Sequential decoding of convolutional codes by a compressed multiple queue algorithm

H.-C. Kuo
C.-H. Wei

Indexing terms: Convolutional codes, Sequential decoding algorithms

Abstract: The conventional multiple stack algorithm (MSA) is an efficient approach for solving erasure problems in sequential decoding. However, the requirements of multiple stacks and large memory make its implementation difficult. Furthermore, the MSA allows only one stack to be in use at a time: the other stacks will stay idle until the process in that stack is terminated. Thus it seems difficult to implement the MSA with parallel processing technology. A two-stack scheme is proposed to achieve similar effects to the MSA. The scheme greatly reduces the loading for data transfer and I/O complexity required in the MSA, and makes parallel processing possible. An erasure-free sequential decoding algorithm for convolutional codes, the compressed multiple-queue algorithm (CMQA), is introduced, based on systolic priority queue technology, which can reorder the path metrics in a short and constant time. The decoding speed will therefore be much faster than in traditional sequential decoders using sorting methods. In the CMQA, a systolic priority queue is divided into two queues by adding control signals, thereby simplifying implementation. Computer simulations show that the CMQA outperforms the MSA in bit error rate, with about one-third the memory requirement of the MSA.

1 Introduction

Convolutional coding is a powerful error-correcting technique for communications over noisy channels [1]. Among the decoding algorithms for convolutional codes, the suboptimal sequential decoding algorithm and the optimal Viterbi decoding algorithm are most commonly used. The primary difference between these is that the Viterbi decoder uses an exhaustive trellis search in a code tree, whereas the sequential decoder searches only parts of this. Thus, the complexity of a Viterbi decoder grows exponentially in proportion to the constraint length of the convolutional code, and becomes infeasible for codes with long constraint lengths [1–3]. The sequential decoder, however, can cope with convolutional codes of any constraint length, and is therefore especially useful in applications where long ones are required [4, 5].

Among the several sequential decoding algorithms, the stack or ZJ algorithm [6, 7] is quite popular. Here the decoder always moves along the path visited with the largest metric until it reaches the end of the code tree. A large storage area, called the 'stack', is required to store all of the paths that have been visited by the decoder. Before paths can be further extended, those in the stack must be sorted to find the best one. This operation is very time-consuming, especially when a large stack is used, and so the decoding speed of the stack algorithm is limited [8]. Chang and Yao [8, 10] proposed an efficient approach to alleviate this. The stack memory is replaced by an array of processors known as a 'systolic priority queue', where the reordering of nodes is completed when they pass through. Although the metrics of nodes in such a queue are not in decreasing order, the node with largest metric can always be delivered quickly in a contant time, regardless of the queue size. Recently, Lavoie *et al.* [11] developed a new type of systolic priority queue, which can operate twice as fast as the standard one.

With systolic priority technology the speed of a stack algorithm will be much faster. Another problem for a stack algorithm is erasure due to input-buffer overflow, which is caused by long searches in the code tree [1]. Chevillat and Costello [9] proposed a multiple stack algorithm (MSA) to alleviate this problem, using, a large first stack and many higher-rank stacks. The decoding is started in the first stack; when this is full the top nodes will be transferred to a higher-rank stack, where decoding continues. Higher-rank stacks are usually much smaller than the first stack, and so when decoding continues in higher-rank stacks, the decoder can soon reach the end of the code tree and obtain a tentative decision, which will avoid the possibility of erasure. Although the MSA is a powerful algorithm for erasure-free sequential decoding, its implementation is not as easy as that of the traditional stack sequential decoding algorithm.

In this paper we present an algorithm which can be easily implemented with a systolic priority queue with properties and performances similar to those of the MSA. Systolic priority queue technology and the operations of a queue are briefly reviewed. As directly mapping the multiple stacks in the MSA to multiple queues will pose many challenges, such as unbounded complexity requirement in memory and I/O, a compressed multiple queue algorithm (CMQA) is proposed. This may be seen as a modification of the MSA. The systolic priority queue is suitable for implementing the CMQA, and its operating principle is introduced here. Comparisons of the bit error rate (BER) performance for both the MSA and CMQA by computer simulations are also given.

Paper 1281I (E5), first received 16th August 1993 and in revised form 17th March 1994

The authors are at the Institute of Electronics and Centre for Telecommunications Research, National Chiao-Tung University, Hsinchu, Taiwan, Republic of China

2 Systolic priority queue technology

One reason for the stack sequential decoder to be less popular than the Viterbi decoder is that the very time-consuming node reordering operation often slows down its speed. Because of this, most practical stack sequential decoders are implemented using the modified algorithm proposed by Jelinek, which is called the stack-bucket algorithm [7]. Although this algorithm can remedy the problem of low and variable speed in metrics reordering, the BER performance is also degraded [1].

To avoid the sorting operation without degrading the performance of the stack algorithm, Chang and Yao proposed an approach in 1986 [8, 10] whereby the stack memory is replaced by an array of processors. These processors, called a systolic priority queue [18, 19] are arranged so as to deliver the node with maximum metric quickly within a constant time interval. Fig. 1 shows a

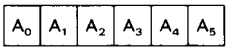

Fig. 1A *Basic structure of linear systolic priority queue*

	A_0	A_1	A_2	A_3	A_4	A_5
		16	6	9	7	
insert 11	11	16	6	9	7	
shift down		11	16	6	9	7
node reordering		16	11	9	6	7
shift up	16	11	9	6	7	
extract 16		11	9	6	7	
node reordering		11	9	7	6	
insert 3	3	11	9	7	6	
shift down		3	11	9	7	6
node reordering		11	3	9	7	6
shift up	11	3	9	7	6	
extract 11		3	9	7	6	
node reordering		9	3	7	6	

Fig. 1B *Example for the operation of the shift register scheme of a systolic priority queue*

linear systolic priority queue and an example of its operations in a shift register scheme [8]. For comparison with the new systolic priority queue described below, the mechanism of the shift-register (SR) systolic priority queue is repeated here.

2.1 Linear systolic priority queue [8]

The result of these operations is the insertion of one new node X into the queue, and the delivery of the best node in the queue. This is performed as follows

(*a*) Insertion of a new node X

(i) $A_0 \leftarrow X$

(ii) $A_{i+1} \leftarrow A_i$, i.e. shifting all nodes one position down

(iii) rearrange the nodes so that $A_{2i+1} \geq A_{2i+2}$, for $i \geq 0$.

(*b*) Deletion of the best node

(i) $A_i \leftarrow A_{i+1}$, i.e. shifting all nodes one position up, and A_0 will contain the best node for deletion. (Note that A_0 is an I/O port.)

(ii) rearrange the nodes so that $A_{2i+1} \geq A_{2i+2}$ for $i \geq 0$.

As these operations are completed simultaneously when nodes travel through the processors, the time to obtain the top node is always constant, no matter how many processors are used. This structure efficiently remedies the speed problem in searching for the best node. However, this linear systolic priority queue permits only one node at a time to go into the queue. To achieve a faster processing speed, Lavoie *et al.* [11] developed a new type of systolic priority queue, and have realised it using a full-custom VLSI chip. Apart from a novel circuit design for more concise operations, the processors in the queue are rearranged so that the queue can receive two nodes at a time and complete all operations in a single clock cycle, thus saving much more time. An example of the operations of this new systolic priority queue is given in Fig. 2*e*. The processors in one chip are divided into many 'slices', each consisting of three processors. Based on the basic operations, the connections between processors can be plotted as shown in Fig. 2*a*. Lines with double arrows represent those paths which should be able to transmit data in both directions, and will be more complex than lines with a single arrow. To make the mechanism of this queue clearer, each basic operation is explained step by step as follows, with the relevant circuits shown in Figs. 2*b*, 2*c* and 2*d*.

2.2 New systolic priority queue [11]

The result of these operations is the simultaneous insertion of two new nodes N_0 and N_1 into the queue, and delivery of the best node it contains. This is achieved by

(i) Insert two nodes simultaneously into the queue and shift all nodes two positions down, as shown in Fig. 2*b*, i.e.

insertion: $N_0 \rightarrow P_0$, $N_1 \rightarrow P_1$; then,

shifting: $P_{2i} \rightarrow P_{2i+2}$, $P_{2i+1} \rightarrow P_{2i+3}$, $i \geq 0$.

(ii) At the same time, rearrangement is conducted in each triplet of processors, move the best node in each triplet to its local top position. The positions of the other two nodes are trivial, as shown in Fig. 2*c*, i.e.

$P_{3i-1} \leftarrow \text{best}\{P_{3i-1}, P_{3i}, P_{3i+1}\}$, $i \geq 1$.

$P_{3i}, P_{3i+1} \leftarrow$ the other two nodes.

(iii) Extract the top node from the queue. At the same time shift the other nodes one position up, as shown in Fig. 2*d*, i.e.

$P_2 \rightarrow P_0$, $P_{i+1} \rightarrow P_i$, $i \geq 2$, P_0 extract out.

(Note that from (i) and (iii), P_0 and P_1 are treated as I/O ports.)

(iv) At the same time, rearrange each triplet of processors again, as shown in Fig. 2*c*, i.e.

$P_{3i-1} \leftarrow \text{best}\{P_{3i-1}, P_{3i}, P_{3i+1}\}$, $i \geq 1$.

$P_{3i}, P_{3i+1} \leftarrow$ the other two nodes.

Although this mechanism consists of many steps, with some circuit design tricks they can all be merged into two. The first consists of insertion, shifting down two positions and triplet sorting. The second consists of triplet sorting and shifting up one position. Thus with a two-phase clocking scheme these steps can be completed in one single clock cycle [11].

In the linear systolic priority queue only a single node is permitted to enter or exit the queue; it would therefore require three clock cycles to retrieve the best node and insert two succeeding nodes. Although each clock cycle for the original queue is shorter than the new queue, the total time required is longer because only one comparison instead of two is performed in a cycle. The timing

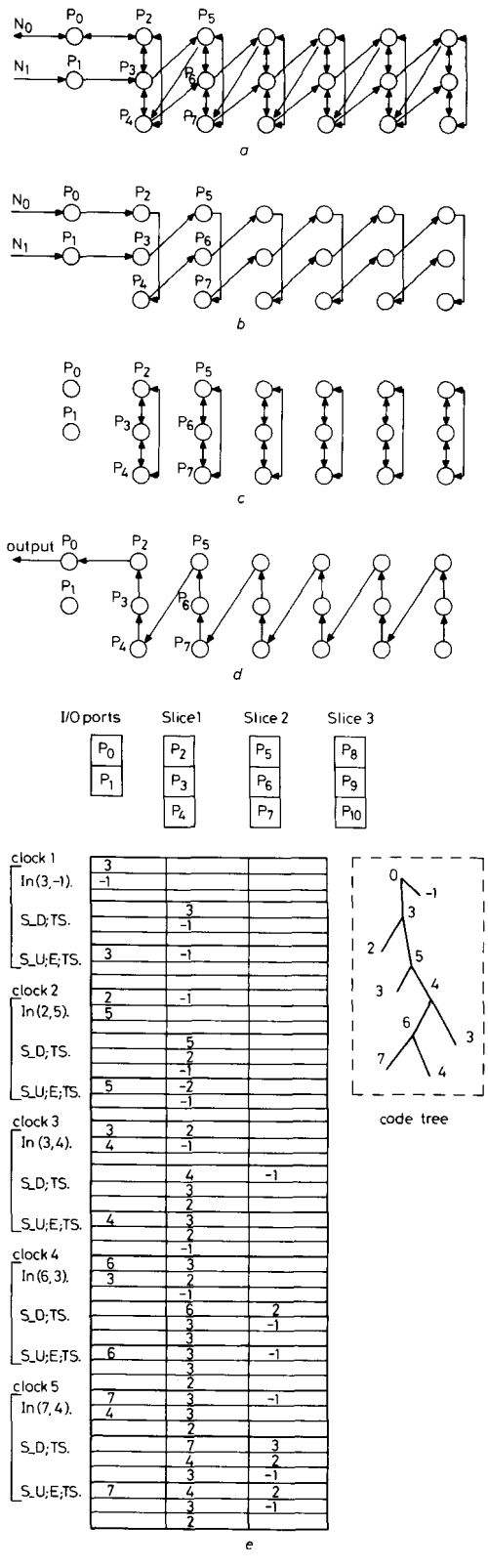

Fig. 2 *Systolic priority queue*

a Block diagram
b Insertion and shifting down two positions (only relevant parts are shown)
c Triplet sorting (only relevant parts are shown)
d Extract the best node and shift up one position (only relevant parts are shown)

Purposes of this step	Corresponding operations
Fig. 2*b*	
Insert two new nodes	$N_0 \rightarrow P_0$, $N_1 \rightarrow P_1$; and
Shift down two positions	$P_{2i} \rightarrow P_{2i+2}$, $P_{2i+1} \rightarrow P_{2i+3}$; $i \geq 0$
Fig. 2*c*	
Node rearrangement (Triplet sorting)	$P_{3i-1} \leftarrow \text{best}\{P_{3i-1}, P_{3i}, P_{3i+1}\}$, $i \geq 1$ $P_{3i}, P_{3i+1} \leftarrow$ the other smaller two nodes
Fig. 2*d*	
Extract best node Shift up one position	$P_2 \rightarrow P_0$, $P_{i+1} \rightarrow P_i$, $i \geq 2$, P_0 extracted out

e Example of operations in a systolic priority queue

In (*a*, *b*)	Insert *a* and *b*
S_D	Shift two positions down
S_U	Shift one position up
E	Extract best node
TS	Triplet sorting

schemes of these two versions of systolic priority queue were shown in Reference 11. As illustrated there, the new type of queue can be twice as fast as the original queue when completing the same jobs. This new systolic priority queue is well defined, so that it can be easily extended to an N-input systolic priority queue. For practical purpose, N is usually equal to 2^b, with $b \geq 0$. The generalised systolic priority queue is shown in Fig. 3. In

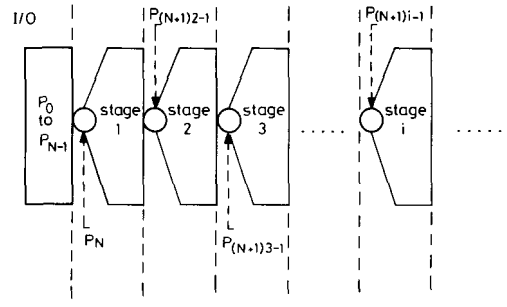

Fig. 3 *Generalised (N-input) systolic priority queue*

In each stage, the best node among $(N + 1)$ nodes should reside at the first place, e.g. at $P_{(N+1)i-1}$, when in stage i

this scheme, the former ith processors, where $0 \leq i \leq N - 1$, are treated as the I/O ports, and N nodes are inserted at a time. All nodes are shifted N positions down and the best node in each group of $(N + 1)$ nodes is kept in the first place, among the $(N + 1)$ positions. That is, the best node in positions $(N + 1)i - 1$ to $(N + 1)(i + 1) - 2$ should reside at position $(N + 1)i - 1$, for $i \geq 1$.

The assertion that the single-input queue always delivers the best node is examined in Reference 8, where it is generalised here to any N-input queue whose mechanism is similar to the linear or the new systolic priority queue.

Theorem 1: In the SR systolic priority queue, after any number of insertion or deletion operations, the ith good node is stored in some register A_k, where $1 \leq k \leq 2i - 1$ [8].

Theorem 2: For an N-input systolic priority queue, after any number of insertion or deletion operations, the ith

good node is stored in some processor P_k, where $N \leqslant k \leqslant (N+1)i-1$.

Proof of Theorem 2: This theorem is proved by induction as follows. The first several steps can be easily checked by observation, and so we may assume that after m steps of operation the ith good node resides in processor P_k, where $N \leqslant k \leqslant (N+1)i-1$. We must then prove that it is still true at the $(m+1)$th step.

A Assume that the $(m+1)$th operation is an insertion. It can be seen that, although N new nodes are inserted, the best node in the queue will still appear at P_N. Now consider the position for the ith good node, where $i \geqslant 2$. At first, if the ith good node before operation resides in P_k, where $N \leqslant k \leqslant (N+1)(i-1)-1$, then after the new N nodes have been inserted this node will still reside somewhere before the $((N+1)i-1)$th position; this is a legal position no matter what nodes are inserted (it should be noted that when $i=2$, the present case will not occur). On the other hand, if the ith good node resides in P_k with $(N+1)(i-1) \leqslant k \leqslant ((N+1)i-1)$, then after N new nodes having been inserted and node-rearranging having been completed, this node will reside at position P_k, with k equal to $(N+1)i-1$. This is because the other nodes to be compared with this previous ith good node all have ranks higher than i, otherwise Theorem 2 is violated at the mth step. As the new rank of this node will not be lower than i, this is again a legal position.

B Assume that the $(m+1)$th operation is a deletion. By the property of Theorem 2, it is easily seen that when the previous best node is deleted and the nodes are rearranged, the new best node will certainly appear at the first position P_N. As the previous second good node is originally at P_k with $N+1 \leqslant k \leqslant (N+1)2-1$, after shifting all nodes one position up this node will appear somewhere among the first $(N+1)$ positions, i.e. at some P_k with $N \leqslant k \leqslant (N+1)2-2$. Thus after rearranging nodes, this new best node will certainly reside at P_N.

Now consider the position for the ith good node. As above, we assume that Theorem 2 holds for the previous m steps, and wish to prove that it still holds at the $(m+1)$th step. At first, if the previous ith $(i \geqslant 3)$ good node resides at P_k with $N < k \leqslant (N+1)(i-1)-1$, then after shifting one position up and rearranging nodes, it can still reside at a legal position. This is because index k of the new position P_k is always less than $(N+1)(i-1)-1$, which is the largest legal position for the $(i-1)$th good node (new rank for that node). On the other hand, if the previous ith $(i \geqslant 3)$ good node resides at P_k with $(N+1)(i-1)-2 \leqslant k \leqslant (N+1)i-1$, then after shifting one position up and rearranging nodes, that node will appear at P_k with k equal to $(N+1)(i-1)-1$. This is because, in the rearrangement the nodes to be compared with the previous ith good node are exactly those nodes located originally at some P_k with $(N+1)(i-1)-2 \leqslant k \leqslant (N+1)i-1$. According to the property of Theorem 2, their ranks are all higher than i.

From **A** and **B**, it can be seen that Theorem 2 still holds at the $(m+1)$th step. By induction, Theorem 2 is always true after any number of steps of the algorithm.

The linear systolic priority queue [8] and the new systolic priority queue [11] may be viewed as special cases of Theorem 2, with N being 1 and 2, respectively. The structure of the case with N equal to 2 will be adopted to implement the algorithm developed in this paper. It is also found that the node arrangement is the key for good nodes to go forwards and the bad nodes to go backwards. If the node rearrangement is inhibited somewhere in the queue for each other cycle, then the data flow there will be controlled to allow only backward transmission. This is important for implementing the algorithm introduced here.

3 Erasure-free decoding

In the decoding process, the stack seqential decoder goes back and forth in the code tree to search for the correct path, and so the received sequence must be stored in an input buffer for later processing. If very long searches occur the input buffer will overflow, causing erasure to take place because the data are lost [1]. The decoding effort of a seqential decoder is a random variable with a Pareto distribution, i.e. the probability $P(C > N)$ that the number of computations C exceeds N decreases for large N is proportional to $N^{-\varphi}$, i.e.

$$P(C > N) = C_{SD} N^{-\varphi} \tag{1}$$

where φ depends on the channel and the rate R only, and C_{SD} is a constant [9, 12, 13]. Because of this property, no matter how large the computation effort and the input buffer, there are always some code words that cannot be decoded completely, and so the erasure problem due to buffer overflow always exists. In fact, the erasure probability becomes the major limitation on the performance of the code, because the error probability of sequential decoding can be made arbitrarily low [14].

Several methods have been proposed to reduce the erasure probability [7, 16], but the BER performances associated with these methods are also degraded. On the other hand, the generalised stack algorithm (GSA) proposed by Haccoun and Ferguson [15] is a method that reduces the erasure problem with no side effects. Furthermore, the MSA proposed by Chevillat and Costello [9] is also successful in conquering the problem. It is found that, by using GSA, lower bit error probability and erasure probability can be achieved by extending several nodes at the top of the stack simultaneously. Also, the variability of computation distribution is reduced. Thus better performance is achieved by multiprocessing, using less computation time and less memory. However, the erasure probability can not be completely alleviated with GSA, because the required computation effort is still Pareto-distributed [15].

The MSA is a method for completely erasure-free decoding. Unlike the stack algorithm, it requires a large first stack and many smaller stacks. Furthermore, during the decoding process many tentative results may be reached and be stored in a special register, which always keeps the best decision up to date. The mechanism of the MSA is illustrated in Fig. 4 and briefly described below.

3.1 Mechanism of the MSA

Step 1: The decoding begins in the first stack. As with the traditional stack sequential decoder, if a terminal node of the tree is reached before the first stack is full, the decoding is completed. However, if the first stack is full before the end of the tree is reached, the top T nodes are transformed to a stack of rank 2, and decoding continues there.

Step 2: Assume that the present stack is of rank i, $i \geqslant 2$. As decoding continues, two things may happen (it is important to note that the i is redeclared each time, and should not be confused with the rank of the previous operating stack).

Case A: If the stack is full before the end of the tree is reached, the top T nodes will be transferred to the stack

of rank $(i + 1)$, and decoding continues there. Go to step 2.

Case B: If a terminal node of the tree is reached, this node is treated as a tentative result. The tentative result

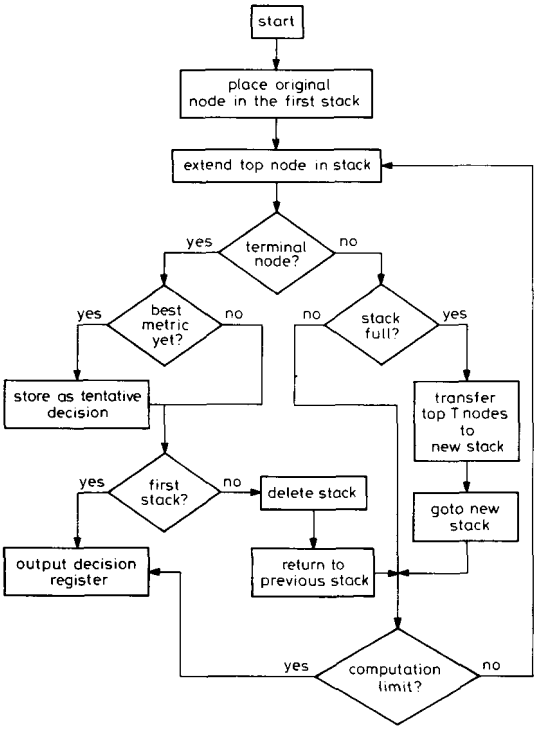

Fig. 4 *Multiple-stack algorithm*

may be stored in a special register, if this result is better or if the register is empty. After that, the present stack will be cleared and decoding continue in the stack of rank $(i - 1)$. Repeat step 2.

Step 2 will be repeated until the present rank is 1 and a tentative result is obtained there. That is, a terminal node is reached in the first stack. The only other case to terminate the decoding process is when the computation limit is reached.

In the MSA, every time a stack is full some nodes at the top are transferred to a higher-rank stack newly formed for further extension. Then only subsets of paths in the code tree, which are more likely to be correct, will be searched immediately. Thus, the decoder can go deeper and deeper into the tree after each transfer between stacks, so that the time needed to obtain a tentative decision is shortened. If u denotes the number of stacks formed before the first tentative decision is obtained, then the probability $P(u > v)$ that u exceeds v will decrease exponentially for sufficiently large values of v. Then the number of computations C_v executed before stack v overflows is given by [9]

$$C_v = Z_1 - 1 + (v - 1)(Z - T) \tag{2}$$

where Z denotes the size of the higher-rank stack, Z_1 is the size of the first stack and T is the number of transferred nodes. From these two properties, it follows that $P(C > C_v)$, i.e. the probability that the number of computations C required for reaching the first tentative decision exceeds C_v, will decrease exponentially with C_v.

The MSA and the stack algorithm can be illustrated from another viewpoint, as shown in Fig. 5. In the stack algorithm, any proper node can be fetched for further extension, i.e. nodes at any place may 'flow' to the branch

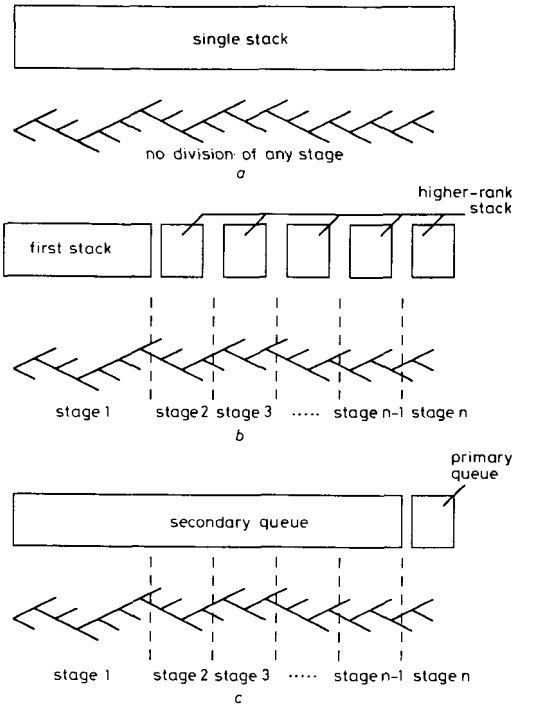

Fig. 5 *Decoding trees for algorithms and the memory configuration*
a Stack algorithm
b Multiple-stack algorithm
c Compressed multiple-queue algorithm

extender. In the MSA, however, such flow direction will sometimes be inhibited. For example, when nodes are transferred to higher-rank stacks, decoding is continued there. Thus nodes not transferred will not be processed now, i.e. cannot 'flow' to the branch extender. Because of this, when the decoder goes through the decoding tree and reaches stages with fewer nodes (see Fig. 5), decoding in the MSA will become much faster, reaching the end of the tree quickly. Unlike most methods designed for remedying the erasure problem, the MSA can perform at least as well as the single-stack algorithm and be totally erasure-free [9]. However, it still has the problem of requiring too large a memory and using sorting to rearrange nodes.

Systolic priority queue technology can remedy the sorting problem in the MSA but many queues will be required, thereby causing a serious problem. As every queue needs an I/O port of several tens of bit-lines per queue, multiple queues will require multiple I/O ports, thus leading to tremendous I/O bandwidth. For example [9], if the size of the newly formed stack is chosen to contain 11 elements, the total stacks needed are about 20–30, which means that about 1000 bit-lines will be connected between the memory and the branch extension unit. This requirement for many stacks will become even more serious if more information bits are to be transmitted in a frame. Such a result will make layout routeing unrealisable, or else a very complex I/O switching control

mechanism must be incorporated to implement the algorithm. Furthermore, only one stack is active at a time for the MSA, and so it will make little difference when parallel processing techniques are adopted. The large memory size required in the MSA, usually in the range of 3000–5000 [9], is another problem to implementing the algorithm with the systolic priority queue, because the result may become infeasible.

4 Compressed multiple queue algorithm

Because the arrangement of I/O ports is very complicated and vast memory space is necessary for the MSA, it is impractical to implement it using a systolic priority queue. Here we present an algorithm that saves these two resources while still keeping the desirable performance of the MSA. This is called the compressed multiple queue algorithm (CMQA), as it uses only two queues to process the jobs traditionally processed by multiple stacks.

4.1 Operating principle of CMQA

The key requirement for the MSA to be erasure-free restricts the node searching operation to the deeper subtree, so that it can reach a terminal node quickly. Thus it seems unnecessary to keep the nodes visited for each searching iteration in so many separate stacks (Fig. 5*c*), that is, the nodes being processed now can be kept in a small stack, and all nodes already visited may be stored in a large stack.

In the CMQA, when the small stack is full the worst nodes are shifted out and put into the large stack, so that decoding can continue in the small stack. Therefore, the multiple stacks required by the MSA are compressed into two, a small primary stack and a large secondary stack. In such a scheme, the primary stack always operates as the newest stack in the MSA, whereas the secondary stack always 'absorbs' the other stacks which are not currently being used.

Although an extra operation for stack compression is needed in CMQA, this algorithm will be more time-saving than the MSA. This is because, when implementing the CMQA with the systolic priority queue, the formation and compression of multiple stacks can be completed at the same time and at the same memory locations. The flowchart of the CMQA introduced below is shown in Fig. 6. For easy comparison with the MSA, the memory used in this algorithm is also referred to as a 'stack'; however, it is actually a systolic priority queue.

4.2 Mechanism of the CMQA

Step 1: At the beginning the operation is conducted in a large stack. If a terminal node is reached before the stack is full, the decoding results are the same as the single-stack algorithm. If the stack is full, go to step 2.

Step 2: Partition the original stack used in step 1 into two parts. Of these, the smaller stack is called the primary stack and contains the nodes from the top of the original stack. The other is called the secondary stack, and is usually much larger than the primary stack.

Step 3: Extend the top nodes in the primary stack and then rearrange the extended new nodes and the nodes in the primary stack. At the same time, rearrange the nodes overflowed from the primary stack and the nodes in the secondary stack. If a tentive decision is obtained in the primary stack, go to step 4; otherwise, repeat step 3.

Step 4: If a tentative decision is obtained in the primary stack, clear the primary stack and then merge the primary stack with the secondary stack. Now, the merged stack is the same as the original stack used in step 1.

Step 5: If the stack is full again, go to step 2. Otherwise, decoding will continue in this stack. The only con-

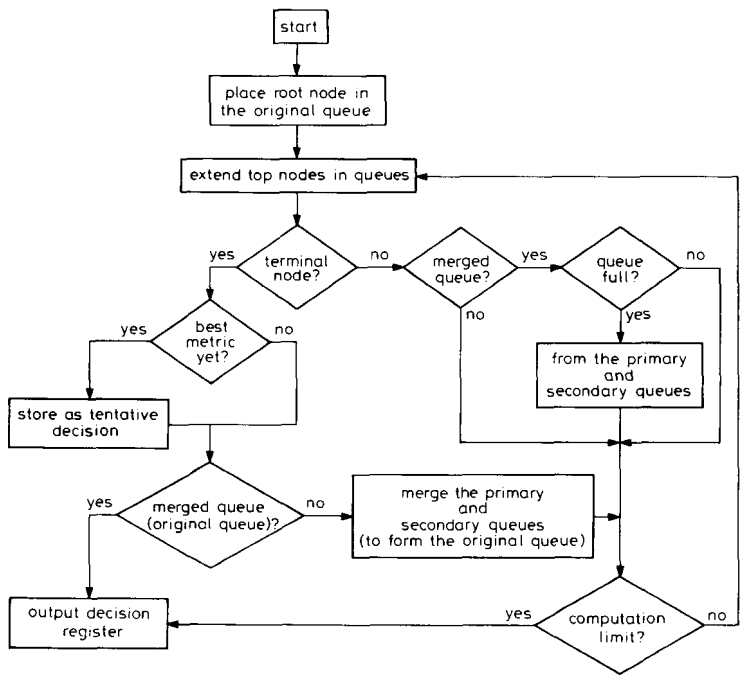

Fig. 6 *Compressed-multiple queue algorithm (CMQA)*

dition for terminating the decoding procedure is when a tentative decision is obtained during step 5, or when it reaches the computation limit.

Examining the procedures for obtaining the first tentative decision in the CMQA and the MSA, it is found that the major operations are conducted in the primary stack (or the highest-rank stack in the MSA). Because of this similarity, it is reasonable that properties 6 and 7 in Reference 9 and eqn. 2 hold for the CMQA (if v in eqn. 2 is replaced by the number of stacks 'absorbed' by the secondary stack). Thus the computation effort required by the CMQA to reach the first tentative decision is also exponentially distributed.

4.3 Implementation of CMQA

As the mechanisms in the above five steps are realised with the systolic priority queue, stacks will be referred to as queues in the following; for example, the primary queue and the multiple-queue algorithm (MSA implemented with the queues).

At first, if the original stack is replaced by a queue in step 1, the algorithm will become a single-stack algorithm implemented with a queue as in Reference 11. In step 2, the original queue is partitioned into two parts. As all nodes still stay where they are in the original queue, no node transfer or other operations actually occur. Therefore no time or hardware redundancy will be required in the queue partition. The operations in step 2 are similar to the multiple-queue algorithm: that is, a new queue is formed and the top nodes are simultaneous transferred into it. However, the size of this new queue is usually smaller than those used in the multiple-queue algorithm.

In step 3, two things are to be completed: the first is to extend the top node in the primary queue and then rearrange the newly extended nodes and the nodes in the primary queue; and the second is to reorder the nodes overflowed from the primary queue with those in the secondary queue.

Rearranging nodes simultaneously in the secondary queue is of crucial importance for this algorithm, as a terminal node may be obtained in the primary queue at any time and the primary queue will be cleared. Simultaneously completing these two jobs is difficult for traditional technology, but is easy with the parallel processing capability of the systolic priority queue. Furthermore, no compression or formation of queues actually happens, although some extra control signals are necessary.

The arrangement of control signals and the partition of queues is illustrated in Fig. 7. At first, as shown in Fig. 7*a*, to form the primary queue the nodes P_k in the original queue (where $k \leqslant 3i + 2$, i may be 1, 2 or some other small integer) are now declared to belong to the primary queue. The other nodes in the original queue then belong to the secondary queue. As the declaration is an abstract idea, the node positions for the primary queue will change with time. For example, when all elements are shifted two positions down, the primary queue is also shifted down, as shown in Fig. 7*b*. Also, when all nodes are shifted one position up, the primary queue and the secondary queue are shifted up, as shown in Fig. 7*c*. The reason the index k for P_k is at first chosen to be $k = 3i + 2$ is that, shifting two positions down, the primary queue contains exactly triplets of processors (see Fig. 7*b*), so that the node reordering operation can be conducted independently in both the primary queue and the secondary queue.

However, after all nodes have been shifted up, if the node reordering operation is conducted everywhere as usual it becomes meaningless to distinguish the primary queue from the secondary queue. This is because, at this time, if the reordering operation occurs on the boundary between these two queues, e.g. P_5–P_7 in Fig. 7*d*, nodes in

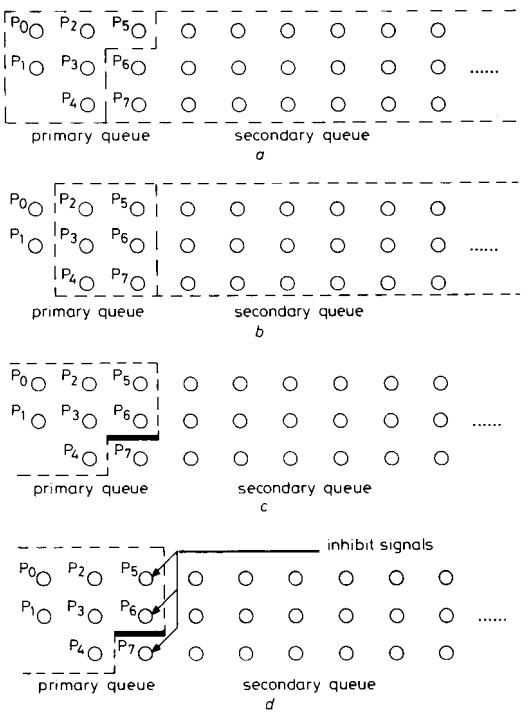

Fig. 7 *The queues*

a Division into primary and secondary queues
b Shifting both queues two positions down
c Shifting both queues one position up (bold line indicates the boundary between queues)
d Adding the inhibit signals

the secondary queue may go into the primary queue. Thus the decoding operations will not be confined within a certain subtree, and the decoder may no longer be erasure-free.

To prevent the elements in the secondary queue from getting into the primary queue, the second node reordering operation (i.e. the reordering operation immediately after the 'shift one position up' operation) should be inhibited on the boundary between two queues. Inhibit signals are used for blocking the paths for node exchanging on the boundaries of queues, as shown in Fig. 7*d*. By enabling or disabling the inhibit signals, the operations in step 2 and step 3 can be conducted on a systolic priority queue. With such inhibition, nodes in the primary queue can still flow to the secondary queue. (As can be seen, P_6 in Fig. 7*d* belonging to the primary queue will belong to the secondary queue in Fig. 7*c*, i.e. in the next iteration.) However, nodes in the secondary queue can no longer get into the primary queue. To merge the queues, as required in steps 4 and 5, the inhibit signals are completely disabled and the original queue can work normally again. With these modifications, the CMQA can be easily implemented with the systolic priority queue.

4.4 Memory considerations

As in the MSA, the control of memory space is a problem for the CMQA. In general, the requirements of large

memory space are usually found in the traditional stack sequential decoders. However, in a practical stack sequential decoder, the control of memory space is simpler. The penetration depths of a correct path in stacks has been studied in Reference 17. It is found that the correct path usually stays near or at the top of a stack. Thus the possibility of correct node loss can be eliminated with moderate memory size.

Because limited memory is used to implement the CMQA, overflow must be allowed in the original queue to achieve better performance. The original queue works until it is divided into two queues, and so its performance will be determined by the queue division time T_d. Different times to divide the original queue will result in quite different performances. For example, if the queue division time T_d is set too small, the probability of the correct path not being found will greatly increase and, worse, the correct node may be lost forever. On the other hand, if the original queue is maintained for a moderate duration even though it overflows, the BER performance will be better. However, because the computation time is also limited in the CMQA, too-late division of queues will increase the possibility of incomplete decoding. To minimise the probability of losing the correct node, and at the same time to avoid incomplete decoding, T_d should be chosen carefully so that it is large enough but below some upper bound. This upper bound is derived as follows.

Every time the primary queue is formed, N nodes are transferred to the queue. Then, in the worst case, to make a possible correct path in this queue one branch deeper into the tree may require N computations. If nodes in the primary queue are all very close to the root at the beginning, then at a conservative estimate it will require $N \times$ (*depth of decoding tree*) iterations of computation to reach a tentative decision, i.e. to avoid incomplete decoding. Accordingly, subtracting this estimated value from the computation limit C_{limit}, the upper bound of queue division time T_d can be determined. For example, in the computer simulations shown later, $N = 4$ and *depth of decoding tree* = 500, so that T_d may be chosen to be (C_{limit} − 2000) or smaller.

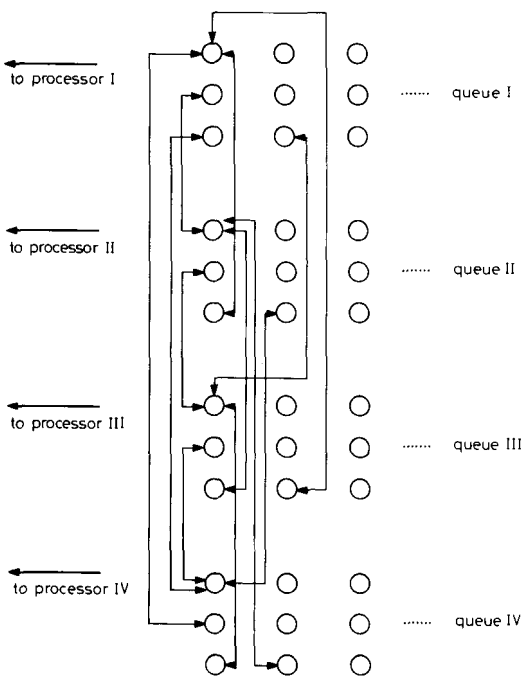

Fig. 8A *Connections between four-processor CMQA for further comparisons between top nodes in queues*

Multiprocessing is a way to improve memory usage efficiency and increase tree searching speed [15]. However, it is not easy to adopt in the CMQA, because the systolic priority queue can extract only the best node, but it will require more than one node in a multiprocessor-type CMQA. Our approach to solving the problem is to use one queue for each processor. The multiprocessor-type CMQA is shown in Fig. 8A; four processors and four queues are used in this example. The double arrowed lines shown in the Figure indicate that further comparisons are required between queues. These are of crucial importance if better performance is desired. As illustrated in Fig. 8B, although not all of the best four nodes are obtained in each turn, the performance will be better than when no further comparisons are included. The reason is that good nodes usually fall into some queue, and without these further comparisons other good nodes may be buried under the best node in that queue. With further comparisons, these good nodes can appear at the top of other queues. A graphic example to illustrate this idea is shown in Fig. 8C. Computer simulations

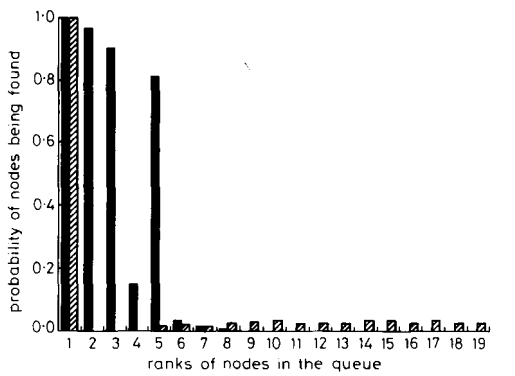

Fig. 8B *Probabilities for the top nodes to be extended*

If further comparisons of top queue elements between queues are adopted, other better nodes can be found

■ further comparisons are adopted

▨ no comparison between queues

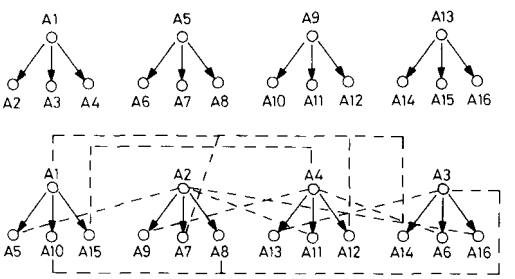

Fig. 8C *Example to illustrate the benefit of further comparisons in the multiprocessor version of CMQA*

The top example shows the worst case if no further comparison is used (where $Ai > Aj$, whenever $i > j$). In this case, the top nodes selected for further extension are nodes of rank one, five, nine, and thirteen. This is because nodes of rank two, three, and four are 'buried' by that of rank one

The lower example shows that if the simple configuration shown in Fig. 7*a* is used, the order of these nodes will become as above. Note that the nodes of rank one, two, three and four are all selected

(The dashed line is used to illustrate the paths connected for comparison)

prove that good performance can be achieved with this scheme.

4.5 Speed considerations

The following compares the operation speeds of the CMQA and the MSA. As some circuit design tricks are adopted in the implementation of a systolic priority queue, and there is still no practical implementation for the MSA, some assumptions are necessary to make the following comparisons more fair. First the node reordering operation is extracted from the integrated operation of a systolic priority queue. This is because the speed of this operation is crucial for the total speed of a single-stack algorithm and the MSA. However, it is important to note that, for a real VLSI circuit of a systolic priority queue, all node reordering and node shifting operations are conducted in parallel with the best-node retrieving (READ) and the new-node insertion (WRITE) operations. That is, only the time delay for one READ and one WRITE is required during a decoding cycle for a systolic priority queue. Secondly, it is assumed that the sorting operation in the MSA is replaced by the best node searching operation. Furthermore, it is assumed that the comparators used in the MSA are as many as required, although in fact only one processor is usually used. Using the comparators to find the best of N nodes $\lceil \log_2 N \rceil$ iterations are required, where $\lceil \alpha \rceil$ is the minimum integer greater than α. Thus $2*\lceil \log_2 3 \rceil$ iterations are required for the CMQA. For the MSA, $\lceil \log_2 Z_i \rceil$ iterations are required, where Z_i is the size of the ith rank stack that the MSA is working on. The operations required for the CMQA and the MSA during one decoding cycle are shown in Table 1. It is found that the

Table 1: Operations for the CMQA and MSA during one decoding cycle

CMQA (when the queue is not divided)	CMQA (when using the high-rank memory)	MSA (when using the first stack)	MSA (when using the high-rank memory)
1 READ 1 WRITE 4 comparisons	1 READ 1 WRITE 4 comparisons	1 READ 1 WRITE 10 comparisons (size of the first stack = 1000) 11 comparisons (size of the first stack = 2000)	1 READ 1 WRITE 4 comparisons (size of the high rank stack = 11)

MSA is slower than the CMQA even with similar implementation technology. If the practical circuit is taken into account, the speed of the CMQA will be faster (only one READ and one WRITE).

5 Computer simulations and discussion

The performance of the CMQA is evaluted by computer simulation. The MSA was also simulated under identical conditions for comparison. A convolutional code with rate R equal to 1/2 and constraint length L equal to 15 was used, where the generator polynomials were

$$G_1(X) = 1 + X + X^4 + X^6 + X^7 + X^8 + X^{10} + X^{11} + X^{13} + X^{15}$$

$$G_2(X) = 1 + X^3 + X^6 + X^7 + X^8 + X^{10} + X^{12} + X^{14} + X^{15}$$

All coded data sequences were transmitted through the same binary symmetric channel with white Gaussian noise; 500 bits per frame were transmitted.

The performances of the CMQA decoder with different queue division time T_d and memory size are illustrated in Fig. 9A. Unlike most stack sequential decoders,

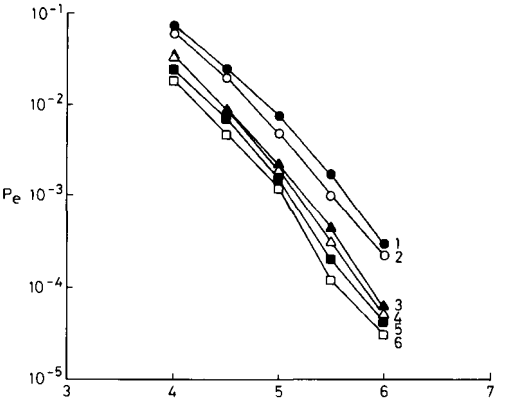

Fig. 9A *Influence of queue division time T_d on the performance of CMQA*

—●— curve 1 (800 elements; T_d = 800)
—○— curve 2 (1000 elements; T_d = 1000)
—▲— curve 3 (1000 elements; T_d = 1800)
—△— curve 4 (800 elements; T_d = 2100)
—■— curve 5 (1000 elements; T_d = 2400)
—□— curve 6 (800 elements; T_d = 3200)

the CMQA decoder with a larger memory does not always perform better than those with a smaller memory. For example, the performance shown by curve 2, which corresponds to a decoder with 1000 queue elements, was poorer than those shown by curves 4 and 6, which correspond to the decoders with 800 queue elements. This illustrates that the queue division time T_d will influence the BER performance. In curve 2, queue division took place just after the queue overflowed. However, in curves 4 and 6, queue division took place after the queue had overflowed thousands of times. As shown in the previous Section, the upper bound to ensure erasure-free decoding in these simulations was $2096(T_d = C_{limit} - N*Depth\ of\ the\ coding\ tree$, where C_{limit} was 4096, N was 4 and *depth of the coding tree* was 500). In curve 6, T_d was much larger than this, so that erasures sometimes occurred. In curve 5, although T_d was slightly larger than the bound, no erasure occurred during the simulation. This is because the bound was derived under the worst case; in most cases this would not happen.

The performance of the multiprocessor-type CMQA is shown in Fig. 9B, where the performance of the single-processor CMQA shown by curves 4 and 5 in Fig. 9A is duplicated for comparison. A four-processor CMQA is used here. As each processor corresponds to one systolic priority queue, four systolic priority queues are used. Using the same the total memory, the size of each queue is one-quarter of that in the single-processor-type CMQA. In a multiprocessor type CMQA, the determination of the queue division time is more important than in the single-processor CMQA. If the queue division time is selected to be equal to the queue overflow time, as shown by curve 1 in Fig. 9B, the performance will become very poor. The performances shown in curves 3 and 5 are good, but erasures sometimes occurred because

T_d was too large. T_d in curves 2 and 4 was properly selected, so that the performances were better than that in curve 1 and no erasure occurred. When compared with curves a4 and a5, which show the performance of a

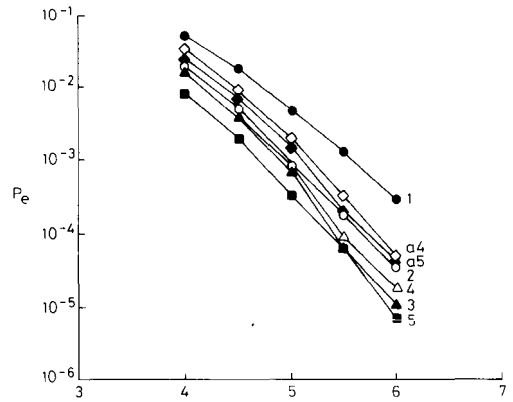

Fig. 9B *Influence of T_d on the performance of multiprocessing CMQA*

—●— curve 1
(1000 elements; T_d = 250)
—◇- - curve a4
(single processor 800 elements; T_d = 2100)
—◆— curve a5
(single processor 1000 elements; T_d = 2400)
—○— curve 2
(800 elements; T_d = 2000)
—△— curve 4
(1000 elements; T_d = 2000)
—▲— curve 3
(800 elements; T_d = 3400)
—■— curve 5
(1000 elements; T_d = 3500)

single-processor-type CMQA, it is found that the multiprocessor CMQA can achieve better BER performance with the same memory sizes.

Comparisons between the performances of CMQA and MSA are illustrated in Fig. 9C. Curves 2 and 4 in Fig. 9B are chosen for these comparisons, because they are truly erasure-free. In the MSA simulated here, the first stack is chosen to contain either 1000 or 2000 elements, and the size for each higher-rank stack is 11 elements. The same computation limit as that in the CMQA (C_{limit} = 4096) or larger (C_{limit} = 8192) is used. It is seen in Fig. 9C that the size of the first stack in the MSA has a great influence on its performance. The performance of the MSA with the first stack size equal to 2000 is better than that of the MSA with the first stack size equal to 1000. For example, when S/N is at 6 dB, the MSA with its first stack equal to 2000 can achieve a bit error rate of 7.35×10^{-5}, but at the same S/N the MSA with its first stack equal to 1000 can only achieve 2.32×10^{4}.

From Fig. 9C it is obvious that the CMQA can achieve better performance than the MSA, using much

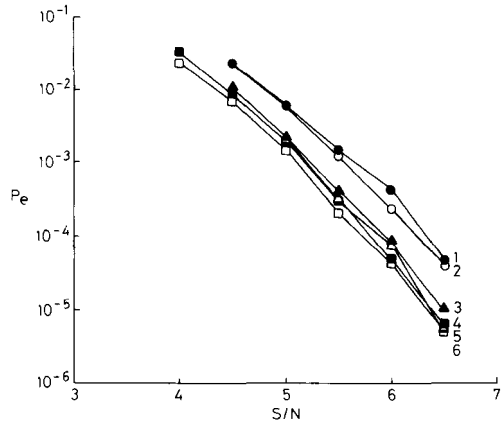

Fig. 9C *Performance comparisons of the CMQA vs. the MSA*

Note: Z1 = first stack size; Z = total memory size
—●— curve 1
(MSA: Z_1 = 1000, Z = 3079; C_{limit} = 4096)
—○— curve 2
(MSA: Z_1 = 1000, Z = 4112; C_{limit} = 8192)
—▲— curve 3
(MSA: Z_1 = 2000, Z = 3101; C_{limit} = 4096)
—△— curve 4
(MSA: Z_1 = 2000, Z = 3090; C_{limit} = 8192)
—■— curve 5
(CMQA: 800 elements, T_d = 2100; C_{limit} = 4096)
—□— curve 6
(CMQA: 1000 elements, T_d = 2400; C_{limit} = 4096)

Table 2: Performance comparisons for CMQA vs. MSA on the AWGN channel with (2, 1, 12) convolutional code G = [42554, 77304], d_{free} = 16

	CMQA	CMQA	MSA Z_1 = 1000	MSA Z_1 = 2000	MSA Z_1 = 1000	MSA Z_1 = 2000
Computation limit	4096	4096	4096	4096	8192	8192
Maximum memory used	800	1000	3112	4035	3244	4156
S/N = 6 dB	6.7×10^{-5}	3.7×10^{-5}	9×10^{-4}	9×10^{-5}	7.2×10^{-4}	9.5×10^{-5}
S/N = 5.5 dB	8.5×10^{-4}	7.3×10^{-4}	5.2×10^{-3}	1.6×10^{-3}	3.66×10^{-3}	1.3×10^{-3}
S/N = 5 dB	3.6×10^{-3}	7.6×10^{-4}	2.1×10^{-2}	8.2×10^{-3}	1.0×10^{-2}	6.6×10^{-3}
S/N = 4.5 dB	1.3×10^{-2}	1.0×10^{-2}	7.0×10^{-2}	3.0×10^{-2}	6.6×10^{-2}	3.6×10^{-2}
S/N = 4 dB	6.0×10^{-2}	4.8×10^{-2}	1.1×10^{-1}	1.0×10^{-1}	1.1×10^{-1}	1.0×10^{-1}

Table 3: Performance comparisons for CMQA vs. MSA on the AWGN channel with (4, 1, 12) convolutional code G = [44624, 52374, 66754, 73534], d_{free} = 33

	CMQA	CMQA	MSA Z_1 = 1000	MSA Z_1 = 2000	MSA Z_1 = 1000	MSA Z_1 = 2000
Computation limit	4096	4096	4096	4096	8192	8192
Maximum memory used	800	1000	2980	3892	1352	2000
S/N = 6 dB	4.0×10^{-6}	3.4×10^{-6}	4.4×10^{-6}	2.66×10^{-6}	4.0×10^{-6}	2.83×10^{-6}
S/N = 5.5 dB	6.0×10^{-6}	5.5×10^{-6}	8.5×10^{-6}	6.3×10^{-6}	7.0×10^{-6}	5.5×10^{-6}
S/N = 5 dB	1.27×10^{-5}	9.4×10^{-6}	1.33×10^{-5}	1.16×10^{-5}	1.21×10^{-5}	1.11×10^{-5}
S/N = 4.5 dB	2.0×10^{-5}	1.9×10^{-5}	2.6×10^{-5}	2.3×10^{-5}	2.45×10^{-5}	1.95×10^{-5}
S/N = 4 dB	4.57×10^{-5}	2.9×10^{-5}	1.1×10^{-4}	8.75×10^{-5}	3.2×10^{-5}	2.15×10^{-5}

Table 4: Performance comparisons for CMQA vs. MSA on the AWGN channel with (4, 3, 9) convolutional code

$$G = \begin{bmatrix} 10 & 03 & 07 & 14 \\ 01 & 15 & 04 & 16 \\ 07 & 00 & 14 & 15 \end{bmatrix}, d_{free} = 8$$

	CMQA	CMQA	MSA $Z_1 = 1000$	MSA $Z_1 = 2000$	MSA $Z_1 = 1000$	MSA $Z_1 = 2000$
Computation limit	4096	4096	4096	4096	8192	8192
Maximum memory used	800	1000	2419	3221	2375	3276
S/N = 7 dB	7.8×10^{-3}	7.25×10^{-3}	2.33×10^{-2}	1.67×10^{-2}	2.3×10^{-2}	1.45×10^{-2}
S/N = 6.5 dB	2.3×10^{-2}	2.1×10^{-2}	6.0×10^{-2}	3.6×10^{-2}	6.45×10^{-2}	4.25×10^{-2}
S/N = 6 dB	5.7×10^{-2}	5.2×10^{-2}	1.3×10^{-1}	9.2×10^{-2}	1.27×10^{-1}	1.06×10^{-1}
S/N = 5.5 dB	1.28×10^{-1}	1.22×10^{-1}	2.09×10^{-1}	1.69×10^{-1}	2.25×10^{-1}	1.77×10^{-1}

less memory. For example, when S/N is equal to 6 dB, with only 800 queue elements, the CMQA can achieve a bit error rate of 5×10^{-5}. At the same S/N, with a first stack of size equal to 2000 and a total memory size equal to 4122, the MSA can only achieve a bit error rate of 7.35×10^{-5}.

Some simulation results of the other three convolutional codes are included in Tables 2–4. The CMQA in all these tables are implemented in a four-processor scheme. In Table 2, a (2, 1, 12) convolutional code is used. It is found that, except for some degradation, the BER performances of the CMQA and the MSA are similar to those for a (2, 1, 15) convolutional code. This is because the possibility of using higher-rank memory (i.e. the possibility that erasure may occur in a single-stack algorithm) is similar for both codes at those S/N ratios. In Table 3, a (4, 1, 12) convolutional code is used. Because this is a powerful code, the BER performance is much better. Furthermore, the performances of the CMQA and the MSA are quite close. This is reasonable, as the possibility of using higher-rank memory is very low for this code at those S/N ratios. Under these conditions, both the CMQA and the MSA operate like a single-stack algorithm, and the minor different in BER performances are due to different memory size and number of processors used. In Table 4, a (4, 3, 9) convolutional code is used. For this code at these S/N ratios the frequency of using higher-rank memory is quite high for both the CMQA and the MSA. From Table 4 it is found that the BER performance for the CMQA is half or one-third of that for the MSA. As the bit errors here are generated mainly in those cases where higher-rank memories are used, the improvement that the CMQA can achieve over the MSA is clear. Thus the CMQA can achieve better BER with less memory than the MSA and is more I/O efficient.

6 Conclusion

The systolic priority queue has been shown to be a promising technique for implementing a high-speed single-stack sequential decoder. A compressed multiple queue algorithm (CMQA) is introduced to implement an erasure-free decoder with the systolic priority queue. Owing to the two-stack scheme proposed for the CMQA, the I/O is much simpler than that required in the multiple-stack algorithm. Furthermore, except for some extra control signals required, the implementation for the CMQA with the systolic priority queue is as easy as that for the traditional single-stack algorithm. Computer simulations show that the CMQA can achieve similar performance to that of the MSA, using only one-quarter to one-third the memory space.

7 References

1 LIN, S., and COSTELLO, D.J. Jr.: 'Error control coding: fundamentals and applications' (Prentice-Hall, New Jersey, 1983)
2 VITERBI, A.J., and OMURA, J.K.: 'Principles of digital communication and coding' (McGaw-Hill, 1979)
3 CLARK, G.C., and CAIN, J.B.: 'Error-correction coding for digital communications' (Plenum Press, New York, 1981)
4 CAIN, J.B., CLARK, G.C. Jr., and GEIST, J.M.: 'Punctured convolutional codes of rate $(n-1)/n$ and simplified maximum likelihood decoding', *IEEE Trans.*, 1979, **IT-25**, pp. 97–100
5 HACCOUN, D., and BEGIN, G.: 'High rate punctured convolutional codes for Viterbi and sequential decoder', *IEEE Trans.*, 1989, **COM-37**, pp. 1113–1125
6 ZIGANGIROV, K.: 'Some sequential decoding procedures', *Probl. Peredachi Inf.*, 1966, **2**, pp. 13–25
7 JELINEK, F.: 'A fast sequential decoding algorithm using a stack', *IBM J. Res. Dev.*, 1969, **13**, pp. 675–685
8 CHANG, C.Y.: 'Systolic array architecture for convolutional decoding algorithms: Viterbi algorithm and stack algorithm' (PhD dissertation, University of California, Los Angeles, 1986)
9 CHEVILLAT, P.R., and COSTELLO, D.J. Jr.: 'A multiple stack algorithm for erasure-free decoding of convolutional codes', *IEEE Trans.*, 1977, **COM-25**, pp. 1460–1470
10 CHANG, C.Y., and YAO, K.: 'Systolic array architecture for the sequential stack decoding algorithm', *Proc. SPIE*, 1986, **696**, pp. 196–203
11 LAVOIE, P., BELZILE, J., TOULGOAT, M., HACCOUN, D., and SAVARIA, Y.: 'VLSI design of a systolic priority queue chip for sequential decoders'. Proc. 1988 Canadian Conf. VLSI, Halifax, Nova Scotia, Canada, 1988, pp. 1–9
12 SAVAGE, J.E.: 'Sequential decoding — the computation problem', *Bell Syst. Tech. J.*, 1966, **45**, pp. 149–175
13 FORNEY, G.D.: 'Convolutional codes III: sequential decoding', *Info. and Control*, 1974, **25**, pp. 267–297
14 FANO, R.M.: 'A heuristic discussion of probabilistic decoding', *IEEE Trans.*, 1963, **IT-9**, pp. 64–74
15 HACCOUN, D., and FERGUSON, M.J.: 'Generalized stack algorithms for decoding convolutional codes', *IEEE Trans.*, 1975, **IT-21**, pp. 638–651
16 FORNEY, G.D., and BOWER, E.K.: 'A high speed sequential decoder: prototype design and test', *IEEE Trans.*, 1971, **COM-19**, pp. 821–835
17 GOULD, T.M., and HARRIS, J.H.: 'Single-chip design of bit-error-correcting stack decoders', *IEEE JSSC*, 1992, **SC-27**, pp. 768–775
18 GUIBAS, L.J., and LIANG, F.M.: 'Systolic stacks, queues, and counters', 1982 Conference on Advanced Research in VLSI, MIT
19 LEISERSON, C.E.: 'Systolic priority queue'. Proc. of Caltech Conf. on VLSI, 1979