Gbit/s Lossless Data Compression Hardware

José Luis Núñez, *Member, IEEE,* and Simon Jones, *Senior Member, IEEE*

***Abstract*—This paper presents the X-MatchPRO high-speed lossless data compression algorithm and its hardware implementation, which enables data independent throughputs of 1.6 Gbit/s compression and decompression using contemporary low-cost reprogrammable field-programmable gate array technology. A full-duplex implementation is presented that allows a combined compression and decompression performance of 3.2 Gbit/s. The features of the compression algorithm and architecture that have enabled the high throughputs are described in detail. A comparison between this device and other commercially available data compressors is made in terms of technology, compression ratio, and throughput. X-MatchPRO is a fully synchronous design proven in silicon specially targeted to improve the performance of Gbit/s storage and communication applications.**

***Index Terms*—Data compression, field-programmable gate arrays (FPGAs), lossless circuits.**

I. Introduction

LOSSLESS data compression, where the original data is reconstructed exactly after decompression is being accepted as a tool that can bring important benefits to a computing system. Its applications have been increasing over the past years thanks to a combination of pressure for more bandwidth allied and to the need to improve storage capacity [1]–[5]. Lossless data compression has been successfully applied to storage systems (tapes, hard disk drives, solid state storage, file servers) and communication networks (LAN, WAN, wireless). Data compression is not being used to its full advantage in systems that operate at bandwidths of over 1 Gbit/s due to performance limitations encountered in the data compression hardware. This paper describes the X-MatchPRO method and architecture that uses a CAM-based dictionary where multiple symbols are processed per cycle to deliver the required performance to avoid becoming a bottleneck in a system operating at a gigabit per second bandwith.

The remainder of this paper is organized as follows: Section II presents a review of the area of lossless hardware-based data compression. Section III describes the characteristics of the X-MatchPRO algorithm. Section IV analyzes the X-MatchPRO compression/decompression architecture. Section V compares the X-MatchPRO compressor with other commercially available data compressor devices. Finally, Section VI concludes this paper.

Manuscript received March 20, 2001; revised January 8, 2002 and September 26, 2002.

J. Nuñez is with the Department of Electronic and Electrical Engineering, University of Loughborough, Leicestershire LE11 3TU U.K.

S. Jones is with the Department of Engineering and Design, University of Bath, Bath BA2 7AY, U.K.

Digital Object Identifier 10.1109/TVLSI.2003.812288

II. Background

A useful classification of lossless data compression systems identifies two main components: a model and a coder [6]. The purpose of the model is to identify where the redundancy is located in the input data and signal repetitive data sequences to the coder. The coder uses the information obtained from the model to replace the input data for shorter codewords and to produce a compressed output. Compression is obtained whenever the ratio of output bits to input bits is less than 1. Although, some coding methods map better than others depending on the chosen model, many different combinations between model and coder are possible.

Modeling can be done mainly in two different ways: statistical or dictionary. Both methods have found their way to hardware and software implementations of lossless data compression systems.

1) Statistical Methods: Statistical methods show a clearer separation between model and coder than dictionary methods. Statistical modeling is based on assigning values to events depending on their probability. The higher the value the higher the probability. The accuracy with which this frequency distribution reflects reality determines the efficiency of the model. The best lossless compression figures reported in the literature correspond to software based statistical methods [7] like Prediction by Partial Matching (PPM) [8] and [9] and Dynamic Markov Compression (DMC) [10]. These methods are based on variable order Markov modeling [11], [12] where predictions are done based on the symbols that precede the current symbol. Statistical methods in hardware are restricted to simple higher order modeling using binary alphabets that limits speed, or simple multisymbol alphabets using zeroth-order models that limits compression. Binary alphabets limit speed because only a few bits (typically a single bit) are processed in each cycle while zeroth-order models limit compression because they can only provide an inexact representation of the statistical properties of the data source. Coding is typically performed with methods like Huffman coding [13] or arithmetic coding [14], the latter being preferred because its efficiency can be made arbitrarily close to the entropy or information content of the model by controlling its precision and therefore is optimal for any model [15]. A few statistical data compressors have been reported in the literature. A zeroth-order model associated with an arithmetic coder is described by Boo *et al.* in [16] for coding of multilevel images. The probabilities in the model are stored in cumulative format using reference probabilities to simplify the update process. The arithmetic coding process has been simplified by truncating the multiplier. An implementation of a parallel binary arithmetic coder is done by Jiang in [17] using an IBM Q-coder [18] as the building block. The Q-coder is a

seventh-order binary Markov model associated with a corresponding binary arithmetic coder. The parallel implementation in [17] processes 4 bits in parallel and since there are only 16 possible input combinations, parallel decoding is also possible. The same technique is used in [19] to obtain a parallel implementation of a multi-alphabet arithmetic coder associated with a byte-based zeroth-order model. The system processes 8 B at a time, but parallel decoding is in this case unfeasible because the number of possible input combinations is 256^8, hence, the complexity of the hardware is too high. The work presented by the same author in [20] is the implementation of a byte-based zeroth-order model associated with a multi-alphabet arithmetic coder. Kuang *et al.* present another high-order binary model in [21] that describes a tenth-order Markov model with associated binary arithmetic coder. In this case, as with the IBM Q-coder, the high-order binary Markov modeling uses fixed-order models and not variable-order models such as PPM, because it is always possible to predict both symbols in a binary alphabet. The chip has been implemented in a 0.8 μm and clocks at 25 MHz. The compression ratio is in the order of 0.5, while speed is data dependent but typically around 3 Mbit/s. Hsieh and Wei describe a byte-based zeroth-order model associated to a multi-alphabet arithmetic coder in [22] for video compression. A similar technique to [16] is used to store the frequency model using some frequency counts as base and others as offsets from the base. This technique simplifies model adaptation. The chip described in [23] by Mukherjee *et al.* does not use arithmetic coding but tree-based codes [24]. Huffman coding is the most popular tree-based code, but others exist [25]. The code is static and it does not adapt with changes in the incoming data source, but since it is not hardwired but mapped to a memory device, it can be changed to suit the application. A compression ratio of 0.5 processing 8-bit symbols results in each symbol being processed in approximately two memory cycles. They report a compression performance of 95.2 Mb/s for compression and 60.6 Mb/s for decompression in a 2-μm SCMOS technology with a clocking frequency of 83.3 MHz. An adaptive Huffman code implementation in hardware is presented in [26]. This design is based on content addressable memory (CAM) modules to speed up the tree adaptation process and achieves a throughput of almost 1 bit/cycle. The model is again a zeroth-order model but no details are available of the hardware implementation.

2) Dictionary Methods: Dictionary methods try to replace a symbol or group of symbols by a dictionary location code. The modeling stage is given extra importance while coding is simplified. Some dictionary-based techniques use simple uniform binary codes to process the information supplied by the modeller. Both software and hardware based dictionary models are very popular, achieving good throughput and competitive compression. Utilities like Pkzip and ARJ in software, or hardware algorithms like ALDC developed by IBM [27], [28] and also available from AHA [29], and LZS developed by STAC/Hifn [30], [31] illustrate this situation. These four examples are Lempel-Ziv-1 (LZ1) derivatives [32]. The ALDC chip is implemented in a 0.8-μm CMOS technology and clocks at 40 MHz to obtain a throughput of 320 Mb/s. The AHA implementation achieves 320 Mb/s at a 40-MHz operation and it is implemented in a 0.5-μm CMOS technology. The STAC/Hi/fn device has been implemented in a 0.35-μm CMOS technology. It clocks at 80 MHz with a throughput of 640 Mb/s. The Hi/fn device is also a full-duplex architecture meaning that it can compress and decompress simultaneously. Both of these chips use CAM memory to store the dictionary and enable parallel searching and adaptation. Surk presents a processing element (PE)-based architecture for the LZ1 algorithm in [33]. Each PE compares the incoming input symbol with the symbol it stores in one cycle and shifts the symbol to its neighbor. The data input rate is constant and post-layout simulation indicates a performance of 700 Mb/s in a 0.5-μm CMOS technology. The basic symbol is 7-bits wide so the compressor is only suitable for the compression of ASCII coded text. Jung and Burleson describe another LZ1 implementation for optimization of wireless local area networks in [34]. The architecture includes multichannel support being able to switch between different dictionaries depending on the communication channel being compressed. This improves compression since each channel has its own dictionary but there is an overhead associated with the multiplexing. A throughput of 50 Mb/s is reported based on 1.2-μm CMOS technology using a clock frequency of 100 MHz. Nusinov and Pasco also present an LZ1 derivative for multichannel compression in [35]. The different dictionaries are stored in RAM memory externally and the appropriate one is uploaded in internal CAM. The chip clocks at 20 MHz and has a throughput of 80 Mb/s.

Lempel–Ziv-2 (LZ2) [36] algorithms have not become as widely used as LZ1 algorithms. The UNIX utility 'compress' uses LZ2 and the data compression Lempel–Ziv (DCLZ) family of compressors initially invented by Hewlett-Packard [37] and currently being developed by AHA [38], [39] also use LZ2 derivatives. The DCLZ family of devices clock at 40 MHz for a throughput of 160 Mb/s based on a 0.5-μm CMOS technology. Bunton and Borriello present another LZ2 implementation in [40] that improves on the [37] DCLZ. This new algorithm uses a similar dictionary structure to [37] but it offers a more advanced dictionary maintenance mechanism where a tag is attached to each dictionary location to identify which node should be eliminated once the dictionary becomes full. The design has been implemented in a 2-μm CMOS technology with a throughput of 160 Mb/s.

Other work that cannot be classified in the range of statistical or dictionary coding includes the genetic algorithms (GA) developed by the DCP Research Corp. in the DCP816 chip [41]. This chip is implemented in a 1-μm CMOS technology and has a throughput of around 1.68 Mb/s clocking at 40 MHz. It supports multiple channels of compression/decompression and uses 512 kB of external RAM per channel. Sakanashi *et al.* presents in [42] a device for printer image compression also based on a genetic algorithm that is able to select the best group of pixels to be used as context to predict the next input pixel depending on the characteristics of the image being compressed. The compressing method is lossless and it is associated to reconfigurable hardware such as an field programable gate array (FPGA) plus a standard IBM QM-coder [43], a derivative from the Q-coder, to perform the compression itself. Our own work, the X-MatchPRO family of devices, belongs to the category of dictionary-based compressors but they are not LZ derivatives.

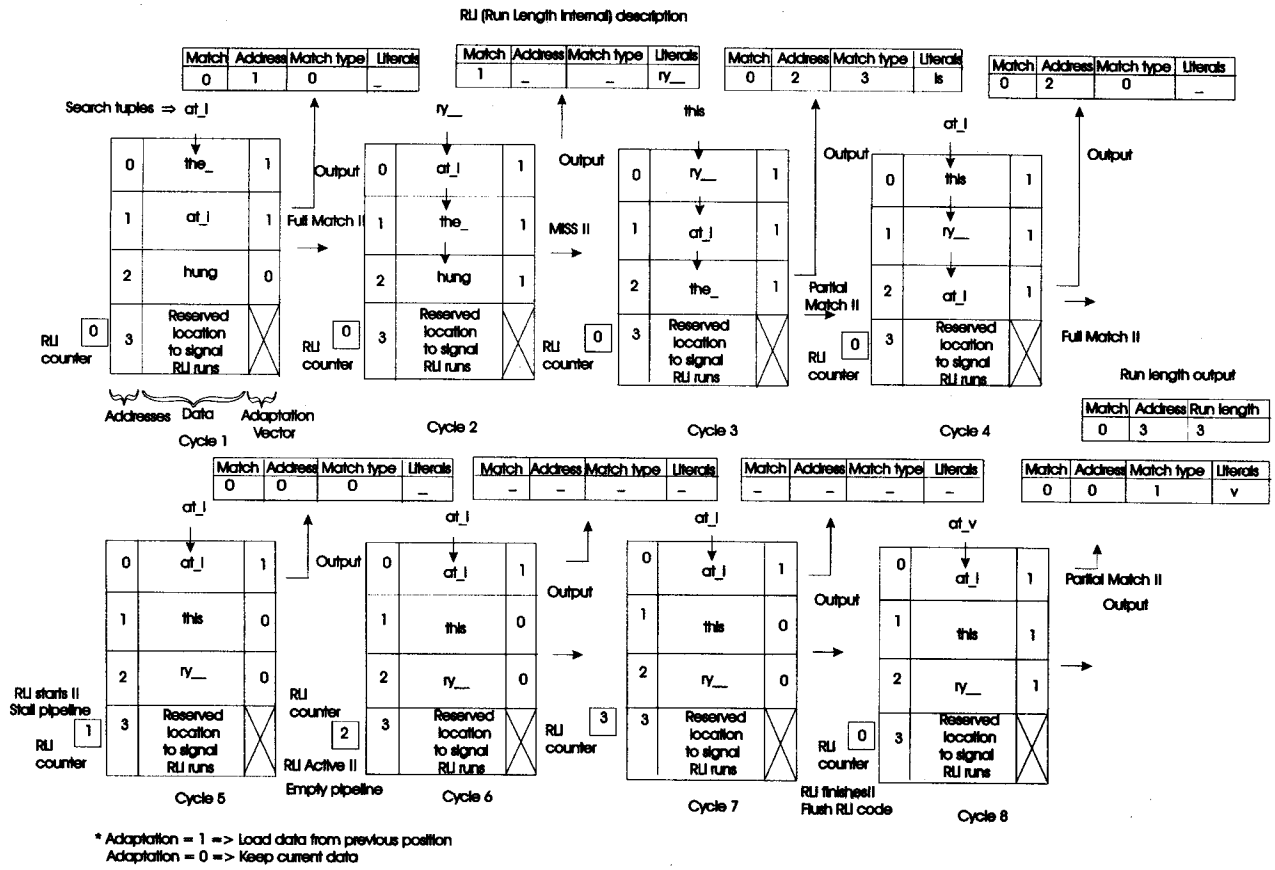

Fig. 1. X-MatchPRO example.

X-MatchPRO originates from our previous research [44]–[48] and advances in FPGA technology. The flexibility provided by using this technology is of great interest since the chip can be adapted to the requirements of a particular application easily. The objective is then to use programmable hardware able to obtain good compression ratios and still maintain a high throughput so that the compression/decompression processes do not slow the original system down.

III. The X-MatchPRO Algorithm

The X-MatchPRO algorithm uses a fixed-width dictionary of previously seen data and attempts to match or partially match the current data element with an entry in the dictionary. Each entry is 4 B (tuple) wide and several types of matches are possible where all or some of the bytes at different positions within the tuple match. Those bytes that do not match are transmitted as literals. This partial match concept gives the name to the procedure- the X referring to 'don't care.' At least 2 B have to match and when no valid match is generated a miss is codified adding a single bit to the four-byte tuple. The dictionary is maintained using a move to front (MTF) strategy [49] whereby a new tuple is placed at the front of the dictionary while the rest move down one position. When the dictionary becomes full the tuple placed in the last position is discarded leaving space for a new one. X-MatchPRO reserves one location in the dictionary to code internal runs of full matches at location zero. Since the MTF strategy forces anything that repeats to be stored at location zero (top of dictionary), this run-length-internal (RLI) technique is used to efficiently code any 32-bit repeating pattern.

The coding function for a match is required to code several fields as follows.

A zero followed by:

If normal code:

1) Match location: It uses the binary code associated to the matching location.
2) Match type: That indicates which bytes of the incoming tuple have matched. This is codified using a static Huffman code based on the statistics obtained through extensive simulation.
3) Any extra characters that did not match transmitted in literal form.

If RLI code:

1) RLI location: The last address in the dictionary is reserved to code RLI events.
2) Run length: 8 bits are used to indicate how many 32-bit repeating patterns have been observed. The maximum run length that it is possible to process in a single code is therefore 255.

The coding function for a miss has two fields as follows:

A one followed by:

1) The 4 B in literal form.

A data tuple (4 B) is added to the front of the dictionary while the rest move one position down if a full match has not occurred. The MTF technique is only applied when dealing with

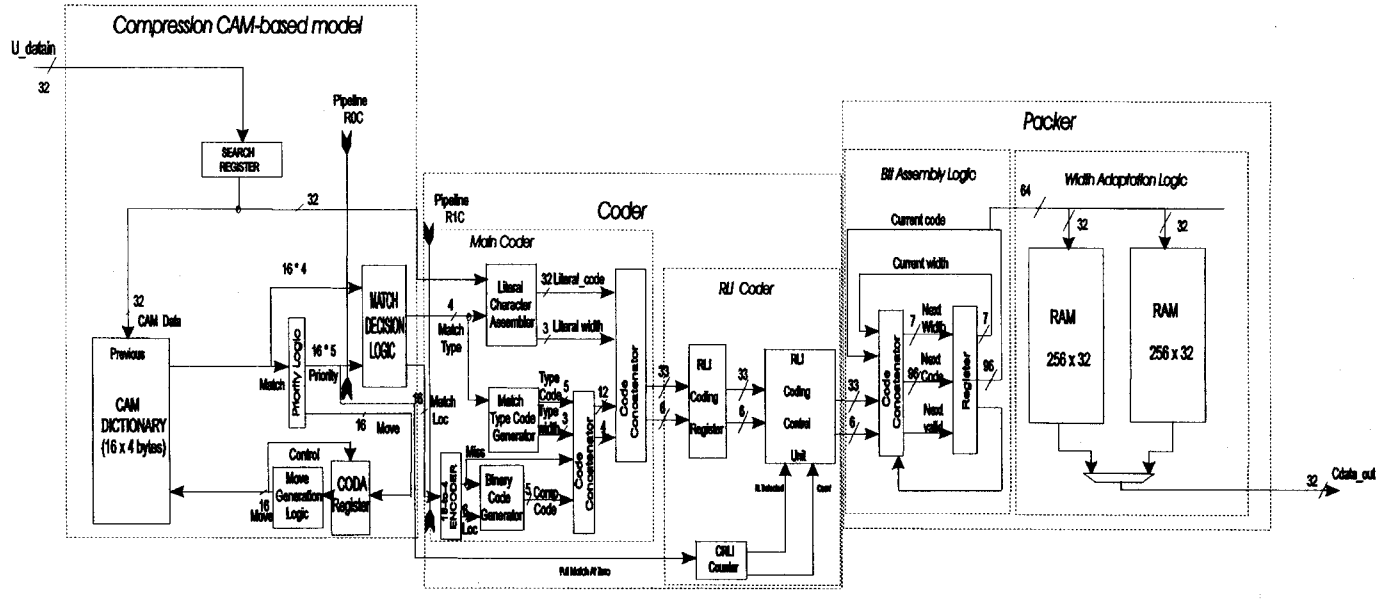

Fig. 2. Architecture of the compressor.

full matches. In this case, the tuples from the first location until the location previous to the matching tuple move down one location, while the matching tuple is placed at the front of the dictionary. The algorithm is illustrated with an example in Fig. 1. The example is based on a small dictionary of only four locations, one of which is reserved to code RLI events. Each dictionary location has a different address. The adaptation vector column defines how the dictionary adapts for the next cycle so that a 1 means load data from north neighboring location, while a 0 means keep current data. Each cycle in the figure corresponds to a different clock cycle. The search data in the cycle 1 of Fig. 1 generates a full match at location 1 and the corresponding output is generated together with a new adaptation vector that will shift the dictionary for cycle 2. The search data in cycle 2 cannot be found in the dictionary so a miss is generated with the four missing bytes being added to the output in literal form. The cycle 3 search generates a partial match where the two first bytes of the search tuple are found in location two. The match type 3 signals this matching condition and the two missing bytes are added to the output in literal form. Cycle 4 generates a new full match this time at location 2. An RLI coding event is active in cycles 5, 6, and 7. The RLI output is generated at cycle 8 when the run stops with a length of 3. The RLI counter only increments when the search data is present at location 0. The code generated at cycle 5 is removed from the output when the RLI counter exceeds 1 because cycle 5 would be coded as part of the runlength. This output code would have been needed if the RLI counter had remained with a count of 1 indicating a single full match at location 0 and not a valid runlength.

IV. The X-MatchPRO Hardware

X-MatchPRO uses a simple coprocessor style interface to communicate with the rest of the system. Compression and decompression commands are issued through a common 16-bit control data port. A 3-bit address is used to access the internal registers that store the commands plus information related to compressed and uncompressed block sizes for reading or writing. A total of six registers form the register bank. Three registers are used to control the compression channel and the other three for the decompression channel. The first bit in the address line indicates if the read/write operation accesses compression or decompression registers. The chip is designed to compress any block size ranging from 8 B to 32 kB. A decompression operation can be requested in the middle of a compression operation and vice versa.

A. Compression Architecture

The compression architecture is based around a block of CAM to realize the dictionary. This is necessary since the search operation must be done in parallel in all the entries in the dictionary to allow high- and data-independent throughput. The length of the CAM varies with three possible values of 16, 32, or 64 tuples trading complexity for compression. Dictionary size is variable to be able to adapt algorithm complexity to the resources available in the selected FPGA. The number of tuples present in the dictionary has an important effect on compression. In principle, the larger the dictionary the higher the probability of having a match and improving compression. On the other hand, a bigger dictionary uses more bits to code its locations degrading compression when processing small data blocks that only use a fraction of the dictionary length available. The width of the CAM is fixed with 4 B/word and its columns can be configured as selectable shift-registers to implement the move to front adaptation policy.

Fig. 2 shows the compression architecture. There are three major components in the compression architecture corresponding to compression model, coder, and packer. Fig. 2 also shows the location of the pipeline registers used to reduce the clock period of the design. There are a total of five levels of registers from input to output and the design supports incremental transmission, which means that transmission of compressed data present in the output buffers can start before

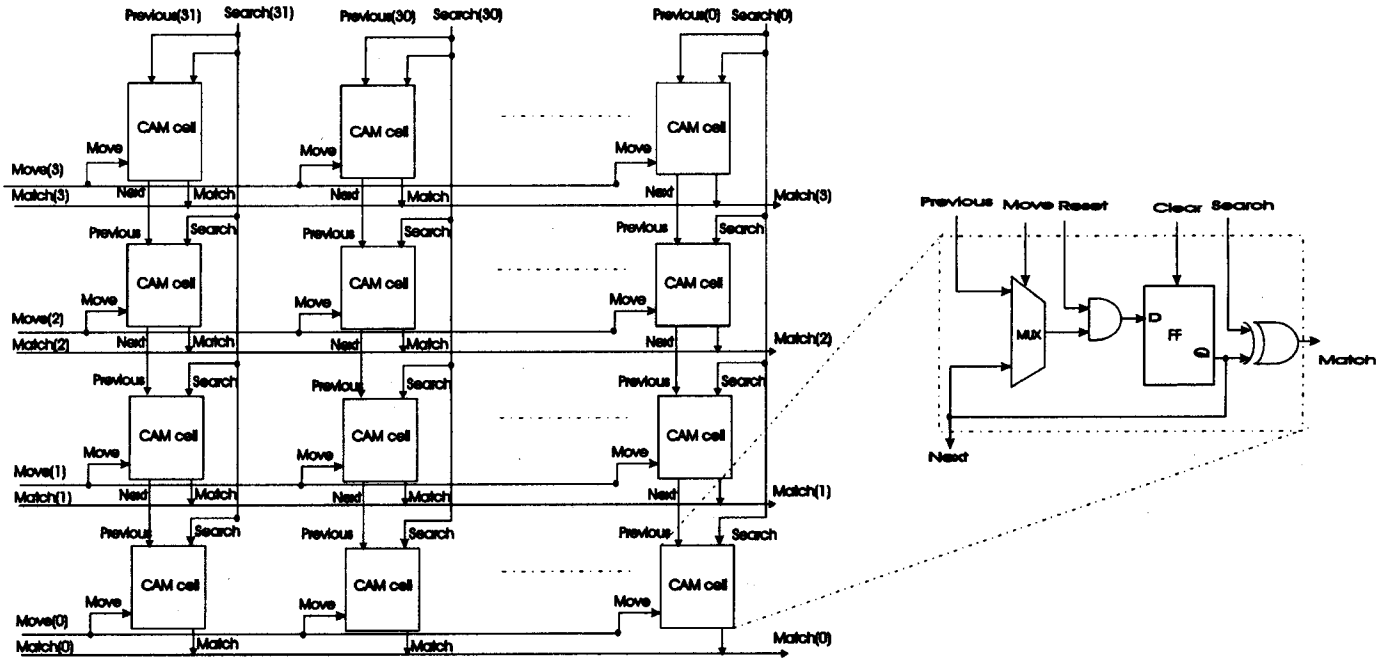

Fig. 3. CAM-Based dictionary architecture.

the whole data block is compressed. These two features help to maintain the latency of the design to a minimum.

1) The Compression Model Comprises:

a) Dictionary: CAM-based dictionary with 16, 32, or 64 tuples. The n-tuple dictionary is formed by a total of nx32 CAM cells. Each cell stores one bit of a data tuple and it can maintain its current data, or load the data present in the cell above. The dictionary architecture is illustrated in Fig. 3. The architecture compares the search data with the data present in the dictionary using one XOR gate to do the comparison of each input bit plus ($\log_2$(dictionary width)) 2-input AND gates tree to obtain a single comparison bit per dictionary position. The delay of the search operation, although in principle is independent of dictionary length, in practice the high fanouts and long wires of large dictionaries degrade its speed considerably. An adaptation vector named *move* in Fig. 3 and whose length equals the dictionary length defines which cells keep its current data and which cells load data from its north-neighboring cell.

b) Move generation logic: The adaptation vector *move* is generated by the movement generation logic using the results of the search operation present in the *match* vector of Fig. 3. The movement generation logic function is to propagate up a match position so all the dictionary cells located over the match position and the match position itself load the data of their north neighboring cells, while all the dictionary cells located down the match position keep the current data. New data is always inserted at the top of the dictionary so when a data element is found in the dictionary it is promoted from its current position to the top of the dictionary in a single cycle. The propagation delay of the movement generation logic is O(log2(dictionary length)) 2-input OR gates. Data flows toward the bottom of the dictionary as it grows older. The oldest data element is always located at the bottom of the dictionary and this is the one evicted from the list when room is required for a data element new to a full dictionary.

c) Out of Date Adaptation (ODA) logic: ODA logic forces the dictionary to adapt with previous match information and breaks the critical path in compression improving speed. In principle, the adaptation vector *move* must be generated using the results of the current search operation available in the *match* vector before the next cycle can start. This search and adaptation operation forms a critical feedback loop specially with large dictionaries because it depends with dictionary size with $O(1 + \log_2(\text{dictionary width} + \log_2(\text{dictionary length}))$ levels of logic and the search operation becomes critical since the fanout of the search register is directly proportional to dictionary length. It is not possible to add a pipeline register in the feedback loop without affecting the algorithm functionality so to further increase the speed of the circuit the algorithm is modified introducing the ODA mechanism. ODA implies that adaptation at time $t + 2$ takes place using the match results generated by the previously process data at time t and not the one at time $t + 1$. This technique breaks the fundamental feedback loop by adding a register between the search and adaptation circuitry. The danger is that dictionary efficiency could be lost if the ODA technique duplicates the same data in different positions in the dictionary. Prior to adding a register between the search and adaptation operations, the adaptation vector at time t provides information to reorder the dictionary at time $t + 1$ and makes sure that data words are unique in the dictionary. In ODA the adaptation vector at time t is not effective until time $t + 2$ so adaptation at time $t + 1$ could insert a data element at the top of the dictionary that already exists in some other dictionary location. After a few cycles the same data could be stored in multiple dictionary positions and dictionary efficiency would be lost degrading compression. The way to avoid this is by forcing the current adaptation vector to adapt not only the dictionary as before but also the next

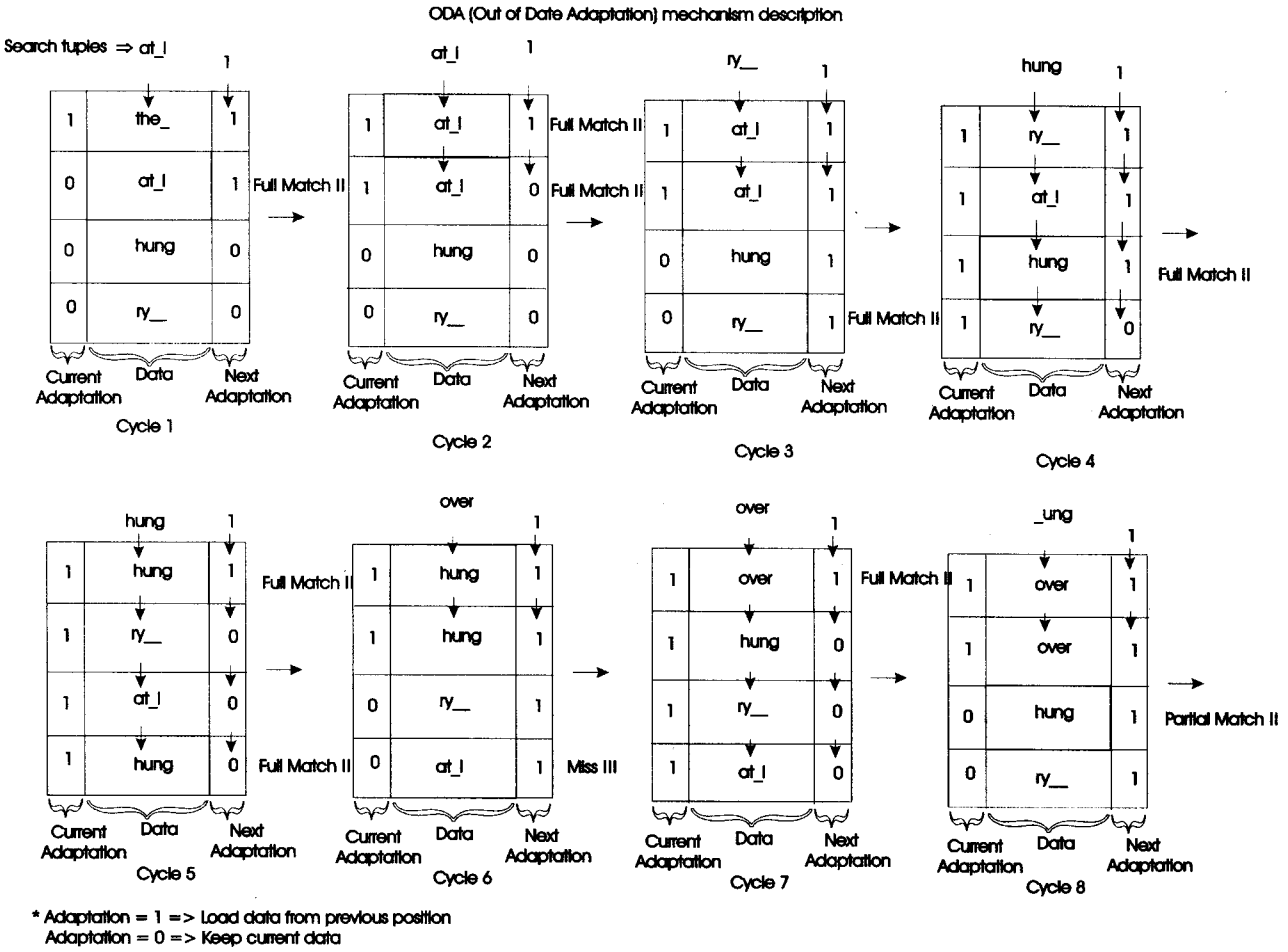

Fig. 4. Out of date adaptation example.

adaptation vector. Fig. 4 illustrates this process using a small dictionary of only four positions in length and 4 B (tuple) in width. Every cycle of Fig. 4 corresponds to a different cycle. The multiple full-match events in cycles 2 and 5 show how the search data could be found simultaneously at position 0 and at position higher than zero, but in this case the match at position 0 is selected as valid. The next adaptation vector depicted at the right of the dictionary depends exclusively on this match information. Fig. 4 shows how ODA adapts the dictionary at time $t + 2$ using a modified adaptation vector originally generated at time t and how data duplication is restricted to position 0 maintaining dictionary efficiency. For example, the current adaptation vector depicted at the left of the dictionary for cycle 3 is generated shifting down the next adaptation vector of cycle 2, as indicated by the current adaptation vector of cycle 2. The current adaptation vector at cycle 3 adapts the dictionary for cycle 4. By using this simple technique, the effect of ODA in dictionary efficiency is negligible because in the worst case only one dictionary position contains repeated information and in the best case all the dictionary positions contain different data. The logic cost of ODA is very small since the basic ODA cell only contains a flip-flop and a multiplexor. Fig. 5 shows the ODA logic plus the movement generation logic for a dictionary of four positions.

d) Priority logic: This logic assigns a different priority to each of the possible matches. A full match has the highest priority while partial matches are assigned priorities according to the number of matching bytes. The higher the number, the higher the priority.

e) Best match decision logic: Logic that selects one of the matches as the best for compression using the priority information.

2) The Coder Comprises:

f) Main coder: Main X-MatchPRO coder whose function is as follows: when a match is detected it assigns a uniform binary code of size $\log_2$(dictionary size) to the match location preceded by a single bit set to 0, a static Huffman code to the match type, and concatenates any necessary bytes that were not found part of a match in literal form. There are 11 possible different match type combinations of 2, 3, or 4 B matching in the tuple. The Huffman tree, obtained after extensive simulation, has only four different code lengths of 2, 3, 4, and 5 bits. The full match is the most probable match type and its Huffman code is only 2-bits long. Matches of three nonconsecutive bytes are the least probable and they are assigned 5-bit long Huffman codes. If instead of a match a miss is detected, the first single bit is set to 1 and the 4 B in literal form follow.

g) RLI coder: The RLI coder detects the existence of multiple full matches at location zero, using a counter. If the counter exceeds the count of 1, then an RLI event becomes active, the pipeline is empty from the previous code, and the output of the chip is frozen while the run length is taking place. A maximum

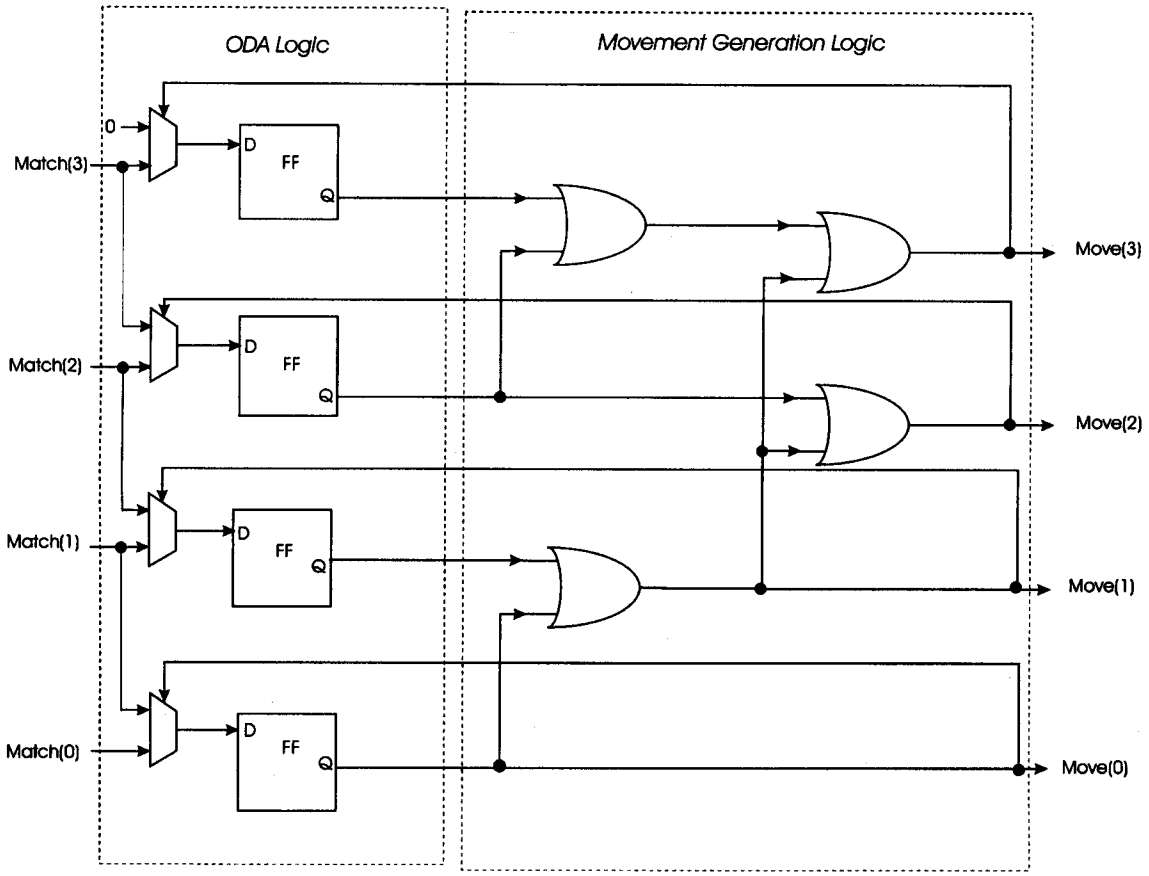

Fig. 5. Adaptation logic architecture.

of 255 full matches at location 0 can be coded in a single RLI codeword. The code corresponding to the last location in the dictionary is reserved to signal RLI events.

3) The Packer Comprises:

h) Bit assembly logic: Logic that assembles the variable-length codewords produced by the coder into 64-bit fixed length codes which are then output to the width adaptation logic.

i) Width adaptation logic: This logic reads in 64-bit compressed words from the bit assembly logic and writes out 32-bit compressed words to the compressed output bus. It performs a buffering function smoothing the data flow out of the chip to the compressed port and it also transforms the data width from 64-bit to a more manageable 32-bit. It contains a total of 2 kB of fully synchronous dual-port RAM organized in two blocks of 256×32 bits to buffer compressed data before it is output to the compressed data out bus.

B. Decompression Architecture

Fig. 6 shows the decompressor architecture. The decompressor channel is also formed by three major components: the decompression model, decoder, and unpacker. The number of registers in Fig. 6 from input to output is again five, so the latency of the compressor and decompressor channels is comparable. The design supports incremental reception so decompression of the compress block can start before the whole data block has been received.

1) The Decompression Model Comprises:

j) Dictionary: Fully synchronous RAM-based dictionary that stores the history data during a decompression operation. The contents of the RAM dictionary during decompression must be the same as the contents of the CAM dictionary during compression in each cycle. Adaptation must take place in exactly the same way to enable correct decompression of the compressed block. The initialization of the compression CAM sets all words to zero. This means that a possible input word formed by zeros will generate multiple full matches in different locations. The algorithm simply selects the full match closest to the top. This operational mode, in effect, initializes the dictionary to a state where all the words with location address higher than zero are declared invalid without the need for extra logic. The reason is that location x can never generate a match until the data contents of location $x - 1$ are different from 0 because locations closer to the top have higher priority generating matches. The MTF adaptation mechanism shifts down the dictionary when full matches are not detected and, therefore, ensures that the last word from this initial state to be deleted from the dictionary is always the word located at location 0 at time 0. This operational mode in compression enables the decompression RAM dictionary to have only location 0 loaded with value 0 during the initialization phase because references to RAM locations higher than zero are not possible before their contents are updated. This technique avoids having a long overhead equal to dictionary size cycles to initialize each position in the RAM to

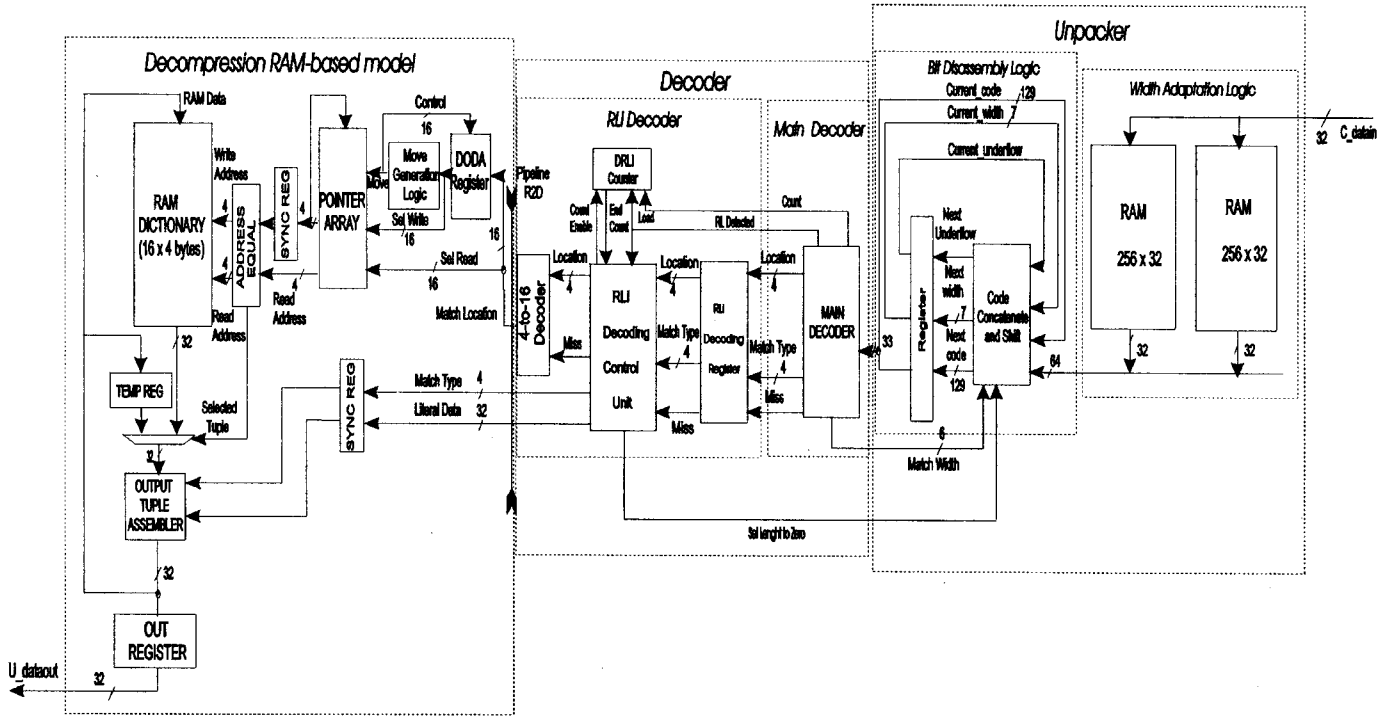

Fig. 6. Architecture of the decompressor.

a predefined value before each decompression operation. The read and write addresses are also monitored for possible collisions. If both addresses are the same, the algorithm needs to read the data that is going to be written in that common address. This data is not present in the memory yet, but it is present in the RAM data in bus. The RAM data is written in the memory normally but it is also latched temporarily in a register. Multiplexing logic selects the output coming from this register instead of the output coming from the memory when the same address is being read and written. The read address is also modified to an unused address to make it different from the write address and avoid corrupting the RAM contents.

k) Pointer array: The pointer array logic performs an indirection function over the read and write addresses that accessed the RAM dictionary. It models the MTF maintenance policy of the CAM dictionary moving pointers instead of data. The pointer array enables mapping the CAM dictionary to RAM for decompression. Since the pointer array is much smaller than the CAM dictionary the savings in complexity allow having the full-duplex architecture in a single device. This is true because the basic pointer word width is 4, 5, or 6 bits depending on the length of the dictionary. On the other hand, the basic data word width is 32 bit. Each position in the pointer array is reset to a value the same as its physical location in the array before each decompression operation.

l) Move generation logic: This logic generates the adaptation vector depending on the match type and match location. The adaptation vector moves the CAM dictionary in compression and the pointer array in decompression.

m) ODA logic: This component forces the pointer array to adapt with previous match information. The ODA logic in decompression is used to replicate the adaptation process in the compression dictionary. They have exactly the same functionality so both dictionaries are maintained in synchrony, although its use to improve the timing characteristics of the design is restricted to the compression channel.

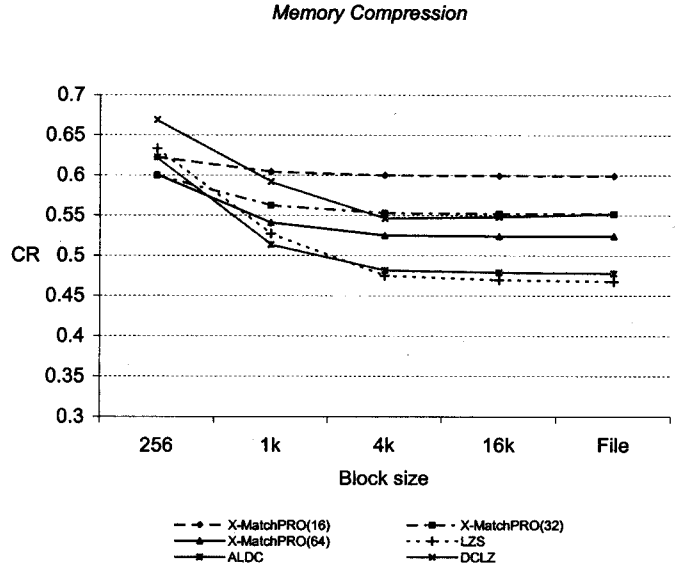

Fig. 7. Compression performance on the memory data set.

n) Output tuple assembler: Module that assembles a decompressed tuple using dictionary information and any literal characters present in the code.

2) The Decoder Comprises:

o) Main decoder: The main decoder obtains a match type and a match location from the codeword supply by the bit unpacker. The first bit defines if a miss or a match follows. If a match is detected the next $\log_2$(dictionary size) following bits in the codeword define the match location. The Huffman code

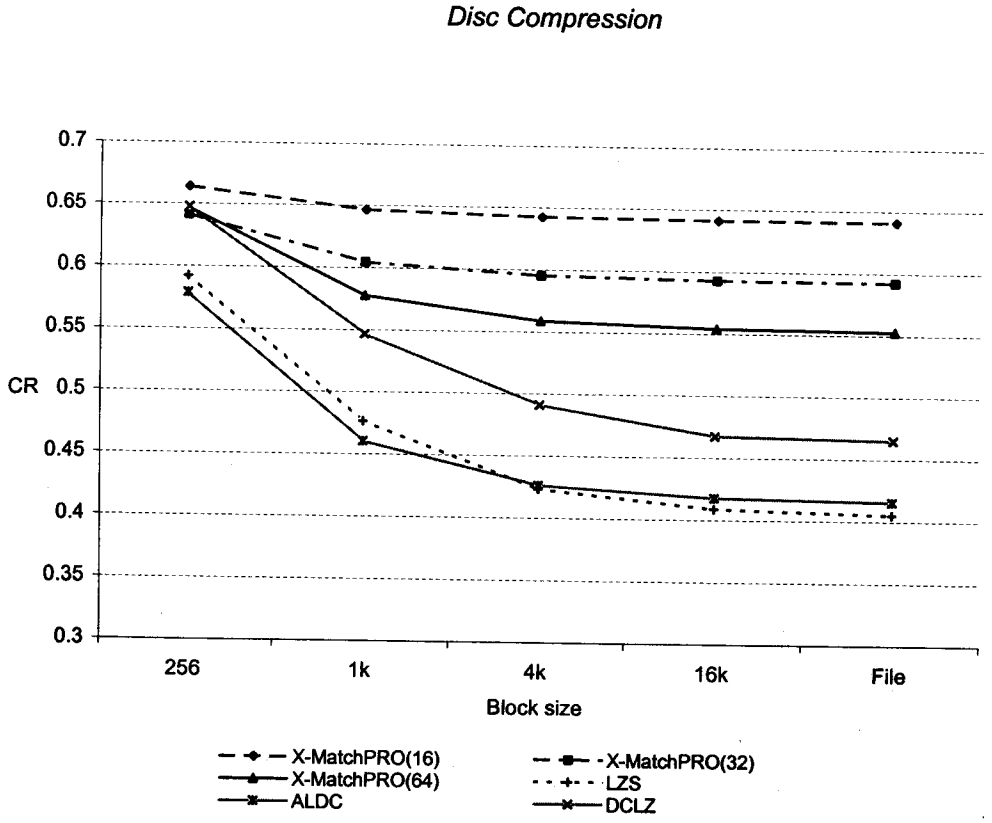

Fig. 8. Compression performance on the disc data set.

for the match type follows the match location code. If the match is partial the missing bytes follow the match type. If instead of a full or partial match a miss is detected the next 32 bits following the first bit correspond to the four missing bytes.

p) RLI decoder: RLI decoder that when the match location in the codeword corresponds to the last position of the dictionary outputs match location 0 and match type 0 as many times as the number of repetitions indicated in the next 8 bits that defined the run length. A counter is loaded with the run length and then it counts up until this value is reached.

3) The Unpacker Comprises:

q) Bit disassembly logic: This logic unpacks 64 bits of compressed data read from the internal buffers into variable-length codewords. To be able to shift out old data and concatenate new data the codeword length must be supplied by the decoder logic. This feedback loop between the decoder logic and the unpacker logic is illustrated in Fig. 6 with the signal *match width* extending from the main decoder module to the code concatenate and shift module. The architecture of this module has been parallelised so concatenation of new data is done in parallel to the decoding operation and only the shifting of old data out must wait for the decoding operation to complete. This feedback loop remains, though, as the critical path of the design and limits the maximum clock frequency.

r) Width adaptation logic: This logic performs the equivalent but opposite function as its counterpart in the compression channel. It reads in 32-bit of compressed data from the input compressed bus and it writes out 64-bit of compressed data to the bit disassembly logic when it requires more data. It performs a buffering function smoothing the data flow in the chip from the compressed port. It contains 2 kB of fully-synchronous dual-port RAM organized in two blocks of 256 × 32 bits each as in the packer.

V. Performance Comparison

Our performance comparison is based on several lossless compression devices currently commercially available. These ASIC compressors are the ALDC1-40S [28] (IBM) and the AHA3521 [29] (AHA) that implement the adaptive lossless data compression (ALDC) (LZ1) algorithm by IBM, the AHA3211 [39] that implements the DCLZ (LZ2) algorithm by AHA and Hi/fn 9600 [31] that implements the Lempel-Ziv Stac (LZS) (LZ1) algorithm by STAC/Hifn. Two data sets have been chosen as representatives of network and computer-originated traffic: the memory data set and the disc data set. The memory data set is formed by data captured directly from main memory in a UNIX workstation used in an engineering environment. The disc data set is formed by typical data found in the hard disk of the same workstation.

Figs. 7 and 8 show the compression performance comparison. It is common in networking and storage applications that data is present in small packets so the performance of these algorithms is evaluated in function of four different block sizes plus file-based compression. The "Y" axis is the compression ratio (CR) defined as the ratio output bits/input bits so the smaller the figure the better the compression. The "X" axis is the block size defined as the number of bytes in a block data to be compressed independently. This means that the dictionary is cleared each time a data block is processed. Fig. 7 shows that the compression performance in memory compression is competitive with other

TABLE I
COMPARISON SUMMARY

X-MatchPROv4 Performance Summary.

DEVELOPERS		IBM	Advance Hardware Architectures (AHA)		STAC Electronics	System Design Group Loughborough University		
CHIP*		ALDC1-40S	AHA3521	AHA3231	Hi/fn 9602	X-MatchPROv4 (16-word dictionary)		
TECHNOLOGY DETAILS	PROCESS	IBM CMOS 0.8 micron triple-level gate array/std cell	0.5 micron CMOS	0.5 micron CMOS	0.35 micron gate array/std cell	0.18 micron SRAM-CMOS FPGA Xilinx VIRTEX-E	0.18 micron SRAM-CMOS FPGA Altera APEX20KE	0.25 micron FLASH-CMOS FPGA Actel A500K ProASIC
	COMPLEXITY	70 Kgates	Not Stated	Not Stated	100 Kgates	5367 LUT's 55 % of a XCV400EBG 432-8	5040 LC's 60 % of a EP20K200EFC 484-1	9039 TILE's 70% of a A500K130-BG456
	CLOCK SPEED	40 MHZ	40 MHZ	40 MHZ	80 MHZ	50 MHZ	50 MHz	25 MHz
THROUGHPUT		40 Mbytes/s	20 Mbytes/s	20 Mbytes/s	80 Mbytes/s	200 Mbytes/s	200 Mbytes/s	100 Mbytes/s
FULL-DUPLEX PERFORMANCE		N/A	N/A	N/A	160 Mbytes/s	400 Mbytes/s	400 Mbytes/s	200 Mbytes/s
ALGORITHM		ALDC	ALDC	DCZL	LZS	X-MatchPRO	X-MatchPRO	X-MatchPRO
EXTERNAL RAM REQUIRED		NO	NO	NO	NO	NO	NO	NO
COMPRESSION RATIO		0.44	0.44	0.52	0.44	0.58 16 word 0.53 32 word 0.51 64 word	0.58 16 word 0.53 32 word 0.51 64 word	0.58 16 word 0.53 32 word 0.51 64 word

implementations with a typical ratio of 0.5. Memory data exhibits a strong 32-bit granularity because it is based on a 32-bit operating system so it suits well the X-MatchPRO algorithm. Compression improves with block size until a 4-Kbyte block size is used. This is the natural block size for memory pages and further increases in block size do not improve compression significantly.

The disc data set of Fig. 8 is more textually bias with a lot of database information so byte-oriented methods such as LZ derivatives have an advantage. It shows that compression improves with packet size until around 16 kB for the LZ-derivatives and 4 kB for X-MatchPRO. The smaller X-MatchPRO dictionaries tend to saturate earlier than their LZ equivalents. The LZ typical dictionary size is much larger with a value of 512 locations in the IBM device and 2048 locations in the STAC/Hi/Fn device.

Table I shows a summary of the features of these lossless data compression devices. X-MatchPRO results are based on three different dictionary sizes: 16, 32, and 64 locations. A dictionary larger than 64 locations improves compression but the flip-flop rich architecture of the dictionary demands larger FPGAs. It is also necessary to replace the uniform binary coding of the match locations by a more complex coding technique. Otherwise, the extra number of bits required to code the match locations in a large dictionary damages the compression ratio specially when compressing small packets. These three X-MatchPRO implementations trade complexity for compression while speed remains invariant. The last column of Table I summarizes the characteristics of the X-MatchPRO algorithm as implemented in Xilinx, Altera, and Actel technologies. The complexity figures correspond to the dictionary with 16 entries. Doubling the dictionary size increases chip complexity by a factor of 1.5 approximately.

The X-MatchPRO chips use a lower-clock frequency than the ASIC implementations, but it can achieve higher throughput thanks to its internal parallel architecture able to process 4 B of input information in a single cycle while all the other solutions only process a single byte. All these chips use CAM circuits to implement the dictionaries and in the case of the fastest Hi/fn9600, ALDC1-40S and X-MatchPRO chips, each input symbol can be processed in a single cycle. Adaptation in the fast LZ1 implementations is based on keeping a window with the most recently seen symbols in the dictionary. Symbols enter and leave the dictionary in a first-in/first-out style, so model adaptation is simplified if compared with X-MatchPRO, where the best match must be resolved before the model is ready for a new cycle. X-MatchPRO solves the adaptation feedback loop that exists in its model with the use of the out-of-date adaptation mechanism that delays the arrival of match information to the dictionary by one cycle without affecting its efficiency. The packing and unpacking of compressed data is also simple in the selected ASIC devices because they map variable-length streams of symbols to fixed length codewords so the boundaries between codewords are easily identifiable. On the other hand, X-MatchPRO codewords are variable in length and their unpacking is a more complex process and indeed, a performance

limitation factor in the chip. The complexity and performance of the Altera Apex and Xilinx Virtex chips is comparable because both use a hierarchical architecture with SRAM switches based on logic cells (Altera) or logic elements (Virtex) with similar complexity and identical feature size. Actel ProASIC devices, on the other hand, use a flat architecture with fine-grained logic cells that increases routing complexity and negatively affects performance [50]. ProASIC feature size is also larger than the Xilinx and Altera feature size and this is also a reason for lower performance.

VI. Conclusion

X-MatchPRO offers an unprecedented level of compression/decompression throughput in a FPGA implementation of a lossless data-compression algorithm for general applications. The hardware architecture has been verified in three different FPGA technologies. The fine granularity of the Actel ProASIC devices has proven very efficient to implement the flip-flop rich X-MatchPRO architecture. The higher granularity of the Altera and Xilinx technologies combined with a more advanced process have enabled throughputs well over the Gbit/s mark. The full-duplex implementation effectively uses the memory resources available in these FPGAs to simultaneously handle a compressed and uncompressed data stream. The architecture is easily scalable so it can be adapted to newer FPGAs with higher gate counts with little effort. We aim to improve compression for the disc data set by increasing dictionary length and introducing more efficient coding techniques than simple uniform binary coding for the match locations. We also expect that an ASIC implementation of our algorithm will be able to improve throughput by a typical factor of 3, if compared with a similar feature size FPGA [Betz98].

References

[1] K. Dickson, Cisco IOS Data Compression, Cisco Syst., San Jose, CA, 2000.

[2] Data Compression, Mitel Remote Access Solutions, Mitel Corp., Mitel Networks, Kanata, ON, Canada, 2000.

[3] R. VanDuine, "Integrated Storage," IBM Corp., Rochester, MN, 2000.

[4] D. Cressman, "Analysis of data compression in the DLT2000 tape drive," *Digital Tech. J.*, vol. 6, no. 2, 1994.

[5] Data Compression Performance Analysis in Data Communications, Hi/fn Inc., Pullman, WA, 1997.

[6] M. Nelson, *The Data Compression Book*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[7] [Online] http://corpus.canterbury.ac.nz/results/cantrbry.html

[8] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Trans. Commun.*, vol. 32, pp. 396–402, 1984.

[9] A. Moffat, "Implementing the PPM data compression scheme," *IEEE Trans. Commun.*, vol. 38, pp. 1917–1921, 1990.

[10] G. V. Cormack and R. N. S. Horspool, "Data compression using dynamic Markov modeling," *Comput. J.*, vol. 30, no. 6, pp. 541–549, 1987.

[11] T. Bell, J. Cleary, and I. Witten, *Text Compression*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[12] Solving the Problems of Context Modeling, C. Bloom. (1998). *http://www.cbloom.com/papers/index.html* [Online]

[13] D. Huffman, "A method for the construction of minimum redundancy codes," in *Proc. I.R.E*, 1958, pp. 1098–1101.

[14] G. Langdon, "An introduction to arithmetic coding," *IBM J. Res. Develop.*, vol. 28, no. 2, pp. 135–149, Mar. 1984.

[15] A. Moffat, N. Sharman, I. Witten, and T. Bell, "An empirical evaluation of coding methods for multi-symbol alphabets," *Inf. Process. Manage.*, vol. 30, no. 6, pp. 791–804, 1994.

[16] M. Boo, J. D. Bruguera, and T. Lang, "A VLSI architecture for arithmetic coding of multilevel images," *IEEE Trans. Circuits Syst. II*, vol. 45, pp. 163–168, Jan. 1998.

[17] J. Jiang, "A novel parallel design of a codec for black and white image compression," *Signal Process. Image Commun.*, vol. 8, no. 5, pp. 465–474, 1996.

[18] W. B. Pennebaker *et al.*, "An overview of the basic principles of the Q-coder adaptive binary arithmetic coder," *IBM J. Res. Develop.*, vol. 32, no. 6, pp. 717–725, Nov. 1988.

[19] J. Jiang and S. Jones, "Parallel design of arithmetic coding," *Proc. Inst. Elect. Eng.*, pt. E, vol. 141, pp. 327–333, Nov. 1994.

[20] J. Jiang, "Novel design of arithmetic coding for data compression," *Proc. Inst. Elect. Eng. Comput. Digital Tech.*, vol. 142, no. 6, pp. 419–424, Nov. 1995.

[21] S. Kuang, J. Jou, and Y. Chen, "The design of an adaptive on-line binary arithmetic—coding chip," *IEEE Trans. Circuits Syst. I*, vol. 45, pp. 693–706, July 1998.

[22] M. Hsieh and C. Wei, "An adaptative multialphabet arithmetic coding for video compression," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 8, pp. 130–137, Apr. 1998.

[23] A. Mukherjee, N. Ranganathan, J. Flieder, and T. Acharya, "MARVLE: a VLSI chip for data compression using tree-based codes," *IEEE Trans. VLSI Syst.*, vol. 1, pp. 203–213, June 1993.

[24] R. M. Fano, *Transmission of Information*. Cambridge, MA: MIT Press, 1949.

[25] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Trans. Inform. Theory*, vol. 21, pp. 194–203, Mar. 1975.

[26] Y. Liu, "Design and hardware architectures for dynamic huffman coding," *Proc. Inst. Elect. Eng. Comput. Digital Tech.*, vol. 142, no. 6, pp. 411–418, Nov. 1995.

[27] J. M. Cheng and L. M. Duyanovich, "Fast and highly reliable IBMLZ1 compression chip and algorithm for storage," in *Proc. Hot Chips VII Symp.*, Aug. 14–15, 1995, pp. 155–165.

[28] ALDC1-40S-M, IBM Corporation, IBM Microelectronics Division, New York, 1994.

[29] AHA3521 40 Mbytes/s ALDC Data Compression Coprocessor IC, Advanced Hardware Architectures Inc, Pullman, WA, 1997.

[30] How LZS Compression Works, Hi/fn Inc, Los Gatos, CA, 1996.

[31] 9600 Data Compression Processor, Hi/fn Inc, Los Gatos, CA, 1999.

[32] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inform. Theory*, vol. IT-23, pp. 337–343, 1977.

[33] Y. Surk, T. Young, and K. Park, "A novel PE-based architecture for lossless LZ compression," *IEICE Trans. Fundamentals*, vol. E80-A, no. 1, pp. 233–237, Jan. 1997.

[34] B. Jung and W. Burleson, "Performance optimization of wireless local area networks through VLSI data compression," *Wireless Networks*, vol. 4, pp. 27–29, 1998.

[35] E. Nusinov and J. Pasco-Anderson, "High performance multi-channel data compression chip," in *Proc. IEEE Custom Integrated Circuits Conf.*, 1994, pp. 203–206.

[36] J. Ziv and A. Lempel, "Compression of individual sequences via variable rate coding," *IEEE Trans. Inform. Theory*, vol. IT-24, pp. 530–536, 1976.

[37] M. Bianchi, J. Katto, and D. Van Maren, "Data compression in a half-inch reel-to-reel tape drive," *Hewlett-Packard J.*, vol. 40, no. 6, pp. 26–31, 1989.

[38] Primer: Data Compression Lempel-Ziv (DCLZ), Advanced Hardware Architectures Inc., Pullman, WA, 1996.

[39] AHA3211 40 Mbytes/s DCLZ Data Compression Coprocessor IC, Advanced Hardware Architectures Inc., Pullman, WA, 1997.

[40] S. Bunton and G. Borriello, "Practical dictionary management for hardware data compression," *Commun. ACM*, vol. 35, no. 1, pp. 95–104, 1992.

[41] DCP816, DCP Research Corp., Edmonton, Alberta, Canada, 1995.

[42] H. Sakanashi *et al.*, *Evolvable Hardware Chip for High Precision Printer Image Compression*, 1998, vol. 1478, pp. 106–114.

[43] M. J. Slattery and J. L. Mitchell, "The Qx-coder," *IBM J. Res. Develop.*, vol. 42, no. 6, pp. 767–784, 1998.

[44] S. Jones, "100 Mbit/s adaptive data compressor design using selectively shiftable content-addressable memory," *Proc. Inst. Elect. Eng.*, pt. G, vol. 139, no. 4, pp. 498–502, 1992.

[45] M. Kjelso, M. Gooch, U. Simm, and S. Jones, "Hardware data compression and memory management for flash-memory disks," in *Proc. ISIC-95, 6th Int. Symp. IC Technol., Syst. Applicat.*, 1995, pp. 161–165.

[46] M. Kjelso, M. Gooch, and S. Jones, "Design & performance of a main memory hardware data compressor," in *Proc. 22nd Eur. Micro Conf.*, Prague, Czech Republic, Sept. 1996, pp. 423–430.

[47] J. Nuñez, C. Feregrino, S. Bateman, and S. Jones, "The X-MatchLITE FPGA-based data compressor," in *Proc. 25th Eur. Micro Conf.*, Milan, Italy, Sept. 1999, pp. 126–133.
[48] J. L. Núñez and S. Jones, "The X-MatchPRO 100 Mbytes/second FPGA-based lossless data compressor," in *Proc. Design, Automation Test Eur., DATE Conf. 2000*, 2000, pp. 139–142.
[49] J. L. Bentley, "A locally adaptive data compression scheme," *Commun. ACM*, vol. 29, no. 4, pp. 320–330, 1986.
[50] V. Betz and J. Rose, "How much logic should go in an FPGA logic block?," *IEEE Design Test Comput.*, pp. 10–15, Jan.–Mar. 1998.

José Luis Núñez (M'00) received the B.S. degree from the Universidad de La Coruna, La Coruna, Spain, the M.S. degree from the Universidad Politécnica de Cataluña, Barcelona, Spain, both in electronics engineering, and the Ph.D degree from Loughborough University, Leicestershire, U.K., in hardware architectures for high-speed data compression, in 1993, 1997, and 2001, respectively.

In 1997, he joined the Department of Electronic Engineering at Loughborough University, where he is currently a Research Fellow. His research interests include lossless data compression, reconfigurable vector architectures, FPGA-based design and high-speed data networks.

Simon Jones (SM'99) is currently Dean of Engineering and Design at the University of Bath, U.K., where he leads a large research group addressing the design of special-purpose processors for data compression and Neural Networks. He also held the ARM/Royal Academy of Engineering Research Chair at Loughborough University, Leicestershire, U.K.

Dr. Jones is Chairman of the Inst. Elect. Eng. Professional Network on System-on-Chip and is currently Chairman of the IEEE for the U.K. and the Republic of Ireland.