High performance PPMC Compression Algorithm

Feregrino Uribe C.
Computer Science Department, National Institute for Astrophysics, Optics and Electronics,
P.O.Box 51 and 216, Puebla, 72000, Mexico
cferegrino@inaoep.mx

Abstract

It has been demonstrated with recent software implementations of context modeling the capability of PPM (Prediction by Partial Matching) [1] type of algorithms to achieve very high compression rates. However, the cost is high in terms of computational complexity and low speed. Hardware implementations of compression algorithms are capable of increasing compression speed by at least an order of magnitude compared with same compression methods implemented in software. In this paper we investigate and study the issues related to simplification of one PPM type of algorithms, the PPMC [2] to achieve high performance.

1. Introduction

Nowadays, there is an ever-increasing demand for faster and better digital communications. The explosive growth of telecommunications requires large amounts of data to be transmitted or stored in the least possible time. Data compression allows systems both to gain space for storage and increase the bandwidth for data transmission, offering a vehicle for cost reduction and efficient operation. Examples of devices that benefit from data compression are routers, hard disks and modems.

There is a continuing demand for improved compression. Network applications make it difficult for software compression to deliver the demanding speeds. Hardware lossless data compression helps digital devices to handle large volumes of data and satisfy the required compression speeds, enhancing the scope and cost-effectiveness of data transmission.

Generally, commercial hardware data compressors are dictionary-based and use LZ-type of algorithms [3]. Statistical algorithms, as PPMC that predict symbols based on statistics, achieve higher compression but at the expense of higher complexity and lower speed. To date, the research into PPMC compression model has generated only software simulations and no hardware implementations have been developed. This paper identifies and analyses the main computational requirements of PPMC, analyzes the interaction and tradeoffs between algorithmic desired characteristics and hardware capabilities to ensure the effective mapping of algorithmic computational requirements into compression architectures.

The remainder of this paper is organized as follows. Section 2 covers related work and explores the PPMC technique. Section 3 identifies its main computational requirements. Section 4 looks into the issues that impact compression performance and hardware design. Section 5 shows the hardware modeling and Section 6 looks into the hardware requirements. Finally Section 7 concludes.

2. Related work

The PPMC model is the first practical implementation of the PPM class of compression algorithms[2]. The scheme maintains a dictionary containing a statistical model of the data, assigns probabilities to the symbols and sends these probabilities to an arithmetic coder[4], which codifies these probabilities into bits.

The statistical model in its simplest form counts the number of times each symbol has occurred in the past and assigns a probability of occurrence to the symbols accordingly. A more sophisticated model is context based, where not just the frequency of the symbol is used to predict but also the particular sequence of symbols that immediately preceded that symbol. This sequence of symbols is called context and its length is the order of the context.

A PPMC model of order *o* reads a symbol *s* and considers the previous *o* symbols as the current context. Then it searches in the dictionary for the symbol *s* preceded by the context of order *o*. If the symbol is found, its probability is sent to the coder. If the symbol is not found, the model estimates the probability of the novel event by 'escaping' to the next lower order *o*-1, transmitting an escape code (*esc*). Then, the process continues until the symbol is found or the model reaches the order –1, where all symbols have the same probability of occurrence. The model is then updated adding the symbol *s* to the corresponding contexts. Next, *s* becomes part of the o^{th} order context to predict the next symbol.

PPMC 'computes' symbol and escape probabilities using the method C (from where the model takes its name) with the following formulas:

$$p(s \mid context) = \frac{f_s}{t+k} \quad \text{and} \quad p(esc \mid context) = \frac{k}{t+k} \quad (1)$$

where *p(s|context)* is the probability that symbol s will occur given that context has occurred; f_s is the frequency count of symbol *s*; *k* is the number of different symbols seen in the current context, and *t* is the sum of the frequency counts of all symbols in the current context.

The PPMC algorithm is carefully tuned to improve compression and increase execution speed by using lazy exclusions [5], taking into account frequency counts in context levels at or above the context in which a symbol was predicted. Then, in the updating process just these frequency counts are updated.

Table 1 shows a 2nd order PPMC model at some stage in the compression process. An 'empty' context means that there is no context to follow; the counts represent the frequency of the symbols. Frequency counts of 0 indicate that the symbol has not been seen in the corresponding context. The 'total' is the sum of the frequency counts (t). Order -1 is a special case that has, and thus predicts, all possible symbols of 8 bits, so t=256.

For example, in English text if the stream '*th*' occurred, it is more probable the next symbol would be '*e*' rather than '*u*'. Then, according to Table 1, if the current context is '*th*' and the incoming symbol were '*u*', the model escapes from 2nd order with $p('esc'|'th') = 0.059$, and the symbol is predicted by order 1 with $p('u'|'h') = 0.009$. Later, during the model updating, the frequency counts and the total are augmented by 1. Table 1 would update '*u*' count in 2nd and 1st orders, from 0 to 1 and from 2 to 3 respectively.

Note that from Table 1 $p('i'|'th')=0.217$ and $p('i'|'h')=0.115$. Then, the entropy, $E_l = - log_2\ p_l$, quantifies the information content, where E_l is the entropy and p_l is the probability of the l^{th} symbol.

According to this formula, symbol '*i*' would be codified with 2.20 and 3.12 bits in 2nd and 1st orders respectively. Generally and as shown in this example, the higher the order of the context the higher the probability of occurrence and the fewer the bits needed to codify the symbol. Thus, the higher the order the better the compression.

The arithmetic coder [4] requires the probabilities in the form of cumulative frequencies (provided by the model) to encode the symbols. Any practical implementation of the model considers restrictions of the maximum frequency counts the model can handle. The model avoids overflow of the counts by halving all of them once a certain threshold has been reached. This technique is called *count scaling*.

Some other variants of PPM models have emerged: PPMD [6], PPMD+ [7], PPM* [8], PPMZ [9], PPMII [10], they differ in terms of escape strategy and order of

Table 1. Example of the PPMC model*

Order	2	1	0	-1
Context	'*th*'	'*h*'	empty	empty
Symbols				
'*a*'	8	33	226	1
'*e*'	51	110	362	1
'*i*'	22	24	188	1
'*o*'	7	16	248	1
sp	6	14	781	1
'.'	1	1	16	1
'*u*'	0	2	84	1
Total (*t*)	95	200	1,905	

* The frequency counts were obtained from a piece of English text of the file alice29.txt, part of Canterbury Corpus [16]

the model. Most of the models maintain digital search trees or tries, particularly suited for fast search operations in software. Other authors explore prefix trees in PPM* [11] to overcome compression of previous versions, or self-organizing lists as elements of hash tables [12] to improve speed. However, none of these variants seem suitable for simplification and practical implementation in hardware.

It was mentioned in [12] that the complexity and space requirements of PPMC algorithm have prevented its practical use, and no hardware implementations have been built. Ten years later, current advances in technology allow fitting millions of gates in a single chip, which makes worth reviewing the algorithm.

3. Key computational requirements

After reviewing PPMC, the searching and updating processes can be identified as the most demanding. As mentioned in [13], the more complex algorithms generally use more complex data structures, and the reduction in speed is generally due to searches and maintenance of these structures.

As well as searching and updating, other issues emerge when implementing the algorithm, related with space constrains, where scaling counts and discarding policy become important. Discarding refers to the measures adopted by developers to continue model adaptation once the dictionary space allocated for the model has been occupied. Next we will describe each of the issues.

The first computational requirement we consider is searching. In PPMC software implementation [2] the searching time is O(log2 *n*), where *n* is the number of items in the trie, this results in a two-fold speed increase from previous PPM implementations, but additional space is required for the maintenance of the structure. Also self-organizing lists have been used for PPMC [12], where searching and updating time is limited by the size of the

list. Frequently-occurring symbols require a lower searching time. Generally, as the order of the model increases, the searching operations become slower.

Second, we consider the model updating process. The complexity of the model updating process depends on two main issues: 1) the data structure that stores the modeling information and 2) how information is stored. Simple data structures and simple data usually lead to simple updating processes. The PPMC model maintenance requires updating symbol and context frequency counts. When counts are not kept in cumulative form they must be computed on-the-fly, which is time consuming. As the model order increases, updating becomes more complex. The insertion time of a new dictionary entry depends on the data structure. A table may require O(1), while a trie may take O(o), where o is itself the order of the model. Updating cumulative frequency counts in a table may take O(q) where q is the size of the alphabet. However, if efficient structures [14] are considered, the time may be reduced to O($\log_2 q$).

Third, the discarding policy is important due to the storage space limitations for modeling information. In PPMC [2], the trie growth proceeds at full speed while memory is available. Once the memory is exhausted, the entire trie is discarded. To avoid inefficient coding at this stage, the model keeps the last 2,048 symbols transmitted and rebuilds the trie from this information. This solution lessens the degradation in compression performance due to rebuilding.

Finally, we consider scaling. Since storage space is restricted, the maximum number of bits for frequency counts is limited. To avoid overflow, the model scales counts. In addition to solving the problem, the compression ratio of the model is slightly improved[5]. However, the time consumption increases and makes the entire compression process slower and probably not worth the increase in compression rates.

Discarding policy and scaling counts are irrelevant to the algorithm itself but they have to be considered for its implementation, and may have a big impact in its performance due to the number and type of operations performed.

One last computational requirement, associated with modeling is the coding process. In particular, arithmetic coding executes adds, multiplications and divisions during the coding process [4]. Due to the complexity of the divide operations, it is expected that arithmetic coder will execute slowly.

Simpler operations are performed faster but care must be taken if compression performance is to be maintained. Additionally, some operations for a renormalization procedure [15] have to be considered.

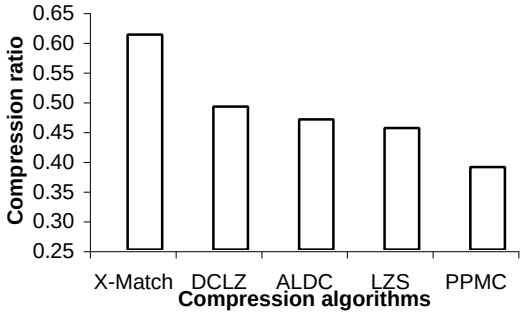

Figure 1. Performance comparison of PPMC and commercial chips

4. Compression performance

In this section we consider several issues that impact the PPMC compression performance and that must be considered for its hardware implementation. The most important is the data structure that will store the model, then the dictionary size, block size, order of the model and discarding policy, as all of these issues have direct impact in its hardware requirements and performance. We compare the performance of PPMC algorithm against several commercial compression chips and later we focus on relevant characteristics for a hardware design of the algorithm.

Experiments show how and to what extent the design issues impact compression performance. Canterbury Corpus [16] is used as a test bed, it includes a collection of 11 files, ranging in size from 3K to 1,029K, from C and LISP source code, html files, technical writings and text files.

4.1 Data structure

The data structure was selected as the most important issue due to the impact in area requirements in a hardware implementation of this algorithm. Area requirements have been one of the issues that have stopped the hardware development of PPMC, since a direct implementation of a binary search tree can be complex. Also, this structure is what stores the modeling information. Binary search trees are tuned for fast software operations and a better hardware replacement must be considered. Literature shows [13, 17, 18] that CAMs (Content-Addressable Memories) are such replacement. They have the ability to search its entire list of available codes in a single transaction.

To observe how PPMC performs using a matrix data structure (that can be directly implemented in a CAM array in hardware), it is compared with commercial

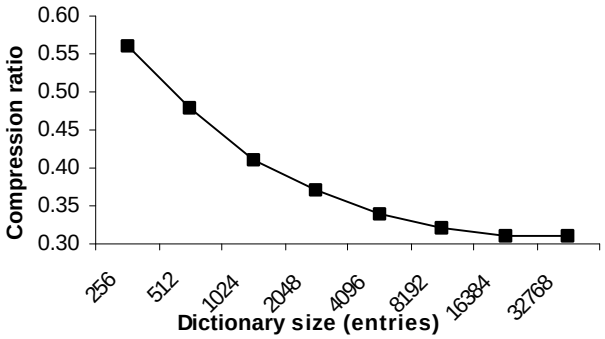

Figure 2. Impact of dictionary size in compression performance of PPMC

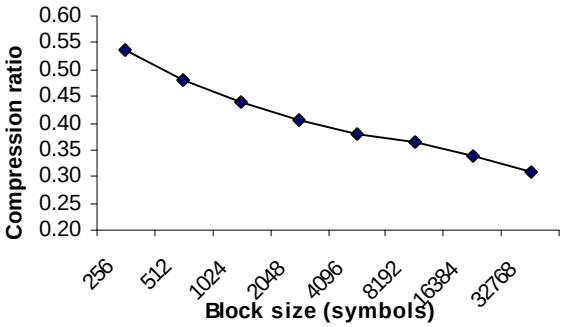

Figure 3. Impact of block size on compression performance

compression chips. All of them use CAMs for model storage and are LZ-based (use LZ[3] algorithms), including LZS [17], ALDC [18] and DCLZ [19]. X-Match [20] is also dictionary-based but uses a different algorithm. To provide fair results, all algorithms are set to similar circumstances. The results from the commercial chips were obtained executing demo versions provided by the companies. Compression ratios result from dividing input bits by output bits, the lower the ratio the better the compression.

In spite of the restrictions imposed to PPMC it is the best performer, see Figure 1. The experiments of the following sections use a similar data structure for model storage because it best resembles a CAM array. Thus it seems sensible to consider in more depth the design of hardware structures to support this.

4.2 Dictionary size

The storage space available limits the dictionary size, which impacts the compression performance. Intuitively, the bigger the dictionary the more accurate statistics of the data can be learn. But also, the smaller the dictionary the lower the space requirements. So, finding a balance between dictionary size and compression warrants high compression performance.

To find that balance, a 3rd order PPMC model is simulated and different dictionary sizes are applied for every run. The results are shown in Figure 2, where each dictionary entry on the X-axis is a 3rd order context plus the symbol it predicts. As expected, compression improve as the dictionary size increases.

With smaller dictionaries, the increase in space requirements are not significant and the gains in compression are considerable. However, as the dictionary size increases, these gains become smaller, until the increase in compression is not worth the increase in space requirements. Additional space for frequencies and/or cumulative frequencies has to be considered.

Table 2. Impact of model order on compression performance

Model Order	Compression rates
0th	0.556
1st	0.467
2nd	0.405
3rd	0.388

4.3 Block size

According to the application of the PPMC compressor, it can compress blocks of different sizes. For example, for software compression used in off-line applications, an entire file can be compressed. However, in on-line or on-the-fly applications, like computer networks, it is not possible to compress entire files, since information is transmitted in packets of a fixed size, depending on the network. So, we investigate the impact of the block size on the PPMC model performance. We have carried out experiments with compressing blocks of different sizes, ranging from 256 bytes to 32,768 bytes. Figure 3 shows the experimentation results. The dictionaries are large enough to allocate the biggest block size tested without reusing space. According to our results, compression performance improves as block size increases. It is expected that large blocks require more space for model storage than small blocks, if the compression ratio is to be maintained. So, a careful selection of block size is required.

4.4 Order of the model

It has been mentioned in section 2 that the higher the order of the model, the better the compression, but it is also true that the compression speed diminishes and the space requirements increase considerably. To find a balance between these requirements and the compression results, it has been carried out some experimentation.

Table 3. Percentage predictions for each order of the model with Canterbury Corpus

Order of the model	Order of the contexts				
	3rd	2nd	1st	0th	-1st
3rd	58.75	13.34	16.12	8.74	3.05
2nd		74.12	15.04	7.91	2.93
1st			89.32	7.80	2.88

As part of a study of the order of the model, we considered the percentage of predictions made by each context order within the model. Orders of the model considered for this experiment are 1st, 2nd and 3rd. The dictionary size is 2,048 positions for 3rd, 2nd and 1st orders and 257 and 256 positions for 0th and –1st respectively. The block size is 4,096 bytes due to this size represents a typical packet size found in many computers and telecommunication systems. The discarding policy is LRU.

The experiment helps to identify how much weight each context order has in the model and if the type of data influence that result. This fact may help later on to determine the size of the dictionary.

Results are shown in Table 2. The figures in the table confirm that the higher the order the better the performance. Higher order models (above 3) were not tested, but there are studies in the literature [8] that reveal that for text, there is a little improvement in compression performance with models of orders higher than 5.

The percentage of predictions is showed in Table 3. The first row shows results for a 3rd order model and indicates the percentage of the order of the contexts that compose it. The second and third rows show results for 2nd and 1st order models respectively. More than 50% of the predictions are made by the 3rd order contexts and surprisingly, more symbols are predicted by the 1st order contexts than by the 2nd order ones.

The statistics of the model gathered by the highest order contexts are more accurate; for this reason, although in higher order models the percentage of predictions in the highest context seems to diminish, the compression ratios improve.

It is worth mentioning that, normally, PPM models have an exponential growth of memory as the order of the model increases [8], see Table 4.

This case considers keeping all possible contexts in the model. However, a practical implementation of PPMC must observe space limitations. An advantage of using a CAM-like structure in a hardware implementation, is that the space requirements can grow linearly allowing significant memory savings.

Table 4. Possible number of contexts as the order of the model increases

Model Order	Possible contexts
0th	256
1st	$(256)^2$=65,536
2nd	$(256)^3$=16,777,216
3rd	$(256)^4$=4,294,947,296

4.5 Discarding policy

A discarding policy is a technique used to continue model adaptation once the space assigned for model storage has been exhausted. The most common discarding policies in practical compression implementations are:

- LRU (Least Recently Used). The system keeps track of the recently used phrases. The phrase to discard is the least recently used.
- LFU (Least Frequently Used). This policy is similar to LRU, except that it maintains frequently used phrases. The phrase discarded is the one with fewer occurrences.
- Climb and Reset (either randomly any entry or the entire model). Climb policy moves up one position the entry that just matched, it can be performed in constant time [21]. Resetting randomly any entry requires generating the random positions to be discarded, this seems to be the simplest policy but the worst performer.

Table 5 summarizes our analysis on these policies. From the literature [2,4,16] and our analysis we know that LRU policy ensures the maximum memory utilization, and it adapts well to changes. However, resetting the entire dictionary is the simplest and more attractive policy to be implemented in hardware.

It has to be considered that both discarding policies, LRU and resetting (the entire dictionary) have considerable advantages for a practical implementation of the PPMC algorithm. On one hand, LRU provides good compression ratios, on the other hand, resetting is a very simple approach to continue adaptation. However, both also have disadvantages, LRU is time consuming and depending on the statistics storage (specially in a hardware implementation if a CAM is to be used) the adaptation can be very complex. Then, a balance has to be done in choosing the better discarding policy according to the application requirements.

4.6 Multi-dictionary model

As two discarding policies have been considered the best in last section for their good adaptation and simplicity respectively, we propose merging them to provide faster and simpler reuse of dictionary positions. This merging

Table 5. Common discarding policies used in compression practical implementations

Discarding policy	Function	Advantages	Disadvantages
Least Recently Used (LRU)	Removes the 'oldest' entry	Discard less probable symbols/contexts	Maintains sorted the used dictionary positions
Least Frequently Used (LFU)	Removes entries used less times	Do not move/add counts	Maintains sorted the use frequency dictionary positions
Climb	Moves only one position to the front	Easy to maintain	Need to 'move' data and frequency counts
Reset Randomly	Frees any entry randomly	Do not need to manage the positions to free	Generates random numbers. Need to update many counts
Reset the Entire Dictionary	Resets the dictionary	Very easy to maintain	May harm compression results

results in a multi-dictionary model, that looks for the simplification of the LRU complexity and the improvement of the compression performance of reset policy. It uses each dictionary for storing contexts of a single order. For example, a 2^{nd} order model has 4 dictionaries, for -1^{st}, 0^{th}, 1^{st} and 2^{nd} order contexts respectively. The strategy to follow is simple; when one of the dictionaries (except for orders –1 and 0) becomes full, it can be reset.

This combined discarding policy has not been tested, so there is not knowledge about the performance of the model under these conditions. The following experiment should provide this knowledge. A 3^{rd} order model is simulated, the dictionary and block sizes are the same as for the last experiment. Obviously, the discarding policies are LRU and resetting the dictionaries that fill up.

When the dictionaries fill up the space assigned to them they are reset, except for orders –1 and 0. To compare the results, a model with separated dictionaries but using the LRU policy was simulated.

Table 6 shows the compression results of this multi-dictionary policy. The first column shows the compression ratio when storing the modeling information in a single dictionary. The second and third columns show results for a multi-dictionary PPMC, using the LRU and resetting an entire dictionary discarding policy respectively. From the table, storing data in separated dictionaries and resetting one of the dictionaries does not harm compression ratios, just a minimum 1% degradation is observed compared with the LRU policy.

Table 7 shows the space requirements for the model that stores the data in a single dictionary and uses an LRU.

Table 6. Compression performance of the compression model

PPMC single dictionary	LRU policy	Resetting a dictionary
0.389	0.388	0.393

Table 7. Space requirements for a single dictionary

Dictionary order	Space requirements (bits)		
	Symbols	Frequencies	Cumulative frequencies
3^{rd}, 2^{nd}, 1^{st}	65,536	73,728	73,728
0^{th}	2,313	3,084	3,084
-1	2,048	2,048	2,048
Total	227,617 bits		

Table 8. Space requirements for separated dictionaries

Dictionary order	Space requirements (bits)		
	Symbols	Frequencies	Cumulative frequencies
3^{rd}	65,536	24,576	24,576
2^{nd}	49,152	24,576	24,576
1^{st}	32,768	24,576	24,576
0^{th}	2,313	3,084	3,084
-1^{st}	2,048	2,048	2,048
Total	309,537 bits		

The extra requirements for maintaining the policy are to be considered, at least 2,048*9 (length of the dictionary * width) bits are required.

Table 8 shows the space requirements for the multi-dictionary model. A few bits more are necessary to keep the number of positions being used in each dictionary.

Comparing Table 7 and Table 8, it can be seen that up to 21% extra space is required when the model is stored in separated dictionaries.

Independently of the discarding policy used, separating the modeling information in several dictionaries does not harm compression ratios, as this experiment showed.

Resetting part of the model when one of the dictionaries has filled up provides compression ratios close to the LRU policy and simplifies considerably the complexity of the model at the cost of higher space requirements. This may be the equivalent to the discarding policy used in [2], where part of the trie is kept to reinitialize the model.

It is worth mentioning that an additional, and very important, gain of using multiple dictionaries is the possibility of searching for the symbols in parallel, *i.e* .it is possible to look in all the dictionaries for the symbol at the same time, what reduces considerably the searching time.

4.7 Resizing the dictionary

From Table 3 we know the percentage of predictions made by each order context and from Table 6 we know that the multi-dictionary model not really harms compression ratios. Taking into account both results, it seems helpful to resize the dictionaries according to the percentage of symbols predicted in each context. This could result in considerable savings in space.

The simulation of the model has been done separating contexts of the same order in different dictionaries and having different space limitations proportionally to the percentage of predictions made by each context order.

The compression ratio obtained was 0.397, just about 1% of degradation compared with the model that uses multiple dictionaries but has 2,048 positions for each of them. In this case, the dictionary of 2nd order contexts saves 75% of the positions and the dictionary of 1st order contexts halves its size.

Then, there are significant savings in space requirements, about 59% as shown in Table 8 and compared with Table 9.

From this experiment we can conclude that it is possible to change the dictionary sizes according to the percentage of predictions given in each context order.

A good measure to minimize space requirements in this model is to further study the order of the model to implement, and the weight that the contexts of each order have in the predictions. If possible, the study of the type of data also helps to define well-balanced dictionaries in terms of size, as this experiment confirms.

5. Hardware Modeling

PPMC hardware suitability is proved by the implementation of a 1st order hardware-modeling unit coupled with an arithmetic coder module. This unit is modeled with the SystemC modeling platform from the Open SystemC Initiative (OSCI). Our system is compiled with the VC ++ compiler, version 6.0, on a Windows NT platform. The assumptions are indicated in Table 10, and the arithmetic coder module uses the code from [4], which was adapted to the new requirements. The coder inputs the output signals from the model.

Both, compressor and decompressor were built as SystemC designs, the compression results were verified against a behavioral C language simulation.

Figure 4 illustrates the pseudocode of the algorithm for the compressor. Input data are entered to a shift register that assembles the context and the input symbol to produce the dictionaries input. When indicated by the control, this context and symbol are searched in parallel in the dictionaries. Each dictionary inputs the contexts according to its order.

Table 9. Space requirements resizing the dictionaries

Dictionary order	Space requirements		
	Symbols	Frequencies	Cumulative frequencies
3rd	65,536	24,576	24,576
2nd	12,288	6,144	6,144
1st	16,384	12,288	12,288
0th	2,313	3,084	3,084
-1st	2,048	2,048	2,048
Total	194,849 bits		

Cumulative frequency counts are transferred from the dictionaries to the output logic. If the search operation in a dictionary is not successful, escape cumulative frequencies are output. The output logic selects the best match and sends it to the coder together with other signals needed to codify them and form the compressed data.

There is one dictionary per every order of the model. The dictionary of order –1 contains the control logic and an array of symbol frequencies in cumulative form, where the index in the array indicates the symbol. Dictionaries of order 0 and above contain a memory block and two registers. They are managed by simple control logic indicating when to search or update. The memory block includes a CAM array to store the input data and two register files to store the frequency counts and cumulative frequency counts of the symbols and their contexts.

The decompressor architecture is similar to the compressor although its task is more complex. Its operations include serial and parallel searches. Serial searches are required when the model looks for symbols. When the model is being updated, it requires searching for escape symbols and, in this case, parallel searches in the dictionaries are performed.

6. Hardware Requirements

The architecture of the model, simulated in behavioral form, was analyzed to get an estimated number of gates required. The model estimated gate count is based on the 1st order model of Figure 4 and is shown in Table 11.

The compressor architecture estimated size is approximately 3 million NAND equivalent gates, from which most of the space is assigned to storage and updating of data.

On average, 3.29 and 7.5 behavioral clock cycles are required to process a symbol for compressor and decompressor respectively. These figures were obtained by dividing the number of behavioral clock cycles required to compress complete data set by the number of symbols compressed. The figures are simulation times and do not necessarily correspond to machine cycles.

```
Clear the dictionaries;
Set LC(longest context) to context;
set CO (context order) to order of the model;

DO
{
        read in a symbol from the data stream;
        search for LC & incoming symbol in all the dictionaries;
        select best match and set BMO = order of best match;
        IF (order of best match = CO)
        {
              output symbol cumulative frequencies;
              update frequencies in dictionary of CO;
       }
       ELSE
       {
            from BMO to CO do:
                  output 'escapes' (CF_Esc) of orders BMO+1 to CO;
                  output CFs from BMO;
                  add LC + symbol to dictionaries of  orders BMO+1 to
                    CO);
                  update frequency counts in dictionary of BMO;
       }
      recompute cumulative frequencies, CFs;
      update LC;
       compute arithmetic coding operations;
} WHILE( more data is to be compressed);
```

Figure 4. Pseudocode for the parallel PPMC

It seems feasible to implement the PPMC model using present day technology, in a single digital VLSI integrated circuit or in other technologies such as FPGAs, e.g. Xilinx FPGA Virtex-II family, that has up to 10 million usable gates. Further work involves taking this design into an FPGA to get accurate performance results.

7. Conclusions

PPMC statistical model has been analyzed, its main computational requirements and its hardware design issues have been identified. A new hardware architecture has been proposed for this algorithm that promises high operating speeds while maintaining its compression performance.

8. References

[1] J. G. Cleary, I. H. Witten, "Data compression using adaptive coding and partial string matching", *IEEE Trans. on Comm.*, 1984, 32 (4), pp. 396-402.

[2] A. Moffat, "Implementing the PPM Data Compression Scheme", *IEEE Trans. on Comm.*, 1990, 38 (11), pp 1917-1921.

[3] Salomon, D. *Data Compression: The Complete Reference*, Springer, USA, 2000, 2nd ed.

[4] I.H. Witten, R.M. Neal and J.G. Cleary, "Arithmetic Coding for data compression", *Comm. of the ACM*, 1987, 30 (6), pp. 520-540.

[5] Bell, T.J., Cleary J.G. and Witten I.H., *Text compression*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[6] Howard P.G., *The design and analysis of efficient lossless data compression systems*, PhD Dissertation, Department of Computer Sciences, Brown University, 1993.

[7] W.J. Teahan, "Probability estimation for PPM", 2nd Second New Zealand Computing Sciences Research Students' *Conference, NZCSRSC*, April, 1995.

[8] J.G. Cleary and W.J. Teahan, "Unbounded length contexts for PPM", *Computer Journal*, 36 (5), 1993, pp. 1-9.

[9] C. Bloom, "New techniques in context modeling and arithmetic encoding" *IEEE DCC*, Los Alamitos CA, March 1996, March, p. 426.

[10] D. Shkarin, "PPM: One step to practicality", *IEEE DCC'02*, Los Alamitos, CA, April 2002, pp. 202-211.

[11] M. Effros, "PPM performance with BTW complexity: a new method for lossless data compression", *IEEE DCC'00*, Los Alamitos ,CA, March 2000, pp. 203-212.

[12] D.S. Hirschberg and D.A. Lelewer, "Context modelling for text compression", *Image and Text Compression*, Kluwer Academic Publishers, Norwell, MA, 1992.

[13] Music Semiconductors, Application notes AN-N11, September 1998, www.music-ic.com.

[14] P.M. Fenwick, "A new data structure for cumulative frequency tables", *Software -Practice and Experience*, 24 (3), pp. 327-336

[15] P.G. Howard and J.S. Vitter, "Arithmetic coding for data compression", *Proceedings of the IEEE*, 82 (6), pp. 857-865.

[16] R. Arnold and T. Bell, "A corpus for the evaluation of lossless compression algorithms", *IEEE DCC'97*, Los Alamitos, CA, March 1997, pp. 201-210.

[17] HiFn Application notes 'How LZS compression works', available from www.hifn.com.

[18] D.J. Craft, "A fast hardware data compression algorithm and some algorithmic extensions", *IBM Journal of Res. and Dev.*, Data Compression technology in ASIC cores, 42 (6).

[19] Application note '10 MBytes/sec DCLZ Data Compression Coprocessor IC', from Advanced Hardware Technologies, available from www.aha.com

[20] M. Kjelso, M. Gooch and S. Jones, "Design and performance of a main memory hardware compressor", *22nd EUROMICRO*, September 1996, pp. 423-430.

[21] R. William, *Adaptive data compression*, Kluwer, Boston, London, 1991.

Table 10. Assumptions for the hardware implementation of PPMC

Model Order	1st order PPMC model
Dictionary size	4096 positions for 1st order, 256 and 257 positions for 0th and –1st orders
Block size	4096 bytes, this size represent a typical packet size found in many computers and telecommunication systems
Data set	Canterbury Corpus
Discarding policy	Not required

Table 11. Estimated system size based on a 1st order PPMC model

Order	Gate count
–1	18,432
0	219,248
1	3,117,056
Other Logic	568
Total Gate Size	3,289,999