Locally Connected VLSI Architectures for the Viterbi Algorithm

P. GLENN GULAK, MEMBER, IEEE, AND THOMAS KAILATH, FELLOW, IEEE

Abstract—**The Viterbi algorithm is a well-established technique for channel and source decoding in high performance digital communication systems. Implementations of the Viterbi algorithm on three types of locally connected processor arrays are described. This restriction is motivated by the fact that both the cost and performance metrics of VLSI favor architectures in which on-chip interprocessor communication is localized. Each of the structures presented can accommodate arbitrary alphabet sizes and algorithm memory lengths. The relative performance tradeoffs available to the designer are discussed in the context of previous work.**

I. Introduction

It is generally agreed that the vastly increased gate density (10^6–10^9 transistors per chip) promised by VLSI technology will rarely be utilized effectively if simultaneous efforts are not undertaken to devise algorithms that can be properly matched to VLSI structures. In this paper, we present several types of processor arrays that will allow us to apply the technological capabilities of VLSI to a specific problem that is of considerable interest in high-speed reliable digital communications. In particular, we pursue the development of high performance VLSI circuits for implementing Viterbi decoders under the assumption of a strictly local interconnection strategy among numerous, simple, identical processors that operate in parallel. This restriction is motivated by the fact that both the cost and performance metrics of VLSI favor architectures [1] in which on-chip interprocessor communication (i.e., wiring) is localized.

A. Historical Perspective

Although the Viterbi algorithm (VA) was discovered independently and first applied to the decoding of convolutional codes by Viterbi in 1967, it was at that time a well-known technique in operations research [2]. Omura was the first to observe this [3], by pointing out that the VA was in fact a special case of forward dynamic programming. Although not reviewed here, further details on the theory behind the VA are widely available in the literature [4]. In 1973, Forney published [5] a comprehensive survey of the Viterbi algorithm that, "reviewed more or less exhaustively all work inspired or related to the algorithm." In this review, Forney observes that for a finite-state discrete-time Markov process that can be modeled as a shift register process the trellis diagram normally associated with the Viterbi algorithm is identical, except for length, to the computational flow diagram of the fast Fourier transform (FFT).

At about the same time as [5], in quite a different body of literature, Stone [6] demonstrated that a concept known as a "perfect shuffle" serves as an important paradigm for connecting processors in parallel processing machines. In particular, he describes how many popular algorithms such as recursive doubling, sorting, polynomial multiplication, convolution, matrix transposition, and the FFT could be efficiently solved on parallel processing networks capable of perfect shuffle and exchange operations. (Conspicuously absent from consideration were shortest path problems.) Although Stone was one of the first to demonstrate that the concept of the perfect shuffle has a wide variety of application to various problems in computer science, the first descriptions of the perfect shuffle appear as early as 1776 in books [7] that describe methods for cheating at card games.[1]

Since the early 1970's, very little research [8] has been done to investigate or exploit the implications of these observations. The work described in [9] explicitly demonstrates that when the state sequence of a shift register process, whose state diagram is defined by a de Bruijn graph,[2] is to be estimated using the Viterbi algorithm it can be done simply on a parallel processor whose interconnection network is defined by a shuffle-exchange graph. (This will be the case for any rate-$1/n$ feedforward convolutional code.) This result is not unexpected when presented in the context of the independent work of Forney and Stone, but linking the Viterbi algorithm directly to the research results on shuffle-exchange graphs does not appear to have been described previously in the literature. The main result is that we can solve shortest path problems of interest in digital communications on a parallel processing network that has been primarily investigated in the discipline of computer science from the point of view of algorithmic structure and computational com-

Manuscript received April 24, 1987; revised September 3, 1987. This work was supported in part by the National Science Foundation under Grant DCI-84-21314-A1, by the Rockwell International Contract INT 6G3052, and by SDIO/IST, managed by the Army Research Office under Contract DAAL03-87-K-0033.

The authors are with Information Systems Laboratory, Department of Electrical Engineering, Stanford University, Stanford, CA 94305.

IEEE Log Number 8718850.

[1]The perfect shuffle is known as the Faro shuffle in magic circles.

[2]This Euler digraph is also referred to as a Good's diagram after the Teleprinter's problem (circa 1940).

plexity. For those interested in the implementation of Viterbi decoders, this appears to be a more powerful vantage point that gives one access to a much broader body of relevant literature.

As a consequence of this perspective, three primary benefits accrue. First, with the advent of VLSI, the question of how to best lay out the shuffle-exchange graph on a grid using as little area as possible has been extensively investigated; especially notable are the results of Thompson [10] and Leighton [11]. Hence, as first noted in [9], the upper and lower bounds on the layout area for the shuffle-exchange graph previously developed in the literature are of great interest in defining area bounds on a VLSI implementation of the Viterbi algorithm. Second, VLSI grid model based area–time complexity results for the FFT and sorting are of great interest despite the fact that these algorithms deal with finite length data sequences, whereas in the Viterbi algorithm, semi-infinite data sequences must be accommodated. The implications are that for various cost functions, we investigate performance tradeoffs of architectures from the point of view of pipeline period or clock period rather than latency. Third, we capitalize on the fact that the algorithm executed by a computational array is uniquely defined by both the processor interconnections and the arithmetic kernel of each processor. There is a pleasant harmony (and much insight to be gained) in the realization that, in certain instances, the Viterbi algorithm is related to so many other types of algorithms.

Orthogonal to this, we observe that there have been persistent efforts in the literature directed towards developing VLSI computational arrays for solving various shortest path problems having a dynamic programming solution [12]–[15]. This is noteworthy in that many channel and source decoding tasks in digital communications, including the coded modulation concepts of Ungerboeck [16], can be solved by a dynamic programming approach to finding the shortest path through a directed weighted graph using the algorithm developed by Viterbi. It is the dynamic programming character of the VA that provides the underlying basis for the algorithmically specialized processor networks presented in this paper. Hence, armed with these insights, it is natural to ask what types of concurrent VLSI architectures are appropriate for realizing Viterbi decoders and what are their relative performance tradeoffs?

B. Outline of the Paper

The Viterbi algorithm is computationally demanding not because its algorithm is complex in a conceptual sense. In fact, the essence of the algorithm is a relatively simple procedure of identical add, compare, select (ACS), and traceback operations. Rather, the computational burden arises because a relatively simple set of operations must be applied to a large number of basic "nodes" or "states" at each discrete-time step. Unfortunately, the number of states N is given by q^{ν} where q is the alphabet size and ν is the algorithm memory length. With the limitations of present fabrication technology there is great incentive to devise algorithms that assign more than one state per processor and/or constrain interprocessor communication such that the area necessary to wire the processors does not dominate the area required by the processors themselves. Locally connected processor arrays are of interest in this context since they satisfy this later constraint.

The most straightforward implementation of the Viterbi algorithm is a *completely sequential* one where every state is evaluated, in sequence, in a single arithmetic logic unit driven by a programmed control unit (i.e., microprocessor). A contemporary example of the sophistication possible in this approach is given in [17]. However, the sequential approach is a degenerate case of a concurrent realization of this algorithm; much better performance is possible, using a given technology, by exploiting the concurrency of the algorithm. The uniprocessor implementation suffers from being processor poor and from being I/O bound. The uniprocessor, though requiring only $O(1)$ area[3] (i.e., constant area), must coordinate $O(q^{\nu})$ "random" accesses to the processor's I/O memory and perform $O(q^{\nu+1})$ arithmetic operations each symbol interval T. Consequently, the hardware logic speed must be $\Omega(q^{\nu+1}/T)$ operations per symbol interval. Since the processor/memory ratio is so low (the design is almost all memory), the throughput of such a system is disappointing relative to other approaches even though this is the "smallest" area solution imaginable.

The other extreme is a *fully parallel* implementation of the Viterbi algorithm where one state is assigned per processor and the interprocessor connection network is a shuffle-exchange graph [9] or in some cases a cube-connected cycles graph. Though $O(q^{2\nu}/\nu^2)$ area is required in a planar embedding [9], the hardware logic speed is only $\Omega(1/T)$ operations per symbol interval. In the context of a VLSI realization, this type of fully parallel layout, though dominated by large interprocessor wire area, is the architectural organization with the greatest possible throughput for a given fabrication technology.

In contrast to these two extremes, this paper considers three alternative types of architectures. Each have multiple processing elements that cooperatively exploit the inherent parallelism in the Viterbi algorithm to varying degrees and yet each have VLSI layouts that require relatively small interprocessor wire area, as only near neighbor interprocessor wiring is allowed. The distinguishing feature of the designs is in how operands are scheduled and transferred between processors. The throughput and layout area requirements are intermediate to those of the completely sequential and fully parallel approaches thus giving the designer a broad spectrum of alternatives from which to choose.

The presentation of this paper is organized into six sec-

[3]A function $f(n)$ is $O(g(n))$ if there exist positive constants k and n_o such that $f(n) \leq kg(n)$ for $n > n_o$. A function $f(n)$ is $\Omega(g(n))$ if there exist positive constants k and n_o such that $f(n) \geq kg(n)$ for $n > n_o$. A function $f(n)$ is $\Theta(g(n))$ if there exist positive constants k and n_o such that $f(n) = kg(n)$ for $n > n_o$.

tions. Initially, we review previous implementations that utilize near neighbor interprocessor wiring. One section is then devoted to each of the three design strategies we propose. The design details of a pipelined cascade of processors are presented in Section III. Sections IV and V present the linear array and mesh architectures for the Viterbi algorithm, respectively. We summarize the throughput and layout area tradeoffs available to the designer in the discussion.

II. Previous Processor Array Implementations

Chang and Yao [18] have recently proposed an interesting method that applies systolic array techniques to the implementation of the Viterbi algorithm. The approach provides pipelining, parallelism, and simple adjacent neighbor interprocessor wiring that is suitable for VLSI implementation.

To illustrate the basic concept [18], consider a convolutional code in $GF(q)$ and total encoder memory length ν. The total number of states N is q^{ν}. Let P be a $1 \times N$ row vector whose ith element, denoted by P_i, represents the accumulated path metric to state i. Let the state transition matrix B be the $N \times N$ adjacency matrix whose ijth element, denoted by b_{ij}, represents the branch metric from state i to state j between two adjacent stages in the trellis diagram. Then the Viterbi algorithm can be formulated as repetitions of the following matrix-vector multiplication problem

$$P^{[k+1]} = P^{[k]} * B$$

where the superscript on P indicates the value of the discrete-time index. The operation $*$ is not ordinary multiplication in the conventional sense, but rather,

$$P_j^{[k+1]} = (P_0^{[k]} + b_{0j}) \otimes (P_1^{[k]} + b_{1j}) \otimes \cdots \otimes (P_{N-1}^{[k]} + b_{N-1,j})$$

where the $+$ operator denotes conventional addition and the $\otimes$ operator denotes the operation of taking maximum. In other words, the above expression means

$$P_j^{[k+1]} = \max_{0 \le i \le N-1} \{P_i^{[k]} + b_{ij}\}.$$

It is well known [19] that systolic arrays can be used efficiently for matrix-vector computations. By interpreting the algebraic kernel of the processing elements in the proper way the VA can be implemented on such systolic arrays where one processing element is provided for each state to be evaluated.

Note that due to the structure of the state transition diagram, there are only q nonzero entries in each row of the state transition matrix and hence only q/q^{ν} of the processors are doing meaningful work at any given time instant. This low efficiency in the utilization of the processor array is due to the sparseness of the state transition table. By combining several stages of the trellis diagram, the density of the state transition table can be increased and hence processor utilization can be improved [18]. In Section IV, we present alternatives to the systolic array concept presented above and in addition propose an entirely new strategy, based on the perfect shuffle paradigm, for performing the Viterbi algorithm on a linear array of processors.

One other approach for implementing the VA that has been reported in the literature is that of a unidirectional ring architecture [20] which contains N processing elements (PE), one for each state to be evaluated. In a unidirectional ring interconnection network, a PE can only receive operands from one adjacent neighbor and transmit operands to its opposite adjacent neighbor such that data flow in either a clockwise or counterclockwise fashion around the ring. Thus, if the routing time to an adjacent processor takes a single clock cycle, the worst case communication time around the ring is $N - 1$ clock cycles.

The ring architecture is appropriate in the sense that it allows one to perform all possible N^2 comparisons among the N-element finite state space. However, this organization suffers from inefficient processor utilization for much the same reason as the systolic linear array since only q/q^{ν} of the processors are doing meaningful work at any given time instant. The next section presents an alternate architecture called a pipelined cascade, that has a ring-like topology but requires only $\log_q N$ processing elements along with interprocessor pipelines and routing switches that improve processor utilization dramatically.

III. Cascade Layouts

In a manner similar to that used for bitonic sorting [21], [22], the Viterbi algorithm can be adapted to run on $\log_q N$ processors each associated with local memory of geometrically varying memory size. A representative implementation is illustrated in Fig. 1 for $q = 2$, $\nu = 3$ hence $N = 8$. If the feedback or recirculation path from processing element $PE_{\log_q N}$ to PE_1 were removed, Fig. 1 would be identical to the "cascade" design for the pipelined FFT computation [23]. The feature to note is that the topology is small, regular, and compact with minimal interprocessor wire area. The regular structure provides for extensibility such that the architecture can accommodate arbitrary problem size instances in a controlled way. The structure also allows one to partition the circuit into processor/FIFO pairs for incorporation onto separate chips (or printed circuit boards) as available technology dictates.

In the case of binary alphabets, each processor contains two sets of ACS circuits arranged pairwise in a butterfly configuration. Associated with each processor PE_j, $1 \le j \le \nu - 1$, are two auxiliary $2^{\nu-j}$ word FIFO queues and a programmable switch S_j, as illustrated in Fig. 1. Various minor rearrangements of memory and switches are possible with equivalent functionality. The total FIFO memory of the system is proportional to the number of states N. Each word in the FIFO is responsible for storing a state metric and an associated survivor sequence or pointer. The switches coordinate unidirectional (counterclockwise) information transfers between processors and

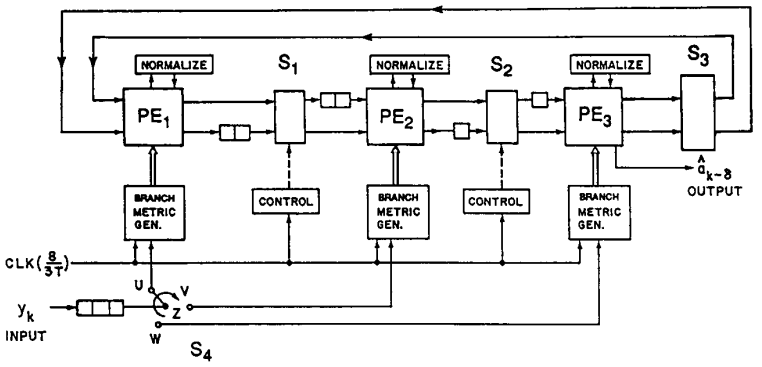

Fig. 1. Pipelined cascade layout. Architecture for binary alphabet and memory three. PE_1, PE_2, and PE_3 are ACS butterfly processors. Refer to Fig. 3 for detailed timing.

FIFO's. They have two modes of operation: mode one where operands are allowed to pass straight through (upper input to upper output, lower input to lower output) or mode two where operands are crisscrossed (upper input to lower output, lower input to upper output). Associated with PE_ν is the switch S_ν which is distinguished in function from every other S_j in that it consists of two components, a programmable switch and memory for the "staging" of switch outputs. Its function will be discussed shortly.

Each symbol interval, the circuit accepts a channel output symbol y_k as input into a dual ported memory or FIFO queue of ν words. Switch S_4, in Fig. 1, routes operands from this queue to the appropriate branch metric generator. Each branch metric generator serves a unique processing element and is responsible for providing a set of four-branch metrics each valid processing cycle (i.e., the clock cycle during which the ACS processor is active) of which there are $q^{\nu-1}$ in ν symbol intervals. If the first path metric generated in a group of $2^{\nu-1}$ processing cycles is subtracted from all others, path metrics can be conveniently normalized to control register overflow. Fixed delay maximum-likelihood estimates $\hat{a}_k$ of the transmitted data sequence a_k are available from the truncated survivor sequence of any state at each stage of the trellis. A convenient method to tap into these survivors is to extract or traceback the last ν items in the survivor list once every q^ν processing cycles which are available from processing element PE_ν. The implicit assumption, of course, is that the survivor sequence list is long enough to guarantee with high probability that the oldest items in the list have merged indicating that all states agree on a common ancestry. An output queue of length ν allows the system to present one output estimate each symbol interval T.

A processing element is responsible for processing states associated with one stage of the trellis. Fig. 2 illustrates the three-stage, eight-state trellis diagram associated with Fig. 1. For the purposes of explaining the operation of the architecture, assume that all states associated with stage k of the trellis are evaluated in PE_1. PE_1 first evaluates in sequence path metrics associated

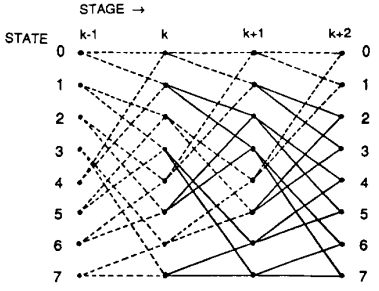

Fig. 2. Three-stage trellis diagram associated with Fig. 1.

with states 0 and 1, then states 2 and 3, then 4 and 5, and finally, states 6 and 7. When PE_1 completes the evaluation of path metrics associated with states 4 and 5 in stage k, PE_2 can proceed to evaluate path metrics associated with states 0 and 1 in stage $k + 1$. Likewise, PE_3 can proceed with calculating the path metrics associated with states 0 and 1 in stage $k + 2$ only after states 4 and 5 in stage $k + 1$ have been evaluated. The data dependency for this last case is highlighted in Fig. 2 by the dashed branches in the trellis. The outputs of PE_3 are staged in switch S_3 and recirculated to PE_1 which then proceeds to evaluate the next stage of the trellis. Pipelining is possible in this recirculation network because newly generated state information produced by processor PE_j can be passed to the immediate neighbor $PE_{j(\mathrm{mod}\,\nu)+1}$ so that states associated with the next stage of the trellis can be processed even before all states associated with PE_j have been evaluated.

A detailed illustration of the flow of data through the pipeline of Fig. 1 is illustrated in Fig. 3. Each instance of time is associated with a channel output symbol. The system clock, operating at q^ν/ν times the symbol rate, controls the input queue which in our example, for instance, takes the channel output symbol(s) associated with time instance "1" and delivers it on clock tick number "3" to the branch metric generator associated with PE_2. The control algorithms for each switch that define the mode of the switch are a function of the current clock tick.

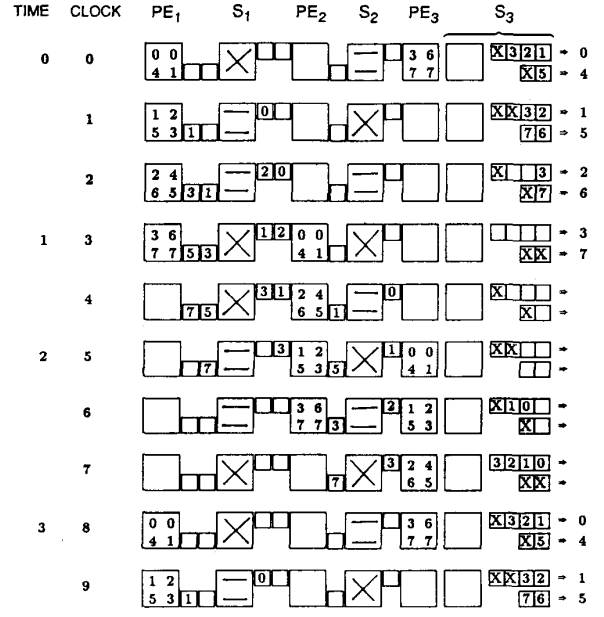

Fig. 3. Schedule for the movement of state information as a function of the system clock for the architecture illustrated in Fig. 1.

The mode of each switch as a function of the clock is illustrated diagrammatically with a "straight-through" or "crisscross" icon. Switch S_3 is unique, however, in that for every clock tick, the contents of each of the two shift registers, which comprise the memory internal to the switch, are shifted one unit to the right after which the two outputs of PE_3 are written in parallel into two register locations as predetermined by the switch algorithm. (As illustrated in Fig. 3, an "X" in S_3 signifies a "DON'T CARE" state.) A final remark on Fig. 3 is that one clock tick was allotted for the ACS operation of a PE. If the computation time of a PE is greater than this it does not disturb the structure other than inserting extra delays within the system.

Pipelining allows ν symbols to be processed each q^ν processing cycles. Consequently, hardware logic speed must be $\Omega(q^\nu/\nu T)$, a respectable linear speedup improvement over the one processor implementation. Even with complete pipelining each processor PE_j is idle for one-half of the processing cycles. Given additional memory and control, the processing elements could be used to decode two data streams simultaneously. Although the layout has been directed toward realizations for binary alphabets, versions of this processor for arbitrary alphabet sizes should be apparent [23].

The cascade, like the uniprocessor, is mainly dominated by storage. The difference being, of course, that in the cascade design the number of processors grows linearly with the algorithm memory length while the number of memory elements grows exponentially with algorithm memory length. As an alternative, the next section presents a layout strategy that contains as many ACS processors as storage elements, to within a constant factor.

IV. Linearly Connected Layouts

In this section, we describe four approaches to implementing the Viterbi algorithm on a linear array of processors. The first two methods are based on the fold-over scheme [24] borrowed from techniques in computational geometry. One other is adapted from the parallel odd-even transposition sorting algorithm [21]. The first three approaches are especially appropriate when the trellis diagram is complete. The last method presented in this section is unique in that it relies on the perfect shuffle paradigm [6] and does not rely on state transition graphs that are complete symmetric digraphs.

As referred to throughout this section, a linearly connected network of processors appropriate for implementing the Viterbi algorithm is illustrated in Fig. 4. It consists of three rows of processing cells where each element of a row is fitted with word-parallel interconnections to its near neighbor. Each row of this architecture is homogeneous in function. This allows us to define a column of three processing cells as a standard building block. Linearly connecting N of these building blocks together allows the assignment of one state to each processor.

The backbone of this architecture is formed by the middle or center row of processing cells, while the top and bottom rows can be viewed as support hardware. The top row takes a clock signal as input and produces as output a set of sequence control signals, one for each processor in the middle row. The bottom row accepts the input signal y_k and generates appropriate branch metrics for each processing element in the middle row. In some implementations, this row may be comprised exclusively of ROM hardware. Each processing cell in the middle row contains routing logic circuitry, a path metric register, survivor sequence register, and add-compare-select circuitry. Each processor is capable of routing the contents of its registers to a neighboring processor in an operation known as a unit-distance route. Operands that must be moved beyond a near neighbor require multiple unit distance routing steps. The unit distance routing operations are performed by the routing logic circuitry that exchanges path metrics and survivor sequences[4] with a near neighbor as dictated by the sequence control signals generated by the top row of processors.

Branch metrics supplied by the bottom row are used to update the path metric registers each symbol interval. As output, the middle row provides fixed delay estimates of the transmitted data sequence. These can be extracted from the truncated survivor sequence resident in the rightmost processing element.

Each of the four different methods proposed here rely on this same basic structure. However, in order to uniquely distinguish the details of their operation, the sequence of events during one symbol interval in the center row of processing cells is illustrated in the subsections that follow.

[4]or alternatively, pointers in a traceback approach.

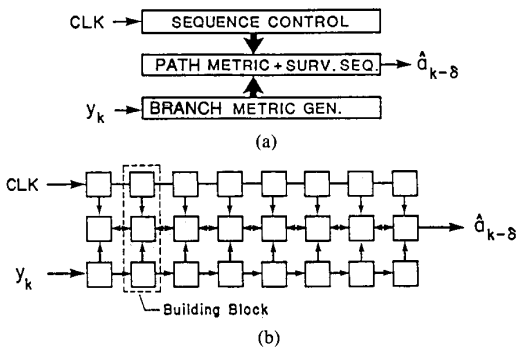

Fig. 4. Linearly connected processor layout. (a) Functional block diagram. (b) Linear array of regular building blocks.

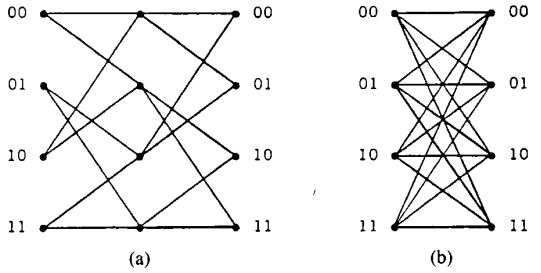

Fig. 5. The strongly connected trellis diagram. (a) Two-stage trellis diagram. (b) Strongly connected trellis equivalent.

A. Systolic Approaches

A complete trellis diagram (or state diagram) is one where every state has a unique state transition to all other states. A state diagram with this property can be exploited to gain full processor utilization every clock cycle in a linear systolic array implementation of the Viterbi algorithm. In considering a systolic, matrix-vector formulation of the VA involving a complete trellis, an analogy with systolic implementations of the DFT [19, p. 290] can be exploited. However, the architectures described in this section rely on the observation that the routing of path metrics and survivor sequences in a complete trellis diagram is analogous to that required in solving the pairwise examination problem [25]. That is, given a set of N elements labeled $(0, 1, 2, \cdots, N-1)$, examine the pairs $(0, 1), (0, 2), \cdots, (0, N-1), (1, 2), (1, 3), \cdots, (1, N-1), (2, 3), \cdots, (N-2, N-1)$.

As defined in Section II, if we let P be a $1 \times N$ row vector whose ith element denoted by P_i represents the accumulated path metric to state i then by performing the pairwise examination problem on the vector $(P_0, P_1, P_2, \cdots, P_{N-2}, P_{N-1})$ and the vector $(P_{N-1}, P_{N-2}, P_{N-3}, \cdots, P_1, P_0)$ simultaneously, we are in effect mimicking the data routing operations required in a single stage of a complete trellis diagram. Two queries remain to be answered. What type of transformation on the state diagram allows us to generate (if it exists) the complete equivalent? How should the transfer of operands be scheduled on a linear systolic array in order to perform the data routing operations indicated by a complete trellis diagram?

Let us respond to the first query. Chang and Yao [18] in their work demonstrate that for simple shift register sequences where the state diagram is a de Bruijn graph, the original trellis diagram can always be transformed to be complete by considering no more than $\log_q N$ stages of the trellis. Fig. 5 illustrates the concept for a simple four-state, two-stage trellis diagram. In the two-stage trellis diagram, if each two-branch metric pair is combined to be a single "composite" branch metric, the trellis diagram will become complete. The transformation required can be developed very naturally from a consideration of the state diagram rather than the trellis diagram. If state diagram is strongly connected then the transitive closure of the state diagram is a complete digraph.

Given that the strongly connected trellis exists, several of the "examine all pairs" systolic algorithms can be adapted to perform the VA on a linear array. For example, consider the simple fold-over scheme presented in Fig. 6. Our intent is to illustrate the movement of state metrics and survivor sequences during one symbol interval. Each processing element illustrated in the first row of Fig. 6 contains two integers in the range 0–$N-1$ inclusive. An integer, let us call it j, indicates the presence of the path metric and survivor sequence associated with state P_j. Hence, at the beginning of the symbol interval, each processor has two copies (denoted upper and lower) of a particular path metric and survivor sequence assigned to it. The basic idea behind the fold-over operation is to regard the upper elements in the systolic array as a strip of paper. The idea is to pick up the strip at the left and fold it over, pulling the leftmost element over from left to right. In a hardware implementation, this operation is effectively accomplished with local near neighbor data tranfers. At the same time, the strip at the right is picked up and the lower elements are folded under pulling the rightmost element from right to left. If each near neighbor data transfer takes one clock cycle the total number of clock cycles required is $2N-1$, which defines the minimum symbol interval.

Selected processing elements receive branch metrics at the end of each clock cycle. Branch metrics are provided to all processing elements that have had new data values folded into it. Initially, the branch metric associated with the state transition $b_{0,0}$ is supplied to the first element of the array. At the end of step 1, the branch metrics associated with the state transition $b_{0,1}$ is provided to the second cell and $b_{N-1,N-2}$ is provided to the second last cell. At the end of step 2, the branch metrics associated with the state transition $b_{1,1}$ and $b_{0,2}$ are supplied to the second and third cell, respectively, while $b_{N-1,N-3}$ is supplied to the third last cell and so on. In general, path metrics are supplied in a manner that corresponds to partitioning the state transition matrix B into a lower and upper triangular matrix. The branch metrics associated with the upper triangular portion of B are used by the upper fold-over scheme. The branch metrics associated with the lower triangular portion of B are used by the lower fold-

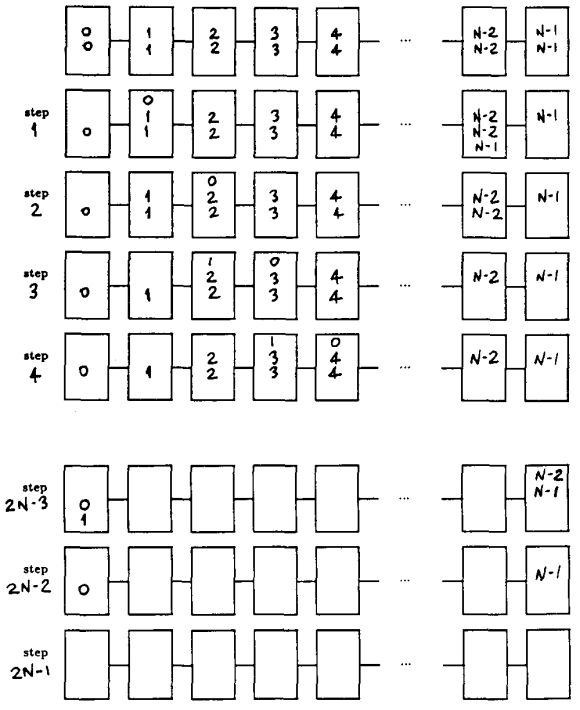

Fig. 6. The fold-over operation for routing state metrics.

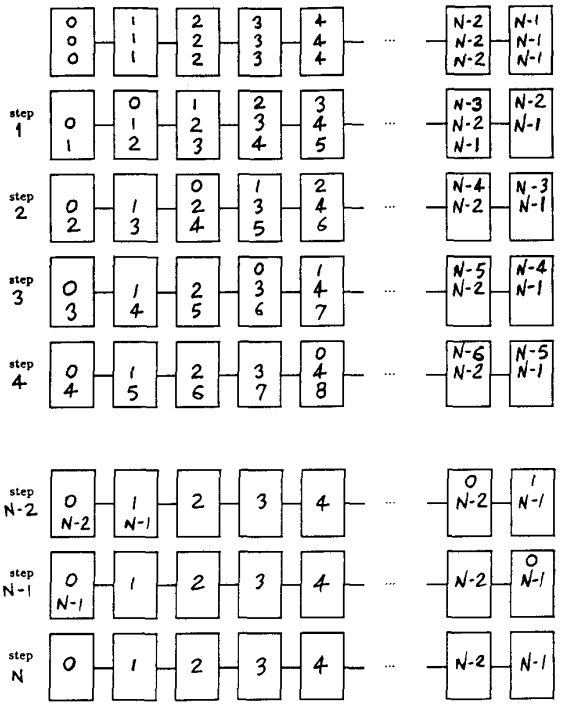

Fig. 7. An improved fold-over scheme.

over scheme. As a branch metric is supplied to a processing element, it is added to the newly arrived path metric and compared to and possibly substituted to reflect the current maximum path metric resident in that processing cell.

The number of clock cycles required for the fold-over operation can be reduced by making an additional local copy of the state information in each PE. The middle element i stays in cell i during the whole operation while the upper elements move to the left and the lower elements move to the right. As new path metrics are transferred into a processing element the corresponding branch metrics are provided by the support hardware. Fig. 7 indicates the movement of operands for each clock tick. The total number of clock cycles required is N, thus defining the minimum symbol interval.

To conclude this section, we will show a systolic Viterbi algorithm for a strongly connected trellis diagram can be derived from the odd–even transposition sort [21]. It should be noted that not all of the parallel sorting algorithms examine every pair of data in the worst case; however, this approach does, which is a requirement for our application. For simplicity, Fig. 8 illustrates the case for N even. At odd numbered time steps path metrics and survivor sequences in cells 0, 2, $\cdots$, $N-2$ are swapped with path metrics and survivor sequences in cells, 1, 3, $\cdots$, $N-1$, respectively. At even numbered time steps, data in cells 1, 3, $\cdots$, $N-3$ are swapped with path metrics in cells 2, 4, $\cdots$, $N-2$, respectively. As in the previous design, when new path metrics are transferred into a processing element, the corresponding branch metrics are provided by the support hardware. Only N clock cycles are required for the routing of operands just as in the case for the improved fold-over scheme.

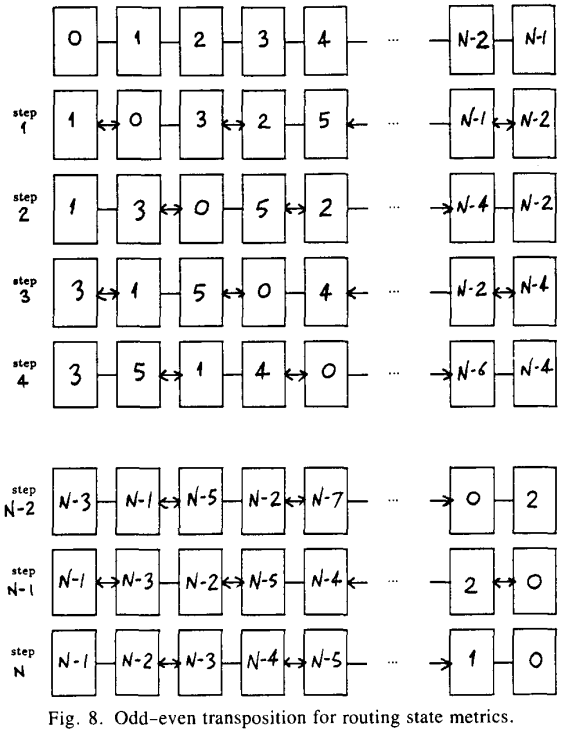

Fig. 8. Odd–even transposition for routing state metrics.

B. The Perfect Shuffle Approach

This section is concerned with the special case of trellis diagrams that can be modeled as being generated by a simple shift-register process. The state diagram, for this case, is a de Bruijn graph, that has an equivalent representation [9] in the form of a shuffle-exchange graph. By exploiting this observation, a linear array can be used to implement the Viterbi algorithm in a very simple way.

To exploit a perfect shuffle approach, the partitioning of the hardware that defines a processing element is defined in a slightly different manner than the previous section. The array consists of connecting $q^{\nu+1}$ processing cells together as illustrated in Fig. 9(b). Each processing cell in the middle row contains routing logic circuitry, a path metric register, survivor sequence register, and one adder. Odd–even processor pairs share a single compare and select hardware element. The routing logic circuitry exchanges path metrics and survivor sequences with a near neighbor as dictated by the sequence of control signals generated by the top row of processors.

The sequence of events during one symbol interval in the center row of processing cells is illustrated in Fig. 9(b). The initial configuration of the array consists of N sets of q identical path metrics (and survivor sequences). In a series of near neighbor transpositions, path metrics are moved (in only $N-1$ clock ticks) to appropriate positions in the one-dimensional array in anticipation of the branch metrics b_{ij} generated by y_k. The specific b_{ij} required by a processing element in Fig. 9(b) is defined by the branch metric labels in Fig. 9(a). This sequence of steps can be viewed as unpacking, without altering their original relative ordering, the items initially in the left half into the even positions in the array, and those in the right half into the odd positions of the array. In fact, the data are rearranged or permuted in a manner identical to that defined by the perfect shuffle operation, under the constraint of near neighbor transpositions only. When illustrated as a function of discrete time, the triangular structure of the series of transpositions is characteristic of the control algorithm required for any size problem instance in $GF(2)$.

After the requisite transpositions and after the branch metrics have been added to the path metrics, the bottom of the transposition "triangle" consists of having each even numbered processor compare its path metric to that of its odd numbered adjacent neighbor. The smallest path metric of this pair is chosen, normalized to prevent overflow and then duplicated in its odd–even processor pair. At the same time the survivor sequence registers are updated and an estimate of a transmitted symbol in the past history is ejected from the last processing cell in the array. The events described in the last few paragraphs are then repeated during the next symbol interval.

A processor as defined here, is allocated for each branch in the trellis diagram, hence, always guaranteeing complete utilization of the processor array during the add, compare, and select operations. Presumably, transpositions can proceed quicker than arithmetic operations.

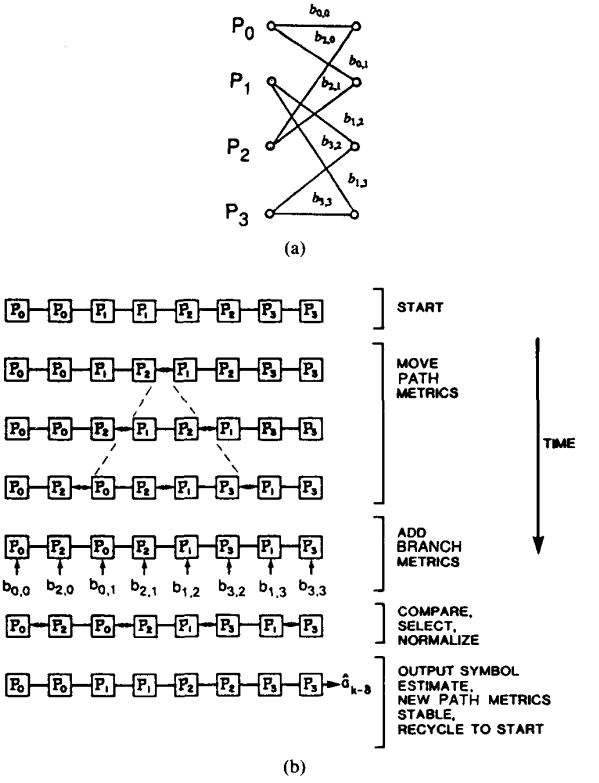

Fig. 9. Linearly connected processor layout of the VA. (a) Trellis diagram with branch metrics labeled. (b) A schematic presentation of the sequence of events in a linearly connected processor layout for the trellis in (a); ↔ denotes transposition.

Overflow control is achieved by selecting one path metric and subtracting it from all others. For an efficient implementation, one of the processors in the center of the array should distribute its path metric into a special register during the transportation operations. At the bottom of the transposition "triangle," each processor will have a copy of this one path metric to be subtracted from all the newly generated path metrics.

The 1-*D* structure of the array allows for the development of strategies [26] to circumvent the effects of low yield (faulty processors) in a wafer-scale implementation. Although this architecture has an unpleasant aspect ratio for algorithm memory lengths of interest, say $\nu > 4$, Leiserson [27, p. 94] can be used to establish the comforting fact that a topologically equivalent layout can be enclosed in a square whose area is, at most, three times the area of the original rectangular layout. One final comment with regard to implementation. Note that the sequence of transposition steps is symmetrical about an axis that cuts the array into two equal halves. Folding this array about this axis of symmetry provides the insight on how one would collapse the design [28] onto half as many processors with a sequence of transpositions that sweep across the array from one end to the other.

Since operands must be transported across the N-element array, hardware logic speed must be $\Omega(q^{\nu}/T)$ in this algorithmically specialized VLSI network. The sim-

pler control algorithm this structure enjoys is paid for by the increased processor area and reduced throughput relative to the cascade design. The performance of this computational array can be improved by utilizing an array with more interprocessor wiring, a two-dimensional array, as demonstrated in the next section.

V. Mesh Layouts

In this section, we demonstrate that the Viterbi algorithm can be implemented on a compact mesh-type recirculation network. The two-dimensional mesh layout has a higher throughput than the linearly connected layouts studied in the previous section because operands do not always have to migrate as far through the network of processors given that most of the processing elements in the array are connected to more than two neighbors. However, the connectivity is inferior and, hence, routing time charges are greater than the large wire area layouts such as the shuffle-exchange layout in that operands in a mesh must sometimes circulate beyond immediate cell neighbors before they reach the correct cell.

The rectangular mesh interconnection pattern consists of $N = q^{\nu}$ identical processors (one for each state), arranged in a two-dimensional array of size $q^{\lfloor \nu/2 \rfloor}$ by $q^{\lceil \nu/2 \rceil}$. Each processor is connected to adjacent neighbors, as shown in Fig. 10(a). Processors at the perimeter have two or three rather than four neighbors; there are no end-around connections. The feature to note is that this structure is a small and compact design that requires essentially little interprocessor wire area.

Each cell is a message driven processing element with the ability to generate forward and receive messages. Each element in this array contains an add–compare–select circuit, a path metric generator (or table lookup), transceiver and multiplexers, and a state machine that serves as a control processor. Two bidirectional communication paths would be provided to each neighbor for path survivors and for state metrics. Each processor is given a numeric label in the range 0 to $N - 1$, in one to one correspondence with each of the possible path metrics P_i. Though path metrics roam throughout the array as determined by the routing algorithm, the processor labels are fixed and correspond to the "home" location of the corresponding path metric (i.e., are meant to indicate the location of the corresponding path metric at $t = 0$). The order in which the processors are labeled or indexed determines the routing algorithm used to move data between processors. The objective is to use index schemes which minimize the time spent in routing. A row-major index scheme, as illustrated in Fig. 10(a), yields a simple routing algorithm for each cell. The routing algorithm is ultimately determined by the trellis diagram which has been rearranged in Fig. 10(b) in a form analogous to that required for a bitonic sort on a mesh. During a route operation, all data move in the same direction; that is, up, down, left, or right. The movement of operands between processors takes the form of either row merging or column merging operations at each clock tick.

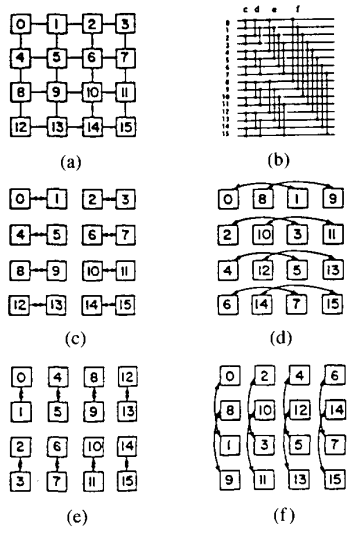

Fig. 10. The VA in a 4 × 4 square mesh. (a) The 4 × 4 mesh of processors labeled in row major order. (b) The rearranged trellis diagram. (c)–(f). Routing steps and migration of state metrics of a 16-state binary VA receiver implemented on a 4 × 4 square mesh. The numerals correspond to path metric labels. The execution sequence is: (c) (d) (e) (f) (c) (d) (e) (f) (c) (d) · · · .

To illustrate the basic concept, consider for binary alphabets, a 16-state Viterbi decoder implemented on a 4 by 4 square mesh. The routing steps and the corresponding migration of state metrics is illustrated in Fig. 10(c)–(f). One stage of an N-state trellis (the trellis diagram for one symbol interval) is implemented by means of unit distance routing steps. The sequence of routing steps, defined by the discrete time index k, repeats every ν symbol intervals since the state metrics are back "home," in their original starting location. Note that some routing steps require multiple unit distance routes, to move information from where it was produced to where it is needed next. The throughput of the VA, executed on a square mesh, is limited by the worst case number of unit distance routes required in a symbol interval and the delay of the ACS processing cells. It can take as much as $O(q^{\lceil \nu/2 \rceil})$ time to rearrange the data among the processors in preparation for the next ACS step. Fortunately, only a few of the ACS + route operations take this amount of time. The average time for a ACS + route for 2^{ν} states is only $O(2^{\lceil \nu/2 \rceil}/\nu)$. Thus, the required logic speed (operations/s) of the structure must be $\Omega(2^{\lceil \nu/2 \rceil}/\nu)$.

Three implementation details of the mesh implementation are particularly important. First, in order to spread the routing step time penalty equally among each symbol interval, a FIFO queue of depth ν should be used on the detector input data stream. The output of the FIFO queue is globally broadcast to all processing cells. This is the only global wiring required in the design (besides power and timing signals). Second, the detector output is available at each symbol interval from the truncated survivor sequence of any processing cell. Third, overflow control of the state metric registers is not straightforward, if we

are determined to use only the local near neighbor communication paths. One possible strategy involves normalizing state metrics once every $\log N$ processing steps (i.e., once every ν steps into the trellis). This provides for the opportunity of selecting one state metric, say state zero, and separating this value from the main computational data path. During the first $\lceil \nu/2 \rceil$ processing cycles, this value can be broadcast to all processing cells, in the same row, using only nearest neighbor broadcasts (on a dedicated connection). At this point in time, each column has at least one processing cell with this value stored in a register. During the next $\lfloor \nu/2 \rfloor$ processing cycles, this value can be broadcast to all processing cells in each column. Now, all state metrics can be normalized using this value. This normalization routine is then repeated in the next ν processing cycles. The penalty paid in this type of scheme is that state metric registers would have to be several bits wider than if state metrics were normalized each processing cycle.

We conclude this section by pointing out that routing algorithms exist where the number of processors (states) is not necessarily a perfect square and where the alphabet size q is not necessarily binary.

VI. Discussion and Concluding Remarks

In this paper, we have presented three architectures; namely, the cascade, the linear array, and the orthogonal mesh, for implementing a dynamic programming algorithm called the Viterbi algorithm. Our objective was to enumerate locally connected processor arrays that exploit the parallelism of the Viterbi algorithm to varying degrees. A summary of the performance of these architectures relative to those already identified as being appropriate for a concurrent implementation of the Viterbi algorithm is presented in Table I. These results are graphically illustrated in Fig. 11. When ordered in terms of increasing interprocessor wire area, an inverse relation with throughput is evident. The conclusion to be drawn is that, for a given fabrication technology, architectures capable of supporting high data rates necessarily require more VLSI implementation area.

One useful figure of merit is the product (*Area*) * (*Period*) denoted by *AP* where the symbol period *P* is defined as the average number of unit times between the appearance of the first input bit of a symbol to the Viterbi decoder and the appearance of the first input bit of the next symbol when operated at the highest possible symbol rate. Illustrated in Fig. 11 are contours of equal *AP* product where *P* is noted to be proportional to the logic speed. The energy consumption during each symbol interval (power) is defined by the *AP* measure of complexity [10] using the VLSI grid model. This measure can also be interpreted as the reciprocal of throughput per unit area.[5] A

[5]The United States Military development program for very high speed integrated circuits (VHSIC) uses a processing throughput per unit area (TP) figure of merit defined by: TP = (gate density) * (clock rate) = *gate*-Hz/mm^2. Contemporary microprocessors, such as the Intel 80386, achieve a TP of approximately 10^{10} *gate*-Hz/mm^2.

TABLE I
Performance Summary

Layout Type	*Logic Speed for Symbol Interval T*	*Layout Area*
Uniprocessor	$\Omega\left(\frac{q^{\nu+1}}{T}\right)$	$o(1)$
Cascade	$\Omega\left(\frac{q^{\nu}}{\nu T}\right)$	$o(\nu)$
Ring	$\Omega\left(\frac{q^{\nu}}{T}\right)$	$o(q^{\nu})$
1-D Mesh (Systolic)	$\Omega\left(\frac{q^{\nu}}{T}\right)$	$o(q^{\nu})$
1-D Mesh (Shuffle)	$\Omega\left(\frac{q^{\nu}}{T}\right)$	$o(q^{\nu})$
2-D Mesh	$\Omega\left(\frac{q^{\frac{\nu}{2}}}{T}\right)$	$o(q^{\nu})$
Shuffle-Exchange	$\Omega\left(\frac{1}{T}\right)$	$o\left(\frac{q^{2\nu}}{\nu^2}\right)$

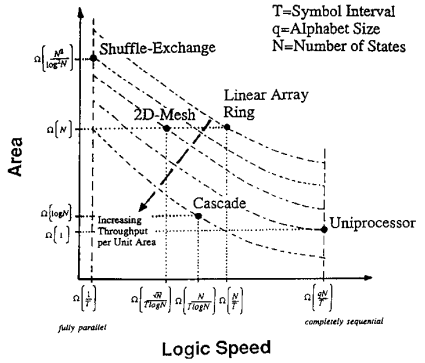

Fig. 11. Area-time performance summary. This figure is not drawn to scale. Only relative relationships are illustrated.

completely pipelined circuit optimal with respect to this criterion can be claimed to make best use of this area. Of those considered, the best design, with respect to this criterion, is seen to be the cascade design.

In this paper, we have assumed that the complexity of the branch extension computations to be low relative to the ACS operation. The reverse may be true for many trellises where the use of ROM's is not feasible. Since we have only dealt with orders of complexity, detailed area–speed comparisons need to be performed before the relative merit of some of the alternatives is completely clear in certain circumstances.

The most straightforward implementation of the Viterbi algorithm is a completely sequential realization. The other extreme is a fully parallel implementation which completely exploits the parallelism of the algorithm to achieve the greatest possible throughput. In contrast to these two extremes, this paper considered three alternative types of architectures. As a guide, we have exploited results from

various types of algorithms the most notable being Fourier transform and sorting algorithms. Each of the designs has multiple processing elements that cooperatively exploit the inherent parallelism in the Viterbi algorithm to varying degrees and yet each has VLSI layouts that require relatively small interprocessor wire area, as only near neighbor interprocessor wiring is allowed. The distinguishing feature of the designs is in how operands are scheduled and transferred between processors. The throughput and layout area requirements are intermediate to those of the completely sequential and fully parallel approaches thus giving the designer a broad spectrum of alternatives from which to choose.

Acknowledgment

Several comments by the reviewers that were helpful in improving the clarity of this paper are gratefully acknowledged.

References

[1] C. L. Seitz, "Concurrent VLSI architectures," *IEEE Trans. Comput.*, vol. C-33, pp. 1247–1265, Dec. 1984.
[2] M. Pollack and W. Wiebenson, "Solutions to the shortest-route problem—A review," *Oper. Res.*, no. 8, pp. 224–230, 1960.
[3] J. K. Omura, "On the Viterbi decoding algorithm," *IEEE Trans. Inform. Theory*, vol. IT-15, pp. 177–179, Jan. 1969.
[4] A. J. Viterbi and J. K. Omura, *Principles of Digital Communication and Coding.* New York: McGraw-Hill, 1979.
[5] G. D. Forney, "The Viterbi algorithm," *Proc. IEEE*, vol. 61, pp. 268–279, Mar. 1973.
[6] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, pp. 153–161, Feb. 1971.
[7] P. Diaconis, R. L. Graham, and W. M. Kantor, *The Mathematics of Perfect Shuffles, Advances in Applied Mathematics, Vol. 4.* New York: Academic, 1983, pp. 175–196.
[8] C. M. Rader, "Memory management in a Viterbi decoder," *IEEE Trans. Commun.*, vol. COM-29, pp. 1399–1401, Sept. 1981.
[9] P. G. Gulak and E. Shwedyk, "VLSI structures for Viterbi receivers: Part I—General theory and applications," *IEEE J. Select. Areas Commun.*, vol. SAC-4, pp. 142–154, Jan. 1986.
[10] C. D. Thompson, "A complexity theory for VLSI," Ph.D. dissertation, Dept. Comput. Sci., Carnegie Mellon Univ., 1980.
[11] F. T. Leighton, *Complexity Issues in VLSI: Optimal Layouts for the Shuffle-Exchange Graph and Other Networks.* Cambridge, MA: MIT Press, 1983, 139 pages.
[12] K. O. Brown, "Dynamic programming in computer science," Dept. of Comput. Science, Carnegie-Mellon Univ., Pittsburg, PA, Rep. CMU-CS-79-106, Feb. 1979.
[13] N. Weste, D. J. Burr, and B. D. Ackland, "Dynamic time warp pattern matching using an integrated multiprocessor array," *IEEE Trans. Comput.*, vol. C-32, pp. 731–744, Aug. 1983.
[14] K. H. Chu and K. S. Fu, "VLSI architectures for high speed recognition of context-free languages," in *Proc. 9th Annu. Symp. Comput. Architec.*, Apr. 1982, pp. 43–49.
[15] L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI implementation of combinatorial algorithms," in *Proc. Caltech. Conf. Very Large Scale Integration*, Jan. 1979, pp. 509–525.
[16] G. Ungerboeck, "Trellis-coded modulation with redundant signal sets, Part I: Introduction, Part II: State of the art," *IEEE Communications*, vol. 25, pp. 5–21, Feb. 1987.
[17] S. C. Glinski, T. M. Lalumia, D. R. Cassiday, T. Koh, C. M. Gerveshi, G. A. Wilson, and J. Kumar, "A processor for graph search algorithms," in *Proc. ISSCC 87*, New York, Feb. 1987, pp. 162–163.
[18] C. Y. Chang and K. Yao, "Viterbi decoding by systolic array," in *Proc. Twenty-Third Annu. Allerton Conf. Commun., Cont. Comput.*, Allerton House, Monticello, IL, Oct. 2–4, 1985, pp. 430–439.
[19] C. Mead and L. Conway, *Introduction to VLSI Systems.* Menlo Park, CA: Addison-Wesley, 1980, p. 275.
[20] W. Bliss, J. Girard, J. Avery, M. Lightner, and L. Scharf, "A modular architecture for dynamic programming and maximum likelihood sequence estimation," in *Proc. ICASSP '86*, Tokyo, Japan, 1986, pp. 357–360.
[21] S. G. Akl, *Parallel Sorting Algorithms.* New York: Academic, 1985.
[22] C. D. Thompson, "The VLSI complexity of sorting," *IEEE Trans. Comput.*, vol. C-32, pp. 1171–1184, Dec. 1983.
[23] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1975, p. 606.
[24] B. Chazelle, "Computational geometry on a systolic chip," *IEEE Trans. Comput.*, vol. C-33, pp. 774–785, Sept. 1984.
[25] Z.-C. Shih, G.-H. Chen, and R. C. T. Lee, "Systolic algorithms to examine all pairs of elements," *Commun. ACM*, vol. 30, no. 2, pp. 161–167, Feb. 1987.
[26] T. Leighton and C. E. Leiserson, "Wafer-scale integration of systolic arrays," *IEEE Trans. Comput.*, vol. C-34, pp. 448–461, May 1985.
[27] C. E. Leiserson, *Area Efficient VLSI Computation, ACM-MIT Press Doctoral Dissertation Award Series.* Cambridge, MA: MIT Press, 1983, 136 pages.
[28] J. P. Fishburn and R. A. Finkel, "Quotient networks," *IEEE Trans. Comput.*, vol. C-31, pp. 288–295, Apr. 1982.

P. Glenn Gulak (S'82–M'83) received the B.A.Sc. degree in electrical engineering (summa cum laude) from the University of Windsor, Windsor, Ont., Canada, in 1978. While on a NSERC postgraduate scholarship, he received the M.Sc. and Ph.D. degrees in electrical engineering from the University of Manitoba, in 1980 and 1984, respectively.

During 1979–1980 he worked for the Burroughs Corporation on signal processing for digital magnetic recording channels. From February 1985 to January 1986, he worked in the Computer Systems Laboratory, Stanford University, Stanford, CA, on the MIPS-X project (a high performance VLSI RISC microprocessor). Since February 1986 he has been with the Information Systems Laboratory where he is currently a Research Associate. His research interests are in digital communications, signal processing algorithms, computer architecture, and VLSI.

Thomas Kailath (S'57–M'62–F'70) was born in Poona, India, on June 7, 1935. He received the B.E. degree from the University of Poona in 1956 and the S.M. and Sc.D. degrees from The Massachusetts Institute of Technology, Cambridge, MA, in 1959 and 1961, respectively.

During 1961–1962 he worked at the Jet Propulsion Laboratories, Pasadena, CA, where he also taught part-time at the California Institute of Technology. He has held shorter term appointments at several institutions around the world, including a Ford Fellowship in 1963, University of California at Berkeley, a Guggenheim Fellowship in 1970, Indian Institute of Science, a Churchill Fellowship in 1977, Statistical Laboratory, Cambridge University, England, and a Michael Fellowship in 1984 with the Department of Theoretical Mathematics, Weizmann Institute, Israel. His research and teaching have been in statistical communications, control, information theory, linear systems, and signal processing.

Dr. Kailath, from 1971 to 1978, has been a member of the Board of Governors of the IEEE Professional Group on Information Theory and the IEEE Control Systems Society. During 1975 he served as President of the Information Theory Group. He has received outstanding paper prizes from the IEEE Information Theory Group (1966), the IEEE Acoustics, Speech, and Signal Processing Society (1983), and the 1986 Education Award of The American Automatic Control Council. He is on the editorial board of several engineering and mathematics journals. He is the author of *Linear Systems* (Englewood Cliffs, NJ: Prentice-Hall, 1980), *Lectures on Wiener and Kalman Filtering* (New York, NY: Springer-Verlag, 1981), as well as Editor of *Modern Signal Processing* (New York, NY: Hemisphere-Springer-Verlag, 1985) and coeditor of *VLSI and Modern Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall, 1985). He is also a member of the National Academy of Engineering, a Life Fellow of Churchill College, Cambridge, England, and a Fellow of the Institute of Mathematical Statistics.