Parallel Architecture for a Pel-Recursive Motion Estimation Algorithm

Emmanuel D. Frimout, Student Member, IEEE, Johannes N. Driessen, and
Ed F. Deprettere, Senior Member, IEEE

Abstract—The paper presents a parallel architecture for a pel-recursive motion estimation algorithm. It is a linear array of processors, each consisting of an initialization part, a data-routing part, and an updating part. The initializing part performs a prediction of the motion vector. The routing part along which previous-frame data are routed from processors to processors that request such data. The router can be data driven or clocked. The latter approach will be presented in more detail. The updating part calculates an update to the predicted motion vector. The architecture we propose is derived in a systematic way and is parameterized with respect to certain window sizes. It is thus completely different from the few existing pel-recursive motion estimation architectures.

I. INTRODUCTION

In an imaging system, motion in a 3-D scene manifests itself as projected motion in the image plane. This projected motion can be represented by a 2-D time-varying vector field—the motion field. The estimation of this motion field is important in two areas: motion-compensated image sequence processing [1] and dynamic scene analysis [2]. Examples of the former are coding, interpolation, and noise reduction. Examples of the latter include robot vision, traffic monitoring, and medical and biological applications. In most of these applications, real-time performance is required. In order to meet this real-time constraint, architectures for VLSI implementation of the motion estimation algorithm have to be developed.

When we consider a digital imaging system, the 2-D time-varying motion field is represented in three dimensions, such that a vector \( d(x, t) \) is attributed to each location \( (x, t) \) with spatial variable \( x = (x, y)^T \) and the sequence variable \( t \). The superscript \( T \) denotes transposition. In the pel-recursive motion estimation, the method used to estimate the vector field contains two parts: a segmentation and an estimation part [3]. In the segmentation part the image is segmented into changed and unchanged areas. This segmentation mask is used in the estimation part.

The pel-by-pel estimation of motion vectors from local time-varying intensities is an ill-posed problem. To circumvent this ill posedness, the motion field is assumed to vary smoothly, except for discontinuities at object boundaries. In the pel-recursive motion estimation algorithm [4], [5] a real-valued motion field \( d(x, t) \) is estimated where one motion vector is calculated for every picture element (pel). This is done in a scanwise manner as follows.

For every pel at the location \( (x, t) \), a real-valued motion vector \( d(x, t) \) is predicted based on previously estimated motion vectors of certain neighboring pixels. The actual motion vector \( d(x, t) \) is obtained through an update scheme that computes the update \( \hat{u}(x, t) = d(x, t) - d(x, t) \), using a causal window of data (fixed size \( N \)), around the pixel in the current frame that is being processed. This is done under the assumption that the motion is constant for all pels within this so-called update window. The basic formula is thus as follows:

\[
\hat{d}(x, t) = d(x, t) + \hat{u}(x, t) \tag{1}
\]

where \( \hat{u}(x, t) \) is the estimated update vector and \( \hat{d}(x, t) \) is the estimated motion vector.

In this paper, the parallelization of a pel-recursive motion estimation algorithm for real-time implementation is discussed. The implementation of the segmentation part, which is based on the Viterbi algorithm, will not be considered; many Viterbi implementations can be found in literature [6], [7]. For the estimation part, a formal design method, which allows for the exploitation of the underlying global parallelism is used. It is in this respect that both our approach and result is quite different from other pel-recursive motion estimation architectures that can be found in the literature [8]-[10]. Our architecture emerges from the exploration of a large solution space while the implementations in [8]-[10] result from heuristic mappings on restricted target architectures. The outline of the paper is as follows. In Section II, the sequential algorithm is presented and the data flow is discussed. These specifications will largely define the subsequent parallelization and the array architecture, which are presented in Section III. The architecture is a linear array of processors, each consisting of an initialization part, a routing part and an updating part. The data router is discussed in Section IV. The main objective of this paper is to derive an architecture from the algorithm. Qualitative evaluation of the algorithm can be found elsewhere [4].

II. SEQUENTIAL ALGORITHM

A. Algorithm

The sequential pel-recursive motion estimation algorithm for an infinite image sequence, with sequence axis \( t \) and with
image (frame) size \( x_{\text{max}} \times y_{\text{max}} \), \( x \) for horizontal and \( y \) for vertical dimensions respectively, is as follows [3]–[5]:

for all \( t \), \( t > 0 \) \{
  for all \( y \), \( 0 \leq y \leq y_{\text{max}} - 1 \)
  for all \( x \), \( 0 \leq x \leq x_{\text{max}} - 1 \)
  \( d_{\delta}\phi(x, t) = \sum_{(x, y) \in S_t} \alpha_{ij} (x - i, y - j, t) \) \( (2) \)
}

for all \( n \), \( 0 \leq n \leq N - 1 \)
\( G_{n,1} = -f_x(x_n - d_{\delta}\phi(x, t), t - 1) \)
\( G_{n,2} = -f_y(x_n - d_{\delta}\phi(x, t), t - 1) \)
\( z_n = f(x_n, t) - f(x_n - d_{\delta}\phi(x, t), t - 1) \)
\}

\( \hat{u}(x, t) = G^T z \)
\( \hat{d}(x, t) = d_{\delta}\phi(x, t) + \hat{u}(x, t) \) \( (7) \)

In this algorithmic description \( G \) is an \( N^2 \times 2 \) gradient matrix; \( G_{n,m} \) are the matrix entries; \( f_x \) and \( f_y \) denote the gradients of the intensity function \( f(x, t) \) in the \( x \) and \( y \) direction, respectively; and \( z \) is a displaced frame difference vector of size \( N^2 \). \( z_n \) are the vector entries. In this algorithm the three external loops assure that the motion estimation process is performed for every pel. We will now elaborate shortly on each algorithmic step, explaining the meaning of the formulae and symbols used in (2) through (7).

B. Prediction Strategy

The prediction strategy relies on the spatial smoothness of the motion field, which is based on the assumption that neighboring pels have similar motion vectors. Thus recursion in the image plane is a very natural choice. Traditionally, pel-recursive algorithms have used a horizontal line predictor [3], [5], in accordance with the scanwise transmission principle. A disadvantage of this predictor is that the estimator needs to converge at each row. This problem can be overcome by performing a 2-D prediction [4] using the spatial prediction model of (2) where \( S_t \) is a causal spatial support region and the \( \alpha_{ij} \) parameters are the corresponding prediction coefficients. The \( \alpha_{ij} \) parameters may be weighted by the segmentation results.

C. Update Strategy

The ill-posedness of the motion estimation problem is circumvented by assuming that the current pel \( (x, t) \) resides in a so-called update window \( U_{\delta}\phi(x, t) \) encompassing \( N \) pixels, \( x_n \in U_{\delta}\phi(x, t), n = 0, \ldots, N - 1 \), that are assumed to have the same motion vector \( d(x, t) \). The update window \( U_{\delta}\phi(x, t) \) is defined such that the algorithm remains causal and can thus be used in a coding environment. Some examples of possible update windows are shown in Fig. 1 for \( N = 4 \), \( N = 9 \), \( N = 5 \), and \( N = 13 \), respectively. Note, however, that in this paper \( N \) is a fixed parameter; the design procedure thus proceeds with a fixed \( N \) without imposing a specific value for \( N \).

The basic equation for the update estimate is as follows [4]:
\( z = G u(x, t) + v(x, t) \) \( (8) \)

Here, the matrix \( G \) is an \( N^2 \times 2 \) gradient matrix, containing the gradients of the image intensities of the previous frame at the displaced update window positions, as given by (3) and (4). The vector \( z \) is the \( N^2 \times 1 \) so-called displaced frame difference (dfd) vector, i.e., the entries of \( z \) are defined as given by (5). The vector \( v(x, t) \) is an \( N^2 \times 1 \) linearization error term that results from working with an initial prediction \( d_{\delta}\phi(x, t) \) of the motion vector \( d(x, t) \). It should be noted that both \( G \) and \( z \) are unique for every pel, the \( (x, t) \) indices have been neglected for the sake of clarity. Thus, by using an update window, an overly dimensioned basic equation is obtained such that a solution for the update vector \( u(x, t) \) can be found.

In order to reduce the complexity without compromising the results, we decompose the real-valued predicted motion vector as follows:
\( d_{\delta}\phi(x, t) = d_t(x, t) + d_f(x, t) \)

where \( d_t(x, t) \) represents the integer part (rounded to the nearest integer) of the predicted motion vector, and \( d_f(x, t) \) represents the fractional part. Experimental results show that the following approximations of (3) and (4) are a good choice:

\( G_{n,1} = -f_x(x_n - d_t(x, t), t - 1) \)
\( G_{n,2} = -f_y(x_n - d_t(x, t), t - 1) \)

This means that the gradients can be calculated for the nearest pel location, and no interpolation will be required to correct for the offset \( d_f(x, t) \). A simple difference operator is used to calculate the gradients at the fixed locations.

The displaced frame difference (dfd), with \( d_{\delta}\phi(x, t) \) as motion vector, needs to be calculated for all pels within the update window (5). Since \( d_{\delta}\phi(x, t) \) is real valued, the displaced image intensities of the previous frame \( f(x_n - d_{\delta}\phi(x, t), t - 1) \) are not available. Contrary to the gradient calculation, experimental results show that it is important to obtain these image values through interpolation. The required interpolation can be done by using the Taylor expansion around the nearest pixel location:
\[ f(x_n - d_{\delta}\phi(x, t), t - 1) \approx f(x_n - d_t(x, t), t - 1) + d_f(x, t) \begin{bmatrix} f_x(x_n - d_t(x, t), t - 1) \\ f_y(x_n - d_t(x, t), t - 1) \end{bmatrix} \]

Thus, the gradients in the previous frame can be used in the calculation of the interpolated dfd. Having formed the dfd vector \( z \) and the gradient matrix \( G \), the approximation \( \hat{u}(x, t) \) of the update vector \( u(x, t) \) from (8) is as given by (6). In this relation \( G^\top \) denotes the pseudo-inverse of the gradient matrix \( G \) [11]. The most attractive method of obtaining this
pseudo-inverse makes use of the singular value decomposition (SVD) of the gradient matrix $G$ [12].

III. PARALLELIZATION

A. Problem Identification

In order to come up with appealing parallel architectures for the pel-recursive motion estimation algorithm, it is advisable to explore a solution space that is as large as possible prior to imposing restrictions in terms of affordable resources. Thus, the first step in the architecture design trajectory is a parallelization step, which consists of deriving a local single assignment version of the given sequential algorithm. An equivalent representation, then, is the so-called dependence graph (DG), which is a graphical state representation of the algorithm. This DG does not suggest any possible structures whatsoever. In the algorithm considered here, the dimensions of the DG will be at least five; one being the image sequence axis and four describing the process of obtaining the predicted motion vector, which is given by (2) in its most general form. This relation shows that the $\alpha$ parameters can propagate [13] in the $(x, y)$ plane and that the $d$ vector can propagate in the $(i, j)$ plane.

In this paper we shall not derive the true dependence graph; instead we shall project the two dimensions $i$ and $j$ onto the $(x, y)$-plane which implies that all points $(x, y, t)$ will somehow store $\alpha$'s as well as the intensities $f(x, t)$. Generally speaking the resulting graph is not a dependence graph, although it may be considered as such if we do not make explicit these storage capacities in the graph, i.e., if we leave them inside the processes to be defined in the nodes $(x, y, t)$, for the $\alpha$'s, or replace them by broadcast from external sources, for the $f(x, t)$. As a result, we define an index space as shown in Fig. 2, with the bound as follows:

$$0 \leq x \leq x_{\text{max}} - 1, \quad 0 \leq y \leq y_{\text{max}} - 1, \quad 0 \leq t < \infty. \quad (13)$$

For every vector that needs to be estimated, we define a motion estimation process $P(x, t)$ which is attributed to each location $(x, t)$. The image intensity values that are needed are assumed to be broadcast as input to these processes. The I/O maps of the processes are as follows:

$$d_0(x, t) = \sum_{(i, j) \in S_x} \alpha_{ij} d(x - i, y - j, t)$$
$$\hat{u}(x, t) = G^+ z$$
$$\hat{d}(x, t) = d_0(x, t) + \hat{u}(x, t). \quad (14)$$

Parallelization of these equations will not be considered in this paper. Instead we will concentrate on the parallelization of the set of processes, obtaining a new ordering as compared to the ordering given in the algorithm.

Process $P(x, t)$ requires image intensities from the update window $U_x$ around pixel $(x, t)$; these are current-frame (cf) input signal intensities. It also requires image intensities from the update window $U_y$ around some pel $(x', t - 1)$ where $x'$ can be any point in the $(x, t - 1)$-plane; these are previous-frame (pf) input signal intensities. The exact location of the required update window in the previous frame is determined by the integer part of the predicted motion vector $(d(x, t))$ calculated by the process.

As a further simplification, we will interpret the $(x, y, t)$ index space as a space-time partitioned index space, meaning that the $t$ dimension provides the evolution of the $(x, y)$ plane. We will now identify the interprocess dependencies in the $(x, y)$ plane itself. Both the pf and the cf dependencies are considered to be inputs from outside the $(x, y)$ plane. The pf data are assumed to be broadcast from a pf memory. The localization of this broadcast is a major problem and will be considered in Section IV. The localization of the cf data is easier and will not be considered in detail.

The prediction strategy defines the dependencies between the processes in the $(x, y)$ index space. Although in general a DG is not trivially obtained from a sequential algorithm, (see [13] for more details), it is straightforwardly set up in the present case. An often-used spatial prediction support $S_d$, with good prediction results, is the quarter-plane support, with the quarter plane prediction relation as follows:

$$d_0 = \alpha_{00} d(x - 0, y - 0, t) + \alpha_{11} d(x - 1, y - 1, t) + \alpha_{01} d(x - 0, y - 1, t) \quad (15)$$

where the $d(x - d, t)$ are known (past) estimated motion vectors. The vectors $d_0 = (1 \ 0)^T, (0 \ 1)^T$ and $(1 \ 1)^T$ are called the dependence vectors.

Other 2-D prediction schemes are possible, such as the nonsymmetric half-plane prediction. We shall pursue, without impairing generality, the quarter plane prediction case.

Consider a single-motion estimation process $P(x, y)$. The quarter-plane (QP) prediction defines three dependencies on the results of neighboring processes. Equivalently, the estimated motion vector computed by $P(x, y)$ will be used by three neighboring processes located at $(x, y + 1), (x + 1, y + 1)$ and $(x + 1, y)$, respectively. If we neglect the dependencies on the image intensities from both the current and previous frame ($f(x, t)$ and $f(x, t - 1)$, respectively) this yields the data dependency representation of Fig. 3 for a single node. The formal description of the input–output map of a typical process $P(x, t)$, suppressing the image intensity inputs, is given in the following relation:

$$d(x, y, t) = F_{x,y} d(x - 1, y - 1, t), \quad d(x - 1, y, t), \quad d(x, y - 1, t), \quad (16)$$

where the map $F_{x,y}$ is as described in (14). With (16) it is
Fig. 3. Data dependency representation for a single node.

Fig. 4. Dependencies between the processes in a single image of the index space.

easy to form the dependence graph given in Fig. 4. In Section III-C, the boundary conditions will be discussed.

B. Projection and Scheduling

We are now ready to allocate processes in space and time. We shall do so using an affine transformation as follows [13], [14]:

\[ I' = T \cdot I \]

(17)

where \( I \) is any point in the \((x, y)\) index space and \( I' = (p, t) \) is the location of the process in space \((p)\) and time \((t)\). The transformation matrix \( T \) can be written as:

\[ T = \begin{pmatrix} p & 0 \\ s & 1 \end{pmatrix} \]

(18)

where \( p \) is a vector perpendicular to the projection vector \( p \), and \( s \) is the schedule vector. The choice of the linear transformation thus amounts to a choice of a projection vector \( p \) and of a schedule vector \( s \). Three criteria are used to come to a unique transformation matrix \( T \); minimize the latency, maximize the throughput, and minimize the size of the processor array.

First we come to a unique choice for the scheduling vector by choosing it in such a way as to minimize the latency of the algorithm. This yields \( s = (1, 1)^T \). Next, a projection vector is chosen in such a way that the throughput is equal to 1, hence the processor efficiency is maximal. That is, \( p \) satisfies:

\[ s^T p = 1. \]

(19)

With the chosen schedule vector the two following projection vectors satisfying (19) can be identified:

\[ p_{10} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad p_{01} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \]

(20)

From these two projection directions, which respectively correspond to a row-wise and a column-wise projection, we choose the one that yields the minimal number of processors. Assuming that \( y_{max} < x_{max} \), projection in the row-direction yields a smaller array. Therefore, \( p_{10} \) is taken to be the projection vector. The resulting transformation matrix is:

\[ T_{10} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}. \]

(21)

Fig. 5(a) shows the transformed index space \((p, t)\) space) after a transformation according to (17). The resulting signal flow graph (SFG) is the conceptual processor array depicted in Fig. 5(b). In the \((p, t)\) space all dependencies cross at least once step. This means that in the resulting SFG all dependencies will have at least one delay; we have thus systolized the algorithm.

C. Boundaries

Up to this point the entire DG has been assumed to be uniform, such that the projection of processes onto each other yields a processor whose functionality is fixed and independent of the location. However, special care has to be taken at the boundaries of the original image frame. Fig. 6 shows the \((x, y)\) index space, where the different domains are marked. The central domain is uniform. All processes in this domain can thus be mapped on identical processors. The processes in the boundary domains, however, are different and, therefore, the processors after projection have to differentiate between boundary and central processes.

The processes in domain I of Fig. 6 estimate motion vectors for the pixels in the first row of the image. It is clear that no previous row is available to be used in the prediction strategy. The first row is to be treated differently, performing a purely horizontal prediction strategy. After projection, the first processor of the processor array, which executes all the processes on the first row, will thus be slightly different from the other processors; it will perform a purely horizontal prediction. In domain II a similar reasoning holds. For the first column of the image frame, a purely vertical prediction strategy is desired. Thus, not only the first processor of the array has to perform a 1-D prediction, but all other processors have to distinguish between a 1-D prediction (vertical) and a 2-D prediction as well. Domains III and IV have similar conditional impact on the processors.

As, after projection, the processes from three different domains (II, central, and IV) are mapped on the same processor, a 2-b control signal is introduced. This control signal allows the distinction between the processes residing in different domains to be made. This control signal can be propagated through the \((p, t)\) space yielding a distributed control. It selects either a 1-D vertical prediction or a 2-D quarter-plane prediction function, respectively. It also indicates whether or not the estimated motion vector is needed by the following process.

D. Input Broadcast Localization

Up to now, we have assumed that input data, both from the current and the previous image frames, are local to the processes in the index space. Obviously we have to provide for a data delivery mechanism that takes care of this data availability.
1) Localization of the Current Frame Input Data: Using the quarter-plane prediction strategy along with a quarter-plane update window (Fig. 1 $N = 4$ or $N = 9$) and the schedule vector $s = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$, the current frame (cf) intensities are input straightforwardly. Indeed, consider the mapping of the current frame storage space onto the index space. With this so-called pre-distribution of the current frame, process $x$ in the index space will contain the intensity at position $x$ of the current frame. Every process requires an update window $U_w$ of data from the current frame. Since the update windows satisfy the causality constraint, all the data required by each process can be obtained from its neighbors. For an update window of size $N$, $N - 1$ local dependencies will be defined for each process. All cf input data can be obtained through these local dependencies.

As the current frame storage space is assumed to lie on top of the $(x, y)$ index space, $x$ pel on $x$ process, it follows that this data plane will undergo a "space-time" transformation, much as the index-space does, by (17), using the transformation matrix in (21). This yields the transformed current frame storage space. When using certain update windows (e.g., $N = 5$ or $N = 13$) the transformed current frame storage space will have to undergo a shift in the $t$ dimension, assuring that all dependencies satisfy the causality defined by the schedule vector. This shift is represented by the following transformation of the current frame storage space.

$$
\begin{pmatrix}
p^* \\
t^*
\end{pmatrix} =
\begin{pmatrix}
p \\
t
\end{pmatrix} - \begin{pmatrix} 0 \\
\eta
\end{pmatrix}
$$

where $(p, t)$ is the original pel position in the transformed current frame, $(p^*, t^*)$ is the new pel position in the transformed current frame and $\eta (\eta > 0)$ is a small offset variable. What is done is that the execution of a certain motion estimation process is delayed $\eta$ steps after its corresponding cf signal input is received.

2) Localization of the Previous Frame Input Data: Each process in the index space requires a window of data from the previous frame. The location of this data in the previous frame is dependent on the integer part of the predicted motion vector, $d_p(x, t)$. Therefore, no a priori knowledge about the dependencies between the previous frame storage space and the processes in the index space is available. Dependencies over long distances boil down to either very long connections, requiring large chip area, or large transfer delays. We therefore decide to a priori limit the extend of the possible dependencies by imposing a maximum $l = \|d_p\|_{\text{max}}$, on the length of the predicted motion vector (in a typical imaging system $l = \pm 8$). In order to assure uniformity throughout the drawings, we will now continue with $N = 4$ without limiting the validity of the discussion. When the update window has the size $N = 2^*2 = 4$ this gives a square displacement window of $(2l + 2)*(2l + 2)$ pixels from which data can be requested.

Fig. 7(a) shows the possible dependencies of one process on the previous frame for a displacement window of $4*4$ pixels (for $l = 1$ and for $N = 4$). Strictly speaking, these dependencies should all be allowed, such that the process can select any subset of the data when the predicted motion vector becomes available. The full connectivity that this requires for each process makes this solution unsuitable for implementation in VLSI. To overcome this drawback we choose for a solution as shown in Fig. 7(b), where the requested window of data is somehow routed to the appropriate process of the index space. In Section IV we elaborate on the routing of pf data.

In order to have this previous frame data available at the array processors, a certain amount of data will have to be stored in the array. Two main questions arise: first, how much data need to be stored within the array and second when and where are these data to be stored.

The amount of pf-input data that needs to be stored internally at any time $t = t_0$ is determined very easily. Indeed, as can be seen from Fig. 8, a processor will have to store only part of an image row. More precisely, any process $P(x, t_0)$ that executes at time $t = t_0$ can request data from a displacement window similar to the displacement window for process $P$ that is shaded in the figure. All these processes together define a band of previous frame data that has to be available in the array at any time $t$. The width $w$ of this band is

$$
w = 4l + w_0
$$

where $w_0$ is a constant dependent on the update window geometry and size. For the windows given in Fig. 1 we have $w_0$ equal to 3, 5, 3, and 6 respectively. Using the $N = 4$ window and with $l = 8$ this gives $w = 35$, which is very little compared to the entire image width.

When the processors execute the "next" processes from the index space, the band of stored data moves one pixel to the right, adding one pixel to the right of each processor's internal memory and removing one pixel from the other side.
Each processor thus has a FIFO memory of length \( w \) into which \( pf \) data will be loaded. As was the case for the current frame and the index space itself, the previous frame also needs to be transformed to the \((p, t)\) index space according to (17), using the transformation matrix given by (21). After this transformation, \( p \) denotes the row processor in which a specific pel will be stored and \( t \) gives the time at which this pel is in the middle of the FIFO register.

3) Physical Inputs: In the architecture described in this paper the rows of input data are entered into the array in a parallel manner, i.e., at each time step the data required by all row processors are entered in parallel. We assume that performing this parallel input will not pose any problem. However, in case this assumption cannot be tolerated, it is not difficult to replace the parallel acquisition by a serial one and to propagate this serially input data to the various processes prior to the execution of processes. We shall not pursue this converor here, for a possible strategy see [13], [15].

E. Gradient Calculation

When a process requests a window of previous frame data to be used in the update calculation, it also needs the gradients at the pel locations within this update window. Two gradient calculation strategies can be distinguished. On one hand, the gradients can be precalculated. For all pixels located within the band of internally stored data, the gradients will be calculated and stored along with the intensities. When a certain update window is requested, then the gradients will be treated just like the intensities; a package containing both the gradients and the intensities of all pixels within the update window will be formed and routed to the request-

F. Parallel Array

In Fig. 9 the signal flow graph is given when quarter-plane prediction (three dependencies on the left) and an \( N = 4 \) update window (three dependencies on the right) are used. In this figure the processor's functionality has been represented by three functions. On the left we find the prediction function (2). The propagation control signal that allows for the selection of the boundary processes is also shown. The routing function, which will be developed in the following section, is in the middle. The update calculation function is on the right side. This part houses most of the algorithm, i.e., (3)-(7). Each processor requires a time \( D \) to do one process from the index space. During this time the motion vector prediction is performed, the routing is done, and an update is calculated.

G. Partitioning of the Array

Up to now we have assigned one processor to every row of image data. Depending on the imaging application, this may yield a very large amount of processors, up to 1152 processors for high-definition television (HDTV). It is likely that this amount of processors is not affordable, so that folding will be necessary. The problem of mapping many computations into a processor array of fixed size is called the partitioning problem [13]. When partitioning a large array (size \( R \)), \( M \) partitions of processors, each of size \( Z = R / M \), are created. Before projection, the DG is thus partitioned into \( M \) blocks of processes.

There are two methods for mapping the partitioned DG to an array: locally sequential globally parallel (LSGP) and locally parallel globally sequential (LPGS) [13], [16], [17]. In the LSGP scheme, one block is mapped onto one proces-
Fig. 9. Signal flow graph of the processor array.

Fig. 10. Cluster processor.

as it determines in which processor of the array the requested data is located. The \( x \) component of the data request (request_\( x \)) merely determines the relative offset of the data in the local memory of the sending processor. Every data request is transmitted along a request/data path to the destination processor. Here the request is acknowledged by forming a data package that contains the data from the local shift register memory pointed to by request_\( x \). The formed data package is returned to the processor from where the request originated. This is done by using an alternate request/data path that runs in the opposite direction.

By using two request/data paths a great amount of concurrency can be achieved. The requests and/or data packages from all router functions can be handled at the same time. This conceptual method defines the router function I/O terminals as shown in Fig. 11, where the up and down paths, both containing an address request path and a data package path, can be distinguished. It is this conceptual model that will be used to further develop the router function. A hierarchy of possible router solutions exists, ranging from an asynchronous (self-timed) router to the synchronized router that we have used in a simulation of the array. We will derive our router by starting off from the easy to understand concept of the asynchronous router. The actual router will be synchronized in two steps. The first step will solve an addressing problem, the second step will solve a buffering and communications problem. The final router mechanism will prove to be the most simple solution with low function complexity and minimal required buffering. However, it is not necessary to synchronize the router. The choice between asynchronous and synchronous routing will depend on the application, the image statistics, etc.

B. Asynchronous Router

In order to consider asynchronous routing we have to step away from the systolic principle with the scheduling performed in Section III, that is, we have to consider the free schedule imposed by the ban passing mechanism; no schedule vector is involved. Thus the asynchronous routing mechanism works as follows. A segmentation result bit and a predicted motion vector are received. When the segmentation bit indicates that the current pel is unchanged then the current process is finished. The router function can thus hand over to the prediction function to receive the predicted mo-
tion vector of the next pel in the row. When the current pel has changed then the routing as described by the conceptual model will be performed. On receipt of the data package the router function passes it to the update function, after which it will wait for the next predicted motion vector.

With this routing mechanism, the motion estimation processing path is data driven. It is thus possible for a certain processor to move ahead on the time index that was determined by the transformation of relation (17), bearing in mind the causality of the prediction scheme, i.e., the prediction dependencies remain valid. Essentially, this means that a processor can advance with respect to the next processor of the processor array but never with respect to the previous processor.

Clearly, when the entire motion vector prediction path is asynchronous, its calculation rate will vary in time. If the input data are supplied at a predefined rate, buffering on the cf and pf inputs will be required. Also, as the speed of different processors may vary, buffers have to be placed within the array. The exact amount of buffering needed will be determined by the image statistics, i.e., the variance in the segmentation results and in the length of the motion vectors. In this kind of architecture an additional problem arises with respect to the addressing used by the requests. For request_y, which points to the processor the request is send to, a relative address can be used. This means that request_y simply equals the integer part of the y component of the predicted motion vector. For request_x this will not be possible, however. As the time at which the individual processes are executed is unknown, the requesting processor does not know the relative offset of the processor it sends the request to. In fact, this relative offset may even change while a request travels from the requesting to the receiving processor. Therefore, request_x will need an absolute address. This means that an address conversion step will be required upon receipt of the request.

In order to resolve the addressing problem, an intermediate solution to the routing mechanism can be found. Thus, as in the previous model, the routing is done asynchronously, performing the routing in minimum time. The difference is that the computation front must be straight as in the systolic case. This means that no processor is allowed to move on to the next process unless all other processors have finished their previous routing tasks. This resolves the addressing problem since the relative x-offset of each processor with respect to any other processor is strictly determined by the distance between the two along the path. However, this introduces a communication problem. When a router function receives its returned package, it needs to wait until all routing is finished. This requires some kind of global communication to allow each router function to know the status of the other processors. This communications problem will disappear when the router works synchronously, as the synchronism itself will provide the mechanism to assure that all processors have performed their job within the same time slot.

C. Synchronous Router

1) Worst-Case Routing: A maximal limit (I) was imposed on the allowed motion vector length in both the x and y directions, limiting the routing for each processor to a square displacement window in the (x, y) index space. This defines a critical path corresponding to a worst-case routing, which is the routing that occurs when the y component of all predicted motion vectors have their maximum length. In this case the sum of the traveling time of the request and the traveling time of the return package must be constant, for all processors.

In this section, the router will be synchronized. This is done by having the request addresses and the returned data packages travel over the maximal distance l, such that the worst-case routing is performed at any time. There are three main advantages to this synchronous routing. First, the buffering requirements will be lifted. Second, the router algorithm becomes much more simple, greatly reducing the complexity of the router functions. Finally, the communication problem, incurred when the addressing problem was handled, is resolved.

2) Synchronous Mechanism: In order to clarify the working of the synchronous routing, an example of an array of 12 processors is considered. The array is shown in Fig. 12(a)–(d), where the processors are numbered 1 through 12. Fig. 12(a) depicts a sample of the routing mechanism where a single request with request_y ≥ 0 is considered, neglecting all other routing that takes place at the same time. Processor 6 sends out a data request with request_y = 3. This means that the predicted motion vector of the process in processor 6 points to a pf data window of which the bottom right corner is stored in processor 3. Since a whole window of data covering two lines (N = 4) is required, this implicitly requests data from processor 2 as well.

Fig. 12(a) shows how the data request from processor 6 will propagate: the request will be passed up the router array where the relative address will be decremented by each processor. When the request reaches processor 3, it will have the value request_y = 0, indicating that data are needed from this processor. The required data from processor 3 will be put in the appropriate locations of a data package transmitted in parallel with the request. The combination of the request and the partially filled data package will be transferred to the next processor, where the request will get the value request_y = -1. This indicates that this is the processor
This is illustrated in Fig. 12(c) and (d). In Fig. 12(c), processor 7 sends out a request for data with request_y = -2. This request is put on the request line in both directions. The request in the up direction is a dummy request.

The request in the downward direction will be transferred down to the maximum distance of l = 5, incrementing request_y at each transfer. When the request reaches processor 12, which is at a distance l from the requesting processor, it will be transferred to the upward path. In this upward path, the operations will be identical to the operations described before; request_y will be decremented at each transfer (Fig. 12(d)). When request_y has a value equal to either 0 or -1 the appropriate locations in the data package that travels in parallel with the request will be filled with the requested data. Again, after a fixed number of transfers, the requested data will become available.

The synchronous mechanism is identical for every request from every processor. The requests and/or data packages from processors on either end of the array need to be able to travel the maximum routing distance l as well. This is made possible by extending the actual processor array on either side with an array of dummy processors of length l, each containing a dummy routing function. Any request for data outside the image boundary will be acknowledged by the appropriate dummy router function with a null package. The array of dummy router functions has been represented in Fig. 9 as a router terminator. It can be seen as an extended LIFO storage register, storing the data/request packages for an appropriate amount of cycles.

3) Evaluation: In both cases, whether data are requested from the up direction or the down direction, the operations are identical. This means that all requests for all rows in the image can be handled at the same time. For all processes executed at a certain time, a predicted motion vector becomes available at the same time. The routing can then be fully parallel for all data requests. Notice that the synchronous routing mechanism enables us to keep the routing complexity minimal. No special buffering is required, the only address conversions are simple increments and decrements and no global communications are required. If the synchronization through an internal counter is found to lack robustness, many other techniques can be used, from an external control signal to extending the data path with a distance counter.

The time cost of synchronizing the router, and therefore the entire processing array, is that no advantage can be taken when no data are to be routed (segmentation = unchanged). Even when none of the processors require pf-data, dummy routing is performed. This disadvantage is to be evaluated against the communication overhead incurred by the other models. The entire algorithm, and in particular the router, have been simulated extensively on sequential and parallel simulators. The parallel simulation confirms the correct behavior of the router and the entire parallel architecture. For the evaluation of the algorithm itself see reference [4].

V. CONCLUSION

We have addressed the problem of an effective and efficient pel-recursive motion estimation architecture. This prob-
problem is of interest in that 1) it is an alternative to block-matching which has attracted much more attention in the literature and 2) it is a nonfixed (data-dependent) algorithm that has a larger architecture design solution space than in the case of data-independent algorithms. We believe that the proposed architecture is the first pel-recursive motion estimation which is not obtained using heuristic mapping techniques.

The formal design method has led to a column parallel architecture in which one processor is allocated to one or more image rows. This array of processors performs the motion estimation process for a whole column of pel in parallel, going from one column to the next in a systolic manner. Each processor consists of three parts: an initialization part, a data-routing part, and an updating part. The most attention is given to the routing parts that constitute a routing path along which previous-frame data are routed from processors that store to processors that request such data. It is this routing that is the central part of the parallel architecture.

ACKNOWLEDGMENT

The first two authors are indebted to R. L. Lagendijk and J. Biemond for the fruitful discussions and continuous encouragement.

REFERENCES


Emmanuel D. Frimout (S’89–M’91) was born in Ukkel, Belgium, on April 13, 1968. He received the M.Sc. degree in electrical engineering in 1991 from the Delft University of Technology, Delft, The Netherlands.

He is currently working as a Research Assistant at the Delft University of Technology in the Information Theory Group. His research interests include signal processing and parallel architectures with applications to image/video coding.

Johannes N. Driessen was born in Maassluis, The Netherlands, on November 30, 1963. He received the M.Sc. degree in electrical engineering in 1987 from the Delft University of Technology, Delft, The Netherlands. His thesis was concerned with image sequence restoration. Since then he has been working toward the Ph.D. degree at Delft University of Technology with prof. dr. J. Biemond as supervisor. His thesis subject is motion estimation for image sequence processing. This research is being supported by the Netherlands Technology Foundation (STW).

His research interests include motion estimation from image sequences and its applications in image sequence coding and filtering and dynamic scene analysis.

Ed F. Deprettere (SM’88) was born in Roeselare, Belgium, on August 10, 1944. He received the A.S. degree from the Ghent State University, Ghent, Belgium, in 1968, and the Ph.D. degree from Delft University of Technology, Delft, The Netherlands, in 1981.

In 1970, he became a Research Assistant and Lecturer at the DUT, where he is now Associate Professor in the Department of Electrical Engineering, Network Theory Section. His current research interests are in modern signal processing—algorithms and VLSI architectures and applications and in methodologies for the mapping of parallel signal processing algorithms, network graphs, and numerical computations onto silicon. He is the editor of the book SVD and Signal Processing: Algorithms, Architectures and Applications (North-Holland, 1988) and of the book Algorithms and Parallel VLSI Architectures (North-Holland, 1991). He is Associate Editor of the Journal of VLSI Signal Processing, the journal Integration, the VLSI Journal, and of the IEEE Transactions on Signal Processing. He coauthored a paper that received a 1989 IEEE SP award. Dr. Deprettere is a member of EURASIP and SIAM.