Gigabyte per second streaming lossless data compression hardware based on a configurable variable-geometry CAM dictionary

J.L. Nunez-Yanez and V.A. Chouliaras

Abstract: A high-throughput lossless data compression IP core built around a CAM-based dictionary whose number of available entries and data word width adjust to the characteristics of the incoming data stream is presented. These two features enhance model adaptation to the input data, improving compression efficiency, and enable greater throughputs as a multiplicity of bytes can be processed per cycle. A parsing mechanism adjusts the width of dictionary words to natural words while the length of the dictionary grows from an initial empty state to a maximum value defined as a run-time configuration parameter. The compressor/decompressor architecture was prototyped on an FPGA-based PCI board. An ASIC hard-macro was subsequently implemented and achieved a throughput of more than 1 gigabyte per second when clocking at 277 MHz on a high-performance, 0.13 μm, eight-layer copper CMOS process.

1 Introduction

Lossless data compression, in which the original data are reconstructed precisely after decompression, is a technique that can bring significant benefits to a computing/storage system. Its applications have been increasing in number over recent years, fuelled by a combination of demand for increased bandwidth along with the need to improve storage capacity [1–5]. Lossless data compression has been successfully deployed in storage systems (tapes, hard disk drives, solid-state storage, file servers) and communication networks (local area network (LAN), wide area network (WAN), wireless). Many of these applications, such as storage area networks (SAN), utilise fibre-channel technology to interconnect high-capacity, high-speed disk arrays and the requirements for throughput and low latency directly influence the specification of any attached data compression hardware. During the past few years, our group has been developing a patented algorithm and associated hardware architecture known as X-MatchPRO, which enables the use of lossless data compression in these high-speed applications [6]. This work presents a very high throughput, ASIC (application-specific integrated circuit) hard-macro implementation named X-MatchPROVW, which now incorporates the capability to adjust the width of the dictionary to a value ranging from 2 to 4 bytes. The variable-geometry dictionary improves model adaptation to byte-based alphabets and enables the new internal run length coder to capture repeating phrases formed by up to 16 bytes in a single-output codeword. These enhancements called for a full architectural redesign and resulted in approximately 20% better compression ratios. In addition, the register transfer level (RTL) description was re-architected so as to include a configuration parameter for the generation of dictionaries of varying size, thus trading silicon area for compression performance. Finally, the newest architectural addition was the introduction of run-time capability to limit the maximum dictionary size from one up to the maximum number of entries physically available in the device. This is achieved via a special dictionary length configuration register (DLCR). This feature guarantees that an implementation with higher complexity (larger dictionary size) can decompress data generated with smaller dictionaries as long as the DLCR register is set to the value corresponding to the smaller dictionary. The dictionary starts empty with a dynamically generated dictionary address determined by the dictionary length and grows to the value set by the DLCR register. This means that a large dictionary can always emulate smaller dictionaries.

2 Background

Lossless data compression algorithms are typically classified as statistical-based or dictionary-based algorithms [7]. Research on statistical-based compression has focused on pushing compression levels to the theoretical limit via highly complex algorithms that, unfortunately, translate to low compression processing speeds such as the prediction by partial matching (PPM) class of algorithms [8–10]. In addition, the algorithmic complexity itself has resulted in only a few relatively simple hardware implementations of statistical-based algorithms [11–16]. Conversely, dictionary-based compression has concentrated on achieving high-throughput and good compression ratios and is based primarily around the two very popular LZ1 and LZ2 algorithms proposed in [17] and [18]. This is reflected in commercial hardware with products implementing the LZS (Lempel–Ziv Stac) [19], ALDC (adaptive lossless data compression) [20, 21] and DCLZ (data compression Lempel–Ziv) [22] algorithms. The LZS algorithm is a dictionary-based compression derivative of the LZ1

IEE Proceedings online no. 20045130

doi:10.1049/ip-cdt:20045130

Paper first received 11th October 2004 and in revised form 28th April 2005

J.L. Nunez-Yanez is with the Department of Electronic Engineering, University of Bristol, Bristol, UK

V.A. Chouliaras is with the Department of Electronic Engineering, University of Loughborough, Loughborough, UK

E-mail: j.l.nunez-yanez@bristol.ac.uk

algorithm. It can sustain up to 300 Mbyte/s throughputs in its latest hardware implementation named the 9630 [23]. It offers good compression ratios that typically reduce to half the original uncompressed size for multiple data types and it has become a *de facto* standard in network compression where a single data type cannot be identified. Popular router manufactures such as Cisco and Intel support LZS compression. Another successful lossless data compression method is the ALDC variant of the LZ1 algorithm originally developed by the IBM Corporation. A hardware implementation of the ALDC algorithm, developed by Advanced Hardware Architectures Inc. (AHA) achieved 80 Mbyte/s, while clocking at 80 MHz. The ALDC algorithm was extended to a parallel implementation in the Memory eXtension Technology (MXT) by IBM [24, 25]. MXT is designed to double the main memory capacity of high-performance servers and relies on an extra compressed level in the memory hierarchy to hide the extra latency introduced by the compression and decompression processes. The compression part of MXT uses four ALDC-based cores working in parallel in different data sections while sharing a common dictionary. A shared dictionary improves compression because more information is available to each core to model the input data. On the other hand, this method does not support incremental transmission (streaming) and suffers from significantly higher latencies because the outputs of all the cores have to be made available prior to adding a header. The individual bitstreams are then concatenated into a combined bitstream, ready for transmission. Decompression has a similar limitation because the whole compressed block must be received before the header can be removed and decompression started. The DCLZ algorithm is a variation on the LZ2 algorithm originally developed by Hewlett-Packard and is today being commercialised by AHA. The device clocks at up to 40 MHz for a throughput of 40 Mbyte/s.

Other research in the area of high-throughput hardware-based lossless data compression has focused on using a number of simple processing elements, organised as a systolic array, to increase the throughput of the Lempel–Ziv algorithm [26–29]. Systolic arrays offer high throughput, because the simple processing elements can be clocked at very high speeds, but the latency typically increases linearly with the number of processing elements. Our own work is based on the X-MatchPRO algorithm, which belongs to the category of dictionary-based compressors that are not LZ derivates. X-MatchPRO originated from the need for very high throughput and low latency lossless data compression. High throughput is achieved by processing multiple bytes per clock cycle, while low latency mandates a pipelined microarchitecture of short length in which transmission of a compressed symbol exiting the compressor data path is performed immediately. The same requirements apply to the decompression pipeline.

3 X-MatchPROVW overview

Figure 1 shows the complete X-MatchPROVW algorithm pseudo-code. The use of a parallel architecture yields a four-fold increment in compression throughput for the same frequency. In addition, the adoption of a five-stage pipeline keeps latencies to a minimum. Compared to MXT, there is only one output stream, resulting in compressed symbols being ready for transmission straight after exiting the compression pipeline and without the need for further assembling. The decompression architecture mirrors this approach and supports incremental reception. Our experiments have shown that the compression ratios achieved with X-MatchPRO are comparable to the LZ algorithm when processing data in machine-readable form (binary), but are significantly worse when the objective is to compress human-readable data (text, html). This is because X-MatchPRO disarranges data that principally exhibit 1-byte granularity rather than the 4-byte granularity searched by the parallel engine. LZ derivatives that process 1 byte at a time can exploit this data feature to locate and eliminate redundancy, thereby providing better compression ratios. It is feasible, however, to exploit a second level of granularity [30] at the natural word level where the number of bytes varies from 1 to 7 bytes per natural word. It is then possible to devise a variable width (VW) dictionary that parses the input to natural words with different lengths instead of a fixed 4-byte (tuple) length. The typical parser for human readable data is the space (ASCII code 32), which is the most common code in data formats such as text or html. Its usage in binary data is much less frequent and ASCII codes 0 and 255 are the most common characters [7]. The VW method can still achieve a high throughput because it processes multiple bytes per cycle and it increases compression because the likelihood of finding a match in the dictionary increases. This technique combines with the adjustment of dictionary length by using a phased binary code (PBC) [31] for the match locations. This means that the geometry of the dictionary varies in its two dimensions depending on the input data. The variable-geometry dictionary works together with the partial matching technique [32] of full 4-byte tuples to offer significant improvement in compression ratios as shown in the following sections. Partial matching is only applied to non-parsed 4-byte words (i.e full words) because it requires at least 2 bytes to match, and partial matches of partial words (i.e. a word with fewer than 4 bytes) offer very limited compression benefits but increase hardware complexity considerably. This means that matches are considered valid when there is a full match of a partial or full word or a partial match of a full word, but never a partial match of a partial word.

4 X-MatchPROVW method

The parsing algorithm analyses four input bytes (tuples) simultaneously and outputs a mask indicating the result of the parsing operation together with the search data for the dictionary. There are five different possible parsing results: the first four cases are generated depending on which byte contains the parser and the fifth case is used when the parser is not found in the input tuple, so a full 4-byte word is generated. If the parser is found at the MSB (most significant bit) of the input tuple, the length of the natural word is 1 byte. This minimal natural word is not searched for in the dictionary because the address width is typically larger than 8 bits and data expansion will take place. Instead, it is treated directly as a miss and coded with a single bit set to 1 to indicate a miss plus a Huffman code of only 1 bit. This alternative procedure efficiently codes the 'orphan' space by replacing the 8-bit code by a 2-bit code. The orphan space is not inserted in the dictionary, so the minimum dictionary width is 2 bytes. The other four possible parsing results are searched in the dictionary and each will generate either a match or a miss.

A successful match produces an output where the search data have been replaced by a pointer to the dictionary location where the match was generated, preceded by a single bit indicating that match. The dictionary location pointer is coded using a PBC. The PBC is a technique used to code the locations of a dictionary that starts empty

```
Set the dictionary to its initial state;
Set next free location counter = 1;
Run length count = 0;
DO {
        read in tuple T from the data stream;
    parse tuple T in natural word W;
    search the dictionary for word W;
    IF ( W length  != 1) {
        IF ( full hit )  {
            increment run length count by one;
            IF ( hit at location zero)  run length active at location zero; }
        ELSE {
            IF  (run length count = 1) {
                output '0';
                output phased binary code for match location = old match location;
                output Huffman code for match type = full match; }
            IF ( run length count > 1) {
                output '0';
                IF (run length at zero) {
                    output phased binary  code for match location =  0;
                    output Huffman code for match type = run length;
                    output Binary code for run length using 8 bits; }
                ELSE {
                    output phased binary  code for
                    match location = old match location;
                    output Huffman code for match type – run length;
                    output Binary code for run length using 2 bits; } }
            set run length count to 0;
            IF (partial hit) {
                determine the best match location and the match type;
                output '0';
                output phased binary code for
                match location = current match location;
                output Huffman code for match type = current match type;
                output any required literal characters of word W; }
            ELSE {
                output '1';
                output Huffman code for miss type = current miss type;
                IF (word is terminated) {
                    output word W minus last  byte; }
                ELSE {
                    output word W; } } }
        IF (full hit)
            move dictionary entries 0 to match location-1 by one  location;
        ELSE {
            move all dictionary entries down by one location;
            increase  next free location counter by one; }
        copy word W to dictionary location 0;  }
    ELSE {
        output '1';
        output Huffman code for miss type = single byte miss type; } }
WHILE (more data is to be compressed);
```

Fig. 1 *X-MatchPROVW*

and then grows as new data is processed. The advantage is that a smaller dictionary uses fewer bits to code its positions so there is a compression gain during the growing stage. A match type code is used to signal which bytes were found in the match location, whereas non-matching bytes are added in literal form. The match types are coded using specially generated Huffman codes [33, 34], which improve compression by assigning fewer bits to the more popular types.

Table 1 shows the different possible match types and the corresponding Huffman codes. This table was obtained after extensive simulation using representative data sets [35]. For example, match type '1110' means that the three MSB were found in the dictionary; this event would be the result of matching the search word mask with the dictionary data mask. The full match '1111' is the most popular match type, so a single-bit Huffman code is assigned to it. Less popular matches are matches of non-consecutive bytes, so the resulting Huffman codes are longer. Match types '1001', '1010' and '0101' do not have Huffman codes assigned to them because their chances of occurrence are too low and will be coded as misses. The priority column in Table 1 indicates which match type is more beneficial from a compression point of view and it would be selected first if a search generates a plurality of possible match types.

Table 1: Valid match types

Match type	Match description	Priority	Huffman code	Code length, bits
1111	full match	1	1	1
1110	3 MSB match	2	010	3
0111	3 LSB match	3	000	3
1101	any other 3 match	4	001111	6
1011			001110	
1100	2 MSB match	5	0010	4
0110	any other 2 match	6	001101	6
0011			001100	

Table 2: Valid miss types

Data type	Data length, bits	Huffman code	Code length, bits
_	8	1	1
a_	16	001	3
ab_	24	0001	4
abc_	32	0000	4
abcd	32	01	2

If the search operation fails to find a match in the dictionary, a miss is generated. The format of the output is a miss type code that signals the number of non-matching bytes following in literal form, preceded by a single bit indicating a miss. Table 2 shows the five possible miss types and associated Huffman codes. The most popular miss-type is the 'orphan' space, so it is given the shortest miss-type code of a single bit. Natural words with more than 4 bytes and phrases formed by more than one word are parsed over several dictionary locations. The Move-To-Front (MTF) [36] dictionary maintenance strategy, whereby a new word is placed at the front of the dictionary and the rest move down one position, maintains these long words and phrases over several consecutive dictionary locations. The overall effect is that multiple matches at the same dictionary location occur when a phrase or long word already present in the dictionary is received again. This event can be coded in a single run-length code where a single dictionary location code plus a number of repetitions are output. The run-length coding technique detects and codes repetitions of matches at any location. Match repetitions at location 0 are generated by the same data being received in the input stream; for example, a stream of spaces in a text file or constant background colour in an image file. Extensive simulation shows that run-lengths at location 0 are the longest ones and improved performance is obtained when 8 bits are assigned to code the number of repetitions. This means that up to 255 repetitions can be coded together. On the other hand, long words or phrases do not generate more than four or five repetitions at locations higher than 0, so only 2 bits are assigned to code the number of repetitions.

Figure 2 shows the format of the variable-length codewords output by the compression method. All the match codes indicate compression and the optimum compression ratio can be obtained when the same data are continuously repeated in the input data source such as when there is a constant background in a picture or the space character in a text file. The maximum compression ratio is then: CR = (1 bit for match + 1 bit for match location + 3 bits for match type + 8 bits for run length)/(255 * 32 bits for input data) = 0.0016. Using an 8-bit value for the run-length means that up to 255 repetitions can be coded in a single run. This means that an input data source of 10 000 bytes will be coded in 16 bytes. All the miss codes expand the original data source except for the code corresponding to the orphan space, as discussed previously. Expansion occurs if the compression ratio is larger than 1 with more output bits being produced than input bits received. The worst-case expansion can be measured as CR = (1 bit for miss + 3 bits for miss type + 32 bits for literal bytes)/(32 bits for literal bytes) = 1.094. This means that an input data source of 10 000 bytes will be coded in 10 940 bytes. Expansion can only be avoided by buffering the compressed data before transmission so the uncompressed version can be sent if expansion has taken place.

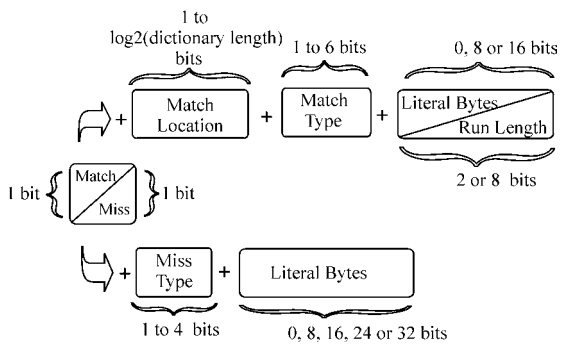

Fig. 2 *Codeword format in VW*

5 X-MatchPROVW example

Figure 3 compares the search mechanism and dictionary maintenance for a non-parsed and a parsed dictionary with 4-byte-wide locations. In both cases the dictionary is shown in cycle 1 as having already processed the sentence *'it is your choice to'*. Spaces are represented with the symbol '_' to facilitate the understanding of the example. The crossed byte locations indicate empty locations in the parsed dictionary. The sentence takes seven dictionary locations in the parsed version and five dictionary locations in the non-parsed version. The example assumes that the sentence *'choice to'* is received as the new data to be processed. The non-parsed version searches in cycle 1 for *'choi'* and in cycle 2 for *'ce t'*. The searched data, although present in the dictionary, are not located in the right positions and two consecutive misses are generated that will result in data expansion. For example *'c'* and *'h'* should be at byte position 3 and 2 in the same dictionary location of the dictionary, but they have been stored at byte positions 0 and 3 at dictionary locations 1 and 3. The MTF dictionary maintenance policy adds the missed data at the top of the dictionary and the rest of the data move down by one location each cycle. On the other hand, the parsing version arranges the data in a way that two consecutive full matches are generated with search data *'choi'* and *'ce'*. The MTF maintenance policy moves the data from the match location to the top of the dictionary. This means that the match location in cycles 1 and 2 is the same (location 2) and this fact can be readily exploited by generating a single codeword with a run-length at position 2 with the appropriate length. Additionally, the data searched in cycle 3 *'to'* (not shown in Fig. 2) will further extend the run-length at location 3 and further improve compression.

6 Hardware architecture

The architecture of the core consists of three major components: the modelling unit, the coding/decoding unit and the packing/unpacking unit. The packing unit function is to pack the variable-length code result of the compression operation into fixed-length codes that depend on the width of the compressed output bus. The unpacking unit performs the inverse operation. The next sections describe the modelling, coding/decoding and packing/unpacking units in more detail based on a dictionary with a maximum length of 16 locations as prototyped on the FPGA board as proof-of-concept. Larger dictionaries are required to obtain LZ-equivalent compression levels and the optimal length from a compression point of view is 1024 entries.

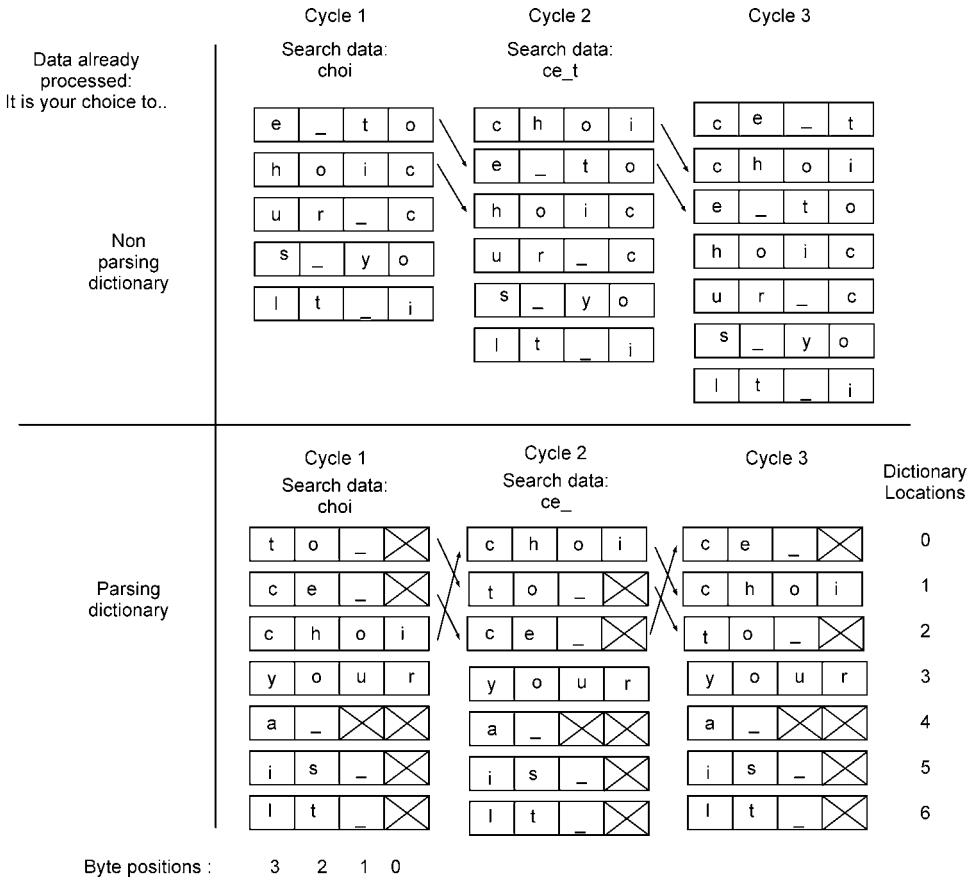

Fig. 3 *X-MatchPROVW example*

6.1 Modelling architecture

6.1.1 Compressor: Figure 4 shows the architecture of the modelling unit of the compressor. An input buffer is used, because, due to the nature of the VW algorithm, fewer than 32 bits of data may be processed per cycle. The parsing unit detects the presence of the parser character in the 32-bit input data bus and generates the appropriate mask bits for the rest of the pipeline. The modelling unit is based on a CAM dictionary of size *32*max dictionary length*, in which the data are stored. The CAM unit is able to perform search and adaptation operations in a single cycle for optimal speed. The columns in the CAM array can be configured as shift registers to implement the MTF maintenance policy described previously.

The VW method uses a mask associated with each dictionary location that tags those bytes in the word that are valid. The mask array must be stored in a CAM with the same structure as the data dictionary. The size of the mask array is *4*max dictionary length*. The priority logic and match decision logic select the best compression match, using the results of the search operation in the dictionary, and forward this information to the coding unit (discussed in the next section). The full match detection unit uses the match information plus the same length information to detect full matches in the dictionary. Owing to the variable-width dictionary locations, a match type such as '1100' could mean either a partial match of the two MSBs or that a full match in a partial word of only 2 bytes has been detected. The full match detection logic resolves this ambiguity and generates the appropriate signals for the adaptation logic and the run-length coding logic. In order to achieve this, the full match detection logic receives the same length data that contains three vectors, each one indicating a same length of 2 bytes, 3 bytes and 4 bytes. The full match detection unit compares the match result and the same length data to issue full matches. For example if the match result is '1100' and both search data and dictionary data have a same length of 2 bytes, a full match has been found. Similarly, full matches are issued when the match results are '1110' and '1111' and the same length of 3 bytes and 4 bytes are active, respectively. All the other cases do not result in a full match.

The adaptation logic implements the MTF maintenance policy, generating an adaptation vector that will shift the dictionary and mask data down. New data are inserted at the top of the dictionary while old data move down one position until the location where a full match (if any) was detected. The rest of the dictionary remains untouched. Misses or partial matches effectively move down the entire dictionary, evicting the data located at the bottom of the dictionary. The Out of Date Adaptation logic (ODA) is used to break the feedback loop present in the search and adaptation operations, enabling a higher clocking rate. Inserting a simple pipeline register will adversely affect compression performance, because it would not be possible to avoid duplicating the same dictionary data in several dictionary positions. ODA means that dictionary adaptation at time $t+2$ takes place using the adaptation vector generated at time t, but it is designed to guarantee that data duplication is restricted to position 0 thereby maintaining dictionary efficiency.

6.1.2 Decompressor: The modelling unit of the decompressor is depicted in Fig. 5. It receives the match (dictionary) location, mask data, match type and literal

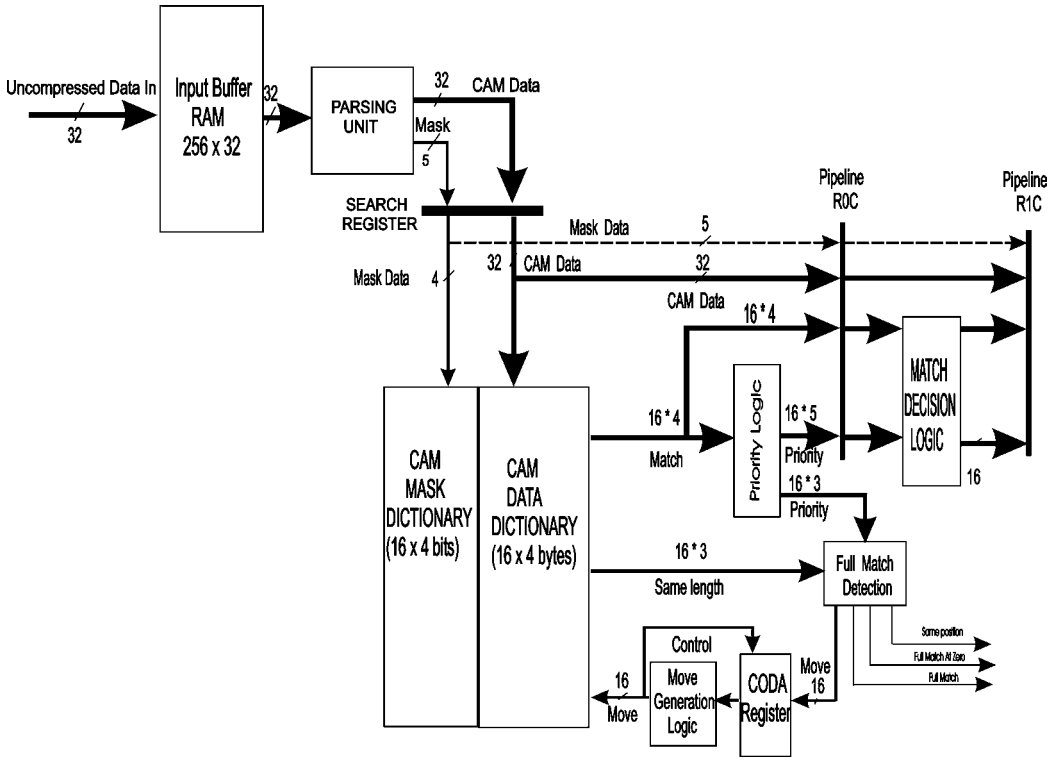

Fig. 4 *Compressor model architecture*

data from the decoder and uses this information to reconstruct the original data element. The decompression dictionary utilises a standard synchronous RAM instead of a CAM, because the received match location is used as the SRAM memory addresses. The pointer array logic performs an indirection function over the read and write addresses prior to accessing the RAM dictionary. It models the MTF maintenance policy of the compressor CAM dictionary by updating pointers instead of moving data. The pointer array enables the mapping of the CAM dictionary to RAM for decompression. Otherwise an extra shift register array would have been needed for the decompression dictionary, thereby increasing the logic complexity of the implementation. Similarly to the compression dictionary, the decompression dictionary stores data and mask information. The number of storage elements in the RAM memory is the same as that of the CAM memory. The output tuple assembler uses the literal data, match type, dictionary data and mask data to output the original word with a variable width ranging from 8 to 32 bits. These data are forwarded to the assembling unit, which performs the reverse operation to that of the compression

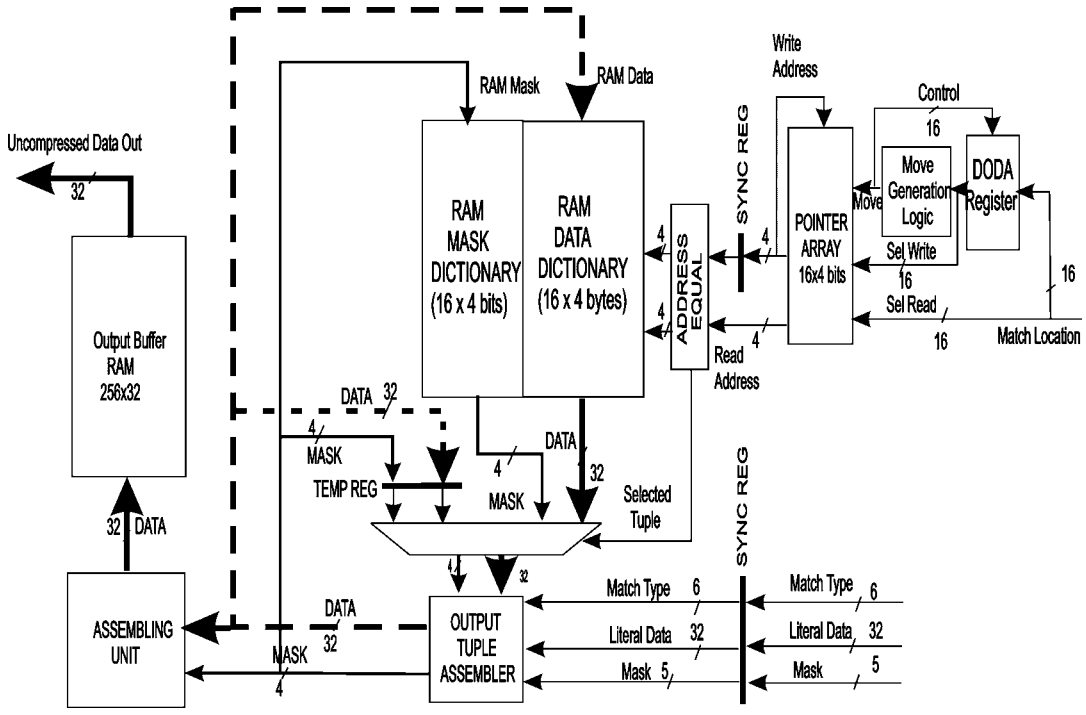

Fig. 5 *Decompressor model architecture*

parsing unit: it assembles the variable width words into fixed-length 32-bit words that are then written into the output buffer.

6.2 Coding/decoding architecture

6.2.1 Compressor: Figure 6 shows the coding architecture associated with the VW algorithm. There are two coders, namely the main coder and the run-length coder. The main coder monitors whether the signals being forwarded by the model correspond to miss or match events. The 16-to-4 encoder codes the unary match location vector into a more manageable log_2(*max dictionary length*) binary representation. This binary match vector is then further processed by the phased binary coder, which assigns a code whose length in bits is defined by how many dictionary locations are active in that coding event. The output is concatenated with the output of the match type coder generator. A second code concatenator *unit* selects between the match event output or the miss event output depending on the miss signal generated when the match type is not valid as defined in Table 1. In parallel to this process, the run-length coding logic monitors the full match signals being forwarded by the model. These signals detect repetitions of full match events at location 0 or at locations above 0.

If two or more full matches occur consecutively at the same dictionary location, the codeword corresponding to the first and second matches are removed from the pipeline and the coding logic stops producing codewords that will be coded as part of a run-length code. The coding event that stops the run length forces the RLI coding control unit to output the run-length codeword followed by the codeword for the event that stopped the run length.

6.2.2 Decompressor: Figure 7 shows the decoder architecture. It receives variable-length codewords of a maximum length of 35 bits (1 bit for miss, 2 bits for miss type and 32 bits for literal bytes), which are then processed in the main decoder to detect possible run-length codes and generate the match location, mask, match type and literal data combinations required to reconstruct the original data. The RLI decoding logic forwards this information to the modelling unit if a run-length code was not detected; otherwise it outputs the match location where the run-length was detected, together with the full match type as many times as indicated by the run-length codeword. There are two feedback paths that are not visible in Fig. 7, called *match width* and *set length to zero*; these paths carry information back to the unpacking unit as this unit needs to know how many bits have been used in the previous decoding step in order to shift out old data and concatenate new data. This feedback loop is the performance limiting factor in the design because it is not possible to add a pipeline register without affecting functionality.

6.3 Packing/unpacking architecture

6.3.1 Packing architecture: Figure 8 shows the packing architecture. The bit assembly logic assembles the variable-length codewords produced by the coder into 64-bit fixed length codes than are then output to the width adaptation logic. A 98-bit register is necessary because in the worst case there could be 63 bits in the buffer waiting to be output and a 35-bit codeword could be generated ($63 + 35 = 98$). The maximum codeword of length 35 is obtained with 1 bit for miss + 2 bits for miss type + 32 bits of literal data = 35 bits. The active code length is stored in a 7-bit register. The 64-bit codeword is then forwarded to the width adaptation logic that 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 bits to a more manageable 32 bits. It contains a total of 2 kbytes of fully synchronous dual-port RAM organised in two blocks of 256 × 32 bits

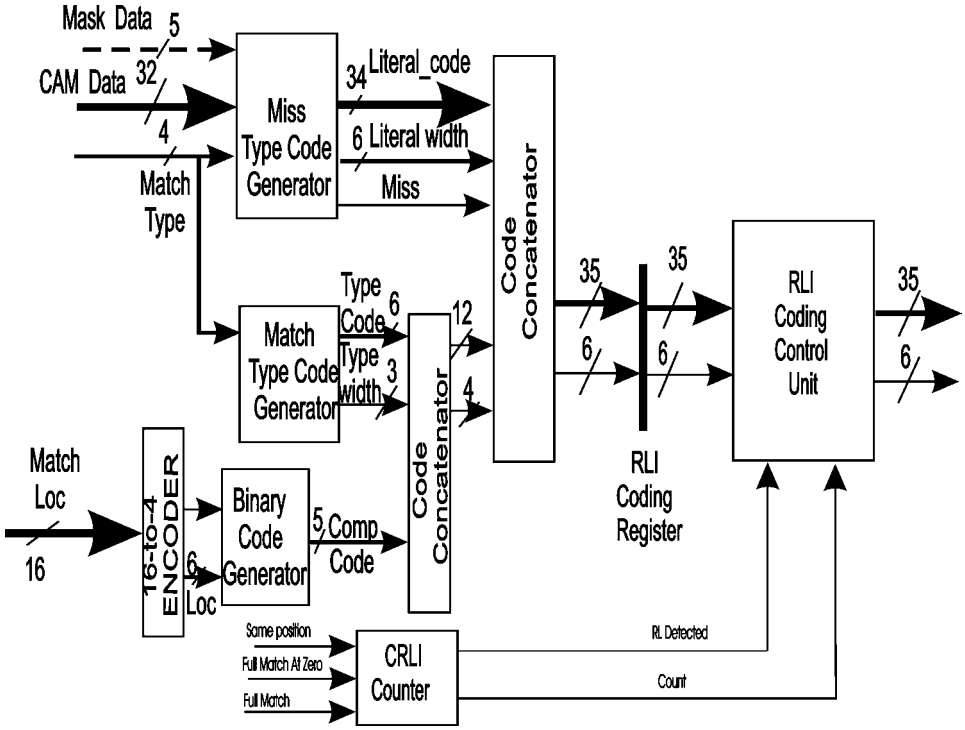

Fig. 6 *Coder architecture*

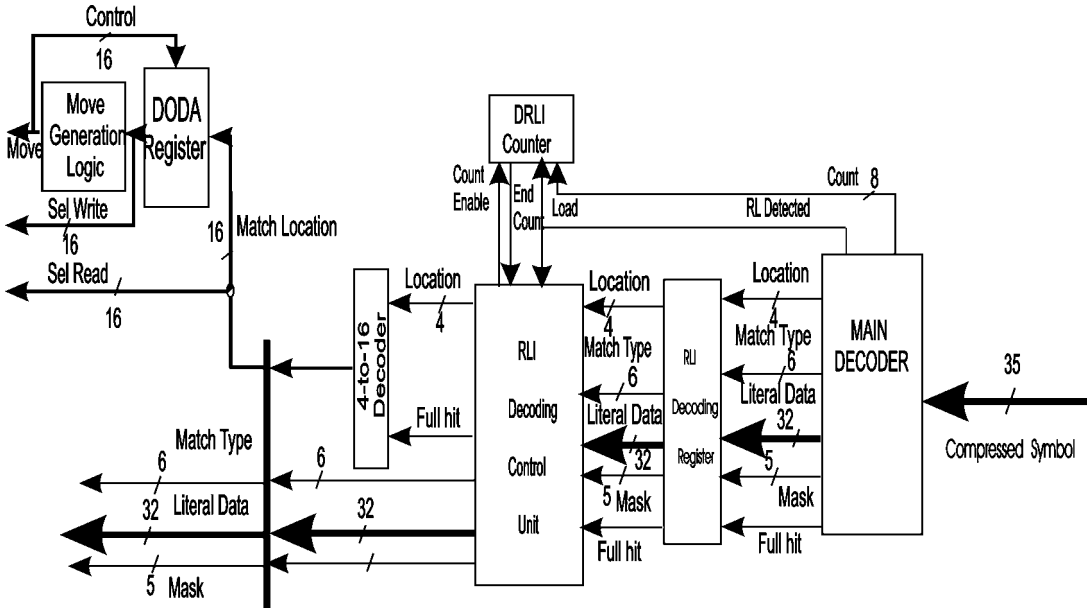

Fig. 7 *Decoder architecture*

to buffer compressed data before being output to the compressed data out bus.

6.3.2 Unpacking architecture: Figure 9 shows the unpacking architecture. The bit disassembly 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 forms a feedback loop difficult to improve. The 64-bit words are provided by the width adaptation logic that performs the equivalent but opposite function as in the packer. It reads in 32 bits of compressed data from the input compressed bus and it writes out 64 bits 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 kbytes of fully synchronous dual-port RAM organised in two blocks of 256 × 32 bits each as in the packer. The design uses a technique where decoding takes place in parallel to concatenation of new data to improve speed. This means that the concatenation of new data must take place before the number of bits decoded in the current cycle is known. In order to guarantee that the next decoding cycle can take place, enough bits must be left in the register in case a maximum number of bits are consumed in the current cycle. The maximum number of bits that can be consumed is 35, so concatenation of new data must take place if fewer than 70 bits are valid in the register. If there are 70 bits valid and the current cycle consumes 35 bits, then 35 bits will be left for the next cycle and the decoding operation can carry on uninterrupted. Because 64 bits are added to old data when the number of valid bits is less than 70, the decoding register is extended to 133 bits (69 + 64 = 133 bits).

7 Performance analysis

There are two variables that define the performance of a data compression architecture, namely the average compression ratio and the throughput it achieves on representative data sets.

7.1 Compression efficiency

We selected three data sets for the compression efficiency analysis. The Calgary and Canterbury [34] data sets are

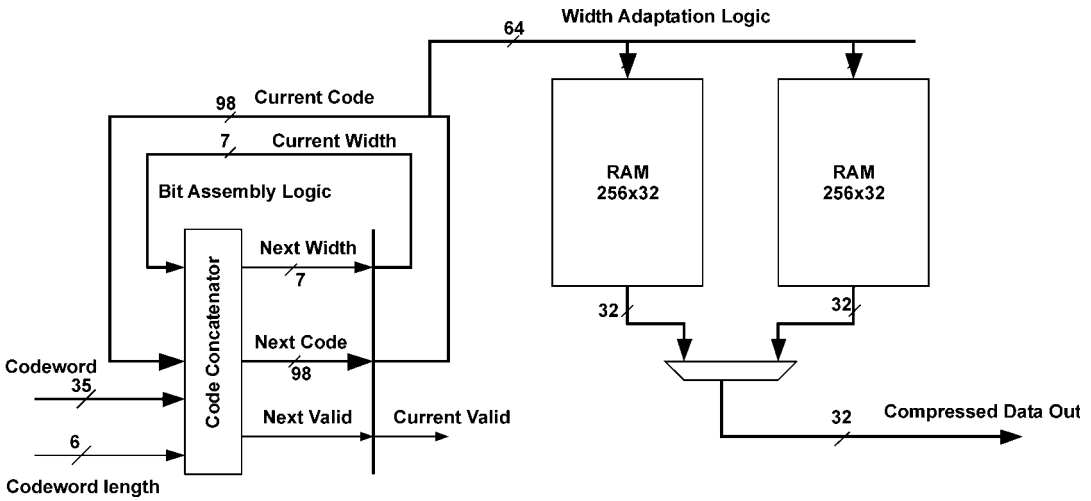

Fig. 8 *Packing architecture*

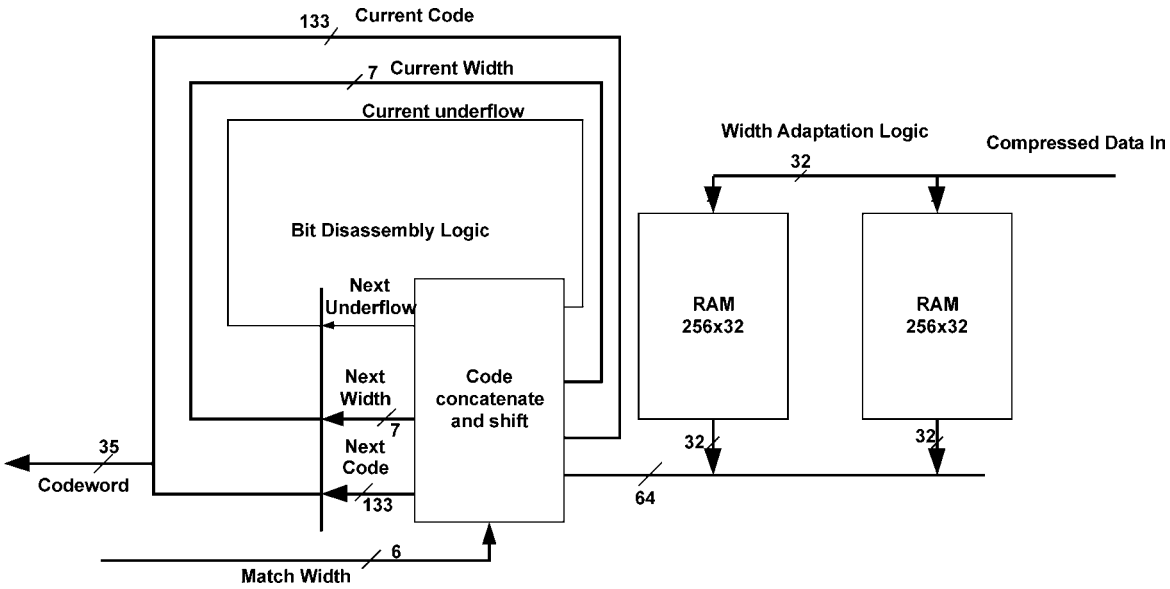

Fig. 9 *Unpacking architecture*

standards used by the data compression community, and the memory data set was developed within the research group to enable studying compressibility of main memory in a computer system.

The Canterbury data set was introduced to replace the ageing Calgary data set and includes representative data found in modern computer systems. Both data sets are biased to data that is textual in nature such as book chapters, poetry, C and Lisp source code and html web pages, but they also include application data, spreadsheet files and fax images.

Figures 10 to 12 show the compression results comparing the parsing (X-MatchPROVW_1024) and non-parsing (X-MatchPRO_1024) versions of the X-MatchPRO algorithm against the three hardware implementations of the algorithms we reviewed in Section 2. These are the ALDC developed by IBM, the DCLZ developed by Hewlett-Packard and the LZS developed by HiFn. These devices are representative of the fastest and best compression technology available today. The dictionary size was increased to the maximum allowed in each algorithm in order to obtain the best compression performance from each of them. Such maximum values are up to 2048 locations for the LZ algorithms and 1024 locations for X-MatchPRO. The horizontal axis indicates the block size; input data are processed as blocks and the dictionary is reset between blocks. This means that no history information is kept from the compression of one block to the compression of the next block. The vertical axis measures the compression performance as a ratio of output bits to input bits, so the smaller the value the better the compression.

The two standard data sets show a similar data compression trend. The non-parsing version of the X-MatchPRO algorithm is the worst performer for all data blocks with approximately a 17% degradation relative to the VW version. The textual nature of these data sets explains the better performance of the variable-width algorithm, which is able to adjust its width to the natural word width of the data source. The non-parsing version uses a width fixed to 32 bits that works well in the machine-readable data subset, but performs poorly for human-readable data. The VW algorithm using the 1024-entry dictionary achieves compression levels similar

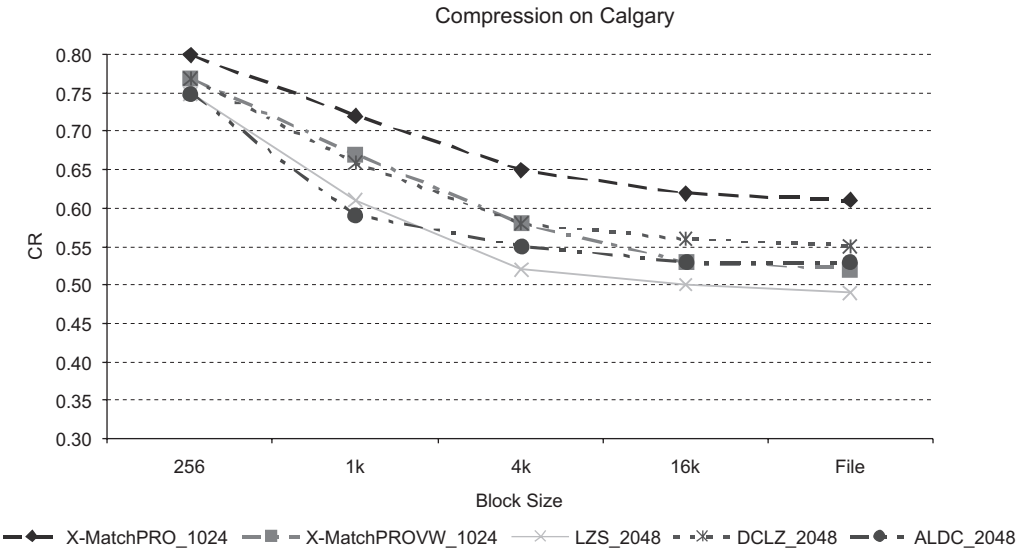

Fig. 10 *Performance using the Calgary data set*

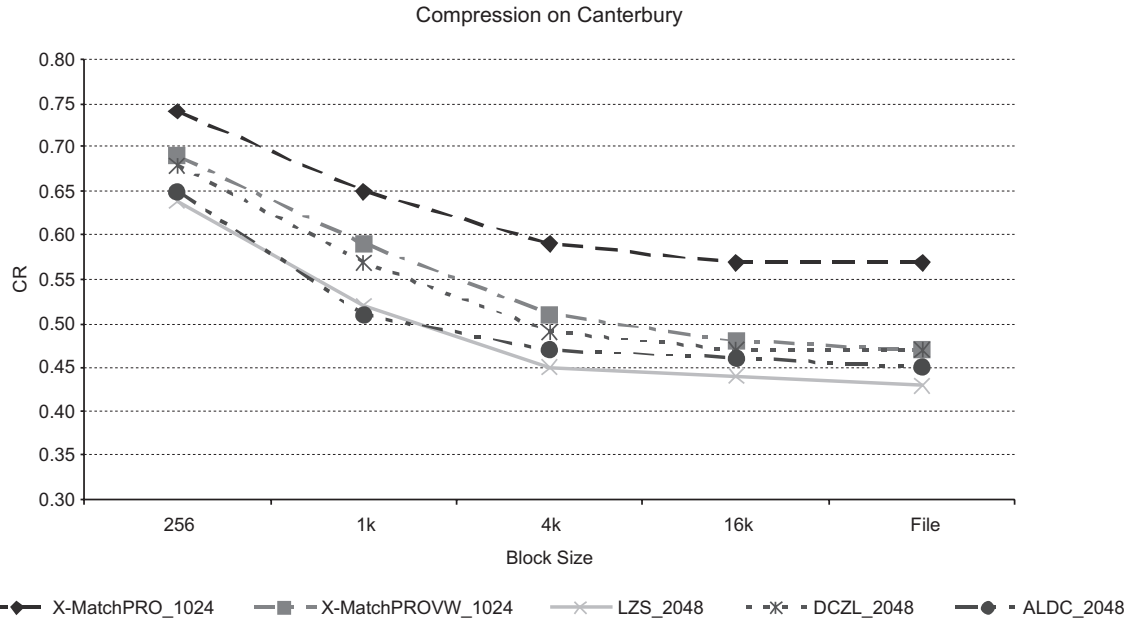

Fig. 11 *Performance using the Canterbury data set*

to the other three commercial algorithms. This translates to compression ratios of slightly better than 0.5 for block sizes greater than 4 kbytes. Compression performance for the 16-entry dictionary is around 0.65. For data blocks larger than 4 kbytes, a saturation effect is noticeable in all algorithms. It can also be observed that the VW algorithm demonstrates improved performance relative to the other algorithms with increasing block size. The reason is that a VW dictionary needs more data to be generated effectively, because up to four LZ dictionary locations can be stored in a single VW location. This means that, in general, the VW algorithm needs a larger data window to achieve optimal performance. The memory data set of Fig. 12 shows that the two X-MatchPRO variants achieve very similar performance levels. This data type has a 32-bit granularity because it is formed by data captured directly from main memory in a 32-bit UNIX workstation while running applications such as EDA tools and web surfing. Under these conditions, the VW algorithm gracefully returns to a non-parsing operational mode. The LZ1 algorithms achieve identical levels of compression, whereas the LZ2 algorithm underperforms the rest for all block sizes.

7.2 Compression throughput

The non-parsing version of the algorithm processes 4 bytes per cycle independently of the data source, which is equivalent to a throughput of 200 Mbyte/s when clocking at 50 MHz. The VW algorithm has a throughput that is data-dependent, because it will parse the input data in data words ranging from 1 to 4 bytes. A natural word of length greater than 4 bytes will be parsed into a number of 4-byte words plus a partial word ranging from 1 to 4 bytes. To evaluate the effects of parsing on data throughput, the average number of bytes processed per clock cycle was measured using the same data sets as for the compression efficiency experimentation. Typically, the throughput on data sets that are textual in nature is around 3.5 bytes per cycle, whereas almost 4 bytes per cycle are obtained for binary data sets.

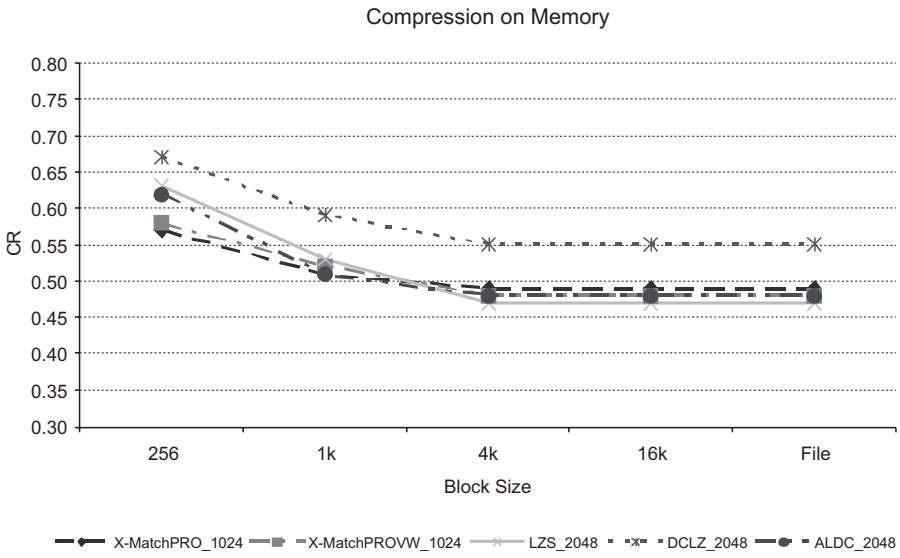

Fig. 12 *Performance using the Memory data set*

We calculated an arithmetic mean of 3.7 bytes per cycle, which translates to 185 Mbyte/s when clocking at 50 MHz. These figures are independent of input data block size.

8 Hardware implementation

Both versions of the X-MatchPRO algorithm were prototyped and validated on an Altera APEX20KE device populating a PCI-based Altera development platform. The FPGA implementation achieved 50 MHz, which is equivalent to a throughput of around 185 Mbyte/s for the VW version of the algorithm. The complexity of X-MatchPROVW in the APEX technology for a dictionary of 16 locations is approximately 8 k FPGA logic cells. This is approximately 15% more than the non-parsing version whose complexity is 6.8 k FPGA logic cells. Most of the FPGA resources are used by the CAM dictionary, which typically accounts for 80% of the total gate count because it is implemented using flip-flops. The requirement to store the 4-bit masks together with the 32-bit dictionary words increases the dictionary size by 12.5%. Chip complexity increases by a factor of 1.5 each time the dictionary size is doubled. This means that the optimal dictionary size of 1024 locations requires approximately 100 k FPGA logic cells with more than 90% of these cells in the CAM dictionary. An ASIC implementation was subsequently undertaken for the UMC 0.13 μm, eight-layer copper process, using the flow outlined in Fig. 13 to measure the performance level obtained with ASIC technology. The design was originally synthesised in Synplify ASIC and then, read into the Synopsys Design Compiler for further logical netlist optimisation. It was then read into the Synopsys Physical Compiler tool and optimised for minimum physical constraints (MPC). The MPC (placed) netlist was then run through Place and Route on the Cadence Encounter platform to verify that the design was indeed routable.

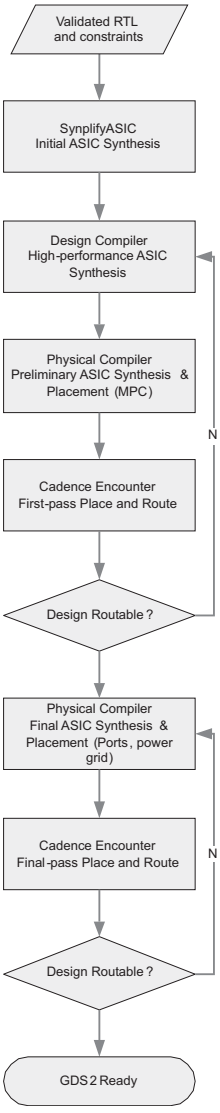

Fig. 13 *Physical implementation flow*

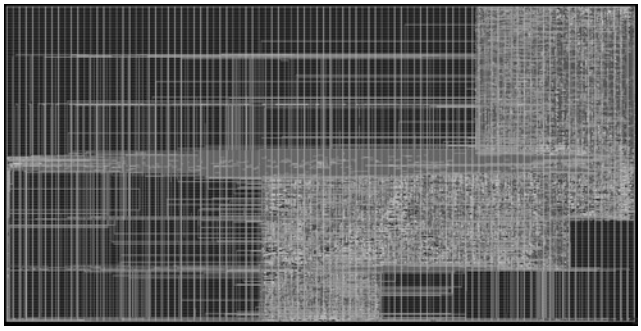

Fig. 14 *X-MatchPROVW layout*

Table 3: ASIC details

Physical parameter	Value
Std cells (hard macros)	31 856 (8 RAMs)
Dimensions	2 mm × 1 mm (2 mm^2)
Std Cell utilisation	58%
Fmax	273 MHz

Once the routability aspect of the design was achieved, the original logical netlist was re-read into the Physical Compiler once more, but now with real physical constraints applied. These constraints specified the utilisation factor, aspect ratio and die size, power ring dimensions, power trunks width and number, pin (port) location and, finally, the power straps characteristics. It was re-optimised and passed to Encounter for the final Place and Route run. Figure 14 depicts the final placed and routed database of the X-MatchPROVW algorithm. The characteristics of the hard macro are given in Table 3. The final hard-macro clocks at a conservative (for the process) 273 MHz. At this frequency, it achieves 1.092 Gbyte/s compression/decompression bandwidth with an initial latency of 14.64 ns and a pipelining rate of 3.6 ns. To the best of our understanding, this is the fastest streaming data compressor/decompressor available either in industry or academia today.

9 Conclusions

This paper presented the novel X-MatchPROVW lossless data compression algorithm and architecture based on a variable-width dictionary in which the input data is parsed into natural words of varying length instead of a fixed 4-byte length. Every dictionary entry has an associated mask that identifies the valid bytes in that position. The mask is stored in the dictionary together with the data, so a match can only be effective over valid data bytes. This new architecture achieves a high throughput, because it processes multiple bytes per clock cycle and increases compression as the likelihood of finding a match in the dictionary increases. Although alternative

definitions of word are possible, parsing is typically done with the space character. The method, therefore, increases the algorithm granularity from the classical 1 byte to that of the natural word, where the natural word length is defined as a maximum sequence of alphabetic characters or non-alphabetic characters limited by the space character. The physical realisation of the method limits the width of the dictionary location in hardware to 4 bytes, but the presence of the internal run-length coder and the move-to-front dictionary maintenance policy keeps the logical connection of a word that extends over several dictionary locations. Additionally, a phase binary coding technique is used so the number of valid words in the dictionary is determined by the degree of redundancy present in the input data source. An ASIC implementation was undertaken and the resulting hard macro achieved a throughput of more than 1 Gbyte/s in streaming data, while maintaining low latency. Automatic configuration of the maximum physical dictionary size (at compile time) and the maximum logical dictionary size (at run-time) enables compressed data blocks generated with small dictionaries to remain compatible with implementations using large dictionaries.

10 References

1 Dickson, K.: 'Cisco IOS data compression'. White Paper, Cisco Systems, 170 West Tasman Drive, San Jose, CA, 2000
2 Mitel Corp.: 'Data compression'. White Paper, Mitel Remote Access Solutions, Mitel Corporation, Mitel Networks, 350 Legget Drive, Kanata, Ontario, 2000
3 VanDuine, R.: 'Integrated storage'. Technical Paper, IBM Corporation, 3605 North Highway 52, Rochester, MN, 2000
4 Cressman, D.: 'Analysis of data compression in the DLT2000 tape drive', *Dig. Tech. J.*, 1994, **6**, (2), pp. 62–71
5 Hi/fn Inc.: 'Data compression performance analysis in data communications'. Application Note, Hi/fn Inc., 2635 Hopkins Court, Pullman, WA, 1997
6 Núñez, J.L., and Jones, S.: 'Gbit/second lossless data compression hardware', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2003, **11**, (3), pp. 499–510
7 Nelson, M.: 'The data compression book' (Prentice-Hall, 1991)
8 Cleary, J., and Witten, I.: 'Data compression using adaptive coding and partial string matching', *IEEE Trans. Commun.*, 1984, **32**, (4), pp. 396–402
9 Moffat, A.: 'Implementing the PPM data compression scheme', *IEEE Trans. Commun.*, 1990, **38**, (11), pp. 1917–1921
10 Cormack, G.V., and Horspool, R.N.S.: 'Data compression using dynamic Markov modelling', *Comput. J.*, 1987, **30**, (6), pp. 541–549
11 Boo, M., Bruguera, J.D., and Lang, T.: 'A VLSI architecture for arithmetic coding of multilevel images', *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, 1998, **45**, (1), pp. 163–168
12 Kuang, S.R., Joung, J.M., Chen, R.D., and Shiau, Y.H.: 'Dynamic pipeline design of an adaptive binary arithmetic coder', *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, 2001, **48**, (6), pp. 813–825
13 Jiang, J.: 'A novel parallel design of a codec for black and white image compression', *Signal Process. Image Commun.*, 1996, **8**, (5), pp. 465–474
14 Pennebaker, W.B., Mitchell, J.D., Langdon, G.G., and Arps, R.B.: 'An overview of the basic principles of the Q-coder adaptive binary arithmetic coder', *IBM J. Res. Dev.*, 1988, **32**, (6), pp. 717–725
15 Kuang, S., Jou, J., and Chen, Y.: 'The design of an adaptive on-line binary arithmetic-coding chip', *IEEE Trans. Circuits Syst. I, Fundam. Theory Appl.*, 1988, **45**, (7), pp. 693–706
16 Hsieh, M., and Wei, C.: 'An adaptative multialphabet arithmetic coding for video compression', *IEEE Trans. Circuits Syst. Video Technol.*, 1998, **8**, (2), pp. 130–137
17 Ziv, J., and Lempel, A.: 'A universal algorithm for sequential data compression', *IEEE Trans. Inf. Theory*, 1977, **IT-23**, pp. 337–343
18 Ziv, J., and Lempel, A.: 'Compression of individual sequences via variable rate coding', *IEEE Trans. Inf. Theory*, 1978, **IT-24**, pp. 530–536
19 Hi/fn Inc.: '9600 data compression processor'. Data Sheet, Hi/fn Inc., 750 University Avenue, Los Gatos, CA, 1999
20 Cheng, J.M., and Duyanovich, L.M.: 'Fast and highly reliable IBM LZ1 compression chip and algorithm for storage'. Hot Chips VII Symposium, 14–15 August 1995, pp. 155–165
21 Advanced Hardware Architectures Inc.: 'AHA3580 80 Mbytes/s ALDC data compression coprocessor IC'. Product Brief, Advanced Hardware Architectures Inc., 2635 Hopkins Court, Pullman, WA, 2001
22 Advanced Hardware Architectures Inc.: 'Primer: data compression (DCLZ)'. Application Note, Advanced Hardware Architectures Inc., 2635 Hopkins Court, Pullman, WA, 1996
23 Information available at www.hifn.com/docs/9630.pdf
24 Tremaine, R.B., Franaszek, P.A., Robinson, J.T., Schulz, C.O., Smith, T.B., Wazlowski, M.E., and Bland, P.M.: 'IBM memory expansion technology (MXT)', *IBM J. Res. Dev.*, 2001, **45**, (2), pp. 271–285
25 Tremaine, R.B., Smith, T.B., Wazlowski, M., Har, D., Mak, K., and Arramreddy, S.: 'Pinnacle: IBM MXT in a memory controller chip', *IEEE Micro*, 2001, **22**, (2), pp. 56–68
26 Henriques, S., and Ranganathan, N.: 'High speed VLSI design for Lempel–Ziv based data compression', *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, 1993, **40**, (2), pp. 90–106
27 Jung, B., and Burleson, W.P.: 'A VLSI systolic array architecture for Lempel–Ziv based data compression'. Proc. IEEE Int. Symp. on Circuits and Systems, June 1994, pp. 65–68
28 Jung, B., and Burleson, W.P.: 'Real time VLSI compression for high speed wireless local networks'. Data Compression Conf., March 1995
29 Storer, J.A., and Rief, J.H.: 'A parallel architecture for high speed data compression', *J. Parallel Distrib. Comput.*, 1991, **13**, pp. 222–227
30 Moffat, A.: 'Word-based text compression', *Softw.-Pract. Exp.*, 1989, **19**, (2), pp. 185–198
31 Jones, S.: 'High-performance phased binary coding', *IEE Proc., Circuits Devices Syst.*, 2001, **148**, (1), pp. 1–4
32 Jones, S.: 'Partial-matching lossless data compression hardware', *IEE Proc., Comput. Digit. Tech.*, 2000, **147**, (5), pp. 329–334
33 Huffman, D.A.: 'A method for the construction of minimum redundancy codes', *Proc. IRE*, 1951, **40**, pp. 1098–1101
34 Parhi, K.K.: 'High-speed VLSI architectures for Huffman and Viterbi decoders', *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, 1992, **39**, (6), pp. 385–391
35 Arnold, R., and Bell, T.: 'A corpus for the evaluation of lossless compression algorithms'. Data Compression Conf., 1997 pp. 201–210
36 Bentley, J.L., Sleator, D.D., Tarjan, R.E., and Wei, V.K.: 'A locally adaptive data compression scheme', *Commun. ACM*, 1986, **29**, (4), pp. 320–330