A Reliable LZ Data Compressor on Reconfigurable Coprocessors

Wei-Je Huang, Nirmal Saxena, and Edward J. McCluskey

CENTER FOR RELIABLE COMPUTING

Computer Systems Laboratory, Department of Electrical Engineering

Stanford University, Stanford, California 94305-4055

{weije, saxena, ejm}@CRC.stanford.edu

Abstract

Data compression techniques based on Lempel-Ziv (LZ) algorithm are widely used in a variety of applications, especially in data storage and communications. However, since the LZ algorithm involves a considerable amount of parallel comparisons, it may be difficult to achieve a very high throughput using software approaches on general-purpose processors. In addition, error propagation due to single-bit transient errors during LZ compression causes a significant data integrity problem. In this paper, we present an implementation of LZ data compression on reconfigurable hardware with concurrent error detection for high performance and reliability. Our approach achieves 100Mbps throughput using four Xilinx 4036XLA FPGA chips. We have also presented an inverse comparison technique for LZ compression to guarantee data integrity with less area overhead than traditional systems based on duplication. The resulting execution time overhead and compression ratio degradation due to concurrent error detection is also minimized.

1. Introduction

Lossless data compression provides an inexpensive method for optimizing resource utilization in communication and storage systems. Redundant information inherent in the source data is removed in a way that there is no loss after reconstruction. Many lossless data compression algorithms for achieving this objective have been proposed, such as Huffman coding [Huffman 52], Lempel-Ziv (LZ) data compression algorithms [Ziv 77, 78], and arithmetic coding [Rissanen 76]. In general, data size reduction is achieved by encoding long but frequently encountered strings into shorter codewords with the help of a dictionary. In Huffman and arithmetic coding techniques, source data statistics are required in advance to construct the dictionary. In contrast, LZ data compression is *universal* because the dictionary is constructed dynamically without any prior knowledge about the characteristics of the source data. LZ data compression has been widely used in various standards since the mid-80's, including the Unix compress functions, GIF compression format, and CCITT V.42bis modem.

The core computation of LZ-based data compression is to search for the maximum-length matching string in a dynamically constructed dictionary. This includes the sequential generation of a dictionary and extensive comparisons between source data and dictionary elements. However, the significant amount of comparisons sets a bottleneck on the encoder throughput, while the sequential dictionary construction raises a critical issue on the data integrity due to error propagation.

To speed up the massive comparisons on general-purpose processors, traditional software algorithms, including hash table lookup and tree-structured searching, have been developed and applied in storage systems [Nelson 96]. For real-time applications over several hundred-Mbps communication channels, coprocessors with dedicated parallel-processing architectures provide an alternative to achieve higher throughput. Two major hardware architectures for LZ compression are Content-Addressable Memory(CAM)-based approach [Jones 92][Lee 95] and systolic array approach [Ranganathan 93][Jung 98]. Implementations of these architectures on Application Specific Integrated Circuits (ASIC) achieve a very high throughput of hundreds of Mbps. However, in general, ASIC lack flexibility in implementing various functions on the same hardware.

Conversely, Adaptive Computing Systems (ACS) are able to obtain versatility and parallelism simultaneously due to the reconfiguration capability. Figure 1 shows a typical ACS architecture. The principal advantage of this architecture comes from the addition of the reconfigurable Field-Programmable Gate Array (FPGA) coprocessors. It provides the flexibility to optimize the system for different figures of merit such as cost, performance, and fault tolerance. From the performance perspective of LZ compression, the abundance of Configurable Logic Blocks (CLB) in current FPGAs [Xilinx 99] provides the capacity to realize highly parallel architectures to boost encoder throughput. From the cost perspective, optimized designs of different applications can be multiplexed in time on the same chip to achieve better efficiency as well.

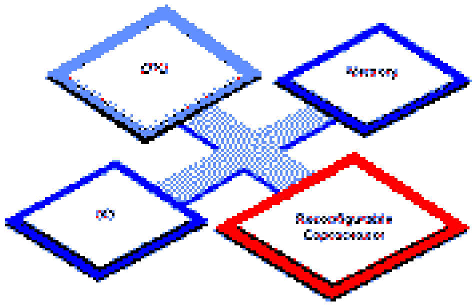


Figure 1: ACS architecture.

In addition to the advantages in performance and cost, ACS has emerged as an excellent candidate to achieve high dependability with small hardware redundancy [Saxena 98]. Once a fault location is diagnosed, the system can be reconfigured to avoid faulty parts and operate in a graceful degradation mode without replacing the whole chips. To achieve dependability, extensive research has been done both in locating faults in FPGA-based reconfigurable systems [Mitra 98][Das 99] and reconfiguring for fault tolerance [Hanchek 98][Emmert 97]. In addition, Concurrent Error Detection (CED) techniques can be used for detecting the presence of faults [Saxena 00]. As we discuss in Sec. 2.2, LZ data compression is prone to error propagation. Hence, CED is especially important for LZ compression in order to obtain high data integrity. For the traditional *N*-Modular Redundancy (NMR) CED approach, replication of hardware resources is required. This makes NMR very expensive for highly parallel architectures such as LZ data compressor. Therefore, we need to develop a cost-efficient CED scheme for our target application.

In this paper, we first present the implementation of a high-throughput LZ data compressor on reconfigurable coprocessors. Second, we develop a cost-efficient CED scheme on FPGAs. The organization of the paper is as follows. In Sec. 2, we briefly describe the principle of LZ data compression and the error propagation problem inherent to the algorithm. Two major hardware implementation approaches, CAM and systolic array, are introduced in Sec. 3. In Sec. 4, we propose a CED technique called *inverse comparison* based on the characteristics of LZ algorithm. In this section, we discuss the hardware overhead, fault coverage, and error recovery scheme related to the proposed CED technique. Section 5 describes the reconfigurable testbeds used in our experiments and presents the emulation results. Section 6 concludes the paper.

2. Overview of LZ Compression Algorithm

2.1 LZ Algorithm

The fundamental concept of LZ data compression is to replace variable-length phrases in the source data with pointers to a dictionary. Unlike Huffman coding, which uses a fixed dictionary optimized for known statistics of source data, LZ data compression uses previously encountered source strings to build the dictionary dynamically. Figure 2 shows the basic operation of the first version of LZ data compression, LZ77 [Ziv 77]. Initially, the dictionary array is empty. In each cycle, the latest source symbol and all the dictionary elements shift one position leftwards, evicting the oldest data symbol in the array. In this way, the dictionary always contains the most recent source phrases and is updated dynamically.

To encode, source data strings are compared with the current dictionary array to find the maximum-length-matching phrase. Once such a matching phrase is found, an output codeword C=(C_p, C_l, C_n) is generated. Each codeword contains three elements: the pointer (C_p) to the starting position of the matching phrase in the dictionary, the length (C_l) of the matching phrase, and the next source data symbol (C_n) immediately following the matching phrase. In practice, in order to reduce the number of bits necessary to encode the length component C_l, there is a limit on the maximum matching length allowed. In the next cycle following the generation of the output codeword, a new source data string enters the system, and a new matching process begins and proceeds in the same way until the source data is completely encoded.

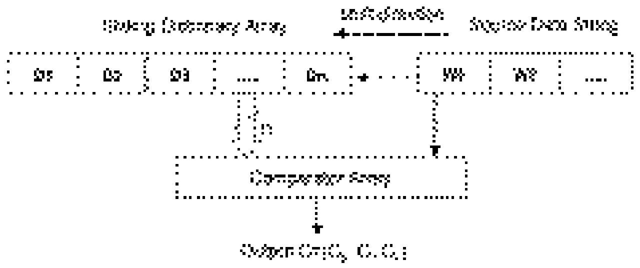


Figure 2: LZ77 encoder diagram.

Since the most recent data patterns are expected to appear again in the near future, we can often replace a long but frequently encountered string with a shorter code if the dictionary is large enough. A simplified example for the LZ77 encoding process is shown in Fig. 3. Here, we assume that the 16-element sliding dictionary is initially filled with "betbedbeebearbe ", and the next source data in the input window is "beta bets". The maximum matching length limit is set to be 7, which requires a 3-bit length component C_l in each codeword. The shaded cells in Fig. 3 represent current matches in the dictionary. In this example, the 9-symbol input string can be coded as two codewords: (0, 3, “a”) and (B, 4, “s”). If the source data is encoded in 8-bit ASCII code, the resulting output size will be 2*(4-bit C_p + 3-bit C_l + 8-bit C_n) = 30 bits. Compared to the original data of 9*8 = 72 bits, the compression ratio is 2.4:1, or 3.33 bits/symbol.

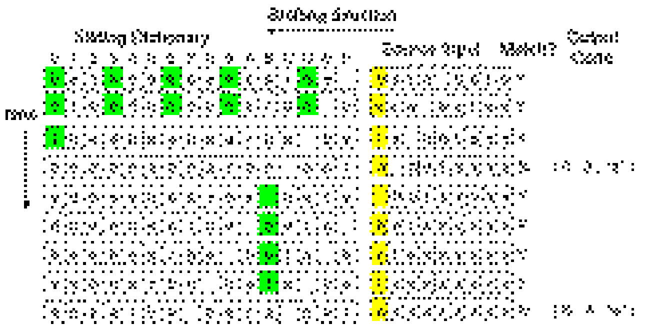


Figure 3: Example of LZ compression.

Unlike encoding, LZ77 decoding is a much simpler process. It is primarily composed of memory accesses only. A decoding dictionary is dynamically constructed in the same way as the encoding dictionary. In other words, once a data symbol is recovered, it is shifted into the decoding dictionary to synchronize the decoder to the encoding process. If an uncompressed codeword is received (C_l=0), the data is recovered by extracting the C_n element from the codeword. Otherwise, the decoder extracts the symbol from the current dictionary position C_p as the reconstructed data. This "extracting-shifting" process continues for C_l cycles to recover the whole maximum-length-matching string, and the next symbol C_n is then appended to it to complete a decoding cycle. Figure 4 illustrates the decoding process for the example in Fig. 3. The shaded cells represent the current extraction position in the dictionary. Note that since the dictionary is updated in the same way for both processes, the initial state of the decoding dictionary is identical to that of the encoder prior to the reception of the first codeword.

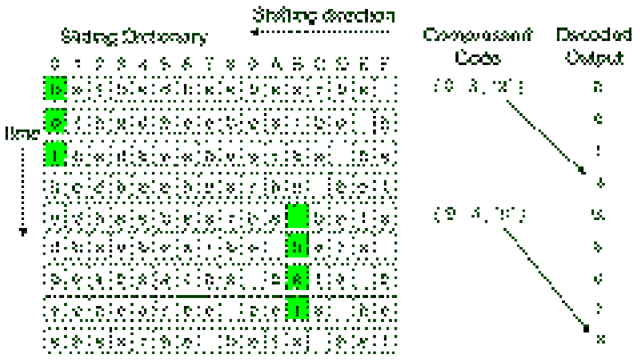


Figure 4: Example of LZ decoding.

2.2 Error Propagation in LZ Data Compression

High data integrity in lossless data compression techniques is extremely crucial because they are frequently applied in digital data coding with bit-precision requirements. However, LZ algorithm is prone to error propagation in the reconstructed data when the encoder is faulty. Consider a scenario where a transient fault in the LZ compressor causes a single bit-flip in C_p. In the decoding end, an incorrect string is referred to and used in both data reconstruction and dictionary update. In the worst case, when the decoding dictionary contains the corrupted phrases, successive reconstructed data may be damaged in an avalanche process even if all the subsequent encoding and decoding operations are fault-free.

For example, let us consider the same encoding and decoding cases in Fig. 3 and Fig. 4. Suppose the encoder hardware suffers from a transient failure that flips one bit of the C_p component in the output. This results in an erroneous compressed codeword (4, 3, "a") in the first output code. The effect in the decoding process is shown in Fig. 5. Here, the decoder makes an incorrect memory reference due to the faulty position pointer and extracts an erroneous string of length three. This string is shifted into the dictionary as if it were correct. Since the following codeword also refers to this incorrect string, the subsequent reconstructed data is corrupted accordingly. As a result, the single-bit error from the encoder causes 6 incorrect decoded symbols in a 9-symbol string. If the initial error occurs in a frequently encountered phrase, the error will spread out at a fast pace and corrupt a significant part of the dictionary.

Note that adding error control codes either *before* or *after* compression may not avoid this situation. Applying error control codes such as Cyclic Redundant Check (CRC) *after* the compressor can improve the system immunity to the noise and interference in communication or storage channels. However, they can neither correct nor detect any hardware failures that occur before them. Moreover, note that this corruption starts with an incorrect memory reference to the previously decoded data. Error control codes applied *before* compression may still be ineffective because these extracted data pass the checks in previous decoding cycles. Therefore, CED is needed in LZ data compressor to avoid the error propagation problem. We will examine the CED schemes in detail in Sec. 4.

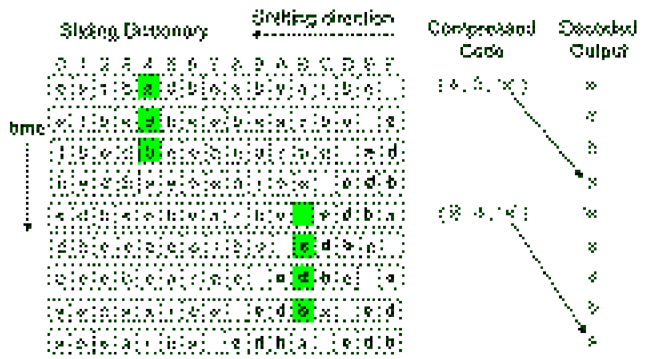


Figure 5: Example of error propagation in the reconstructed data.

3. Hardware Architectures

3.1 CAM-based Approach

A content addressable memory (CAM) cell is composed of a standard storage device and a comparator. With comparison functionality built into each cell, the CAM array can achieve massive parallelism in matching operations. Also, an incremental pointer representing the current write address in the CAM can be used to implement the shift operations in the sliding dictionary.

Figure 6 shows the architectural layout and the timing diagram of the CAM-based LZ encoder. The encoding process for each source symbol can be pipelined into two stages. In the first stage, a source data symbol is fed into the CAM array to be compared with all dictionary components. The match result of each cell in the previous cycle is propagated to the next cell to realize the string comparison. Also, the results are collected to encode the matching position pointer and determine the global *Matched* signal, which indicates whether the current maximum-length-matching phrase is found. In the second stage, a position encoder is used to determine C_p. The encoder is a priority encoder such that when several inputs are asserted in the same cycle, only the one with the highest priority is selected as the output. To simplify the design, priorities can be assigned in either an increasing or decreasing order with the cell index. In this cycle, the compared data also shifts into the CAM to update the dictionary, and the next source symbol enters the system to start the next comparison operation. The output stage follows immediately after the *Matched* signal is disabled.

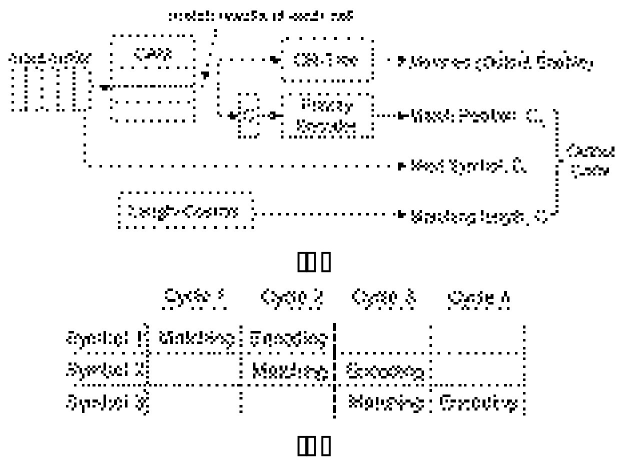


Figure 6: CAM-based LZ encoder: (a) architecture; (b) pipelining diagram.

The advantage of the CAM approach is the high throughput brought by massive parallelism. The processing rate of source data is 1 symbol/cycle. Compared to software approaches that takes at least *O(logN)* cycles to complete matching in an *N*-entry dictionary for one source symbol, the throughput of the CAM approach running with slower clock rates can outperform C-programs running on CPUs. For this reason, CAM-based LZ compression hardware is popular in many ASIC implementations, such as IBM ALDC compressor core [Craft 98].

However, one drawback for the CAM approach is the low resource utilization. For example, in the last five cycles in Fig. 3, the source string to match is " bets". Since only the cell in address *B* matches the first symbol of the string, all the other cells are disabled during the successive comparisons for the string. The effective utilization for the last four cycles in this case is thus 1/16 only, and it decreases as both the dictionary size and matching length increase. Since the probability of matching long phrases generally increases with the dictionary size, there is a tradeoff between compression ratio and CAM utilization.

3.2 Systolic Array Approach

In searching for the maximum-length-matching phrase in LZ data compression, source symbols have to be processed in the original order to ensure the correct sequential relationship after reconstruction. This leads to data dependencies among comparisons of successive source symbols and adjacent dictionary contents. As a result, systolic arrays of Processing Elements (PE) can be applied to achieve better area and power efficiency [Jung 98]. The idea is to separate the comparators from the CAM-based dictionary cells, and to balance the tradeoff between throughput and the number of array components. A special high-performance case proposed in [Jung 98] with the same throughput as the CAM-approach is shown in Fig. 7(a). In this architecture, the sliding dictionary is implemented in shift registers. The detailed PE structure is shown in Fig. 7(b), with the match result activated by its previous output to account for string matching.

The timing diagram of this high-performance systolic array LZ encoder is the same as that of the CAM approach in Fig. 6(b). The major advantage over the CAM approach is the flexibility of processing elements. Since PEs are detached from the memory cells, we can schedule idle PEs for other purposes, such as error detection, and increase the hardware utilization. We will examine this feature in Sec. 4.2 as we discuss the hardware overhead of CED.

3.3 Mapping on Reconfigurable Coprocessors

In both Fig. 6 and 7, the majority of hardware is consumed by the parallel matching units. The dictionary cannot be implemented on RAM modules, since each element needs to be retrieved for processing in each cycle. The recently released Altera APEX 20KE Programmable Logic Device (PLD) has built-in CAMs to achieve a better performance over discrete off-chip CAM approach [Altera 99]. For general PLD or FPGA without on-chip or off-chip CAMs, however, the CAM or shift register dictionary of a systolic array scheme can only be realized in flip-flops. Normally, synthesis of memory using flip-flops is not area-efficient. Therefore, hardware consumption in the dictionary usually dictates the overall number of FPGA or PLD chips required. This makes the dictionary size a critical balancing factor between compression ratio and hardware consumption.

Table 1 lists the average compressed percentage and the number of flip-flops needed in the dictionary of sizes ranging from 512 to 4096. The maximum matching length is set at 63. The average compressed percentage is measured over 18 Calgary text compression corpus files [Calgary 90]. The size of these files ranges from 12K to 769K bytes. Since modern FPGAs such as Xilinx Virtex and XC4000 series have capacities ranging from 1K to 20K flip-flops [Xilinx 99], the LZ data compressor can be implemented using a reasonable number of FPGA chips.

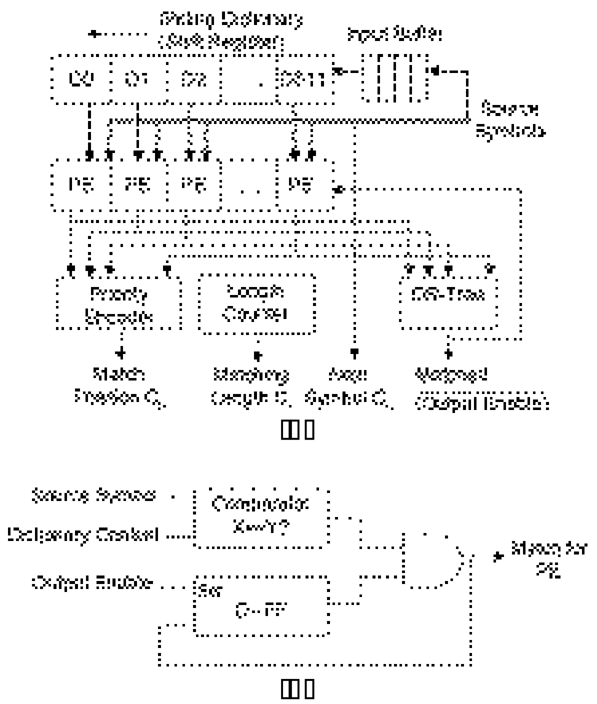


Figure 7: High-performance systolic array for LZ compressor: (a) architecture; (b) PE structure.

Table 1: Hardware consumption in LZ dictionary vs. average compressed percentage.

[illegible]	512	1024	2048	4096
[illegible]	4096	8192	16384	32768
[illegible]	38.67	43.21	49.5	52.96

4. Concurrent Error Detection Scheme

4.1 Inverse Comparison vs. NMR

In Sec. 2.2, we analyzed the error propagation problem in LZ data compression. In order to improve the data integrity and use the reconfiguration capability for fault tolerance, CED technique is required as a first step to detect the presence of faults. The traditional *N*-Modular Redundancy approach (NMR) replicates the hardware to perform multiple copies of identical computations. A comparator at the output stage detects if an error occurs. For a duplex system with two copies of identical computations, reliability can be improved by significantly reducing the undetected error rate [Saxena 98]. Nevertheless, the price of duplicating all hardware resources is very expensive, especially for the parallel architecture in LZ data compression.

For LZ compression, the uniqueness of its inverse transformation enables another way to verify the output integrity. For the encoder, the fundamental reliability requirement is to ensure that the source data can be reconstructed in the decoder. If we perform the inverse process on the encoder outputs, the resulting data should perfectly match the original source input. Therefore, instead of performing the same computation twice in the duplex system, data integrity can be guaranteed by checking whether the source data can be reconstructed from the encoded output without loss. This is the basic concept of our proposed *inverse comparison* CED scheme, which is shown in Fig. 8. An additional decoder decompresses the encoded codewords, and a checker compares the reconstructed data with the delayed source input. Note that if the throughput of the decoding process can match that of the encoder, we can pipeline the error detection computations so that the overall execution time overhead on the total cycle count is small.

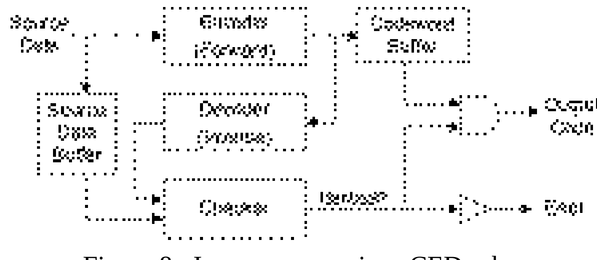


Figure 8: Inverse comparison CED scheme.

In general, inverse comparison CED technique can be used in any 1-to-1 mapping application. To avoid large hardware overhead, the complexity of the inverse process should be much smaller than that of the forward computation. As the discussion in Sec. 2 shows, LZ decompression is basically a memory access, which is much simpler than the string comparison in the encoder. Therefore, we expect this approach to have less hardware redundancy than a duplex LZ compressor. Next, we will analyze practical issues related to the LZ compressor with inverse comparison CED, including hardware overhead, fault coverage, and error recovery scheme.

4.2 Hardware Overhead for Inverse Comparison

The hardware overhead for the inverse comparison technique has three components. They are (1) the decoding dictionary, (2) the error checker implementation, and (3) source data buffers and output codeword buffers.

There are two possible approaches for implementing the decoding dictionary. One is to use a separate memory, and the other is to share the dictionary with the encoder. For the separate memory approach, since LZ decompression does not involve parallel comparisons, a dual-port RAM will suffice to match the encoder throughput of 1 symbol per cycle. Since the most recent FPGAs have efficient on-chip RAM solutions, hardware overhead due to a separate decoding dictionary can be minimized. For example, for Xilinx XC4000XLA series [Xilinx 99], one CLB has the capacity to house a 16x1 dual-port RAM module, while it contains only 2 flip-flops in the same block. Since the encoding dictionary has to be implemented in the flip-flops as discussed in Sec. 3.3, the CLB consumption in the separate decoding dictionary is only 12.5% of that in the encoding part. This is much less than the 100% overhead in the duplex system.

On the other hand, there may be situations where the flip-flop resources are still abundant, but the on-chip RAM is unavailable. A separate decoding dictionary using on-chip RAM would be impractical in these situations. One way of solving this is to recycle the encoder dictionary since both processes should refer to identical copies. Since the decoding process in inverse comparison CED does not begin before an output codeword is generated, it lags behind the encoding at most for L_{max} cycles, where L_{max} is the maximum matching length limit. Therefore, an extra "victim buffer" of size L_{max} is required for the shared dictionary to accommodate the latency in decoding. For an N-entry dictionary, this causes an overhead of L_{max}/N.

In addition, sharing dictionary between the encoder and decoder raises a reliability issue on the correctness of memory contents. Since the inverse comparison technique targets its protection at the datapath hardware, any faulty input to the processing arrays will not be detected if the error checker compares two identical copies of faulty input. Hence, error control codes are necessary to protect the shared dictionary. Obviously, adding extra parity checks creates another tradeoff between error detection capability and hardware overhead. If we target at detecting single stuck-at faults in the memory, it would be sufficient to use a simple parity check on each memory cell. The resulting overhead in the shared dictionary is (12.5%+L_{max}/N), which is larger than the separate dictionary approach.

The second source of hardware overhead comes from the error checker block in Fig. 8. As discussed in Sec. 3, if the systolic array approach in Fig. 7(a) is applied, we can schedule idle PEs to execute the comparison since the number of effective PEs is generally low during the search of long matching strings in the encoder. The scheduling can be done according to the match result of each cell in the previous cycles. To reduce the area overhead caused by routing and scheduling, we need to limit the number of PEs available for error checking purposes. In this case, we should select the PEs starting from the oldest entry in the dictionary (D0) since old data statistically have a smaller chance to be referred to in the encoding process again.

Another approach is to use an extra comparator dedicated to error checking. Note that for this method, we do not need more than one error checker since the encoder throughput is one source symbol per cycle in structures of both Fig. 6 and 7. Therefore, hardware overhead incurred by the error checker is not substantial compared to the other two sources of overhead. This approach is expected to have less execution time overhead because the error checking is not competing with the encoding process on PE resources.

The third source of hardware overhead, output codeword buffers and source data buffers, is needed to minimize the throughput degradation caused by inverse comparison CED. For example, let the latency for decoding one symbol be t cycles. If a matching string of length n is encoded, the first source symbol of the string has to wait (n+t) cycles before it can be compared in the error checker. In addition, after this codeword is generated, it takes (n+t) cycles to complete checking this codeword in a pipelined fashion. Hence, if n is very large, both the source data buffer and the output codeword buffer will be occupied by the corresponding elements of this codeword for a long period of time. If sufficient buffer space is not available, subsequent source data symbols and output codewords will be blocked from advancing in the pipeline, which causes the encoding to stall.

To eliminate this effect, we can consider the most congested situation where a matching string of length L_{max} is followed by unmatched single-symbol phrases. Since the first symbol and the codeword of matching length L_{max} will occupy the head of corresponding buffers for (L_{max}+t) cycles, the minimum capacity of a non-blocking buffer is equal to (L_{max}+t+1). Normally, L_{max} is chosen between 16 to 128, depending on the dictionary size. In our implementation, we choose a 512-entry dictionary with L_{max}=63. The resulting codeword length is thus (9-bit C_p + 6-bit C_l + 8-bit C_n)=23 bits. If we implement 64-entry buffers using on-chip 16x1 dual-port RAM configurations in Xilinx XC4000XLA series, the resulting hardware consumption will be (64/16)*(23+8)=124 CLBs. Compared to the encoding dictionary implemented in 4K flip-flops occupying 2K CLBs, the hardware overhead in the memory device due to the non-blocking buffers is about 6%. Also, even if the on-chip RAM is unavailable, the source data can be extracted from the shared dictionary in the encoder, and the output codeword buffer can be implemented using flip-flops. The resulting memory overhead (without considering routing overhead) will be (64*23)/(512*8)=35.9%.

In summary, we expect the overall CLB overhead to be below (12.5%+6%)=18.5% if we use the on-chip RAM of Xilinx XC4000XLA FPGA series with a separate decoding dictionary. If the on-chip RAM is unavailable, and the shared dictionary approach with 1-bit parity check per cell is used, the CLB overhead is expected to be below (12.5%+64/512+35.9%)=60.9%. The overhead of both schemes is much smaller than the 100% in a duplex system. Note that this overhead estimation only accounts for the CLB consumed by the dictionary and buffers. Since the comparator array and subsequent combinational logic blocks in the encoding process are not considered, the actual overhead due to inverse comparison CED scheme should be significantly lower than the estimated data.

4.3 Fault Coverage

In the LZ encoder architecture described in Sec. 3, we can classify the functional blocks into two categories according to the error-detection mechanism. The first group is the dictionary array. If the dictionary array is shared by both encoding and decoding parts, it is protected by error control codes. A simple parity check appended to each of the 8-bit source symbol can provide odd-error detection capability in each memory cell. Instead, more sophisticated error control codes can be applied to achieve better detection and correction capability at the cost of larger hardware overhead. On the other hand, if a separate decoding dictionary is used, the dictionary array is equivalent to a duplex system in terms of error detection capability. Suppose the probability that one copy of a dictionary element is faulty in a cycle is p (p<<1), and we assume that the failure occurrence is independent between the two copies. The error detection failures resulting from a faulty dictionary occurs only when both copies have the same faulty result in the same period. Therefore, the probability of undetected errors is p^2, which is much smaller than the original error probability p.

The second group is the datapath hardware, including PEs in systolic array or comparators inside CAM cells, the length counter, and other combinational logic blocks. These blocks are protected by the proposed inverse comparison CED technique, which guarantees that the source data can be correctly reconstructed in the fault-free decompression end. However, there are situations where faults cannot be detected but the source data can still be correctly recovered in the decoder. Although the encoded output is safe in these cases, there is a certain degree of degradation in compression ratio.

Suppose the faults are modeled by some signals being stuck at some logic value. Two kinds of faults can escape from inverse comparison CED. The first category has the same effect as if one or more PE (or CAM cell) outputs are stuck at "unmatched" logic value. This includes the stuck-at-unmatched PE outputs and the case where intermediate signals in the OR-tree are stuck at 0. For transient failures of this type, the worst case occurs when the error hits the longest matching phrase position and results in a shorter matching length. For the permanent stuck-at-unmatched faults, the OR-tree always encounters "unmatched" results from corresponding PEs, and thus these PEs are never encoded as the match position element. Equivalently, the effect of permanent stuck-at-unmatched PEs is the reduction of the number of elements in the dictionary, which degrades the compression ratio in the encoder.

Figure 9 shows the simulation result of the average compressed percentages when the number of permanent stuck-at-unmatched PEs grows from 0 to 400. Again, benchmark data are from [Calgary 90], and there are originally 512 entries in the dictionary. The simulation is done assuming that n stuck-at-unmatched PEs result in a dictionary of size (512-n), with the number of bits in the C_p pointer remaining at 9. From the resulting plot, even when one of the intermediate gates in the 9-stage OR-tree fails such that 1/4 of the PEs are effectively disabled, the compression ratio degradation incurred is still within 9%. When 1/2 of the PEs are disabled, the degradation is around 23%. Therefore, the compression ratio degradation due to a fair amount of stuck-at-unmatched PEs is graceful.

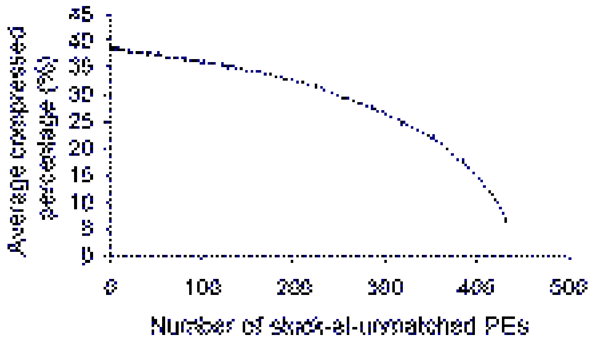


Figure 9: Average compressed percentage in the presence of stuck-at-unmatched PEs.

The second category of faults that might not be detected occurs when PE outputs are stuck at "matched" value temporarily. Since the position encoder resolves the matching pointer according to a certain priority order, a transient stuck-at-matched PE may not change the correct output as long as it occurs in the position with a lower priority than the fault-free result. Also, inverse comparison CED only checks the final resulting codeword of a string matching process instead of every intermediate outcome. Therefore, it is not guaranteed to detect transient stuck-at-matched PEs if they occur in the middle of the maximum-length searching process. Fortunately, in fact, the only escapes for this type of faults are those which can be masked without changing the fault-free final output. Once a stuck-at-matched PE alters the output codewords, it can always be caught by the inverse comparison CED since some unmatched reconstructed data will eventually be detected in the error checker. Note that for permanent stuck-at-matched PEs, the error can always be detected by the inverse comparison CED, which is different from the effect of permanent stuck-at-unmatched PEs.

To prevent the compression ratio from further degradation due to permanent stuck-at-unmatched faults in PE outputs or equivalent cases, we can schedule a testing mode in between two successive compression jobs to check if such type of faults exists in the PE array. Our primary goal here is to detect any fault in the OR-tree network and PE outputs that lead to the effect of stuck-at-unmatched PEs. A simple strategy is to force the PEs to produce “matched” results and to check them one by one in each cycle. In the testing mode, a matching symbol is shifted into the dictionary once in the beginning of the test, and the dictionary cells are compared with this test symbol. In each cycle, only the cell that holds the matching symbol should produce a "matched" result in fault-free conditions. The test result can be observed at the *Matched* signal in Fig. 6 and 7, and this signal should remain at 1. In this way, we can also get the knowledge of possible permanent stuck-at-unmatched locations in the OR-tree or the corresponding PE once an error (*Matched*=0) is observed during the testing mode. The test latency for an *N*-entry dictionary design is *N* cycles.

4.4 Error Recovery

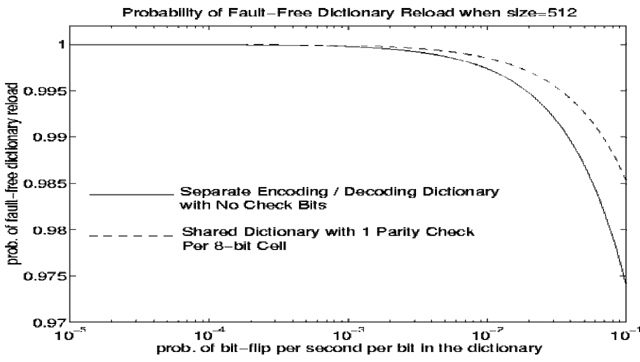
By reconfiguring the system to avoid faulty parts, reconfigurable coprocessors have the advantage of error recovery once a failure is diagnosed. Also, in situations where hardware resources are significantly reduced due to the diagnosed clustered faults, we have the flexibility to switch to degraded operations.

To exploit such an advantage, the application running on the ACS should be scalable. For LZ compression with inverse comparison CED, we can first reduce the hardware consumption by shrinking the sizes of source data and output codeword buffers in the CED scheme. This only pays the price of throughput degradation without hurting the performance in compression ratio and error detection capability. When the system is further damaged, we can tradeoff the compression ratio by reducing the size of the dictionary. The effect of reducing the dictionary size is equivalent to the presence of stuck-at-unmatched PEs. From the simulation result in Fig. 9, the resulting compression ratio degradation is graceful.

However, before the reconfiguration begins, we need to distinguish transient faults from permanent failures. If it is transient, avoiding the temporarily faulty block can be a waste. For this purpose, error recovery technique from transient faults is necessary once a fault is detected. One simple technique for LZ compressor is to retry only the encoding process for the string that encounters the error. In this case, we need to reconstruct the state of the machine to the state before the detection of fault.

One method to realize this is to enable 2-way shifting in the sliding dictionary and to recover the original state by shifting the dictionary entries backwards. However, this method can double the routing and control efforts and cause significant hardware overhead.

Instead, rebuilding the dictionary completely from the input end is a more feasible approach. In this way, the only hardware overhead is the extra register storing the memory address of the head of dictionary in the beginning of the corresponding string matching process. In addition, note that the compression task does not restart completely from the very beginning. Previous correct outputs are still valid and need not be regenerated all over again, and the retrial process starts from the beginning of the current faulty string.



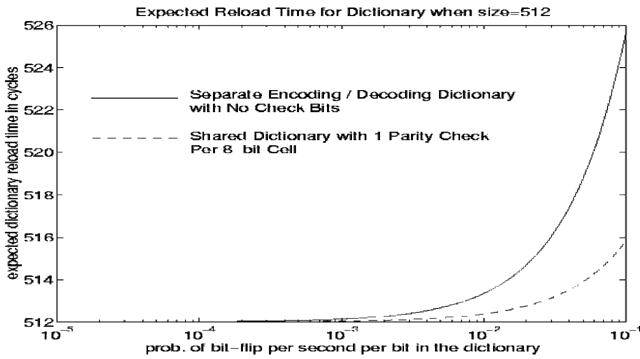
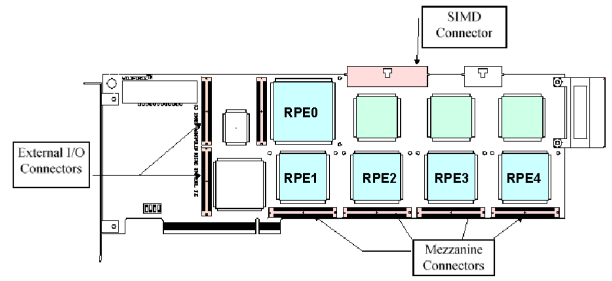


Figure 10: Statistics of 512-entry dictionary reload.

The only drawback of this retrial scheme is that the execution time overhead grows with the dictionary size since the normal operation cannot start before the dictionary is fully reloaded. A long retrial has a greater chance to be hit by another upset during its process. In the worst case, the system is always stuck in retrial cycles and the normal operation may never begin. Figure 10 is the analytical result showing the probability of fault-free dictionary reload process and the expected reload latency for a 512-entry dictionary with respect to different bit-flip probability *p* (bit flips per second per bit). It is clear that even when *p* is as large as an order of 10^{-2}, the system is still safe from deadlocking in retrial cycles.

5. Emulation Results

To demonstrate that the LZ compressor with inverse comparison CED achieves high throughput with small hardware overhead on reconfigurable coprocessors, we emulated the design on WILDFORCE reconfigurable computing engine [Annapolis 99]. The board layout and architectural diagram are shown in Fig. 11. The WILDFORCE system contains five Reconfigurable Processing Elements (RPE), RPE0 to RPE4, with a maximum clock rate of 50MHz. Each RPE is a Xilinx XC4036XLA FPGA chip, which provides the reconfiguration capability. The RPEs are connected to one another through 36-bit bi-directional systolic data paths and a 36-port crossbar implemented by three identical Xilinx XC4006E FPGA devices. With five RPEs on the board, WILDFORCE has a reconfigurable capacity of 6480 CLBs. In addition, there are three 512x36-bit bi-directional FIFO RAM modules. These FIFOs provide a bridge for the RPEs to communicate with external hosts via a PCI bus interface.



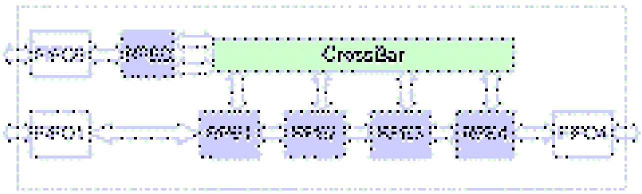


Figure 11: WILDFORCE layout and architecture.

In our LZ compressor emulation, we use a 512-entry dictionary with a maximum matching length L_{max}=63. We use the high performance systolic array architecture in Fig. 7 to provide the flexibility in the scheduled error checker approach. The data compression benchmark files are from [Calgary 90]. First of all, to evaluate the resulting throughput degradation due to inverse comparison CED, we did a Verilog simulation to compare the average cycle count overhead with respect to different inverse comparison schemes. Here, the different schemes include different error checker implementations (scheduled PE or dedicated PE) and varying sizes of output codeword buffer. In the scheduled error checker approach, we select four PEs corresponding to dictionary entries D0 to D3 in Fig. 7(a).

From the simulation results in table 2, we can see that with one extra PE dedicated for error checker, the execution time overhead can be significantly improved. Also, note that when the buffer size is close to L_{max}, the inverse comparison CED scheme approaches zero execution time overhead since the buffer rarely overflows. Therefore, in the following emulation, we choose the dedicated error checker with 64-entry buffers to avoid throughput degradation.

Table 2: Cycle count overhead of different inverse comparison CED schemes of LZ compressor.

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	**64**	**8**	**64**	**8**
[illegible]	**12%**	**18%**	**0.5%**	**3%**

Table 3 lists the comparison of hardware consumption for different CED schemes. The number of CLBs is reported by Xilinx Alliance 2.1i place and route tool [Xilinx 99]. CLB overhead percentage is calculated with respect to the number of CLBs without any CED. As predicted in Sec. 4.2, CLB overhead for both inverse comparison approaches is smaller than the estimated number calculated by considering only memory overhead. It is clear that the inverse comparison CED has a much smaller hardware overhead than the traditional duplex system.

Table 3: Comparison of the hardware consumptions.

[illegible]	[illegible]	[illegible]	[illegible] [1]	[illegible] [2]
[illegible]	**4,219**	**8,440**	**4,607**	**5,296**
[illegible]	**0**	**100%**	**9.20%**	**25.53%**
[illegible] [3]	**4**	[illegible]	**4**	**5**

The comparison of encoder throughput, measured on WILDFORCE emulation and C programs on general-purpose processors, is shown in table 4. The algorithm of

[1] Using a separate decoding dictionary without parity check.
[2] Using a shared dictionary between encoding and decoding with 1-bit parity check per cell.
[3] Each of the five Xilinx XC4036XLA FPGAs on WILDFORCE board contains 1296 CLBs.

the C codes is from [Nelson 96]. Here, the inverse comparison CED with the minimum execution time overhead is applied in WILDFORCE system, while the C programs on general-purpose processors do not have any CED scheme. Note that since the decoding process is not in the critical path, the clock rate on WILDFORCE emulation is identical with or without inverse comparison CED. From table 4, it is clear that even with a slower clock rate, the WILDFORCE approach can achieve more than 30 times speed up due to the massive parallelism. With more advanced FPGAs that can run at faster speeds and have greater CLB and routing capacities, we expect the application to fit on one chip and have a higher clock rate and throughput.

Table 4: Performance benchmarks.

[illegible]	[illegible] F [illegible] 4036 [illegible]	[illegible]	[illegible]
[illegible]	4	1	1
[illegible]	16 [illegible]	300 [illegible]	450 [illegible]
[illegible]	128 [illegible]	3[illegible]0 [illegible]	4[illegible]2 [illegible]

6. Conclusions

In this paper, we proposed a reliable LZ compressor implementation on reconfigurable coprocessors. With highly parallel architectures, this approach achieved a very high throughput even with slower clock rates. Both analytical and experimental results proved the feasibility to map our approach to current FPGA technologies.

To avoid the error propagation problem, we proposed the inverse comparison CED technique to apply to our FPGA LZ compressor. Practical issues regarding the reconfigurable coprocessor realization have been investigated, and this approach has minimized both the CLB overhead and the throughput degradation. With the error detection capability, the scalable feature of our proposed scheme, and the reconfiguration ability of FPGA, reliability can be achieved with graceful degradation.

7. Acknowledgements

The authors would like to thank Dr. Santiago Fernandez-Gomez, Philip Shirvani, Subhasish Mitra, and Siegrid Munda for their valuable feedback and suggestions. This work was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-97-C-0024.

References

[Altera 99] Altera Corporation, "Application Note: AN 119 (Implementing High-Speed Search Applications with APEX CAM) ver. 1.01," *http://www.altera.com*, July 1999.

[Annapolis 99] Annapolis Micro Systems Inc., *http://www.annapmicro.com*, 1999.

[Calgary 90] Calgary Text Compression Corpus, available from Univ. of Calgary, Canada. *ftp://ftp.cpsc.ucalgary.ca*.

[Craft 98] Craft, D. J., "A Fast Hardware Data Compression Algorithm and Some Algorithmic Extensions," *IBM J. Research and Development*, Vol. 42, No. 6, Nov. 1998.

[Das 99] Das, D., and N.A. Touba, "A Low Cost Approach for Detecting, Locating, and Avoiding Interconnect Faults in FPGA-Based Reconfigurable Systems," *Proc. of IEEE International Conference on VLSI Design*, pp. 266-269, 1999.

[Emmert 97] Emmert, J. M., and D. Bhatia, "Partial Reconfiguration of FPGA-Mapped Designs with Applications to Fault Tolerance and Yield Enhancement," *7th Int'l Workshop of Field-Programmable Logic*, 1997.

[Hanchek 98] Hanchek, F., and S. Dutt, "Methods for Tolerating Cell and Interconnect Faults in FPGAs," *IEEE Trans. on Computers*, Jan. 1998.

[Huffman 52] Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes," *Proc. IRE*, Vol. 40, pp. 1098-1102, 1952.

[Jones 92] Jones, S., "100-Mbps Adaptive Data Compressor Design Using Selectively Shiftable CAM," *IEE Proc. G*, Vol. 139, No. 8, pp. 498-502, 1992.

[Jung 98] Jung, B., and W. P. Burleson, "Efficient VLSI for Lempel-Ziv Compression in Wireless Data Communication Networks," *IEEE Trans. on VLSI Systems*, Vol. 6, No. 3, pp.475-483, 1998.

[Lee 95] Lee, C. Y., and R. Y. Yang, "High-Throughput Data Compressor Design Using Content Addressable Memory," *IEE Proc. G*, Vol. 142, No. 1, pp. 69-73, 1995.

[Mitra 98] Mitra, S., P. P. Shirvani, and E.J. McCluskey, "Fault Location in FPGA-Based Reconfigurable Systems," *IEEE Intl. High Level Design Validation and Test Workshop*, 1998.

[Nelson 96] Nelson, M., and J. L. Gailly, *The Data Compression Book*, 2nd edition, M&T Books, 1996.

[Ranganathan 93] Ranganathan, R., and S. Henriques, "High-Speed VLSI Design for Lempel-Ziv-based Data Compression," *IEEE Trans. Circuits and Systems*, Vol. 40, pp. 96-106, Feb., 1993.

[Rissanen 76] Rissanen, J. J., "Generalized Kraft Inequality and Arithmetic Coding," *IBM J. Res. Develop*., Vol. 20, No. 3, pp. 198-203, 1976.

[Saxena 98] Saxena, N. R., and E. J. McCluskey, "Dependable Adaptive Computing Systems," *IEEE Systems, Man, and Cybernetics Conf.*, pp. 2172-2177, Oct. 11-14, 1998.

[Saxena 00] Saxena, N. R., S. Fernandez-Gomez, W.-J. Huang, S. Mitra, S.-Y Yu, and E. J. McCluskey, "Dependable Computing and On-Line Testing in Adaptive and Configurable Systems," *IEEE Design and Test*, Vol.17, No. 1, pp. 29-41, 2000.

[Xilinx 99] Xilinx Inc., *http://www.xilinx.com*, 1999.

[Ziv 77] Ziv, J., and A. Lempel, "A Universal Algorithm for Sequential data Compression," *IEEE Trans. Information Theory*, Vol. IT-23, No. 3, pp. 337-343, 1977.

[Ziv 78] Ziv, J., and A. Lempel, "Compression of Individual Sequence via Variable-Rate Coding," *IEEE Trans. Information Theory*, Vol. IT-24, No. 5, pp. 530-536, 1978.