A Configurable Statistical Lossless Compression Core Based on Variable Order Markov Modelling and Arithmetic Coding

Jose Luis Nunez-Yanez, Department of Electronic and Electrical Engineering, University of Bristol, BS8 1UB, UK, e_ mail:j.l.nunez-yanez@bristol.ac.uk,
Vassilios Chouliaras, Department of Electronic and Electrical Engineering, University of Loughborough, LE11 1TU, UK, e_mail: v.a.chouliaras@lboro.ac.uk

Abstract— This paper presents a practical realisation in hardware of the concepts of variable order Markov modelling using multi-symbol alphabets and arithmetic coding for lossless compression of universal data. This type of statistical coding algorithms has long been regarded as being able to deliver very high compression ratios close to the information content of the source data. However, their high computational complexity has limited their practical application in embedded environments such as in mobile computing and wireless communications. In this work a hardware amenable algorithm named PPMH and based on these principles has been developed and its architecture and implementation detailed. This novel lossless compression core offers innovative solutions to the computational issues in both stages of modelling and coding and delivers high compression efficiency and throughput. The configurability features of the core allow efficient use of the embedded SRAM present in modern FPGA technologies where memory resources range from a few kilobits to several megabits per device family. The core has been targeted to the Altera Stratix FPGA family and performance, coding efficiency and complexity measured for different memory configurations.

Index Terms— Markov modelling, statistical compression, lossless compression, arithmetic coding.

I. INTRODUCTION

The past 5 years have witnessed an explosion in wireless networking demand and capability mainly motivated by the huge success of handheld and mobile devices such as cellular phones

and Personal Digital Assistants (PDA) [1]. Wireless networks deliver the power and freedom of mobility at the expense of lower bandwidth and reliability (packet loss) when compared to their wired counterparts. Lossy Data compression is already heavily utilized in voice and video signals and global standards exist such as MPEG4 [2] for video or G729 [3] for voice. These methods, because of their lossy nature, cannot be applied to general data such as text, html content, database information or application data and binaries since exact reconstruction of these data types is mandatory after decompression. Advanced Lossless data compression for this application area can reduce bandwidth requirements significantly and simultaneously deliver energy savings of great importance in mobile computing. Energy savings can be achieved as long as the energy cost of transmitting a block of data are higher than those associated with compressing it and transmitting the resulting compressed block. This paper presents a practical realisation of a high-order Markov model and associated arithmetic coder in the form of a compression IP core able to deliver high throughputs and compression ratios superior to classical dictionary-based algorithms such as GZIP and WinZIP.

II. BACKGROUND

Current lossless data compression technology makes a distinction between dictionary-based and statistical-based algorithms [4]. Dictionary-based compression has been traditionally more popular in software and hardware due to its inherent simplicity. Examples of dictionary-based compression implementations in software are the popular WinZIP or GZIP algorithms commonly used for archiving and distributing large amounts of data in desktop systems. Also, the hardware devices available from leading companies such as IBM [5], AHA [6] and HiFn [7] microelectronics use dictionary-based compression methods based on the original LZ77 [8] and LZ78 [9] algorithms. Statistical compression is not so popular, although it is recognised as able to offer superior compression ratios [10]. However, this has been only achieved with complex software implementations [11] that consume vast amounts of memory

and have very low throughputs, in the range of thousands of CPU cycles per byte. This means that power-hungry, Pentium 4 class microprocessors running at GHz rates are needed to provide the computing power to run these advanced statistical algorithms in real time. This combination of complex software and complex microprocessor solution is unsuitable for battery-power wireless devices or embedded systems. Few statistical hardware compressors have been reported in the literature. A particularly successful example is the IBM Q-Coder [12] device targeted to the compression of black and white fax images. This chip is based on a simple high-order (7^{th}) fixed binary model since only two possible symbols are present in the input data source. It will therefore offer poor compression if it is used with data of unknown nature in an application domain such as wireless networking. The fixed-order model is viable with a binary alphabet because no escaping [4] can take place to lower orders since both possible symbols (black or white pixel) always have a probability higher than 0. Assigning a probability *p* to one of the symbols automatically assigns the rest *1-p* to the other so a single count must be stored and manipulated. The binary alphabet considerably simplifies the algorithm and hardware design although it cannot efficiently extract the redundancy present in general data which is typically of a byte-oriented nature. This device achieves a throughput of 64 Mbits/second implemented in the CMOS 5S (0.35 um) technology from IBM. The work conducted by the Taiwanese team leaded by Professor Jou has also investigated a similar concept of combining a fixed high-order binary model with an arithmetic coder [13]. The model order is increased from 7^{th} to 10^{th} order and additional tuning steps are added to improve compression efficiency. Efforts at using a byte-based alphabet have been limited to 0^{th} order context-free models due to their complexity. The results shown by the Spanish team led by Prof. Bruguera [14] showed that the compression efficiency of this simple model cannot compete with dictionary-based compression. Similar results are presented by other researchers in the area [15,16]. Research performed by the New Zewland team led by

professors Cleary and Witten [17] has shown the power of blending several model-orders using multi-symbol alphabets and arithmetic coding in the PPM (Prediction by Partial Matching) class of lossless compression algorithms.

III. Algorithm Overview

Statistical coding is based on performing predictions on symbol distribution and then coding the most probable symbols with fewer bits. Variable order Markov modelling exploits the fact that a prediction can be made with much more certainty by observing the symbols that preceded the current symbol. The symbols used for the prediction are called context while their number is called model order. The proposed compression system identifies three different processing stages. The first processing stage is context modelling followed by probability estimation and finally arithmetic coding. The first two stages correspond to the variable-order Markov Model while the arithmetic coding module obtains a compressed bit stream using the probability data provided by the model. Fig. 1 shows a detailed data flowchart of the different stages involved in the proposed data compression process that will be described over the following sections. Our novel hardware amenable compression algorithm has been named PPMH (Prediction by Partial Matching in Hardware).

A. Context modelling overview

The upper section of Fig. 1 shows the main steps associated with context modelling in the PPMH algorithm. During context modelling a finite number of symbols (model order) that preceded the current symbol and constitute its context are searched in a context tree built dynamically as more data is seen. The first symbol to be searched is the symbol that preceded the current symbol and corresponds to order 1. Order 0 is located at the root of the context tree and no searching is required.

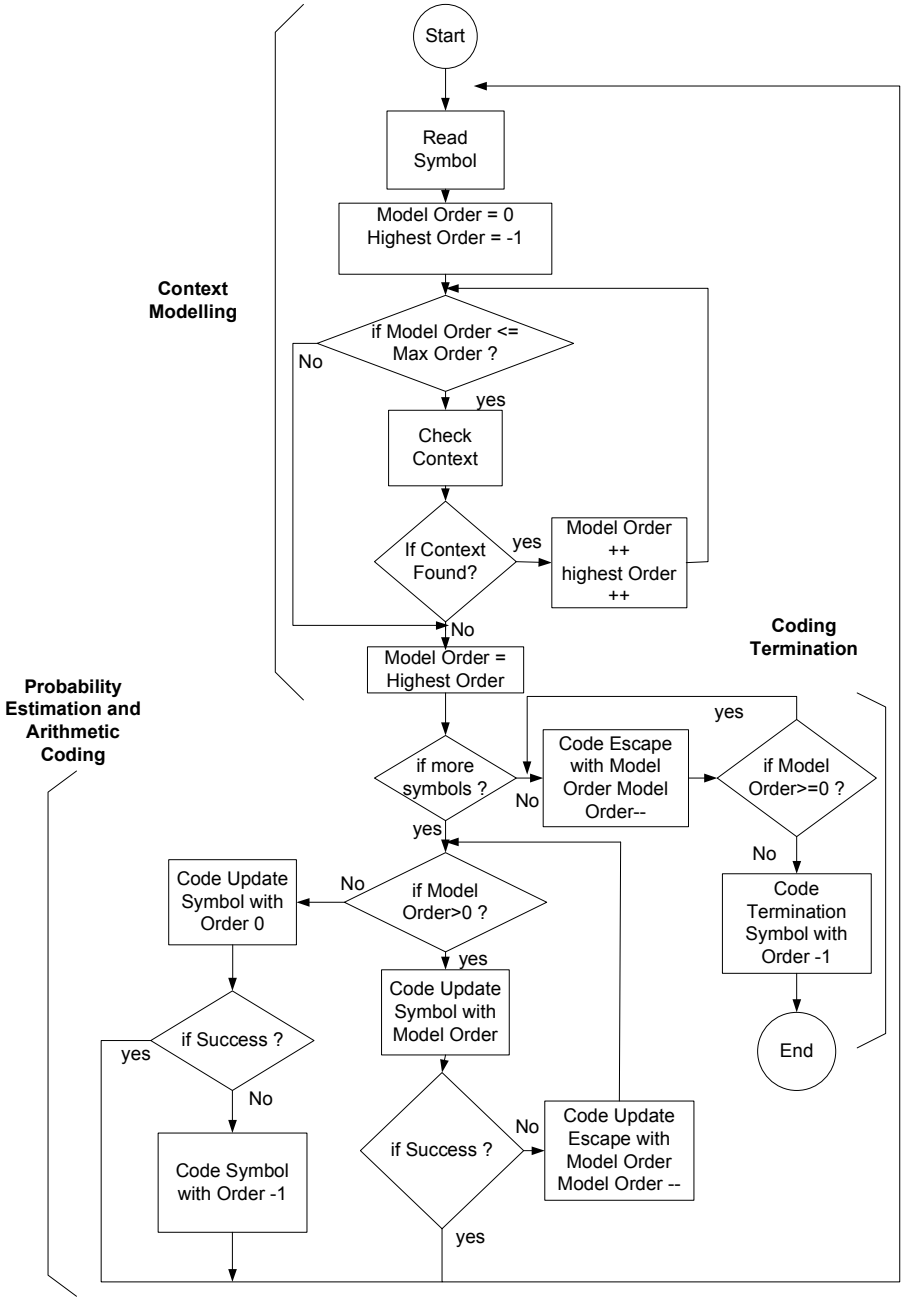

Fig 1. Detailed Flow Chart of Statistical Compression Algorithm

Three different outcomes are possible from context searching:

1. *Context symbol found.*
2. *Context symbol not found but free context entry found.*
3. *Context symbol not found and free context entry not found.*

Outcome 1 means that the search operation was successful and the process is repeated for the next higher order until the maximum model order is reached or outcomes 2 or 3 are reached.

Outcome 2 means that a new context symbol can be inserted effectively increasing the context tree size. Each context tree node contains a pointer to the context area that stores all the probability data plus any required control bits. To speed up the context modelling stage the maximum number of nodes in the context tree and the number of available context areas are not equal. There are 25 % more entries in the context tree than contexts areas physically present (memory allocated) in the algorithm. This means that outcome 2 could be reached but no context area could be available to be assigned to the context tree node. This event will also result in a condition equivalent to outcome 3. Outcome 3 will be reached when the search function is unable to reach outcome 1 or outcome 2 in a predetermined number of search cycles. The number of search cycles is limited to reduce the cycle cost of context modelling and avoid infinite search loops. Once context modelling of a particular symbol completes the next phase of probability estimation starts.

B. Probability estimation overview

Probability estimation extracts the context area indices from the contexts nodes maintained by the context modeller and uses them as pointers to the memory area holding the probability information. The probability estimator starts with the highest model order reached during context modelling and tries to obtain a valid prediction for the current symbol with that context. Success is achieved as long as the current symbol has a probability value larger than 0 in that particular context. Otherwise an escape event is coded and the algorithm tries to use the next lower order until model order 0 is reached. Model order 0 is also allowed to fail and generate an escape if the symbol has not been seen before. In this case order –1 is used where all the symbols get a probability larger than 0 and equal to 1/alphabet_size. The probabilities in order –1 are fixed and probability estimation can never fail. Immediately after probability estimation the algorithm increases the prediction value of the current symbol over all the contexts that have been used except for order -1. The objective of the algorithm is to use

always the highest possible order where probabilities tend to be more skewed and generate higher compression ratios. Lower orders typically accommodate more symbols and distribute the available range over all of them. The probability data obtained during this stage is finally forwarded to the arithmetic coder that uses it to generate an optimal compressed bit stream.

C. Arithmetic Coding

The final stage of the coding process is arithmetic coding. The arithmetic coder is based on a software algorithm known as the Z-coder and developed by AT&T labs [18] as a generalization of the Golomb/Rice coder for lossless coding of bilevel images. Golomb/Rice coding is used to code a run of *r* consecutive occurrences of a most probable symbol (MPS) followed by a single occurrence of a least probable symbol (LPS), using a parameter *m* to control how many MPS symbols fit in one bit of code and also how many bits of code are required to code a LPS symbol. The code has two components: the first component is *r/m 1's*, followed by a single 0, while the second component is *r mod m*, coded as an ordinary binary number with $log_2(m)$ bits. Although easy to implement, the limitation of Golomb codes is that the chosen parameter *m* is only good for a single probability distribution but a general compression system has to be able to deal with arbitrary sequences of events with different probabilities. The Z-coder aims to solve this limitation. Z-coding is the same as Golomb coding with the advantage that the parameter *m* can be changed for each symbol being coded. The extra complexity of the algorithm is small and more details can be found in the original paper [18]. Our work has focused on maintaining the simplicity of the Z-coding algorithm while increasing its suitability for hardware implementation. The resulting MZ-coder balances the complexity of coding the MPS and LPS symbols, simplifies the precision of the arithmetic and handles special hardware borrow conditions while maintaining coding efficiency and achieving high performance.

IV. CODING TERMINATION

A particular case that deserves special attention is how to indicate to the decoder that all the symbols have been decoded and the process must stop. A simple solution will be to insert a header in the compressed file that will indicate the total number of symbols that must be recovered after decompression and let the decoder use this value to control the reconstruction process. The problem is that this assumes that the coder either knows how many symbols are in the uncompressed data block before compressed data can be output or buffering of the whole compressed data block must take place before transmission can start. Both alternatives increase latency and hardware complexity. The solution we propose is the insertion of a special bit sequence at the end of the compressed bit stream so that it can be identified as a flag signalling the decoder to stop decoding bits. Adding a new symbol to the alphabet for this purpose increases complexity and wastes range (degrading compression) since this special symbol is used only once per data block. A more efficient solution adopted in this work is to use the features of the variable-order Markov model to generate a bit sequence that cannot take place during normal coding operations and can be easily detected by the decoder. An explanation of the developed mechanism follows. Each time order 0 is used by a particular symbol that symbol is chosen as the current *virtual* termination symbol. This means that the termination symbol is not constant but changes as different symbols make use of order 0. The probability of the termination symbol in order 0 is guaranteed to be larger than 0 since the update sequence increases it after the symbol is coded. Therefore, if the algorithm codes this termination symbol using a special sequence that escapes through all the orders including order 0 and finally codes it in order –1 an abnormal output is generated. The decoder can detect this abnormal output and interpret it as an *end of block* flag. The reason that this mechanism works is that if the termination symbol is being coded as a normal symbol (not to *indicated end of block*) it would have been stopped at least by order 0 and order –1 would

have never been reached. This mechanism is illustrated in Fig. 1 when the *if more symbols* decision takes the *no* branch.

V. Core Overview.

The compression core that implements the PPMH algorithm described in the previous also sections consists of 3 main modules: context modelling, probability estimation and arithmetic coding. The main characteristic that enables the efficient mapping of all the algorithm features into hardware gates is the decomposition of the coding of each symbol into a sequence of binary coding events. This means that the core operates logically at the symbol granularity level but physically at the bit level. Consequently, the hardware only needs to support operations at the bit level although compression is unaffected by maintaining a model operating with a multi-symbol alphabet. The first implementation of the PPMH compression algorithm uses a byte-based alphabet (256 different symbols) and has been named Byacom-1 [25]. The architectural, performance and complexity details of Byacom-1 are described over the following sections.

VI. Statistical Modelling Architecture

A. Context Modelling

Fig. 2 shows a simplified diagram of the context modeller. The context FIFO stores the symbols that preceded the current symbol and form its context. The FIFO width is 1 byte to match the width of the symbol while its length is configurable and depends on the maximum model order (Fig. 2 assumes model order 3 for illustration purposes). The hardware implementation of the context modeller is based on a hashing tree that enables fast search operations with low complexity. The tree is stored in standard SRAM memory and maintains its logical structure using a pointer mechanism.

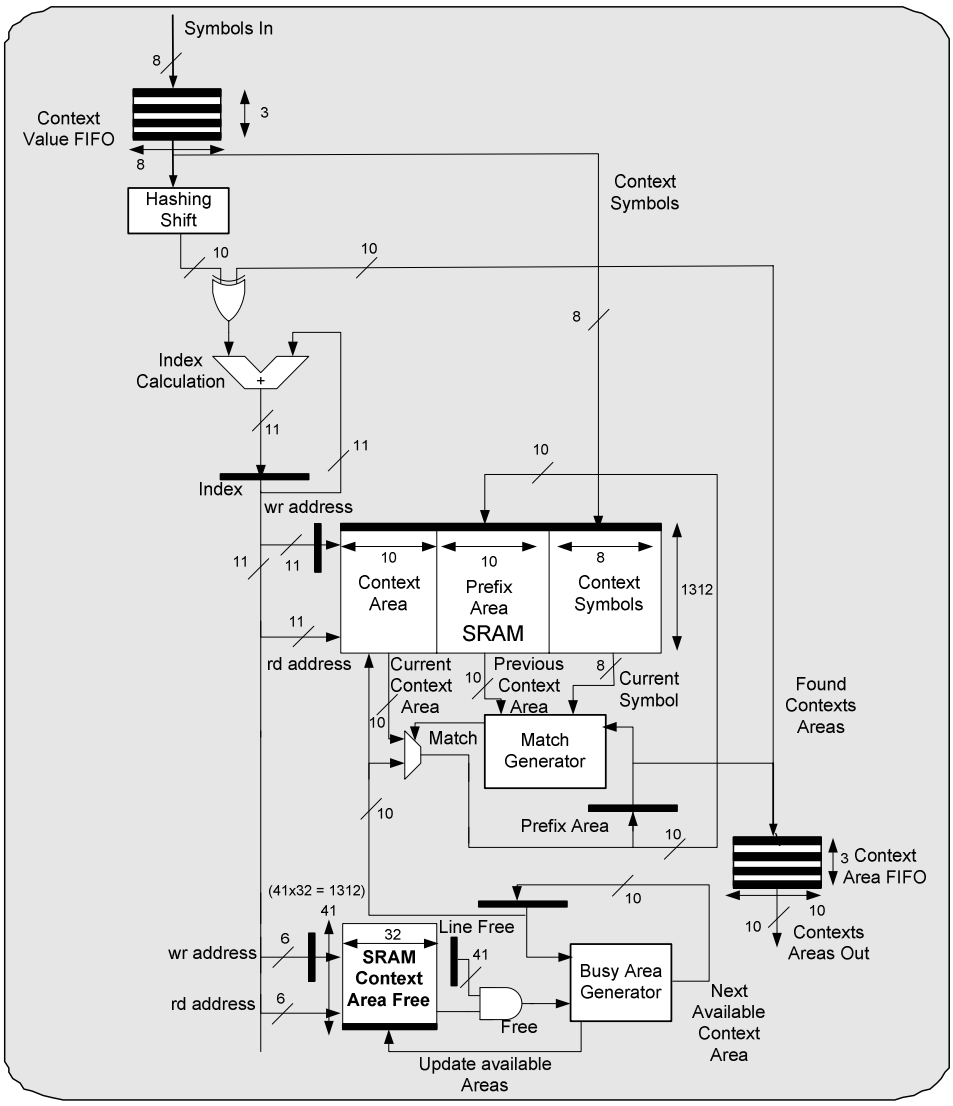

Fig 2. Context Modeller Architecture

The hashing shift and the XOR gate in Fig. 2 are used to generate an index to be used to address the SRAM memory that stores the context tree. The tree memory is divided into three sections. Section 1 stores the context area memory address where the probability data for that particular tree node can be found. The other two sections implement the pointer mechanism that maintains the logical structure of the tree. Section 2 stores the context area index of the tree node parent of the current node in the tree structure. Section 3 stores the context symbol stored at the current tree node. Table 1 illustrates the contents of the 3 sections of the tree

memory after the sequence *"aaacaaaccab"* has been processed. The corresponding logical tree structure is shown in Fig. 3.

Memory Address/index	Context Area	Prefix Area	Symbol	Order
0	1	0	a	1
1				
2				
3				
4	2	1	a	2
5				
6	4	0	c	1
7	3	2	a	3
8				
9	6	2	c	3
10	5	1	c	2
11	10	5	c	3
12				
13	7	3	c	4
14	8	4	a	2
15	9	4	c	2
16				

Table 1. Context Tree Memory Organization

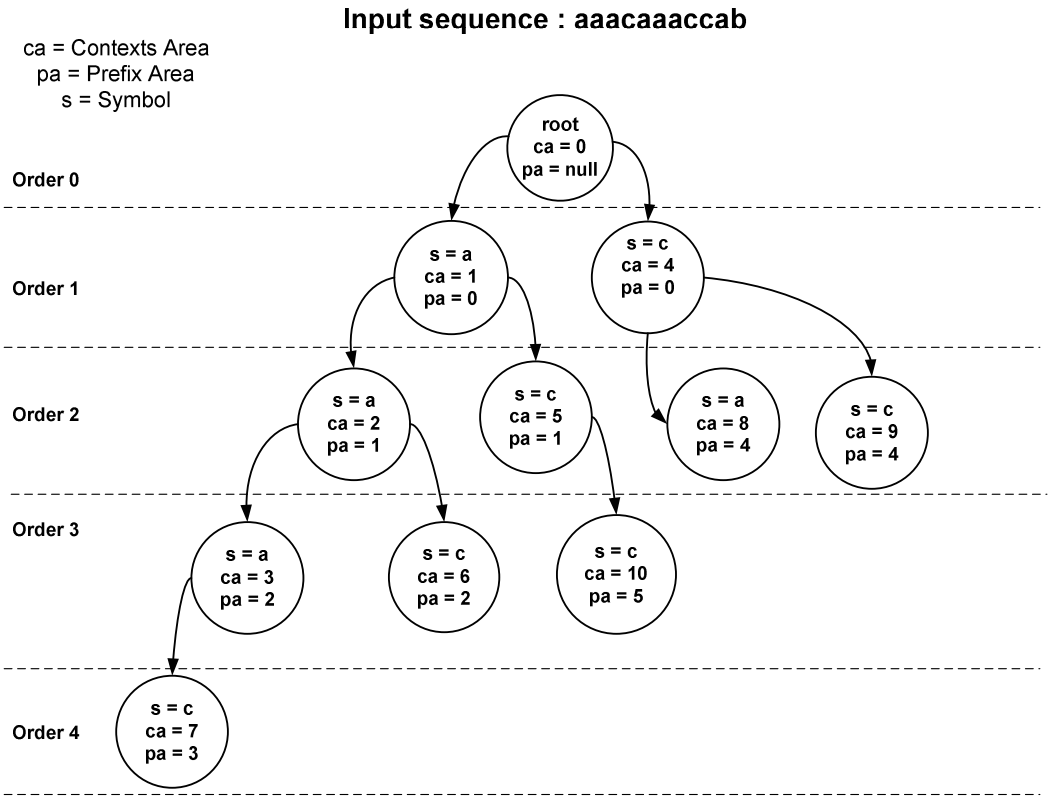

Fig 3. Context Tree Logical Structure

A match is valid when the context symbol and the prefix area are equal to the current symbol and the context area of the previous found context symbol. Successful matches result in the output context area FIFOs being populated with context area information and the maximum active context model order being incremented. An unsuccessful match results in the process being repeated with a new index obtained by adding a small increment to the previous index. The number of cycles the process repeats after an unsuccessful matched is limited to 10. This number was chosen after extensive simulation. The count is reset at the start of a new search operation. New search operations are generated with different context symbols until the maximum model order is reached. However, if the maximum count of 10 is reached or a free location is found the process is immediately stopped and the next hardware stage (probability estimator) is activated. This means that the search for order $m + 1$ does not take place if order m is not active. Finding a free location is equivalent to reaching a leaf in the tree depicted in Fig. 3. The leaf will then be extended with a new child as long as model order is lower than the maximum order and there are context areas available. The *SRAM area free* memory and the *busy area* generator shown in Fig. 2 enable a single-cycle reset state without having to reset the table memory with a multi-cycle table walk operation. A table walk would have had a very negative effect on throughput when dealing with small blocks since the number of cycles needed to reset the table could typically be larger than the number of cycles needed to compress the block. A single valid register, named *line free* in Fig. 2, is reset after processing each block and this automatically invalidates all the locations in the table memory. This register has a similar function to the register holding the valid bits in a direct-mapped cache. Each of the register bits is shared by several table locations and in order to distinguish which context tree nodes are busy and which context tree nodes are free the *area free* memory contains 1 bit per context tree node signalling a free or busy node. If the valid register bit is set to zero all the tree nodes associated with that valid register bit are considered invalid. The

found context areas are stored in two equivalent buffers. When the first buffer is being filled with context areas by the context modeller, the second buffer is being emptied by the probability estimator. Once both stages have completed their operation the buffers functionality is reversed and the process restarted. This double buffering mechanism increases the throughput of the system avoiding idle stages.

B. Probability Estimator

The probability estimator uses a balanced binary tree with 256 leafs corresponding to each of the symbols in the alphabet. The context area obtained from the context modeller identifies a memory area where the probability data of the symbols seen in that context is stored. An additional symbol is the escape symbol used to blend different model orders when no valid prediction is possible because the symbol is new in the current context. Fig. 4 illustrates the binary tree evolution for an alphabet of only 4 symbols plus the escape. Initially, all the range is assigned to the escape but as new symbols are received the values stored in each of the tree nodes change to reflect the new distribution. The depth of the tree in this example is $(log_2(alphabet\_size) + 1) = 3$. A full alphabet of 256 symbols will have a tree depth of 9. The important point to notice is that to fully code a symbol using this binary tree is enough with coding the binary decisions (left or right) taken place at each level of the tree when the tree is transversed from root to leaf. This procedure means that after 9 binary decisions a symbol is fully coded. There are two main advantages obtained from using this binary tree. Firstly, the arithmetic coding stage does not need to be based on a complex multi-alphabet arithmetic coder but a simple and fast binary arithmetic coder would suffice. Secondly, the maintenance of the frequency counts is achieved with a single update operation per node visited [19]. Fig. 4 shows how the frequency counts stored in each node are updated to reflect the new probability distribution each time a new symbol is coded. The associated probability values for each symbol are also shown in Fig. 4.

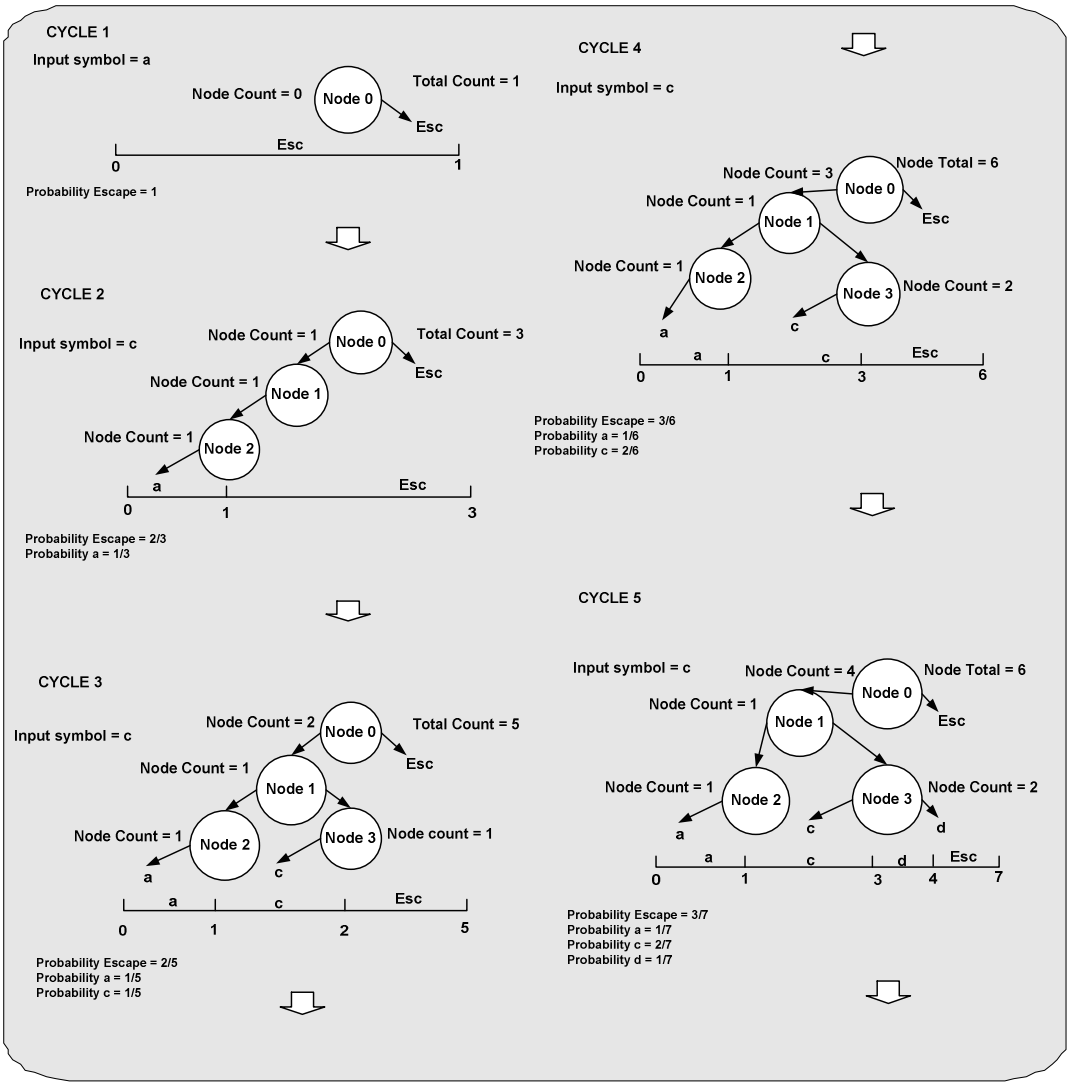

Fig 4. Probability Estimator Tree Evolution

Probability and frequency counts are related with equation (1):

$$P(symbol_i) = \frac{F(symbol_i)}{\sum_{j=0}^{255} F(symbol_j)} \qquad (1)$$

Where $P(symbol_i)$ and $F(symbol_i)$ are the probability and frequency values respectively. Traditionally, the frequency counts in a statistical algorithm have to be maintained in cumulative form [17] so that each symbol has a corresponding fully-defined range that identifies it without any ambiguity. The problem with maintaining frequency counts in cumulative form is that updating counts at the bottom of the range affects the counts for all the symbols in the alphabet and a multi-cycle update operation is required. Software

implementations typically locate more common symbols at the top of the range so the update operation affects a few frequency values in most of the cases. A direct hardware implementation of this technique would generate a variable cycle count with a worst case of 256 cycles to maintain the frequency data. The binary tree has the property that one update operation per tree level is sufficient to maintain the range values in the correct range [19]. Therefore, any possible symbol needs a constant cycle count of 10 to transverse the tree generating coding events and updating all the frequency data simultaneously. The escape symbol exists at the top of the tree and consequently only 2 cycles are needed to coded it. The binary tree architecture enables the high compression ratios possible with multi-symbol alphabets (a better match of data granularity) and simultaneously achieves low hardware complexity which also helps to achieve a higher clock frequency. The binary tree is projected first right to obtain 9 processing elements and then down to reduce it to a single processing element. This single processing element walks through the tree from root to leaf forwarding two frequency count values and a binary decision to the binary arithmetic coder. The two frequency count values (*cum0* and *cum1)* divide the range into a left probability and a right probability. The binary arithmetic coder uses this information to perform a series of arithmetic operations that modify its internal state and produce a compressed bit stream. The whole process is numerically efficient and using 9 coding events instead of 1 coding event per input symbol produces no significant redundancy. Fig. 5 illustrates the architecture of the processing element that implements the binary tree node assuming a context population of 1024. The *total value* memory contains the total frequency count for a particular context while the *probability storage* memory contains all the probability data associated with each of the nodes in the tree. The low frequency value for each binary decision is always 0. The frequency value stored in each tree node defines the *middle value* (cum0) while the *top value* (cum1) is obtained from the previous tree level.

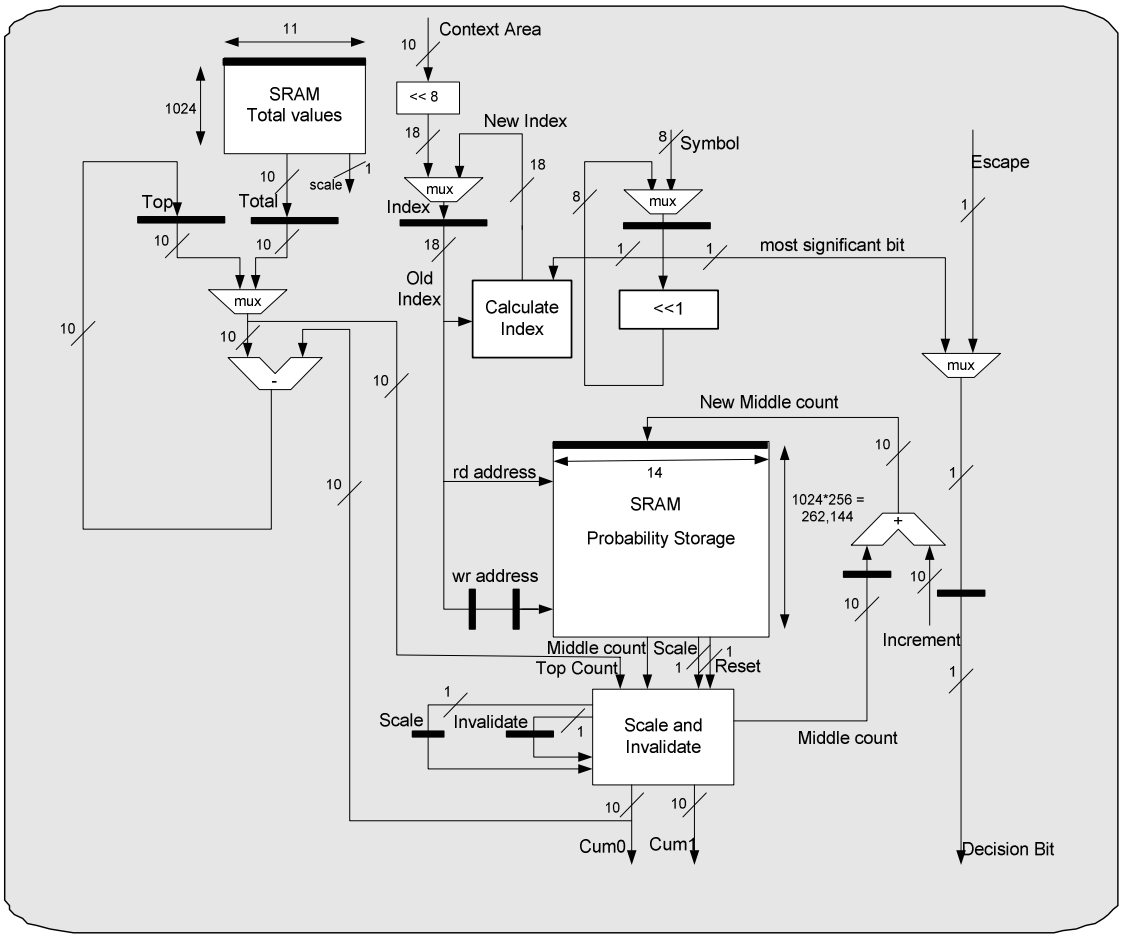

Fig 5. Probability Estimator Tree Node Architecture

The control logic generates a new *top* for the next lower tree level with the current *middle value* if the binary decision is left or the current *top value* minus the *middle value* if the decision is right. This new top is stored in the *top* register to be used in the next cycle. The top value is the total count obtained from the *total value* memory only when the current tree node is the root node. Model adaptation takes place every cycle to increase the probability values of the symbols being propagated down the tree. The increment rate depends on the active order. In general, higher order models increment faster since they contain fewer different symbols. This variable increment rate improves compression.

1) Scaling and Resetting

The frequency counts stored in the probability estimator memories need to be initialised to a known value before each data block is processed. This is similar to the case of the context modeller memories discussed in previous sections. In principle, this could mean that all the frequency memory locations need to be accessed in order to reset them to a known value. This, however, will degrade performance considerably specially when compressing small blocks of a few hundred bytes. As there are 256 nodes per context tree and a typical implementation could contain 1024 contexts a total of 262,144 locations would need to be reset. To enable single cycle resetting we use the busy/free bits from the context modeller stage and propagate them down the tree simultaneously to the coding of the symbol itself. This means that each tree node needs to store not only its frequency count value but also two additional control bits indicating if the left sub-tree or the right sub-tree have a reset operation pending. A similar strategy can be used to scale the probability data. Scaling is used when the total count value exceeds a maximum count and can potentially overflow the allocated storage space. In the Byacom-1 implementation 10 bits are used to store each of the tree frequency counts so the count cannot exceed the value of 1024 at any time. It is possible to simply freeze model adaptation once the maximum count value is reached but this deteriorates compression efficiency. Scaling could be achieved simply by resetting the model to its initial state but this will affect compression since the context will loose all the history information after the scale operation. A better solution is to scale by dividing all the counts by 2 and this can be readily accomplished in hardware by a simple shift operation. The problem is that if a scale event is required the model must be stopped and a state machine activated to visit and scale all the nodes in the context tree affected. This solution degrades throughput and decreases the performance of the core. The preferred solution uses the same approach as resetting and adds two control bits to each of the tree nodes to indicate if a scale operation is

pending on the left or right sub-trees. The scale operation propagates down and updates the frequency count value present in the tree node before any other operation is performed. These two solutions increase the storage requirements from 10 bits to 14 bits per node location but they guarantee high performance and limit worst-case latency to a small value independent of block size and scale frequency.

2) Coding speculation

Due to the nature of the binary tree, coding events take place speculatively, prior to success of the coding operation. A coding operation will fail when the symbol probability in the active context is 0, in which case an escape symbol must be coded and the next lower order used. The escape symbol always has a probability larger than 0 and the final order (order –1) always assigns a probability value larger than 0 to all the possible input symbols, so that the coding operation can never fail. The decomposition of the coding operation into a sequence of 9 binary coding events means that a few of these binary coding operations could be completed successfully before one of them fails because one of the sub-tree paths that the symbol needs to follow (left or right) has a probability of 0. The arithmetic coder would have coded a sequence of binary decisions that needs to be undone before the escape symbol can be coded and the next lower order activated. In order to achieve this, all the register state in the arithmetic coder has an equivalent shadow copy that only gets updated once the symbol has completed the whole coding sequence successfully. If the coding sequence fails the values stored in the shadow registers are used to update the visible registers and the state of the arithmetic coder recovers to a known correct state. Finally, it is possible that due to coding speculation a few bits have been output by the arithmetic coder to the output buffers that need to be removed from the coded bit stream. The output buffer uses a double counting mechanism that keeps track of this situation so a similar update/commit mechanism can be used to remove coded bits from the output buffer. This process decreases latency and

buffering requirements compared to waiting for a probability estimation operation to be successful before committing the frequency values to the arithmetic coder.

VII. Modelling Performance Analysis

This section analyses the effects of maximum model order and context area population. These are 2 key configuration parameters in the Byacom-1 core and they have a major effect on compression, complexity and throughput so a good understanding on how they relate to each other will help in the selection of the right parameters for a particular implementation. . The data set selected to perform this experiments is the standard Canterbury Corpus [20]. The Y axis shows compression ratio as a ratio of output to input bits so the lower the figure the better the quality of results. The block size in the X axis defines the amount of data in bytes that is compressed independently of other data present in the file or channel. Block-based compression is useful in communication channels where packets with a few hundred bytes are compressed independently to avoid propagating errors between packets. Shorter blocks tend to compress worse since less data is available to build an accurate model so it is important to evaluate the effects of block size in compression ratios. Figs. 6 to 9 show the compression gain achieved over model order 0 by model orders 1, 2, 3 and 4 varying the context population. The smallest configuration uses a total of 64 contexts and the largest uses 8192 contexts. Model order 2 is the best option if fewer than 1024 contexts are available while model order 3 is preferred if 1024 contexts or more are available. The performance of context configurations 1024, 4096 and 8192 is very similar when the block size is in the range of 4 Kbytes or smaller and the larger configurations only have a positive impact when larger blocks in the order of hundred of Kilobytes are compressed. Model order 3 is the best performer with up to 90% better compression over model order 0 for file-based compression.

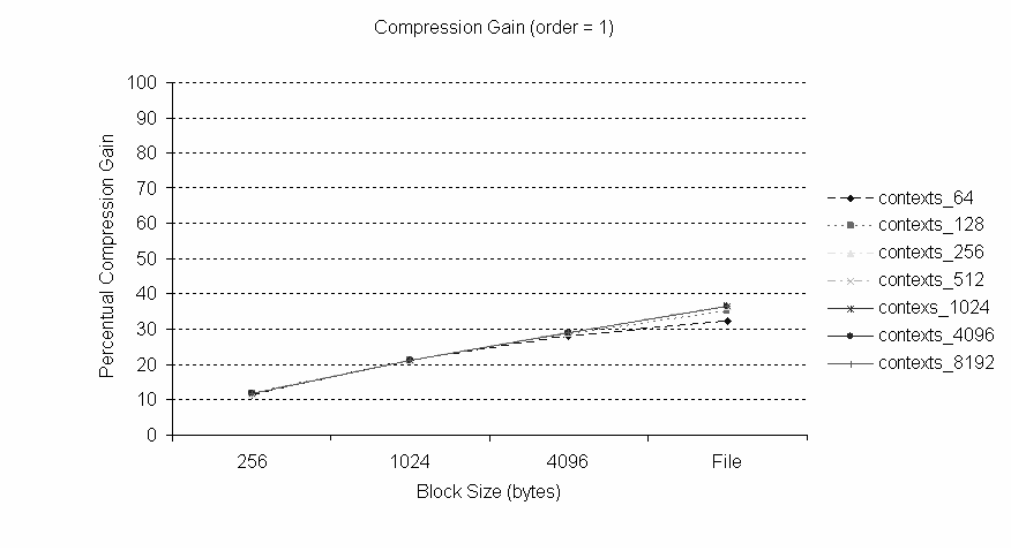

Fig 6. Model Order 1 Compression Analysis

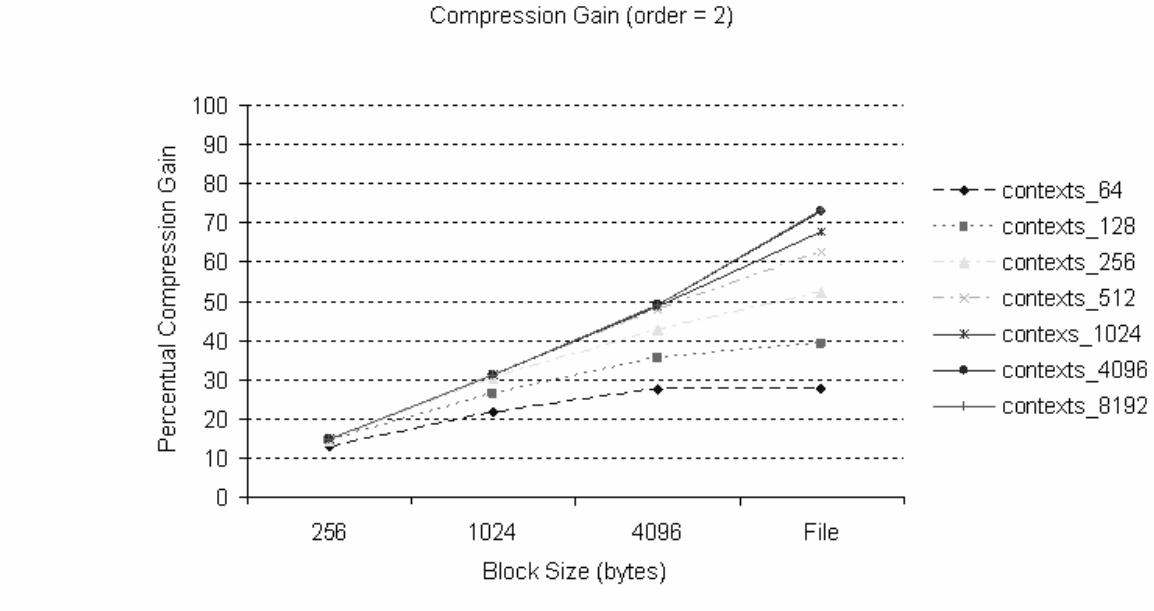

Fig 7. Model Order 2 Compression Analysis

Fig 8. Model Order 3 Compression Analysis

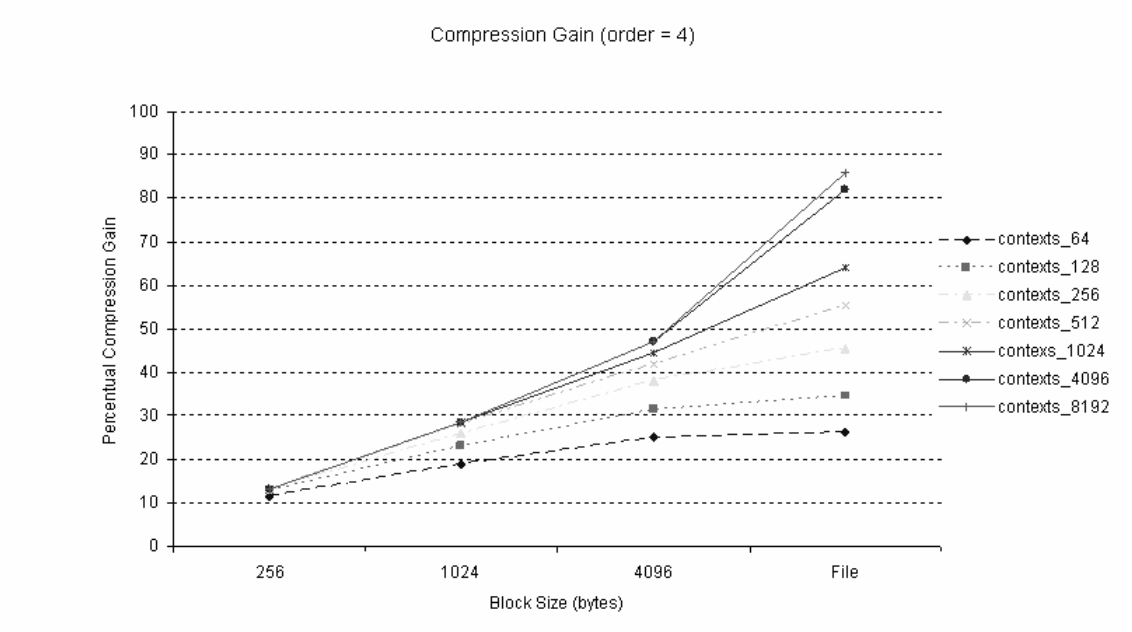

Fig 9. Model Order 4 Compression Analysis

Model order 4 does not improve compression for this particular data mix. We have observed a compression gain using order 4 when files larger than 1 Megabyte are compressed as a single block. The average file size in the Canterbury corpus is around 256 Kbytes and this could explain why larger model orders do not improve compression. The best compression ratio achieved for the Canterbury corpus is 0.285 (model order 3, file-based compression and context population of 8192) that means that 100 Megabytes of original data would be compressed to 28.5 Megabytes.

VIII. ARITHMETIC CODING

Fig. 10 shows the internal organization of the multiplication-free arithmetic coding module. A total of 6 pipeline stages are identified to improve the clock ratio of the design. The lack of a renormalization loop in the MZ algorithm means that one decision bit is processed per clock cycle. The functionality of each of the pipeline stages is briefly described over the following sections due to space limitations. More details on the AC engine can be obtained in [21].

1)LPS table 512x7. The Least-Probable-Symbol table transforms the 2 frequency count values obtained from the probability estimator module into a single truncated probability that

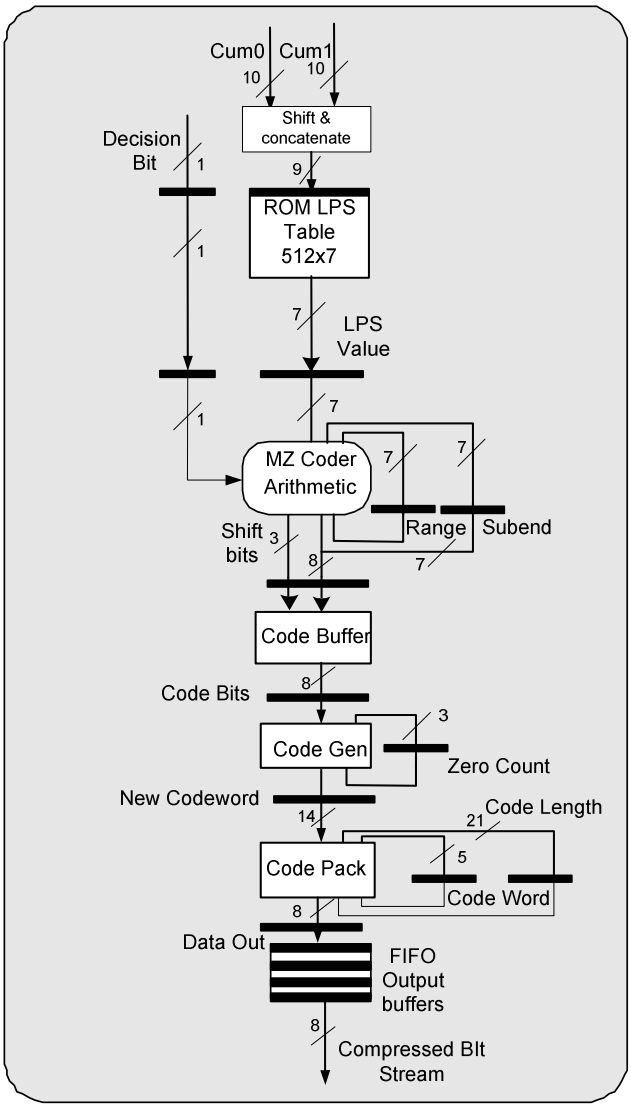

Fig 10. Arithmetic Coder Architecture

approximates the results of dividing both frequency values. This table can be implemented using a standard ROM memory but since many of its values are set to 0 and others are repeated across table entries logic synthesis of the table results in a combinatorial logic block with a small gate count more efficient that a full ROM memory with 512x7 bits.

2) MZ coder arithmetic. The MZ arithmetic uses the *range* and *subend* coder state values and the LPS value to generate the codewords. The renormalization is done in parallel for both *range* and *subend* and in the same pipeline cycle as the rest of the MZ arithmetic.

3) Code buffer. The code buffer stage is required to control possible borrow bits originating in the previous stage that could affect the value of the bits contained in the code buffer.

4) Code generator. The code generator takes the bits produced by the code buffer ranging from 0 up to 7 and the zero run count to build a code of up to 14 bits. The zero run register counts the number of consecutive 0 bits in the input. These bits are the equivalent of the *bits to follow* variable used by software arithmetic coders.

5) Code packer. The variable number of bits produced by the code generator are finally pipelined to the code packer. The functionality of the code packer is to pack the variable length codewords into fixed-length 8-bit codewords ready to be output.

IX. Performance Comparison

This section analyses the performance of the core in terms of compression ratio and throughput and compares it with other state of the art universal data compression algorithms implemented in both hardware and software.

A. *Compression.* The core can be configured at compile time with different values of context population trading complexity for compression efficiency. Approximately, 3.6 Kbits of memory are required per context including the memory used in the context modeller and the probability estimator. The maximum model order is a parameter that can be configured at run-time to a value ranging from order 0 (empty context) to order 4 (4 symbols are used to perform a prediction). This parameter basically defines the maximum depth that the context modeller tree is allowed to reach. The results shown in Figs. 11 and 12 are based on a configuration with 8192 contexts and model order 3 that were determined to be the best values for the PPMH algorithm as discussed in section 7. Fig. 11 compares the compression efficiency of the PPMH algorithm with the well known software-based algorithms. We have selected the popular open source Lempel-Ziv implementation known as GZIP and equivalent

to other commercial algorithms such as PKZIP and WinZIP as a fast and efficient dictionary-based algorithm.

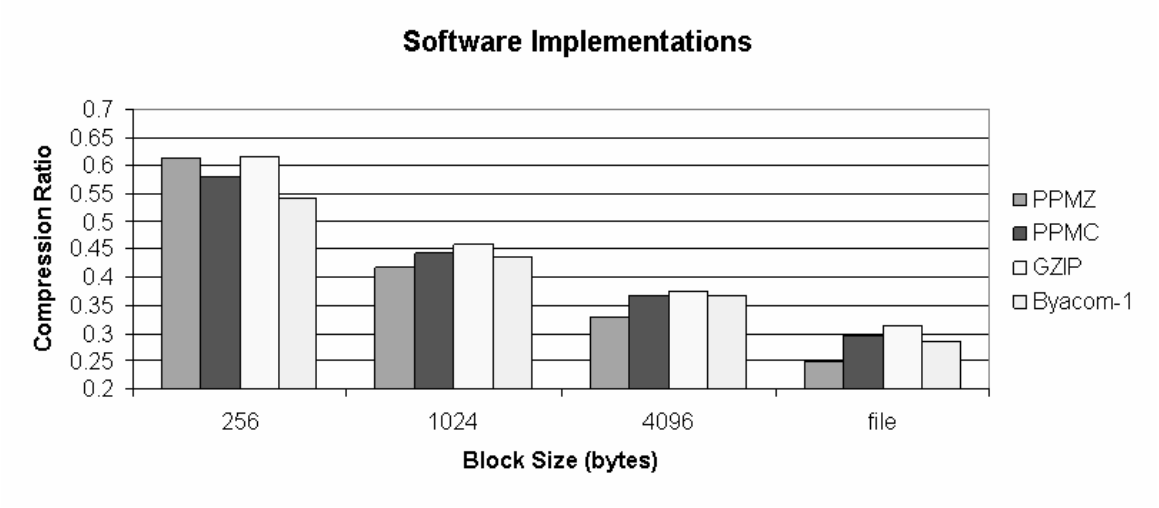

Fig 11. Software Compression Performance Comparison

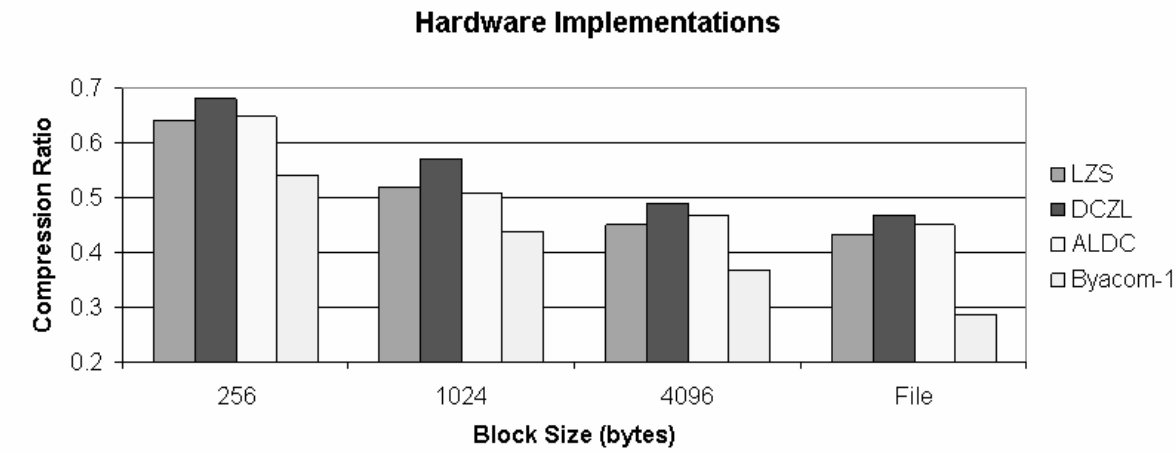

Fig 12. Hardware Compression Performance Comparison

Two statistical compressors have also been selected for this work: the PPMZ [11] and PPMC [17] algorithms. The PPMZ algorithm is a very sophisticated implementation that uses local order estimation to select the best possible model order from a maximum of 8. PPMC uses a similar modelling strategy to PPMH although there are major differences on model implementation and in the arithmetic coding algorithm itself (range coder [22] for PPMC and MZ coder for PPMH). Nevertheless, the performance of PPMC and PPMH is very similar

with an slightly advantage for the PPMH algorithm mainly due to a more sophisticated arithmetic coding algorithm and some implementation details such as the adjustable increment mechanism so model orders adapt at different speeds. PPMZ does not perform particularly well for small blocks of fewer than 1024 bytes since its complex data structures need more data to operate effectively. Once blocks larger than 1024 bytes are used PPMZ outperforms the rest of the algorithms. PPMH has the best compression ratios for small blocks of around 256 bytes and is the second best performer after PPMZ for the rest of the block sizes. Fig. 12 compares PPMH with other hardware-based lossless compression algorithms. The three algorithms selected are popular dictionary-based algorithms used in commercial applications such as routers and tape drives. LZS and ALDC are both based on the LZ-77 [8] algorithm while DCLZ is based on the LZ-78 [9] algorithm. PPMH compresses better than these algorithms for all block sizes with the difference being more noticeable for large block sizes where more data is available to improve the accuracy of the predictions done by PPMH.

B. *Throughput.* One of the main features of PPMH is the decomposition of the prediction and coding of a symbol (byte) into a sequence of binary decisions that take place as the symbol moves a long a binary tree with all the symbols plus the escape symbol occupying the leafs of such tree. This architecture enables an efficient circuit with a reduced gate count and consequently high clock frequencies. The disadvantage is that throughput is limited in the best scenario (no overheads) to 1 bit per clock cycle unless multiple cores are used to process multiple symbols in parallel. This kind of parallelism typically involves duplicating the data path while the storage area and consequently the probability distributions are shared among each of the functional units. This paper focuses on the single data path implementation leaving multiprocessing capable architectures for future research. Fig. 13 shows the

throughput of the Byacom-1 implementation of the PPMH algorithm in terms of clock cycles per bit for different block sizes.

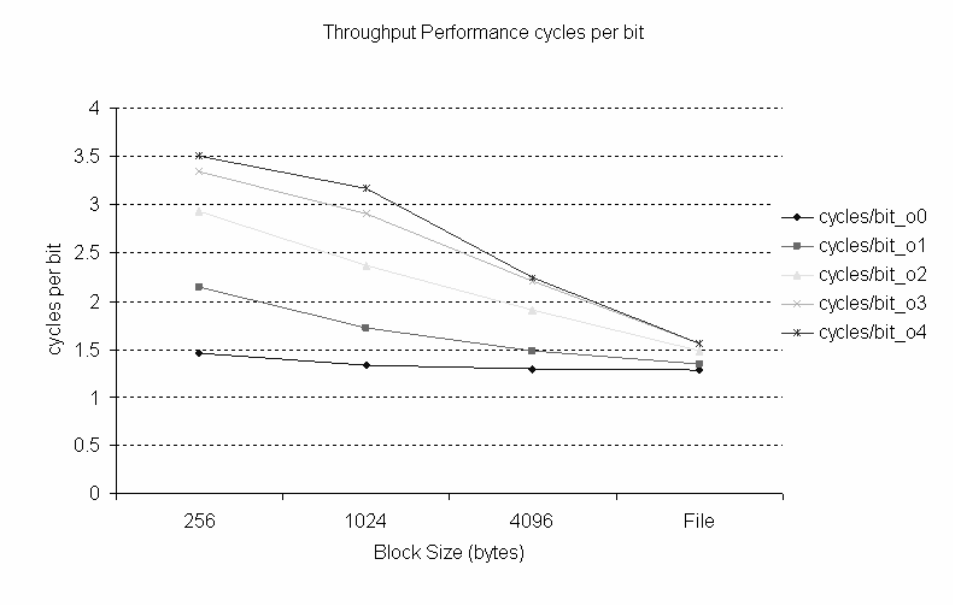

Fig 13. Byacom-1 Throughput Analysis

Throughput improves with block size since the escaping mechanism is used less often as the high orders start making a higher proportion of valid predictions. The small blocks do not provide enough data for the higher order models to make a valid prediction so multiple tree transversing is needed for each byte of input data. Effects on compression efficiency are limited by the fact that a context just created will automatically assign all the range to the escape symbol producing no redundant output bits when coding a symbol. On the other hand throughput will be affected since the minimal condition of 10 cycles per bit (9 memory accesses to synchronous RAM are needed) will increase to at least 22 cycles (2 cycles to code the escape plus extra 10 cycles to code the symbol in the next lower order). If the symbol will again fail to be coded more cycles will be used. An implementation with 1024 contexts and model order 3 delivers an average throughput of 1.4 cycles per bit compressing the whole file as a single block. We have analysed the performance that the software algorithms PPMZ, PPMC and GZIP could achieved on a typical single-issue in-order embedded processor. This

work is based on the cycle accurate SimpleScalar [23] processor simulation toolset that models a MIPS-like microprocessor. We have configured the simulator with two 16-Kbyte 4-way set-associative level-1 cache for instructions and data and no level 2 caches. After compiling the software using the SimpleScalar GCC compiler with optimisations enabled we have written a script to automatically process all the data present in the Canterbury corpus and collect results including dynamic instruction count, cycle count and memory allocated. Fig. 14 shows the results obtained for the three software algorithms using a logarithmic scale in the Y axis to measure average counts.

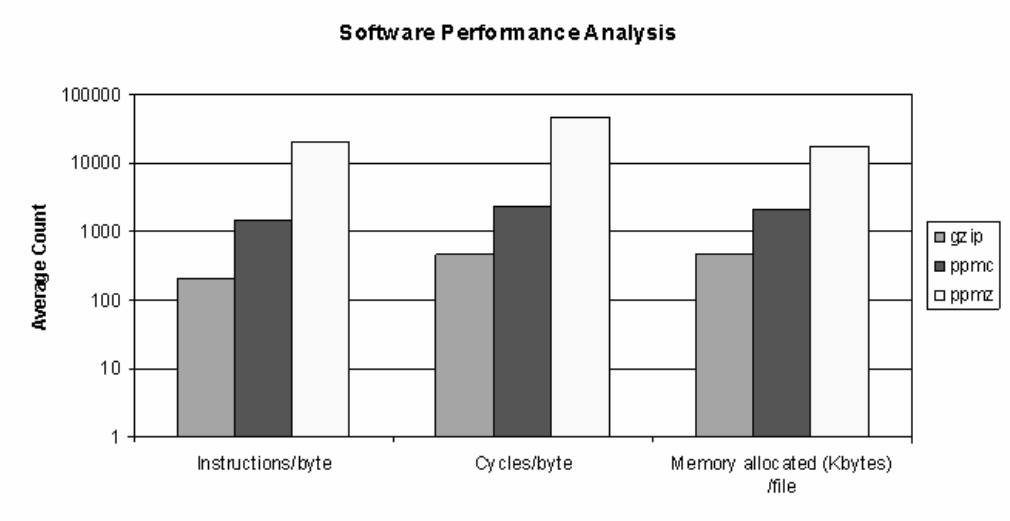

Fig 14. Software Throughput Analysis

The logarithmic scale enables the highly different counts to be represented in a single graph. The fastest algorithm is the GZIP with an average cycle count of 450 cycles per byte. PPMC increases this value to 2,600 cycles per byte. The computational complexity of PPMZ with more than 48,000 cycles per byte is overwhelming and clearly out of the reach in an embedded application. It is also apparent that better compression ratios imply an exponential increase on computational complexity with relatively small gains in terms of compression. Similar conclusions can be reached in the case of memory requirements which range between 466 Kbytes for the GZIP algorithm to 17,580 Kbytes for the PPMZ algorithm. It is also

important to notice that these results have been obtained compressing each of the files in the Canterbury Corpus as a whole and then averaging by the number of files. They correspond to the right point of the X axis of Fig. 13. If the files were blocked as in the rest of the X axis points of Fig. 13 the cycle count per compressed byte will increase due to the overheads of algorithm initialisation that would be needed once for each of the individual blocks. Comparing the value of 11.2 cycles per byte (1.4 cycles per bit) in the Byacom-1 hardware implementation the complexity reduction is considerable specially when compared with the sophisticated PPMC and PPMZ statistical coders. Although, it is expected that more sophisticated embedded processors exploiting microarchitectural features such as out-of-order execution and superscalar execution will deliver a performance increase, compression is fundamentally a sequential task (bytes are compressed sequentially so each byte can use the previous bytes as history information) where advanced microprocessor features such as SIMD computing or multithreading have a limited, if any, role to play. The throughput of the dictionary-based hardware implementations of Fig. 12 is clearly higher since they processed 1 byte per cycle with the help of CAM dictionaries that enable single cycle search operations. They deliver however significantly inferior compression performance and also CAM dictionaries tend to consume a lot of energy which is a disadvantage for battery-powered wireless devices that constitute the primary focus of this work.

X. IMPLEMENTATION

The compression core that includes the context modelling, probability estimator and arithmetic coder modules has been implemented in an state-of-the-art Altera 0.13 μm Stratix FPGA technology. Stratix devices are particularly well suited for this memory intensive algorithm thanks to their Trimatrix [24] memory architecture where embedded memory is available in three different sizes (512 bits, 4 Kbits and 512 Kbits). The larger 512 Kbits memory blocks are very useful to store the probability data associated with each of the

contexts in the PPMH model. The core has been configured with 64, 128, 256, 512 and 1024 contexts. The complexity in terms of FPGA logic cells and memory requirements are given in Figs. 15 and 16 respectively. The X axis shows that the main components of the coder data path including the input and output buffers and register file used to write commands. The component that consumes most logic resources is the arithmetic coding engine with 1,400 cells. The overall figure is around 3,000 logic cells for the compression data path. This figure increases slightly with the increase in memory cells but remains overall largely invariant. Most of the complexity concentrates in the memory blocks needed to store the data associated with the hashing tree in the context modeller and the probability data in the probability estimator. Fig. 16 uses a logarithmic scale along the Y axis to measure the memory in bits used by each of the configurations. The requirements vary from 237,760 bits for the 64 context configuration to 3,723,424 bits for the 1024 context configuration increasing linearly with the context count.

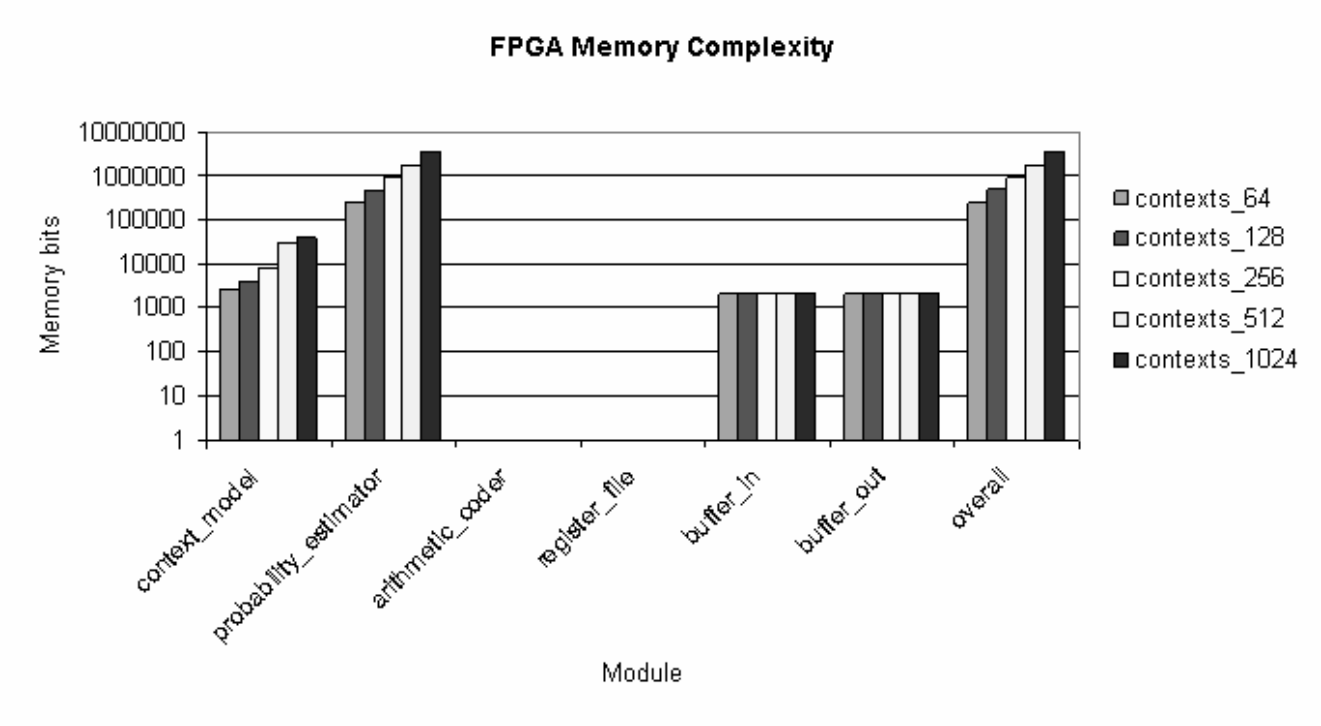

Fig 15. Byacom-1 Memory Requirements

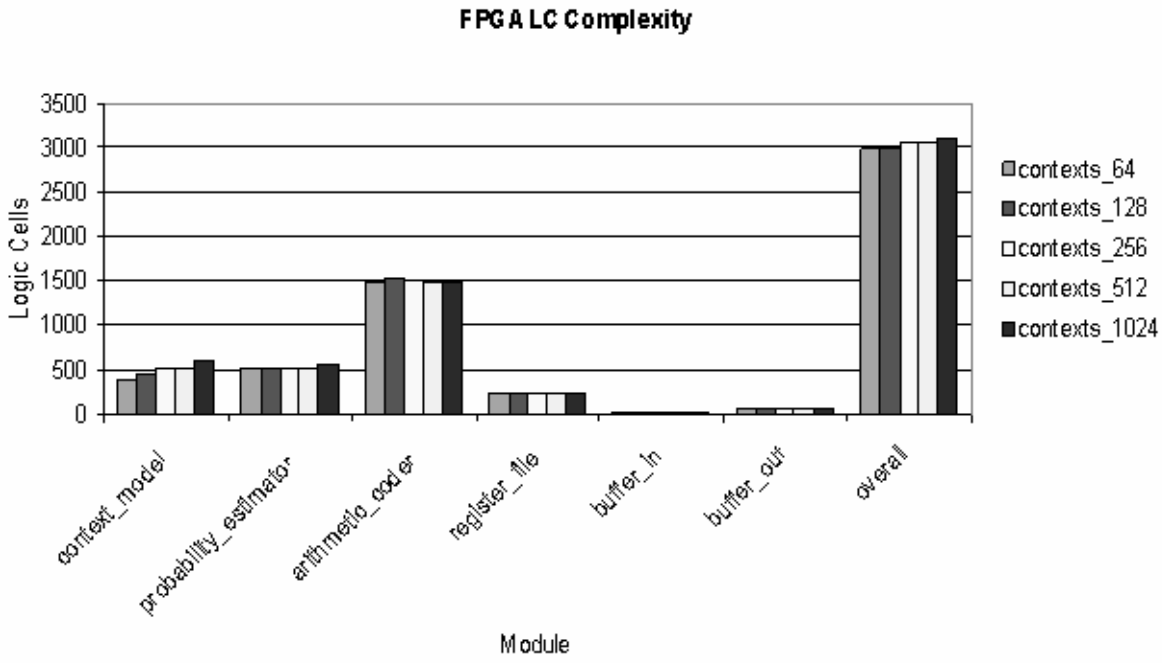

Fig 16. Byacom-1 Logic Cell Requirements

The device used in this work is the EP1S80F1020C5 with total memory resources of 7 Mbits so the larger configuration uses around 50% of the memory available. The clock rate for all the configurations remains almost constant at 70 MHz (69.7 MHz for the largest configuration to 71 MHz for the smallest configuration). This clock rate is remarkably constant despite the increase in used memory blocks that could complicate routing and degrade performance. We have observed that the Trimatrix memory is an enabling microarchitectural feature for this memory intensive core. The large memory blocks enable the packing of the logic and memory in a reduced area. Experiments conducted using another state-of-the-art FPGA family that only includes a single type of embedded memory block of 18 Kbits showed a significant performance degradation as context population increases. The reason is that the memory blocks are spread over the silicon die and long wires are needed to route logic and memory reducing the achievable clock frequency.

XI. Conclusions

This paper has presented the hardware-amenable PPMH statistical algorithm for lossless compression of general data. The hardware architecture and implementation of the compression data path have also been developed. The decompression data path is currently under development and will be presented in future work. The IP core has been targeted to a high-density FPGA family where a clock ratio of 70 MHz has been achieved resulting in a throughput of 50 Mbits/second. An analysis of the compression performance has shown to be competitive with the best software-based statistical algorithms and superior to current dictionary-based methods. Throughput is 2 orders of magnitude higher than software-based statistical algorithms running on a typical embedded microprocessor. The core is implemented using a low logic cell count but it is memory intensive with several megabits of memory needed for optimal performance on large block sizes. This technology could be very beneficial in a reconfigurable system where memory can be shared between different processing functions. The main memory blocks used by the probability estimator are standard single-port SRAM memories and modern SoCs can routinely accommodate several megabits of them. The selected FPGA technology offers enough embedded memory for a reasonably large 1024 context implementation. Future FPGAs will enable larger context population configurations. The IP has been designed parametrically so configurations can be generated at compile-time by changing some constants in the RTL description. Future work includes research into multiprocessing variants and the extension of the parametric model to be able to target efficiently 2-dimensional data such as that presented in medical or space imagery. Future work will also look into adding preprocessing stages based on predictive coding for image data. We will also like to investigate the configuration of different alphabet sizes extending the current byte-based alphabets to multiple-bit alphabets for lossless compression

of scientific data obtained from high-resolution analogue-to-digital converters. Executables and information can be obtained at www.byacom.co.uk.

1. G. Lawton, 'New Technologies Place Video in Your Hand', IEEE Computer, Vol. 34, No. 4, pp. 14-17, 2001.
2. S.Vassiliadis, G. Kuzmanov, S. Wong, 'MPEG-4 and the New Multimedia Architectural Challenges', Proc. 15th International Conference on Systems for Automation of Engieering and Research (SAER-2001), pp. 24-31, Bulgaria, 2001.
3. R. V. Cox, P. Kroon, 'Low Bit Rate Speech Coders for Multimedia Communications', IEEE Communications Magazine, Vol. 34, No. 12, pp. 34-41, 1996.
4. M. Nelson, J. Gailly, 'The Data Compression Book', 2nd edition, M&T Books, New York, NY 1995.
5. J.M.Cheng, L.M.Duyanovich, 'Fast and Highly Reliable IBMLZ1 Compression Chip and Algorithm for Storage',Hot Chips VII Symposium, August 14-15, pp. 155-165, 1995.
6. 'AHA3521 40 Mbytes/s ALDC Data Compression Coprocessor IC', Product Brief, Advanced Hardware Architectures Inc, 2635 Hopkins Court, Pullman, WA, 1997.
7. '9600 Data Compression Processor', Data Sheet, Hi/fn Inc, 750 University Avenue, Los Gatos, CA, 1999.
8. J.Ziv, A.Lempel, ' A Universal Algorithm for Sequential Data Compression' IEEE Trans. Inf. Theory, Vol. IT-23, pp. 337-343, 1977.
9. J. Ziv, A. Lempel, 'Compression of Individual Sequences Via Variable Rate Coding', IEEE Transactions on Information Theory, Vol. IT-24, pp. 530-536, 1976.
10. A.Moffat, N.Sharman, I.Witten, T.Bell, 'An Empirical Evaluation of Coding Methods for Multi-symbol Alphabets', Information Processing & Management, Vol. 30, No. 6, pp. 791-804, 1994.
11. C. Bloom, 'Solving the Problems of Context Modelling', http://www.cbloom.com/papers/index.html, 1998.
12. M. J. Slattery, J. L. Mitchell, 'The Qx-Coder', IBM Journal of Research and Development, Vol. 42, No. 6, pp. 767-784, 1998.
13. S..Kuang, J. Jou, Y. Chen, 'Dynamic pipeline design of an adaptive binary arithmetic coder', IEEE Trans. on Circuits and Systems-II: Analog and Digital Signal Processing, Vol. 48, No. 6, pp. 813 –825, Sep 2001.

14. M. Boo, J.D. Bruguera and T. Lang, ‘A VLSI Architecture for Arithmetic Coding of Multilevel Images’, IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing, Vol. 45, No. 1, pp. 163-168, January 1998.
15. J. Jiang, ‘Novel design of Arithmetic Coding for Data Compression’, IEE Proc.-Comput. Digit. Tech., Vol. 142, No. 6, pp. 419-424, November 1995.
16. M. Hsieh, C. Wei, ‘An adaptative Multialphabet Arithmetic Coding for Video Compression’, IEEE Transactions on Circuits and Systems for Video Technology’, Vol. 8, No. 2, pp. 130-137, April 1998.
17. J. Cleary, I. Witten, ‘Data Compression Using Adaptive Coding and Partial String Matching’, IEEE Transactions on Communications, Vol. 32, No. 4, pp. 396-402, 1984.
18. L. Bottou, P. G. Howard, Y. Bengio, ‘The Z-coder adaptive binary coder’, In Proceedings of the Data Compression Conference, pp. 13-22, March 1998.
19. R. Stefo, J.L Núñez, C. Feregrino, S. Mahapatra, S. Jones, ‘FPGA-based modelling unit for high speed lossless arithmetic coding’, 11th International Conference on Field Programmable Logic and Applications FPL'2001, Belfast, Northern Ireland, UK, pp. 643-647, August 27-29, 2001.
20. R. Arnold, T.Bell, ‘A Corpus for the Evaluation of Lossless Compression Algorithms’, Data Compression Conference, pp. 201-210, 1997.
21. J. Nunez, V.A Chouliaras, ‘High-Throughput Arithmetic Coding Hardware for the H264 Advanced Video Compressor’, submitted to IEEE Transactions on Circuits and Systems for Video Technology, March 2004
22. Information available at http://www.compressconsult.com/rangecoder/
23. Information available at www.simplescalar.com
24. Information available at www.altera.com