UC Irvine
ICS Technical Reports

Title
Systolic implementations for transpose coding

Permalink
https://escholarship.org/uc/item/3p10849c

Authors
Stauffer, Lynn M.
Hirschberg, Daniel S.

Publication Date
1991-11-15

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Z
699
C3
no. 91-69

Systolic Implementations for Transpose Coding

Lynn M. Stauffer
University of California, Irvine
Irvine, CA 92717
stauffer@ics.uci.edu

Daniel S. Hirschberg
University of California, Irvine
Irvine, CA 92717
dan@ics.uci.edu

Technical Report 91-69

November 15, 1991

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Contents

	Page
Introduction	1
List Compression	2
Related Work in Move-to-Front Coding	4
Parallel Transpose Coding with Fixed-Length Words	5
Systolic Implementations ENC1 and DEC1	6
Systolic Implementations ENC2 and DEC2 with Reduced Delay	13
Parallel Transpose Coding with Arbitrary Words	16
Topics for Future Investigation	17
References	18

1. Introduction

Data compression attempts to remove redundancy from data and thereby increases the density of transmitted or stored data. Traditionally, there has been a tradeoff between the benefits of employing data compression versus the computational costs related to encoding and decoding. Parallelism represents a means for speeding up data compression performance. The problem of compressing data as effectively as possible is a challenging one that has been extensively researched in the sequential setting [W91, BCW90, S88, LH87]. Included in this vast collection of sequential methods is Move-to-Front coding which maintains a dynamic list of words (to be encoded by their list position) using the move-to-front self-organizing list strategy [BSTW86, E87, R87, HC87]. A systolic array implementation of Move-to-Front has been described [TW89]. In this paper, we present systolic array implementations of Transpose coding, which uses an alternative self-organizing list strategy but otherwise is similar to Move-to-Front coding. We present implementations for fixed-length word lists which provide improved system bandwidth by accelerating Transpose coding.

The state-of-the-art in software data compression systems is the UNIX[1] *compress* utility which is based on a variation of Ziv and Lempel coding due to Welch [W84, ZL78]. The UNIX *compress* system provides compression savings of up to 80% at a relatively high input bandwidth of 30 Kbytes per second on a 1 MIPS machine [TW89]. Higher compression savings are achieved by high-order Markov models and improved versions of *compress* which operate at limited input bandwidths of approximately 10 Kbytes per second on a 1 MIPS machine. A systolic array implementation of Move-to-Front running on a 40 MHz clock operates at a bandwidth of 40 Mbytes per second with compression savings ranging from 20% to 70% [TW89]. Several parallel compression systems based on dictionary coding achieve similar compression at input rates exceeding 25 Mbytes per second using a

[1] UNIX is a trademark of AT&T Bell Laboratories

40 MHz clock. Our implementations operate at a bandwidth commensurate with the systolic Move-to-Front system.

Our algorithms are implemented on a systolic array. Systolic arrays consist of a linearly-connected collection of synchronized rudimentary processing elements. Each processor has its own local memory and is assigned a unique identification number. An advantage of the systolic implementation is that a larger pipe can be fabricated by placing a sequence of processing elements on a single chip, and then joining a series of chips on a board. Another benefit is that the length of interprocessor connections are constant and independent of the array size.

Section 2 describes the basic list compression method and several of its variants. Related work on systolic implementations of Move-to-Front coding is presented in Section 3. Our systolic designs for Transpose coding are given in Sections 4 and 5. Conclusions and areas for further investigation are the focus of Section 6.

2. List Compression

Data compression schemes can be categorized according to the method used to parse the input stream into individual encodable messages. In *defined-word* schemes, the context determines a set of source messages or words[2] (sequences of input symbols) that are candidates for encoding. There are a number of suitable definitions for the composition of a word. For instance, in text file compression, a word may be defined to consist of an individual character or a sequence of characters delineated by a space.

This paper considers a class of compression algorithms which maintain a sequential list of words using a self-organizing heuristic so that frequently accessed words appear near the front of the list. To distinguish the collection of compression techniques which utilize a self-organizing list from dynamic dictionary or Ziv and

[2] Bentley et al. refer to these source sequences as “words” [BSTW86].

Lempel compression schemes, we will refer to this collection as *list compression* methods under a particular update heuristic.

A list compression method uses a self-organizing data structure to maintain a list of source messages and a variable-length encoding of the integers to compress list indices. To compress a word, it is located in the dynamic word list and encoded by its list position. After a word has been referenced, the list is reorganized appropriately. In Move-to-Front coding, the encoded word is removed from its current position and placed in the first list position. In Transpose coding, the encoded word is exchanged with the contents of the preceding list entry. By directing the encoded list position to a fixed-to-variable coder, the output is further compressed by assigning short codewords to positions near the front of the list.

The move-to-front and transpose list organizing strategies are two update heuristics among a collection of many others (see [HH85] for a survey of self-organizing linear search). After reaching a steady state, where many further search requests are not expected to significantly impact the expected search time, the expected access cost will be less for transpose than for move-to-front. But the convergence time or number of accesses required to reach a steady state is greater for transpose than for move-to-front [HH85]. There are applications for which move-to-front and transpose outperform each other. Moreover, Horspool and Cormack report that the transpose heuristic performs as well as the move-to-front and is easier to implement since updates involve only local rearrangement [HC87]. For any particular application, simulations are necessary to determine the superior heuristic. This paper, by furnishing a systolic array implementation of transpose, provides the option of choosing between transpose and move-to-front for other applications.

3. Related Work in Move-to-Front Coding

Dictionary coding algorithms (which include list compression algorithms) function by replacing blocks of input with references to earlier occurrences of identical data. Systolic implementations have been developed for several variants of the general dictionary scheme. Parallel dictionary methods are surveyed in [SH91A]. Previous findings in parallel list compression are described below.

Thomborson and Wei investigate parallel implementations of Move-to-Front coding [TW89]. They distinguish two major algorithmic variants of Move-to-Front compression. The simpler procedure permutes a byte-level fixed-length list of symbols and the other approach divides the input stream into "words" and transmits words by a move-to-front code. VLSI implementation issues are the root of this distinction. That is, a general defined-word scheme requires words of arbitrary length to be present in the word list. In a systolic array, arbitrary word lengths place unreasonable demands on the number of input/output pins that must be placed on each processing element. Thomborson and Wei examine various alternatives, such as placing a limit on the length of words, and find that even permitting short words can be prohibitive.

For the simpler byte-level fixed-length Move-to-Front coding, Thomborson and Wei describe an array of 256 processing elements each of which stores an 8-bit number corresponding to an ASCII code [TW89]. The input stream enters the pipe and is encoded by detecting matches between the input characters and the bytes stored in the processing elements. When a match is detected, the input character is replaced by the identification number of the matching processor. By depositing the input character in the first processor as it enters the pipe and then cascading previous processing element contents down the array, the move-to-front behavior is realized. The output of the array, consisting of a sequence of 8-bit list indices, is fed into a fixed-to-variable length coding system. This byte-level

design achieves compression savings of 19% to 38% and operates at a bandwidth of 40 Mbytes/second running on a 40 MHz byte clock.

Thomborson and Wei describe a systolic design for approximating general defined-word schemes [TW89]. The idea is to map variable-length words to an 8-bit hashcode using a hardwired hash table. These 8-bit codes are entered into the Move-to-Front list of target strings and manipulated as in the byte-level systolic encoder and decoder arrays. A closed hashing scheme with no collision resolution is used to obtain a high-speed, high-bandwidth design. These performance improvements, however, come at the expense of poorer compression performance. Unlike the sequential Move-to-Front codes in which the least-recently-used target word "falls" off the end of the list, the hashing approach randomly eliminates list words. This random behavior of the systolic design yields compression savings ranging from 25% to 65% and an input bandwidth of 40 Mbytes running on a 40 MHz clock.

4. Parallel Transpose Coding with Fixed-Length Words

Parallel transpose list compression is described by the following general paradigm. Encoder and decoder maintain identical word lists using the transpose heuristic. Namely, after a word is used it is exchanged with the word stored in the position immediately preceding its original position. In general, to transmit word w on the systolic array, w is compared to the list entries of successive processors. If w matches the list entry in processor i, it is encoded as i. The encoder then updates the list by transposing the list entry (w) of processor i with the list entry in processor $i-1$. When the decoder array receives list index i, it decodes it as the list word stored in processor i (which will be w) and then updates the list by exchanging w with the previous list entry stored in processor $i-1$. Since several

matches can be detected in parallel the list update procedure needs additional consideration.

In the sequential setting, a sequence of words that match the list structure in successive entries are handled in the same way as other matches. However, in the systolic environment, matches corresponding to successive entries in the array impose additional constraints when the list of words is being manipulated in parallel. That is, simultaneous matches occurring in different locations in the array may force global communication among the processors to determine the contents of the updated list. To illustrate this difficulty, consider the input string "abcdefgh" and the word list " h, g, f, e, d, c, b, a". Sequential transpose list compression outputs the sequence of positions 8, 8, 6, 6, 4, 4, 2, 2 and the final word list is identical to the original. In a sense, each pair of matches "cancels" the effect of their updates. On the systolic array, the input string pipes into the array and all eight matches are detected simultaneously. Handling the subsequent update may require global communication.

A systolic implementation may have difficulty allowing non-fixed-length words because of the unreasonable pin requirements [TW89]. Our initial designs are for defined-word methods which permute a list of fixed-length source messages. Section 5 describes systolic implementations which approximate the more general dynamic list compression method.

4.1. Systolic Implementations ENC1 and DEC1

This section gives systolic implementations (which we call ENC1 and DEC1) for encoding and decoding using the transpose fixed-length list compression model. For a list of length N, ENC1 consists of N processing elements (PE's) linearly connected by a two-way communication channel. PE i stores the list entry which is currently in position i in the list and a copy of the input word PE i considered on the previous clock cycle. The list entry will be referred to as $entry_i$ and the

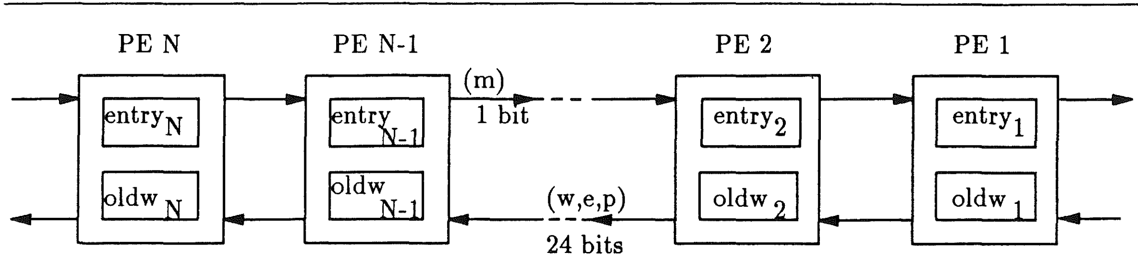


Figure 1

Systolic array for ENC1

prior input word as $oldw_i$. The input stream enters the pipe from the right (PE 1) and the encoded message exits at the left (PE N). A schematic of the architecture is shown in Figure 1. For $N = 256$, the array consists of 256 processing elements each storing an 8-bit byte list entry corresponding to an ASCII symbol.

In order to prevent a contiguous sequence of matches from occurring concurrently, our design allows input packets to enter the array only on every other clock cycle. The word list is updated at the start of each encoding cycle. Later in the cycle, word matches are detected and encoded.

At the beginning of the ENC1 clock cycle, PE i receives 3-tuple $(w,\ e,\ p)$ from PE $i-1$ and bit m_{i+1} from PE $i+1$. w is a word to be matched, e is the current contents of $entry_{i-1}$ that may be needed for a transpose update, p is the list position of w (0 is no match found yet) and m_{i+1} is a bit flag which is set if PE $i+1$ detected a match in the previous clock cycle. If m_{i+1} is set then PE i overwrites $entry_i$ with $oldw_i$. If w matches $entry_i$ then PE i carries out three tasks. Namely, PE i sets $p = i$, flags $m_i = 1$, and (if $i > 1$) overwrites $entry_i$ with input e (equivalently $entry_{i-1}$ obtained from PE $i-1$).

At the close of the clock cycle, PE i overwrites $oldw_i$ with w and transmits $(w,\ entry_i,\ p)$ to PE $i+1$ and (m_i) to PE $i-1$. Contention is avoided as a result of restricting input to every other cycle. An example of this encoding procedure is given in Figure 2.

For $N = 256$, the ENC1 communication channel is 25 bits wide and each processing element has three 8-bit registers, an 8-bit identity comparator, two 8-bit multiplexers, and additional control logic. The critical path in PE i passes through three hardware components. First, $entry_i$ and $oldw_i$ pass through a multiplexer triggered by flag m_i. Second, $entry_i$ and the input word w enter the identity comparator. Finally, the output of the comparator determines if the input word should be encoded as i by triggering a second multiplexer. The critical path compares to the move-to-front systolic array [TW89].

Our first decoding algorithm DEC1a resembles ENC1. As in ENC1, input enters the pipe on every other clock cycle. At the outset of the cycle, the word list is updated. Unlike ENC1, where only a single bit is passed from PE i back to PE $i-1$ to facilitate updating, DEC1a requires $entry_i$ be transmitted along with the single match bit. Following the list updating, list indices are replaced by word list entries.

DEC1a proceeds as follows. At the beginning of the clock cycle, PE i receives 3-tuple (w, e, p) from PE $i-1$ and 2-tuple (m_{i+1}, f) from PE $i+1$. p is the encoded list index, e is the current contents of $entry_{i-1}$, w is the unencoded word occurring in position p of the list (Λ if $p < i$), m_{i+1} is a flag indicating if PE $i+1$ decoded a list index on the previous clock cycle and f is the prior contents of $entry_{i+1}$ that may be needed for a transpose update. If m_{i+1} is set then PE i copies f into $entry_i$. If $p = i$ then PE i replaces w by $entry_i$, sets $m_i = 1$, copies $entry_i$ into f, and (if $i > 1$) finally overwrites $entry_i$ with e. At the end of the clock cycle, PE i sends (w,$entry_i$,p) to PE $i+1$ and (m_i, f) to PE $i-1$. Figure 3 provides an example of DEC1a.

Our second decoding algorithm, DEC1b, processes packets on every system cycle and therefore operates at twice the rate of ENC1 and DEC1a. However, DEC1b restricts the input to a fixed predefined alphabet of size S. The symbols in

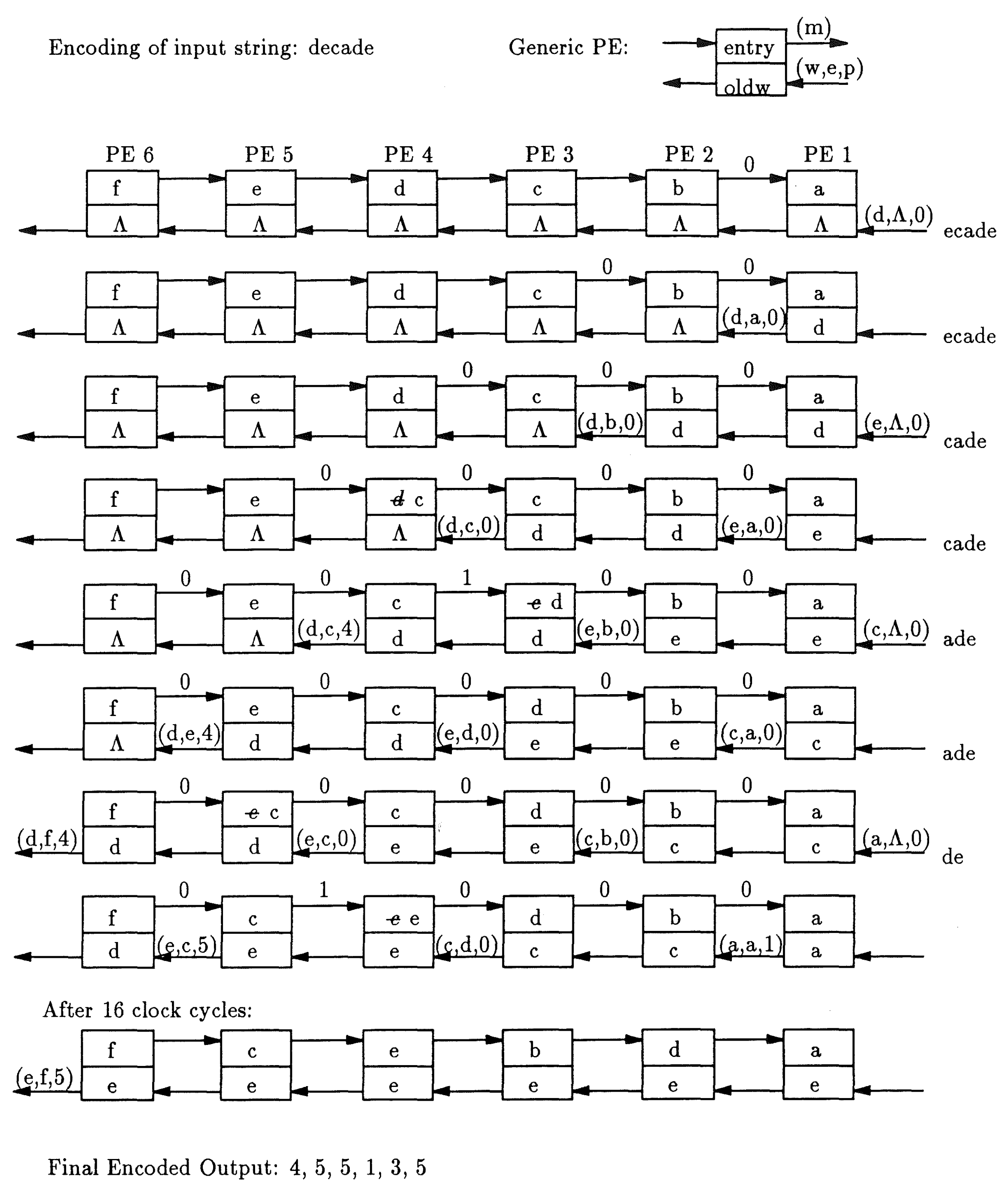
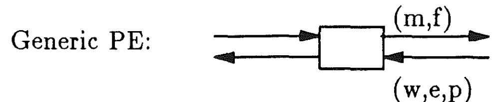


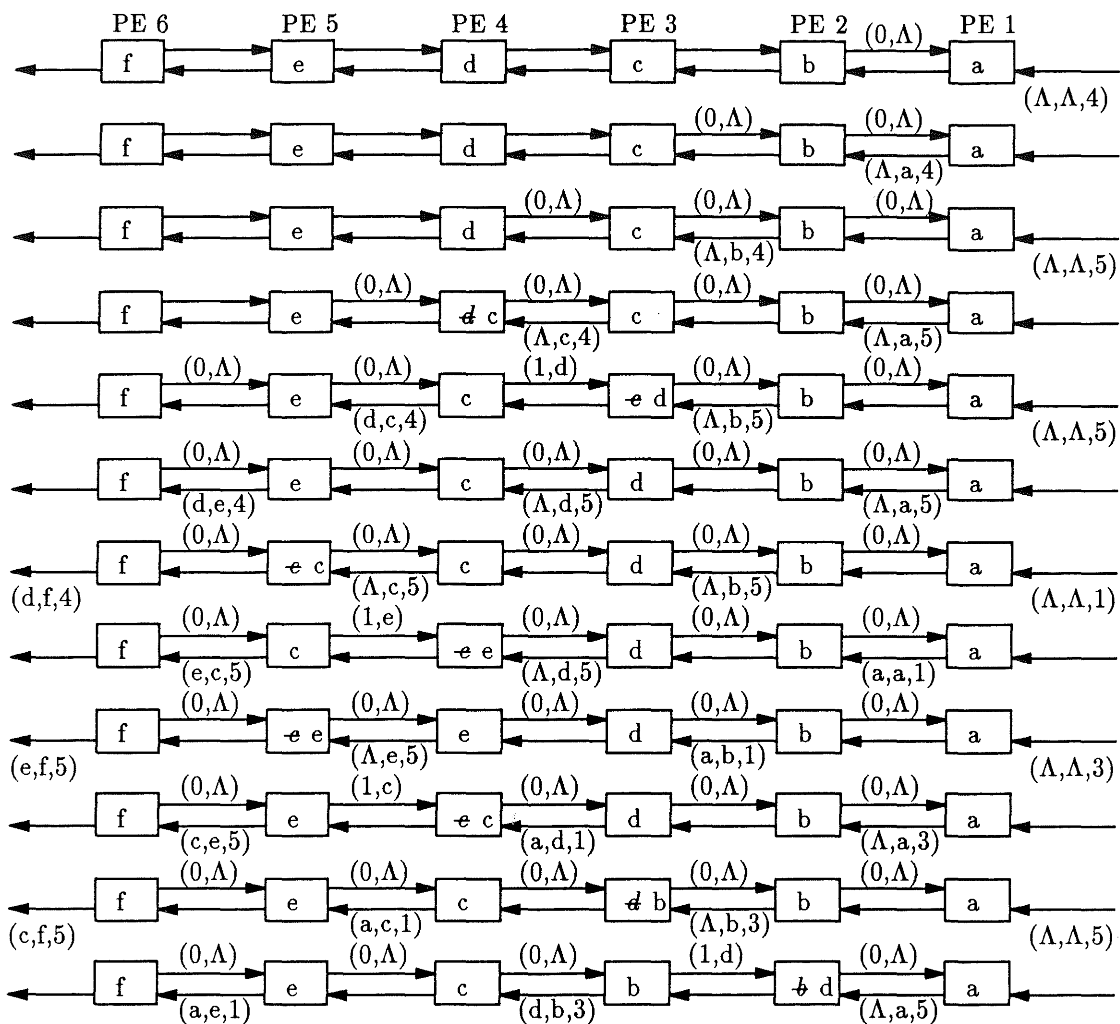
Figure 2

Example of ENC1 on input word "decade"

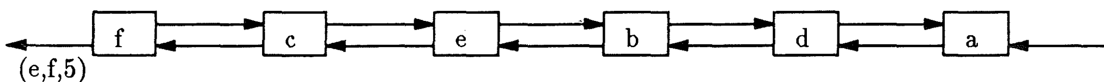
this alphabet are arbitrarily given indices 1 through S. The output of ENC1 is a

Decoding of input string: 4, 5, 5, 1, 3





After 16 clock cycles:



Final output: decade

Figure 3

Example of DEC1a

sequence of list positions which DEC1b receives as input and decodes in two stages. The first stage decodes list positions into indices and the second stage translates indices into symbols. PE i stores pos_i which is the list position of the alphabet symbol with index representation i.

At the start of the clock cycle, PE i receives input packet (w, p) from PE $i-1$. As in DEC1a, p is the encoded list position and w is the unencoded word occurring in position p of the list (Λ if $p < i$). If $p > 1$ and $p = pos_i$ then w is set to i and pos_i is decremented. If $p = pos_i - 1$ then pos_i is incremented. All necessary manipulations of list positions transpire before a packet arrives at the processor which will carry out the decoding.

After leaving the DEC1b array, the decoded symbol indices are replaced by their corresponding alphabet symbol by a special purpose processor located at the end of the pipe. The DEC1b algorithm is illustrated in Figure 4.

For a fixed-length list, such as the 256 different 8-bit ASCII characters, each processor is initialized to contain one of the 8-bit bytes. Alternatively, new words can be added to the list until the list becomes full. Moreover, if an input word w is not in the current list of size K $(1 \leq K \leq N)$ the word is encoded by the index $K + 1$ followed by the word w and the list is updated by transposing w with the list entry K. If $K = N$ (i.e., the list is full), word w replaces the last list entry. The decoder "learns" the word list in a similar fashion. In the systolic array, an additional flag bit in each processor is used to delimit the current list. The processor holding the flag is designated as the first empty list entry. Initially, the flag bit in PE 1 is set.

4.1.1. Empirical Evaluation

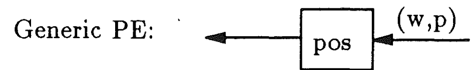
Compression findings for fixed-length word implementations of Move-to-Front and Transpose coding are reported in Table 1. A systolic array consisting of 128 processing elements each initialized to contain a 7-bit ASCII character was

alphabet = a, b, c, d, e

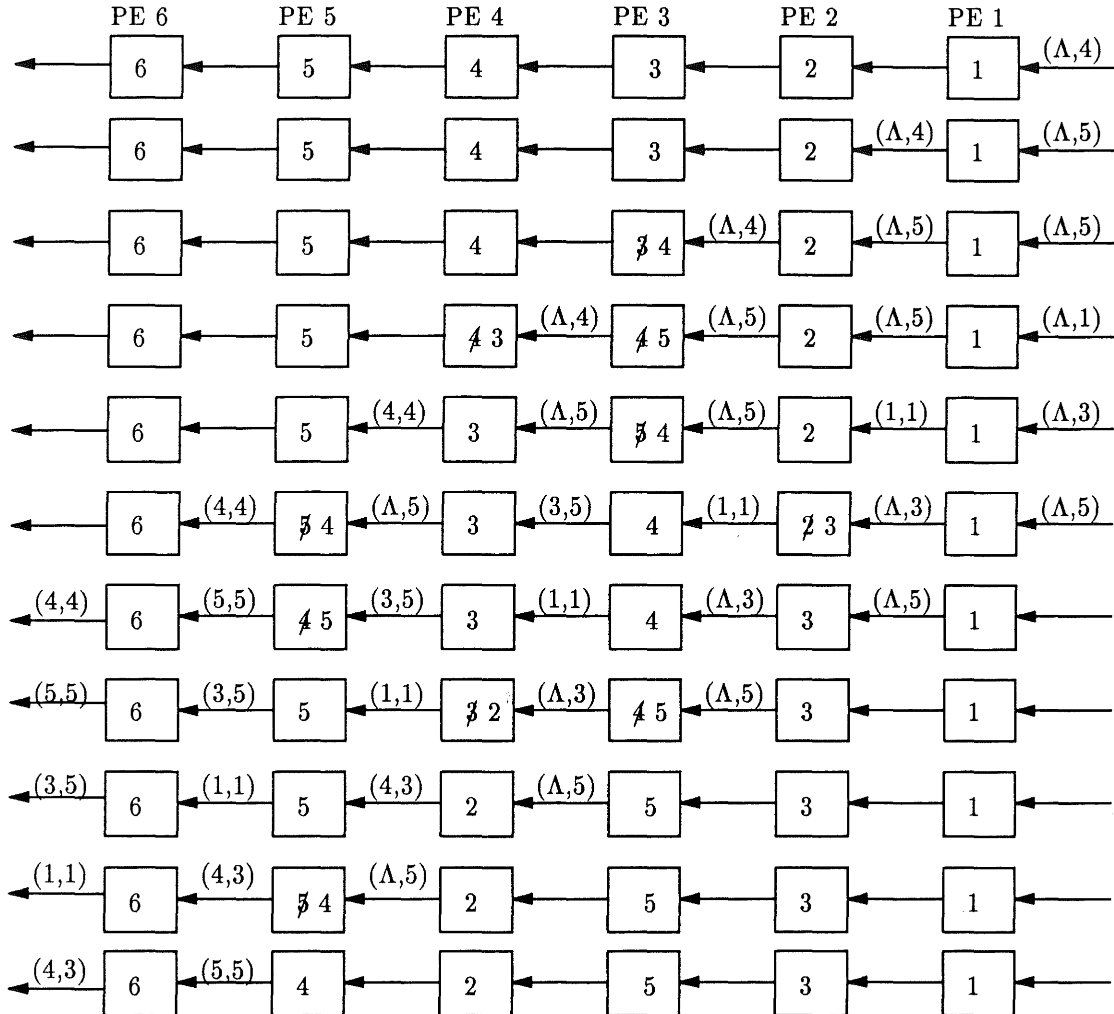
symbols: a b c d e f

indices: 1 2 3 4 5 6

Generic PE:



If PE i has contents j then symbol with index i is in list position j and thus $pos_i = j$.



Ouput of stage 1: 4, 5, 3, 1, 4, 5

Output of translator (indices into alphabet symbols): decade

Figure 4

Example of DEC2

simulated. The test files used are part of the Calgary/Canterbury text compression

Input File	Size (Bytes)	Compression Savings – MTF	Compression Savings – Transpose
bib	111261	29.44	32.24
book1	768771	37.91	41.90
book2	610856	36.98	39.54
paper1	53161	34.19	34.33
paper2	82199	37.49	39.07
paper3	46526	36.63	36.81
paper4	13286	35.20	29.75
paper5	11954	33.71	26.64
paper6	38105	35.45	33.79
news	377109	31.48	34.43
progc	39611	30.72	30.70
progl	71646	38.72	38.61
progp	49379	35.26	34.99

Table 1

Compression savings delivered by Transpose and Move-to-Front Coding

corpus [BCW90]. Transpose coding provides superior compression performance for large text files but, for small files (under 40 Kbytes), Move-to-Front gives better compression. These findings support the theoretical expectation that Transpose coding takes longer to reach a steady state but, after reaching a steady state, has a smaller expected access cost.

4.2. Systolic Implementations ENC2 and DEC2 with Reduced Delay

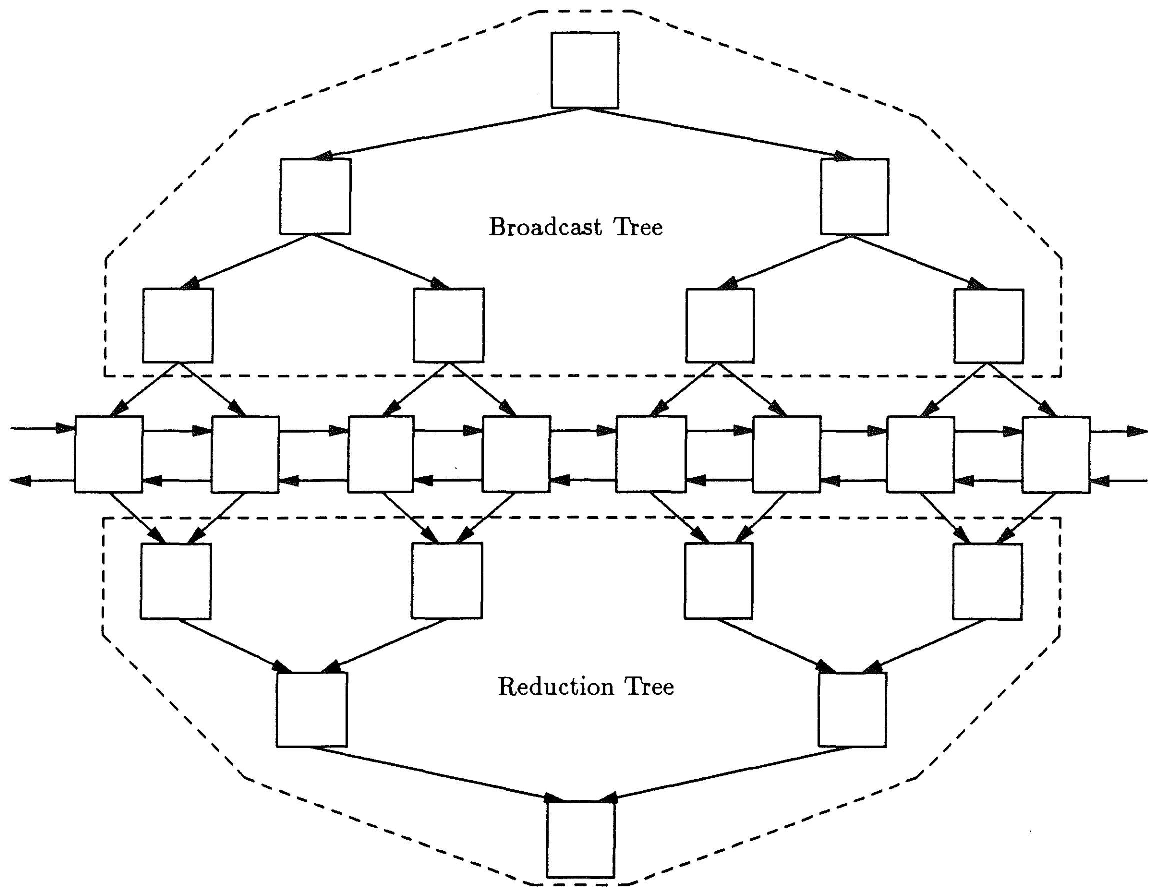
The linear delay of the previous designs is determined by the piping of the input from PE 1 through to PE N. In this section we describe an architecture

which combines a systolic array with trees, resulting in a logarithmic delay. The trees broadcast the input to the array processors and reduce the simultaneous outputs of the processors. A similar architecture for dictionary coding is described in [Z90]. In addition to decreased delay, the restriction allowing data to enter the pipe only on every other system cycle is eliminated in systolic implementation ENC2 for Transpose coding.

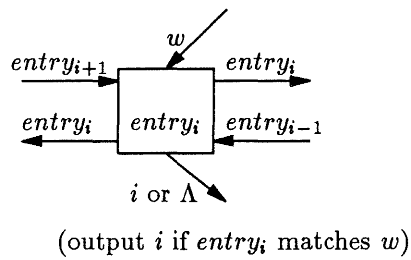
For a list of size N, the ENC2 architecture consists of $3N - 2$ processors. The first N processors are arranged in a systolic pipe and the remaining processors are configured as two binary trees (each containing $N-1$ processors) synchronized with the systolic array (see Figure 5). One of the binary trees (the *broadcast tree*) is placed on top of the systolic array. Input enters at the root of the broadcast tree and is propagated down the tree to each array processor. The other binary tree (the *reduction tree*) is placed below the systolic array. Results of the array processors are reduced to a single non-null output via the propagation toward the root of the reduction tree. The tree interconnect provides total delay of $2 \log_2 N$.

Processor PE i in the systolic array stores the list entry which is currently in position i in the list. As in ENC1, this entry is referred to as $entry_i$. PE i receives input from the broadcast tree and from PE $i - 1$ and PE $i + 1$. PE i transmits $entry_i$ to PE $i - 1$ and PE $i + 1$ and outputs match information to the reduction tree. Processors in the broadcast tree simply pass their input to their outputs. Reduction processors receive two inputs which are either both zero or one is non-zero. In the first case, the reduction tree outputs zero. In the second case, the reduction processor transmits the non-zero input.

After the input has propagated down the broadcast tree to the array processors, encoding proceeds as follows. At the beginning of the clock cycle, PE i receives (w) from the broadcast tree, $(entry_{i-1})$ from PE $i-1$, and $(entry_{i+1})$ from PE $i+1$. w is the word to be encoded. If w matches $entry_i$ then w is set to i and



Generic PE i in systolic array:



(output i if $entry_i$ matches w)

Figure 5

Broadcast/reduce architecture for ENC2 and DEC2

$entry_{i-1}$ is written into $entry_i$. If w matches $entry_{i+1}$ (received from PE $i+1$ at the start of the cycle) then PE i overwrites $entry_i$ with $entry_{i+1}$ and sets w to 0. Otherwise, PE i sets w to 0. At the close of the clock cycle, PE i transmits $entry_i$ to PE $i+1$ and PE $i-1$ and sends w to its neighboring processor in the reduction

tree. Thus, at the end of each clock cycle, exactly one array processor (the one which matched the input symbol) outputs a non-zero value into the base of the broadcast tree. This non-zero value is propagated to the root of the reduction tree and finally output. Decompression (DEC2) mirrors ENC2.

5. Parallel Transpose Coding with Arbitrary Words

The designs of Section 4 included only fixed-length words in the list. The more general word list compression scheme permits words of arbitrary lengths to be present. A straightforward extension of our previous designs allows arbitrary words. However, the 3-tuples that travel through the array, consisting of two words and a pointer, may lead to unreasonable VLSI implementation requirements. For example, the fixed-length implementations consisting of 256 processing elements require 24 bits per tuple or 48 input/output pins per systolic element. For words of length 8 or less on a systolic array of similar size, the pin requirement jumps to 272 input/output pins per chip. One simple solution to the problem of unbounded pin requirements is to place a bound on the maximum allowable word length. This bound enforces a limit on the pin requirements. The appropriate maximum word length is dependent on the application.

In order to avoid the potential VLSI issues, Thomborson and Wei approximate Move-to-Front coding by using a hardwired hash table to map arbitrary words onto an 8-bit byte[TW89]. This single byte is piped into the array and encoded as a fixed-length word. The increased throughput provided by the systolic design comes at the expense of lower compression savings. Their design provides compression savings ranging from 25% to 65% and an input bandwidth of 40 Mbytes running on a 40 MHz clock. This is considerably lower than the compression savings of 30% to 75% obtained by the sequential Move-to-Front codes. Implementing this hashing approach on our systolic transpose coders is straightforward. We suspect that the

random behavior of the hashing scheme would result in compression savings similar to those of the Move-to-Front systolic implementation [TW89].

6. Topics for Future Investigation

Although we have outlined the design and compression performance of several systolic compressors for high-bandwidth applications, extensive empirical evaluations of our methods are needed to fully describe their behavior.

As supported by the empirical findings in Section 4.1.1, Move-to-Front coding performs well on small files and Transpose coding is superior for large files. This suggests the examination of a hybrid scheme which combines Move-to-Front and Transpose coding. Initially, Move-to-Front coding is utilized and at some point control is switched to a Transpose coding method.

The broadcast/reduce implementation for Transpose coding described in Section 4.2 has a delay of logarithmic length. Further investigation into a broadcast/reduce for Move-to-Front coding may result in similar reductions in the delay.

The systolic designs for arbitrary word lengths are limited by the imperfect approximation behavior of the hashing scheme. Alternative methods are necessary to better emulate general sequential list compression.

As suggested by Thomborson and Wei, a high-bandwidth architecture for adaptive Huffman coding or other fixed-to-variable length coders which is capable of operating at a rate that is commensurate to the systolic list compression systems is also of great interest. Potentially, this high-bandwidth coder, when coupled with a systolic list compressor, will lead to improved compression performance.

References

[BCW90] Bell, T. C., Cleary, J. G., and Witten, I. H. *Text Compression*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[BSTW86] Bentley, J. L., Sleator, D. D., Tarjan, R. E., and Wei, V. K. A locally adaptive data compression scheme. *Commun. ACM 29*, 4 (April, 1986), 320-330.

[E87] Elias, P. Interval and recency rank source coding: two on-line adaptive variable-length schemes. *IEEE Trans. Inform. Theory IT-33*, 1 (Jan., 1987), 3–10.

[HH85] Hester, J. H. and Hirschberg, D. S. Self-organizing linear search. *ACM Comp. Sur. 17*, 3 (Sep., 1985), 295–311.

[HC87] Horspool, R. N., and Cormack, G. V. A locally adaptive data compression scheme. Technical Correspondence. *Commun. ACM 30*, 9 (Sept., 1987), 792–794.

[LH87] Lelewer, D. A. and Hirschberg, D. S. Data compression. *ACM Comp. Sur. 19*, 3 (Sep., 1987), 261–296.

[R87] Ryabko, B. Y. A locally adaptive data compression scheme. Technical Correspondence. *Commun. ACM 30*, 9 (Sept., 1987), p. 792.

[SH91a] Stauffer, L. M. and Hirschberg, D. S. Parallel data compression. Technical Report 91-44. Info. and Comp. Sci. Department, University of California, Irvine.

[S88] Storer, J. A. *Data Compression Methods and Theory*, Computer Science Press, Rockville, Maryland, 1988.

[TW89] Thomborson, C. D. and Wei, Belle W.-Y. Systolic implementations of a move-to-front text compressor. In *Proceedings 1989 ACM Symposium on Parallel Algorithms and Architectures*, Sante Fe, New Mex., ACM, New York, 1989, pp. 283–290.

[W84] Welch, T. A. A technique for high-performance data compression. *Computer 17*, 6 (June, 1984), 8–19.

[W91] Williams, R. N. *Adaptive Data Compression*, Kluwer Academic Publishers, Norwell, MA, 1991.

[Z90] Zito-Wolf, R. J. A broadcast/reduce architecture for high-speed data compression. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, 1990.

[ZL78] Ziv, J. and Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory 24*, 5 (1978), 530–536.