.1. The Approach

It is very difficult to obtain the compression ratio/speed relationship by analyzing the compression algorithm only, because the compression ratio is highly related to the characteristics of raw data input. Our strategy to overcome this obstacle is to compress lots of samples and summarize the relationship, if any, from the outcomes. Since the target to be compressed, as we stated before, is the intermediate data during distributed computation, the samples should be snapshot from the images of the main memory space. In fact, the samples are extracted from each host machine's virtual memory swap partitions which exactly contain images from the main memory. The sampling sizes we use are 1, 2, 4, and 8*Mbytes*. Results of other sizes can be calculated by interpolation or extrapolation.

At first, we try to record the compression ratio and time of each fixed-size sample, and figure out its compression speed index by dividing the original sample size with its compression time. However, it was soon proved to be an unwise choice. The recorded compression time values cannot be stabilized because they are affected not only by the CPU type, but also by many run-time factors such as system load, size of physical memories, and so on. Without a precise measurement of compression speed, our prediction mechanism would be inaccurate and thus useless; briefly, we need to take another approach instead.

Fortunately, there does exist an alternative. Recall that the compression speed is closely related to the compressed data size. Also note that the output of LZ-based compression algorithms is basically a sequence of tokens which are encoded from the ($maxmatchlen$, $maxmatchpos$) pairs

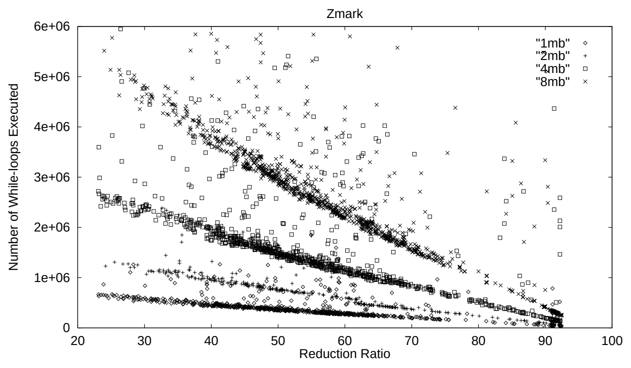


Figure 3. The relationship between compression ratio, time and original data size.

(see the pseudo-code in Figure 2). Such observations indicate that we can use the number of produced tokens as an index of compression speed—the more tokens are generated, the more slowly the compression algorithm runs. Furthermore, from Figure 2 we find the main body of our compression procedure is a big loop which creates exactly one token per iteration. Since each iteration of the main loop in our compression procedure takes roughly the same time to execute, we can also count the number of executed iterations of the main loop as an index of compression speed. The token/iteration count approach eliminates all influences of both machine-dependent and run-time factors, providing a universal index of speed measurement.

4.2. Experimental Results

To formulate the ratio/speed relationship of our LZHUF algorithm, we conducted an experiment on four Pentium-based machines running Linux. Each host is equipped with 128*MBytes* of physical memory and 128*MBytes* of swap space, respectively. For each different sample size, we randomly extract 128 samples from swap partitions of each host machine; i.e. totally 512 samples per sampling size are examined. The results are summarized in Figure 3, in which the horizontal axis represents the compression ratio, the vertical axis is the number of executed iterations of the main loop in LZHUF algorithm, and each pixel represents a data sample, respectively.

From Figure 3 we find that clearly most data samples indeed converge into four straight lines, which coincide with the cases of 1, 2, 4, and 8*Mbytes*, respectively. Such a phenomenon strongly suggests that the relationship between compression ratio and the number of iterations executed is almost linear. In order to further validate this outcome, we calculated by regression the slopes of the four straight lines, which are (listed from the lowest one to the uppermost one) -8409.86, -19528.8, -35259.3, and -70030.5, respectively. The slope sequence is also proportional to the sequence of sampling sizes (i.e. 1:2:4:8), which also strongly supports that the relationship between the compression ratio and the number of iterations executed is linear. From the analyses shown above we conclude that *given a fixed input size, the compression ratio of the input is about linearly proportional to the number of executed iterations of the main loop in our LZHUF algorithm*. The establishment of the relationship provides a solid basis of both speed measurement and ratio prediction because we can compute the compression time from the estimated compression ratio immediately.

4.3. A Relative Compression Speed Index

With the relationship established in the previous section, we can now exploit the connection between compression ratio and time on a specific hardware platform. What we still have to do is to measure how long a single iteration of the main loop in LZHUF takes, on a specific host machine. To achieve this, we set up a reference standard platform, a 233-Mhz Pentium/MMX machine with 128*MBytes* of physical memory, and compressing a lot of samples to measure its absolute compression performance. Speed results of all other real-world host machines will be normalized against the compression performances of our reference standard to get a relative compression speed index. We call such an index the *Zmark* value. The Zmark value of a target machine (including our reference standard) is evaluated after rebooting it into the single-user mode and removing all unnecessary programs and services. Since the whole system dedicates itself to the Zmark task only, it should reach nearly its ideal compression performance. On our reference standard machine, whose Zmark index is defined to be 1.0, it takes 1 second to execute roughly $330,000$ iterations of the main loop in our LZHUF incarnation, or equivalently, 3 microseconds per iteration. On other machines, the Zmark value is proportional to the compression power: the faster a target processor is, the higher the Zmark value it delivers. For example, a hypothetical target machine with Zmark index equalling to 0.5 should take 6 microseconds to execute an iteration of our LZHUF incarnation, and deliver half compression performance of our reference standard.

4.4. Incorporating Real-World Factors

Given the compression ratio of the input data block and the Zmark index of a target system, we are ready to figure out the remaining unknown time estimations by taking into consideration the run-time factors, i.e. the current system load, $Load_{now}$, which is defined as the number of pending jobs in the system, and the current network speed, $Speed_{net}$, in bytes per second. In most operating systems such as UNIX or Linux, these two parameters can be obtained by invoking proper system calls, or examining corresponding global variables; for example, our test platform reports it transmits 100*KBytes* per second over a 10*Mbits/s* Ethernet-based local area network.

Details of calculating remaining time estimations are listed as follows. First, we estimate the compression ratio of a given data block with our prediction algorithm, and compute the number of iterations executed within our core LZHUF algorithm from the estimated compression ratio and the original data size, $Size_o$. Next, the required compression time, T_c^{est}, can be obtained by multiplying the predicted number of iterations executed with the average running time of one iteration on the reference standard machine, and $Load_{now} + 1$, then dividing by the Zmark index of the system. Finally, we can compute T_{to}^{est} and T_{tc}^{est}, the transmission times of the original and compressed data, respectively, by the following equations:

$$T_{to}^{est} = \frac{Size_o}{Speed_{net}}, \qquad (6)$$

and

$$T_{tc}^{est} = \frac{Size_c}{Speed_{net}} = \frac{Size_o \times (1 - Ratio_c)}{Speed_{net}}. \qquad (7)$$

5. Compression Ratio Prediction

From previous sections, we see that an effective and efficient compression ratio prediction mechanism is indeed the key to success of our compression scheme. Developing such a prediction mechanism is quite challenging because we rely on the accuracy of predicted ratio to correctly render T_c^{est} and T_{tc}^{est}, and such a predictor must also be efficient enough to minimize T_p, the overhead of the predictor itself. In this section we devise a heuristic called *prediction-by-sampling*, to solve both problems simultaneously.

5.1. Prediction-by-Sampling

The concept of prediction-by-sampling is simple. Given a large raw data block, we arbitrarily select a number of small samples (short strings) from within that data block and calculate their individual compression ratio estimations. The average compression ratio estimation of all samples can thus be used as the predicted compression ratio of the whole input data block. If the sample size and the number of samples are properly chosen, the predicted compression ratios should be quickly obtained and very close to the actual ones. To put this concept into practice, however, we must first determine how to estimate a sample's compression ratio, in addition to how large a sample should be, and how many samples should be drawn.

At the first glance, one may try to directly use the core compression mechanism to compress a sample and measure its actual compression ratio. Unfortunately, most dictionary-based algorithms, including the LZHUF we use, are not efficient to compress small patterns because they impose a relatively large initializing overhead. Therefore, we adopt the following alternative approach instead: Each sample is matched with all substrings within a history "window" range located just before that sample in the input raw data block. Call longest matched substring $maxmatch$ and its length and position $maxmatchlen$ and $maxmatchpos$, respectively. Observe that the character sequences of $maxmatch$ occur twice—first time in the "window," second in the sample—the second occurrence in the pattern can thus be replaced by the ($maxmatchlen$, $maxmatchpos$) pair to save space. If m bits are required to encode the ($maxmatchlen$, $maxmatchpos$) pair, then the estimated compression ratio of that sample can be calculated by the following formulas:

$$\frac{\#\, of\, bits\, of\, maxmatch - m}{\#\, of\, bits\, of\, maxmatch} = \frac{maxmatchlen \times 8 - m}{maxmatchlen \times 8} \tag{8}$$

$$= 1 - \frac{m}{maxmatchlen \times 8}. \tag{9}$$

In fact, such an approach is identical to the core string-matching operations of dictionary-based compression algorithms (see Figure 2), except that we do not have to initialize or maintain the dictionary in those compression algorithms. Eliminating dictionary-manipulating overhead can greatly improve the speed of estimating compression ratio of the samples. Note that such an prediction-by-sampling approach also implicitly determines the maximal sample size, which is bound by the size of the history "window."

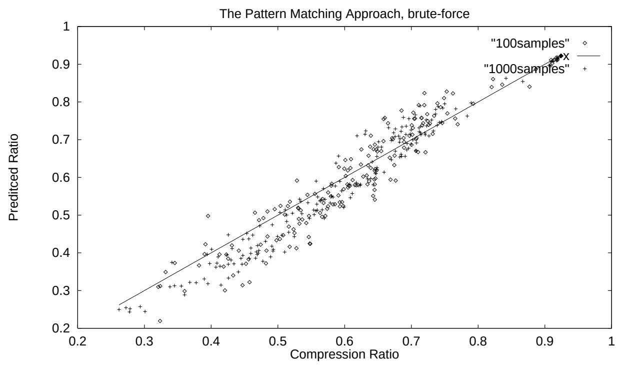
Once the compression ratios of all samples are calculated, the predicted compression ratio of the whole input data block can thus be calculated easily, which is

$$\frac{\sum_{i=1}^{N} maxmatchlen_i \times 8 - \sum_{i=1}^{N} m_i}{\sum_{i=1}^{N} maxmatchlen_i \times 8} = 1 - \frac{\sum_{i=1}^{N} m_i}{\sum_{i=1}^{N} maxmatchlen_i \times 8}, \tag{10}$$

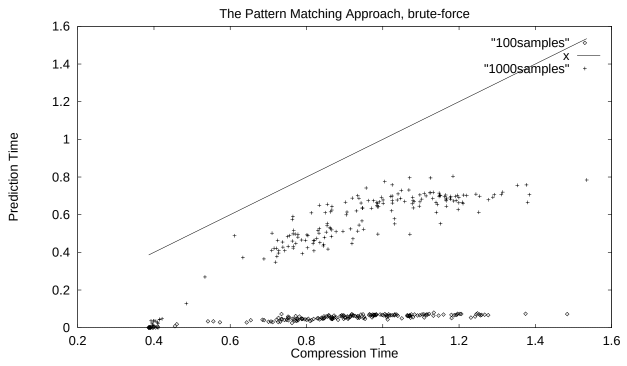
where N is the number of samples selected from the input data block and should be determined by both the input data length and the sample size.

5.2. Experimental Results and Analyses

To investigate the efficiency and effectiveness of our prediction-by-sample heuristics, we conducted an experiment similar to that of Section 4.2 on the same reference standard platforms. 200 data blocks of 1*MBytes* are extracted from each platform's swap partition as the raw input data blocks. In our predictor implementation the history window size is fixed as 4*KBytes*, and the maximal sample size is 32*Bytes*. To make our predictor behave as similar to our LZHUF algorithm as possible, The latter two parameters are set to be the same as those used in LZHUF. We first draw 100 samples from each input data block, and compute the predicted compression ratio and record the elapsed time for prediction. Next, the whole procedure is repeated with the number of samples increased to 1000. The results of predicted and actual compression ratios are plotted in Figure 4(a), and the ratios of elapsed prediction vs. actual compression time are plotted in Figure 4(b), respectively, to contrast the prediction and actual compression outcomes.



(a) Compression ratio (closer is better)



(b) Time used (lower is better)

Figure 4. Results of the brute-force approach.

From Figure 4(a) we observe that the predicted ratios are quite close to the actual ones in both 100- and 1000-sample cases. In fact, the average prediction errors of the 100- and 1000-sample cases are around 7.38% and 6.59%, respectively. Clearly, for a 1*M-Byte* input 100 32-*Byte* samples have already provided adequate prediction accuracy. A closer inspection reveals that although the sample size is small apparently, the coverage of each sample is indeed as large as the size of the history window, because each sample will be matched with all substrings within its 4*K-Byte* history window. Therefore, the total coverage of 100 samples can be up to 400*KBytes*, or equivalently 40% of the input data size at best, enough to retain good prediction accuracy.

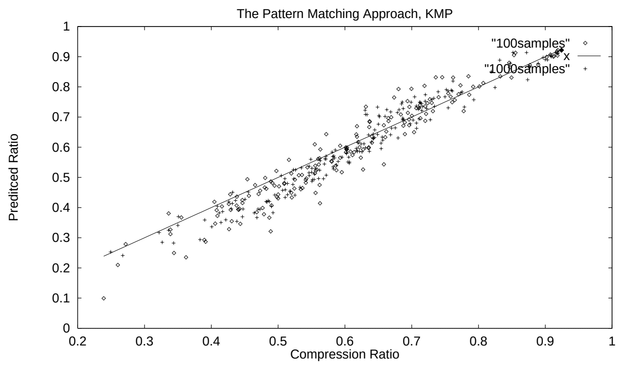
From Figure 4(b) we find that on average, evaluating 100 and 1000 32-*Byte* samples of a 1*M-Byte* input takes respectively 5% and 55% of the actual compression time to finish. For most practical uses, the time cost of evaluating 1000 samples on a 1*M-Byte* input is intolerably high. Indeed, the 1000-sample latter case suffers from a phenomenon called *oversampling*; the total coverage of 1000 samples is indeed $4MBytes$, which is substantially larger than the original input size. Obviously, the improvement of accuracy from 7.38% to 6.59% does not justify the 10x overhead of 1000 samples. When the number of samples falls back to 100, the overhead level of our prediction mechanism becomes far more acceptable (only 5% of the actual compression cost); on the other hand, we have not yet incorporated any performance enhancement features into our predictor, hence the efficiency can be further improved.

As a summary, our prediction-by-sampling mechanism achieves very high (93%) prediction accuracy, while its efficiency is also good enough (provided that the input is not oversampled), yet can be further improved.

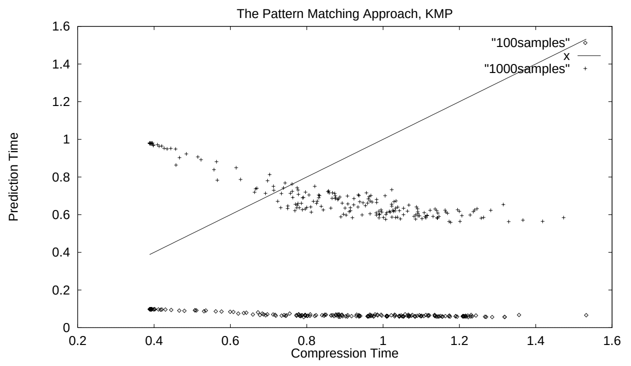
5.3. Optimizing Prediction Efficiency

One way to improve the performance of our predictor is to replace the "brute-force" string matching routine by a more efficient algorithm. The algorithm we adopt is the famous Knuth-Morris-Pratt's (KMP) algorithm [4, 7], which utilizes a special skip table to decide how many characters should be skipped if a mismatching is encountered, effectively reducing the number of substrings to be compared.

With the enhancement of the KMP algorithm, the whole experiment of Section 5.2 is repeated and the new results are plotted in Figure 5(a) and (b), respectively. Also note that although suffered from the oversampling problem, the 1000-sample results are still kept for symmetry. From Figure 5(a) we find that the accuracy levels are virtually the same as those of the previous section. This is quite reasonable because the change of string matching function should not affect the matching length/position results; only the matching speed changes. Unfortunately, from Figure 5(b) we discover that the substitution of the KMP algorithm does not improve the prediction efficiency at all. On the contrary, the efficiency of the predictor is even further lowered: the average overhead level of the 100-sample case arises to 9.63%, and sometimes a 1000-sample prediction is even slower than directly compress the whole input data block. A closer investigation discloses that the speedup in string matching is overshadowed by the large initial overhead of the KMP algorithm in preparing the skip table; i.e. the input and sample sizes are too small for the KMP algorithm. Hence on large input data and samples, the KMP-enhanced predictor may be able to outperform its predecessor.



(a) Compression ratio (closer is better)

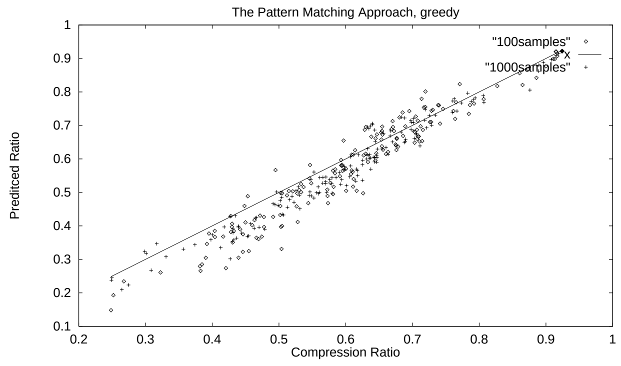


(b) Time used (lower is better)

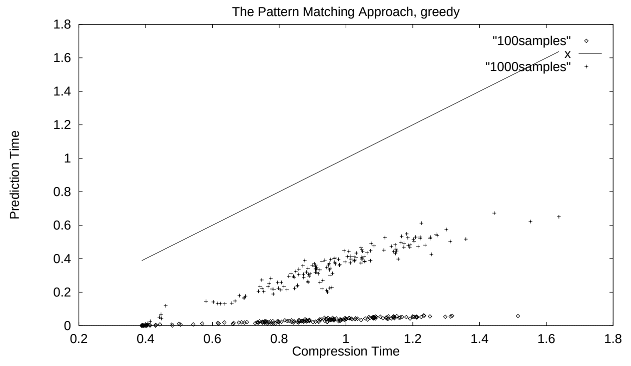
Figure 5. Results of the KMP approach.

Another approach we take is called *greedy heuristics* to accelerate string matching. We observe that, the (partial-) string matching of a sample and its associated history window within the input data is indeed an incremental process; that is, if a partial match of length, say, l has been found, all other partial matches whose lengths are smaller than l are useless and will be discarded. Consequently, if the current maximum matching length is l, we can first search the $(l + 1)$-th character of the sample with the remainder of the history window. If we do find an identical one, then we can jump back l characters to see if a match of length $l + 1$ or more is really found. Note that if we need to find a complete match of the whole sample, such a heuristics does not reduce the amount the same as the brute-force method; however if only partial matches are required, like the case of our predictor, this heuristics is likely to reduce the number of unnecessary comparisons.

After integrating the greedy heuristics, we repeat the experiment of Section 5.2 to examine the characteristics of our new predictor and exhibit the results in Figure 6(a) and (b), respectively. From Figure 6(a) we find that, similar to the previous two versions, the greedy predictor achieves basically the same accuracy level. On the other hand, from Figure 6(b) we can observe that the greedy heuristics sig-



(a) Compression ratio (closer is better)



(b) Time used (lower is better)

Figure 6. Results of the greedy approach.

nificantly boosts the prediction efficiency by $\frac{1}{3}$: the 100- and 1000-sample predictions now take only 3.2% and some 34% of the actual compression time to execute, validating the effectiveness of the greedy heuristics.

6. Conclusions

In this paper we proposed a compression ratio prediction mechanism based on the prediction-by-sampling heuristic. We also presented a compression speed estimation scheme which renders estimated compression time/speed from a predicted compression ratio. Our prediction mechanism is both effective and efficient, achieving 93% of prediction accuracy at the cost of only 3.2% of the execution time of actually compressing. The high efficiency and high accuracy of our prediction-by-sampling mechanism give our compression algorithm extra flexibility to quickly adapt itself to the characteristics of the input data. When the input data stream is extremely compressible, our algorithm behaves as if it was a regular unconditional dictionary-based compression algorithm. On the other hand, when the input data stream is not suitable for compression, our algorithm behaves as if there was no compression at all. In most other cases, our algorithm is able to deliver better performance than the both two extremes. Such an flexibility makes our algorithm especially suitable for a highly dynamic communication environments such as distributed systems. Also, our scheme and methodology can incorporate other compression algorithms and prediction mechanisms, although the compression ratio/speed relationships and the speed index (Zmark) should be re-formulated.

References

[1] M. A. Bassiouni and A. Mukherjee. Data compression in real-time distributed systems. In *IEEE GLOBECOM '90*, volume 2, pages 967–971, 1990.

[2] A. Y. D. Cheung. Data transfer using controlled compression. Master's thesis, Dept. of Computer Science, Univ. of Waterloo, Canada, 1996.

[3] R. Chow and T. Johnson. *Distributed Operating Systems and Algorithms*. Addison-Wesley, 1997.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[5] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. IRE.*, 40(9):1098–1101, 1952.

[6] C. B. Jones. An efficient coding system for long source sequences. *IEEE Trans. Inf. Theory*, IT-27(3):280–291, 1981.

[7] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[8] A. I.-C. Lai and C.-L. Lei. Data prefetching for distributed shared memory systems. In *Proc. HICSS-29*, volume 1, pages 102–110, 1996.

[9] G. G. Langdon, Jr. An introduction to arithmetic coding. *IBM J. Res. Dev.*, 28(2):135–149, 1984.

[10] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the sprite network file system. *ACM Trans. Comput. Syst.*, 6(1):134–154, 1988.

[11] V. Phalke and B. Gopinath. Compression-based program characterization for improving cache memory performance. *IEEE Trans. Comput.*, 46(11):1174–1186, 1997.

[12] D. M. Royals, T. Markas, N. Kanopoulos, J. H. Reif, and J. A. Storer. On the design and implementation of a lossless data compression and decompression chip. *IEEE J. Solid-State Circuits*, 28(9):948–953, 1993.

[13] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, 1994.

[14] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[15] J. A. Storer and T. G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982.

[16] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, 1995.

[17] J. S. Vitter and P. Krishnan. Optimal prefetching via data compression. *J. ACM*, 43(5):771–793, 1996.

[18] R. N. Williams. *Adaptive Data Compression*. Kluwer Academic Publishers, 1991.

[19] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, IT-23(3):337–343, 1977.

[20] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory*, IT-24(5):530–536, 1978.