Journal of VLSI Signal Processing, 4, 331–342 (1992)

Error Detection in Arrays via Dependency Graphs*

EDWIN HSING-MEAN SHA AND KENNETH STEIGLITZ
Department of Computer Science, Princeton University, Princeton, NJ 08544

Received August 9, 1991; Revised December 3, 1991.

Abstract. This paper describes a methodology based on dependency graphs for doing concurrent run-time error detection in systolic arrays and wavefront processors. It combines the projection method of deriving systolic arrays from dependency graphs with the idea of input-triggered testing. We call the method ITRED, for *Input-driven Time-Redundancy Error Detection*. Tests are triggered by inserting special symbols in the input, and so the approach gives the user flexibility in trading off throughput for error coverage. Correctness of timing is proved at the dependency graph level. The method requires no extra *PE*s and little extra hardware. We propose several variations of the general approach and derive corresponding constraints on the modified dependency graphs that guarantee correctness. One variation performs run-time error correction using majority voting. Examples are given, including a dynamic programming algorithm, convolution, and matrix multiplication.

1. Introduction

Reliability is often a critical issue in applications of high-performance systolic or wavefront array processors, and for that reason much recent work has addressed the problems of on-line error detection (see, for example, [1]). We consider in this paper a flexible and general methodology for incorporating error detection in array design.

The two general approaches pursued in the literature for error detection are *hardware* and *time redundancy*. That is, one can detect errors by introducing additional computing hardware, perhaps duplicating *PE*s, or one can do duplicate computations using the same hardware. In general, there is a tradeoff between the decrease in throughput caused by the time redundancy, and the cost of the extra hardware used for hardware redundancy. A high degree of time redundancy can achieve good error detection, but at the cost of decreased throughput; a high degree of hardware redundancy can do the same without the attendant decrease in throughput, but at the cost of more hardware.

Much previous work takes advantage of the regularity of systolic arrays. For example [1] describes algorithm-based techniques that are especially suited to systolic arrays, but these are applicable only to a subset of linear systems, and it is unclear how to use them on problems like the substring comparison we consider in Section 2. The work in [2], [3] uses dual-module redundancy to detect errors; the essentially time-redundant technique of [4] applies only to unilateral linear arrays and results in a slowdown by a factor of two; [5] also deals with special classes of systolic arrays and agains halves the throughput rate using time redundancy. The method of *roving spares* described in [6] uses limited hardware redundancy, but it is not clear how to extend the method to bilateral arrays or more complicated structures.

This idea of using tokens to trigger error detection appears to have been introduced in [7]. They use both time and space redundancy, and a fixed periodic pattern of inserting tokens. In the case of unilateral linear arrays, the number of inserted tokens in the array at any instant cannot exceed the number of extra *PE*s. Thus, the frequency of token insertion is predetermined by the number of extra *PE*s. In the case of bilateral linear arrays, they make use of the idle *PE*s and idle cycles in the original computations for space and time redundancy, so only one extra *PE* is needed.

We will combine two ideas to achieve run-time error detection: First, as in [7], we introduce special symbols in the input that signal the processors to perform comparisons for the purposes of detecting discrepancies. Typically, this is done by having two (or more) adjacent processors perform the same computation and comparing results. In contrast with [7], however, the frequency of insertion of these special symbols is determined by

*This work was supported in part by NSF Grant MIP-8912100, and U.S. Army Research Office-Durham Grant DAAL03-89-K-0074.

the user at run time, rather than being pre-determined by hardware constraints. Second, we introduce the special symbols at the level of the dependency graph, and follow the effect through the projections used to arrive at a systolic or wavefront array [8].

There are several advantages to this general approach over more specialized or *ad hoc* approaches. First, it allows the user to determine the frequency of error checking at run time. Thus more error checking can be done when a lower throughput is acceptable. A second advantage stems from the fact that the method is expressed in terms of the dependency graph. This allows us to use previous work [8] on scheduling and projection to prove the correctness of the resulting working architectures. A third advantage is that the approach requires no extra *PE*s, and little extra hardware.

In the next section we briefly describe dependency graphs using the problem of finding minimum substring-distance as an example. In Section 3 we describe the general methodology of ITRED. In Section 4 we discuss our fault model at the level of array nodes, nodes in the signal flow graph that are mapped to the working architecture. The details of implementing ITRED for unilateral linear arrays, which include the minimum substring-distance problem and convolution, are discussed in Section 5. Section 6 then shows how to extend ITRED to more general problems, using matrix multiplication as an example. We prove correctness in Section 7. Finally, in Section 8 we show how ITRED can be adapted to handle some special design requirements.

2. Minimum Substring-Distance

In this section, we introduce as a working example the problem of finding minimum substring-distance. We use this problem to illustrate the dependency graph *DG* and the mapping method for transforming a *DG* to an array architecture [8]. String comparison is a time-consuming and important operation in many applications, such as information retrieval, databases, artificial intelligence, pattern recognition, and DNA pattern matching.

The *edit distance* between two strings is the minimum number of basic operations (insertion, deletion and substitution) necessary to transform one string to the other. For example, `chao` can be transformed to `sha` by a sequence of three operations as follows:

```
chao (delete c) --> hao (delete o) -->
                                ha (insert s) --> sha.
```

But two transformations suffice:

```
chao (substitute s for c) -->
                                shao (delete o) --> sha.
```

In fact this is minimum, so the edit distance between the two strings is two.

Systolic arrays for computing edit distance between two strings have been described in [9]–[11]. In [12], Landau and Vishkin consider the problem of finding a substring of a string S most similar to a given pattern P. Given string S and pattern P, let $S(i:j)$ be the substring of S from position i to position j and let $dis(S(i:j), P)$ be the edit distance between $S(i:j)$ and P. The *minimum substring-distance* is the minimum distance $dis(S(i:j), P)$, where i and j range from 1 to the length of S. Thus, the minimum substring-distance between the string "I like Systolic VLSI arrays," and "Systolic arrays" is five.

The problem of minimum substring-distance can be solved by two-dimensional dynamic programming, which in turn can be implemented by a one-dimensional systolic array.

An input instance of the problem is

$$S = s_1 s_2 \ldots s_n\text{: a (long) string}$$

$$P = p_1 p_2 \ldots p_m\text{: a (short) string}$$

The output of the problem is the minimum of all edit distances of substrings $S(i-k:i) = s_{i-k}s_{i-k+1}\ldots s_i$ from the pattern P, where $1 \le i \le n, 0 \le k \le i-1$.

The dynamic programming algorithm proceeds as follows. Let $D[i, j]$ denote the minimum distance of all substrings as s_i from the prefix $P(1:j)$, where $1 \le i \le n$, $1 \le j \le m$. Initially,

$$D[i, 0] = 0 \qquad \text{for every } i \text{ and}$$

$$D[0, j] = j \qquad \text{for every } j.$$

If we think of the $D[i, j]$ as being in a two-dimensional array, each $D[i, j]$ can be computed from the entries above, to the left, and above and to the left, as follows:

```
for i = 1 to n do
  for j = 1 to m do
    D[i,j] = min ( D[i − 1, j] + 1, D[i, j − 1] + 1,
      D[i − 1, j − 1] if s_i = p_j or
                  D[i − 1, j − 1], otherwise )
```

When this double loop is completed, the entries $D[i, m]$ contain the minimum distance of all substrings ending at s_i from the pattern P. If we consider each *min* operation as a node and represent each dependence

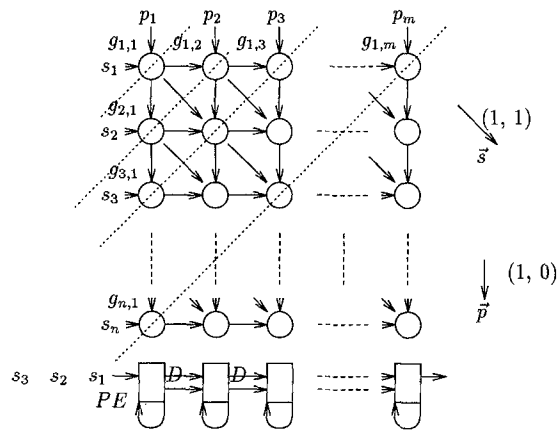

Fig. 1. Dependency graph for minimum substring-dist.

of an operation on data as a directed edge between two nodes, the resulting dependency graph *DG* is as shown in figure 1. The graph *DG* is acyclic and therefore computable.

We call a node in *DG* a *computation cell*, or *cell*. As described in [8], the two design steps of *processor assignment* and *scheduling* can be used to map such a *DG* to a lower dimensional *signal flow graph SFG*. We call a node of the signal flow graph a *Processor Element* (*PE*), this being justified because the signal flow graph is very close to a hardware specification for a SIMD systolic or wavefront array. Let an *equiprocessor curve* be a curve containing all the cells of the dependency graph that are projected onto one *PE* of the signal flow graph of lower dimension, and let an *equitemporal surface* be a surface containing all the computation cells that are active at a given time.

Usually, the equiprocessor curves are parallel straight lines, in which case we let $\vec{p}$ be a vector parallel to the equiprocessor lines, called the *projection vector*. Further, it is often the case that the dependency graph has a *linear schedule*; that is, all equitemporal surfaces are parallel hyperplanes, and so have a unique normal direction. Let $\vec{s}$ be a vector in this normal direction, called the *schedule vector*.

Kung [8] showed that given a projection vector $\vec{p}$, necessary and sufficient conditions for a linear schedule to be *permissible*, that is, represent a realizable computation in the signal flow graph, are the following:

(1) $\forall$ edge $\vec{e}$ in *DG*, $\vec{s}^T\vec{e} \geq 0$.

(2) $\vec{s}^T\vec{p} > 0$.

In our example of the minimum substring-distance problem, we can choose the projection vector $\vec{p} = (1, 0)$ and the permissible linear schedule $\vec{s} = (1, 1)$, as shown in figure 1. This leads to a signal flow graph with m processors, where m is the size of the pattern P, and that is reasonable since n, the size of the string S, is usually very much larger than m.

3. ITRED: General Approach

In this section we discuss ways of modifying dependency graphs to achieve error detection, and we will call a specific algorithm for doing so a *strategy*. The strategy determines the way in which special symbols are inserted in the input data stream. We propose two approaches. In the first, we derive some strategies that allow every *PE* to be tested if the user chooses to provide the right inputs. In the second approach not only can every *PE* be tested consecutively by choice of the input stream, but the computation results themselves can be produced by majority vote. We begin with the first approach, which is actually a special case of the second.

We use a special input symbol, called α, which serves the purpose of informing a *PE* to do error detection (as in [13]). When PE_i receives an α symbol, PE_i will do the same operation as PE_{i-1} and compare its result with that of PE_{i-1}. (We assume here that PE_i is in fact capable of performing the same operation as PE_{i-1}. If all processors are not identical, this requirement might require augmenting the capabilities of some of the processors.) If the results are not the same, an error has been detected. The user has the freedom to decide how frequently an α symbol is inserted in the original input. At one extreme, the user inserts no α symbols, in which case there is no decrease in throughput. At the other extreme, the user inserts an α symbol before each input data point in the original input stream, so the throughput becomes at most half the original speed. Thus, the tradeoff between speed and error coverage is under user control.

DEFINITION 3.1. We say a strategy for inserting α's into the input stream is α-successful if all *PE*s are tested at least once and all computation cells have the correct timing.

Actually, ITRED can be easily extended so that every *computation cell* is tested, but sometimes we may need to add extra *PE*s so the computation cells on the border can be tested.

We want to think of adding the α symbols into the original dependency graph; to do this we add special

cells called α *cells*. In the dependency graph, the effect of an α symbol is similar to a delay, since when PE_i receives an α symbol, it will save its state, discard what it produces after it simulates PE_{i-1}'s computation, and then restore its previous state.

For simplicity, we first consider the case of a two-dimensional dependency graph G like the one in figure 1, with m columns and n rows. Without loss of generality, we assume that data for a particular problem instance enters along a row (*row input*), and flows from column to column. Let $g_{i,j}$ be a computation cell, where $1 \leq i \leq n$, and $1 \leq j \leq m$.

To insert an α symbol in the input stream that travels from *PE* to *PE*, insert a complete *row* of α cells in the dependency graph, as shown in figure 2. If this row is inserted before row i, this splits G into two parts, the part from row 1 to row $i-1$, and the part from row i to the last row. Keep the edges that went from row $i-1$ to i in the first part. Let $\vec{\alpha}$ be the vector normal to the added row, so $\vec{\alpha}$ is (0, 1). Note that in other, more general situations the inserted α symbols may not form a hyperplane, and therefore there may not be a well defined $\vec{\alpha}$ vector. We will see an example of this in a later section.

Let α^j, $1 \leq j \leq m$ be the row of added α cells, ordered in the direction of increasing time. If column j is projected to PE_j, add the directed edge $(\alpha^j, g_{i,j})$. Call these edges *delay edges* and denote by c^j the computation cell pointed to by the delay edge leaving α^j. Since α^j and c^j project to the same *PE*, the difference between their coordinate vectors is a vector parallel to $\vec{p}$. Figure 1 shows the original dependency graph for the minimum substring-distance problem and figure 2 shows the dependency graph modified in the way just discussed.

An α stream inserted into the dependency graph in this way can be regarded as a surface, which we call an α-*surface*. When the α-surface is a hyperplane, we can call it an α-*hyperplane*. We say that an α-surface is a *cutting surface* if removing it separates the dependency graph into disconnected pieces. We say that a cutting surface is *unicutting* if all the edges crossing this surface cross it in the same direction. Cutting or unicutting hyperplanes are defined analogously.

We next derive constraints on the way in which the original dependency graph should be modified so that testing takes place correctly. We prove later that these conditions are sufficient to ensure that a strategy is α-successful. Observe first that since we need to test every *PE*, the vector $\vec{\alpha}$ cannot be perpendicular to the vector

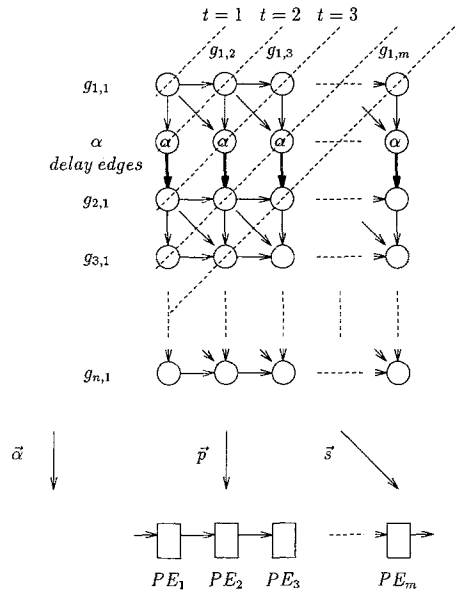

Fig. 2. Modified dependency graph for minimum substring-dist.

$\vec{p}$, and in fact every *PE* should be the image under projection of at least one α cell. Furthermore, because we do not intend to increase the number of *PE*s, we also require that each *PE* be the image under projection of at least one computation cell.

We know that different *PE*s should be tested at different times, so the vector $\vec{\alpha}$ *cannot be parallel to the vector* $\vec{s}$. (When working architecture is a wavefront array, this sequential property of the testing will be naturally ensured by the fact that the testing is data-driven.) Since each α^j is basically a delay for some later operation c^j by the same *PE*, the delay edge should be in the same direction as the vector $\vec{p}$.

Let PE^j be the *PE* to which α^j is projected. We know that whenever a *PE* receives an α, this *PE* needs to do the same operation as its neighboring *PE* will do. Thus, for each α^j there should exist a computation cell (not an α cell) that is projected to PE^j's neighbor at the same time that the α cell is projected to PE^j. We summarize the constraints discussed above in the following, which we call the Σ constraints for hyperplanes.

Σ constraints for hyperplanes:

0. $\vec{\alpha}$ is not parallel to $\vec{s}$
1. $\exists$ an α cell on the border at which data arrives
2. all delay edges are parallel to $\vec{p}$
3. $\forall PE$, $\exists$ an α cell which is projected to *PE*
4. $\forall PE$, $\exists$ a computation cell which is projected to *PE*

5. $\forall \alpha^j$, $\exists$ a non-α computation cell that is in the same equitemporal hyperplane as α^j and is projected to a neighboring *PE* of PE^j
6. The α-hyperplane is unicutting

As noted above the zeroth constraint is not needed at all when the working architecture is a wavefront array, so we assume without loss of generality that the working architecture is a synchronous, systolic array, rather than a wavefront array. Actually, the zeroth constraint is implied by the fifth constraint, so it is redundant and can be omitted. If the equitemporal surface or the α surface is not a hyperplane, we can generalize the above constraints easily as follows:

Σ constraints:

1. $\exists$ an α cell on the border at which data arrives
2. all delay edges are parallel to $\vec{p}$
3. $\forall PE$, $\exists$ an α cell which is projected to *PE*
4. $\forall PE$, $\exists$ a computation cell which is projected to *PE*
5. $\forall \alpha^j$, $\exists$ a non-α computation cell that is in the same equitemporal surface as α^j and is projected to a neighboring *PE* of PE^j
6. The α-hyperplane is unicutting

If the projection, schedule, and modified dependency graph satisfy the above constraints, we say that this dependency graph is *correctly modified*. We leave for Section 7 a proof that a correctly modified dependency graph is α-successful.

In the second approach to modifying the dependency graph, majority voting is applied. In this scheme k adjacent *PE*s will perform the same operation, the output will be the majority result, and error detection will be performed at the same time. We introduce $k-1$ special symbols $\alpha_1, \ldots, \alpha_{k-1}$, which play roles similar to the α symbol. For simplicity, we assume that k is 3, but it is straightforward to extend k to be any odd number. When PE_i receives an α_1 symbol, it performs the same action as before—it simulates a computation in the adjacent *PE*, say PE_{i-1}. If PE_{i+1} receives an α_2 symbol, it simulates the computation of a *PE* which is distance-2 from it, say PE_{i-1}. We need to guarantee that PE_{i+1} receives α_2 and PE_i receives α_1 at the same time, and at a time when they can both simulate the same computation by PE_{i-1}, do the error detection, and output the majority result.

Therefore, α_2 should immediately precede α_1 in the α stream. The constraints analogous to the Σ constraints for performing majority voting are given below, with all terms previously used now indexed by the same index i as the corresponding symbol α_i. For example, $\vec{\alpha}_i$ is the normal vector for the α_i hyperplane.

Σ_{maj_k} constraints for hyperplanes:

1. all the $\vec{\alpha}_i$ are parallel to each other
2. the $\alpha_{k-1}, \ldots, \alpha_1$-symbols are in the same equitemporal hyperplane, and are projected to $k-1$ adjacent *PE*s
3. the α_1-hyperplane satisfies the Σ Constraints

The corresponding more general constraints for the case of surfaces are:

Σ_{maj_k} constraints:

1. all the α_i-surfaces are parallel to each other
2. the $\alpha_{k-1}, \ldots, \alpha_1$-symbols are in the same equitemporal surface, and are projected to $k-1$ adjacent *PE*s
3. the α_1-surface satisfies the Σ Constraints

For example, the modified dependency graph in figure 3 satisfies the above Σ_{maj_k} constraints. Note that if we want every computation cell in the dependency graph to be tested k *PE*s, we may need to add some

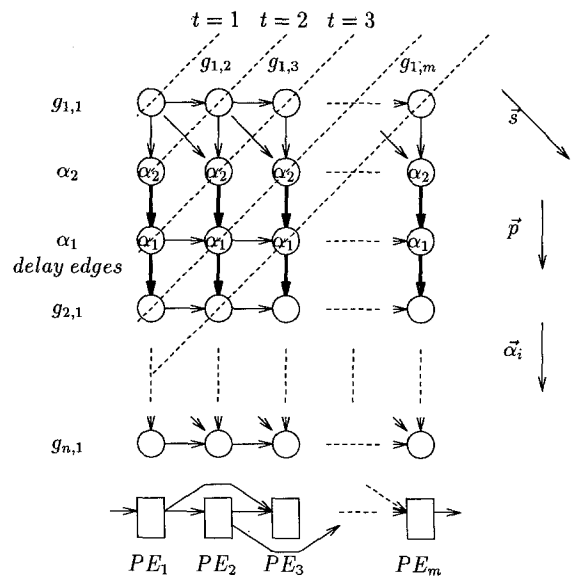

Fig. 3. Modified dependency graph for the minimum substring-distance problem (approach 2).

extra *PE*s to take care of the cells on the border of the dependency graph.

In the remainder of this paper we assume that ITRED uses the first approach (no majority voting), unless we explicitly state otherwise.

4. Fault Model

Given a dependency graph, we project it to a lower dimensional signal flow graph [8], and map this signal flow graph to a working architecture. Each cell of the signal flow graph that is mapped to the real working architecture is called an *array node*, which can usually be regarded as a *PE*. We use array and fault models similar to those in [2], [3], [7].

Each *PE* is composed of two parts: the buffers and the processing unit (*PU*). The buffers can be divided into two parts: the *data* buffers (*DB*) and *internal* buffers (*IB*). *DB* holds the input data and *IB* holds the state necessary to perform the next operation.

In our first approach to run-time error detection, every two consecutive *PE*s do the same operation and compare results. In the second approach, a majority vote determines the outcome if a discrepancy occurs. The comparator and majority voter can be implemented to be totally self-checkable [14], [13], and faults in buffers or communication can be detected and corrected by using coding techniques [14], [13]. The extra hardware for error detection in ITRED is so simple, and therefore can be built so reliably, that we can assume all faults occur in *PE*s.

A fault here will mean a functional fault, not the traditional gate-level stuck-at fault. In the first approach it is usually convenient to assume that when two adjacent *PE*s have their outputs compared, and they are *both* faulty, then their incorrect outputs are different, so that an error is detected immediately. Similarly, in the second approach, where we compare the outputs of k adjacent *PE*s operating on the same inputs, we assume that no k adjacent faulty *PE*s whose outputs are compared produce identical (incorrect) results.

5. One-Dimensional Linear Arrays

In this section we give details of the application of ITRED in the simplest case—one-dimensional linear arrays. Two-dimensional meshes and more complicated topologies are considered in the next section. As mentioned in Section 3 the constraints for introducing α symbols are more stringent for systolic arrays than wavefront arrays, so we restrict attention to the former. We say a linear array is *unilateral* if data flows in only one direction (see figure 4 for an example). We say a linear array is *bilateral* if data can flow between two *PE*s in both directions. We begin with details for the first approach in the unilateral case, and discuss the second approach and the bilateral case subsequently.

Let PE_1 be the leftmost *PE* and PE_i the ith *PE* from the left. For the case of a linear systolic array, this first approach yields a result similar to the one in [7], but no extra *PE* is needed. When PE_i receives an α symbol, it will do the same operation as PE_{i-1} and compare both results. If the results are not the same, an error has been detected. If there are c α's, as long as there is at least one input data value between any two consecutive α's, c different pairs of *PE*s can concurrently check their results. In figure 4, there are two α's and we show the sequence of pairs which do error detection at different clock times.

We next explain the details of the extra hardware required to implement ITRED. As mentioned above, the *PE*s are divided into processing unit *PU*, and buffers—which in turn are divided into the data buffer *DB* and the ouptut buffer *IB*. The buffer *IB* normally stores *PU*'s previous output. We index *PU*, *DB* and *IB* according to their corresponding *PE*.

Without loss of generality, we assume a three-phase clock. In the ordinary situation (without error detection), during the first phase (input phase) PU_i loads data from DB_{i-1} and some part of IB_{i-1} into DB_i. During the second phase (processing phase) processing unit PU_i gets input from DB_i and IB_i, and performs its operation. During the third phase (output phase) PU_i loads its result to IB_i (again, assuming no error detection).

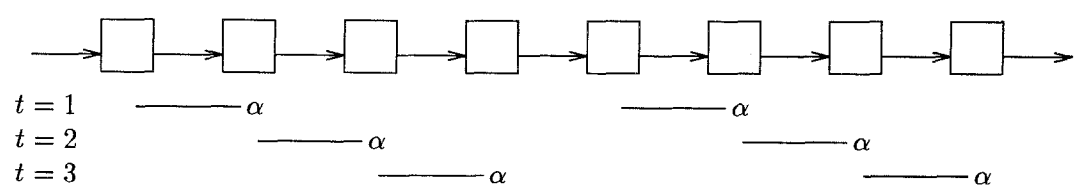

Fig. 4. An example of a unilateral linear array using ITRED.

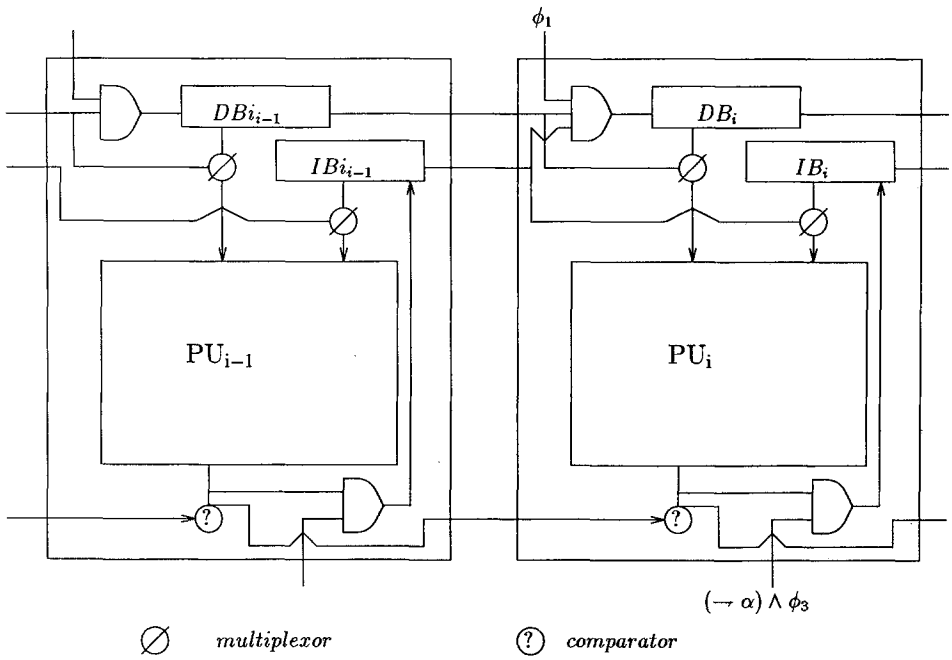

Fig. 5. The PE cells.

In error-detection mode, when PE_i receives an α symbol, it will do the same operation as PE_{i-1} and compare results. The input phase is as before, passing along the α symbol in the input data stream. In the processing phase, PU_i needs to get its input from DB_{i-1} and IB_{i-1}. In the output phase, PE_i will *not* load its results to IB_i, so as to preserve the old contents of IB_i for further use. The only extra thing PE_i needs to do in the output phase is to check its output with the output from PE_{i-1}. A block diagram for PE_i and PE_{i-1} is shown in figure 5.

Now we need to make sure that PE_i will be in the correct state and get the correct input after an α symbol has passed through it. When PE_i receives an α symbol it does not perform its real operation but performs the same operation that PE_{i-1} does. At the next clock tick, say time j, since IB_i did not change at time $j-1$, and data to DB_i is also delayed one tick (because of the α symbol in the input stream), PE_i can perform the same operation as it would have without the α symbol.

We give a simple example in figure 6. Assume the original input data is first a, and then b, c, and write a_i to indicate the state of PE_i after processing a. The succession of PE states without error detection is shown at the top of figure 6. Next, consider what happens when the user inserts an α after a to do error detection. We write a_i^* to indicate that PE_i's internal buffer has not changed, which happens when PE receives an α symbol. The bottom of figure 6 shows the modified succession of events, and verifies the fact that each PE receives the correct inputs and is in the correct states at the right times. From this example, we can see that the timing under a particular strategy may not be obviously correct. A general proof of correctness for ITRED will be given in Section 7.

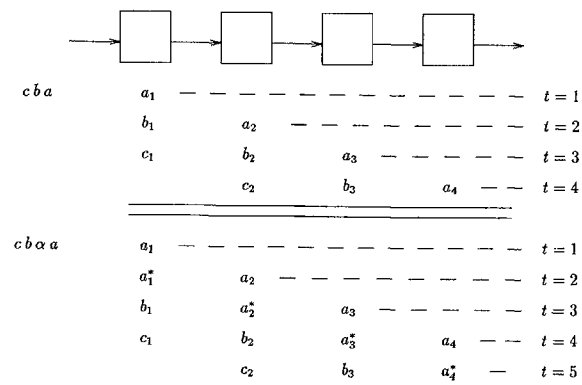

Fig. 6. An example showing correct timing for a unilateral array.

We next discuss the second approach, where the results of more than two computations are compared.

For the purpose of discussion, we assume that the parameter k is 3, so there are two special symbols α_1 and α_2. Since PE_{i+1} now needs to simulate the computation in PE_{i-1}, there needs to be a new data line from PE_{i-1} to PE_{i+1}. We need also to include a majority voter in every PE, which entails only a simple modification of the hardware in figure 5. One correctly modified dependency graph for the above example is shown in figure 3, and a more condensed version of the same systolic array is shown in figure 7. In the next section, we will demonstrate the application of this approach to a two-dimensional systolic array for matrix multiplication.

Next we illustrate the application of ITRED to the case of bilateral linear arrays using the example of convolution. Given two sequences $x(j)$ and $y(j)$, $i = 0, \ldots, n-1$, the convolution for x and y is

$$z(i) = \sum_{j=0}^{n} x(j)y(i-j),$$

where $i = 0, \ldots, 2n-2$. The dependency graph is shown in figure 8.

We first modify the dependency graph to add α symbols, taking care to satisfy the Σ constraints. The vector $\vec{p}$ can be chosen to be (1, 1), which results in a bilateral linear array. Inserting rows of α's results in a unicutting α-hyperplane, and the vector $\vec{\alpha} = (1, 0)$. We then add delay edges that are parallel to $\vec{p}$, shown as bold edges in figure 9. Finally, we choose a schedule, which results in the signal-flow graph shown below the dependency graph. Note that this choice of schedule results in equi-

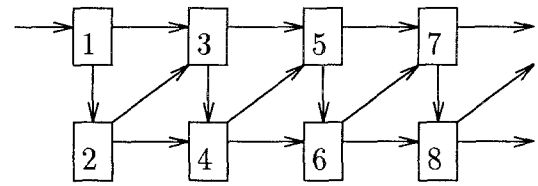

Fig. 7. More condensed systolic array.

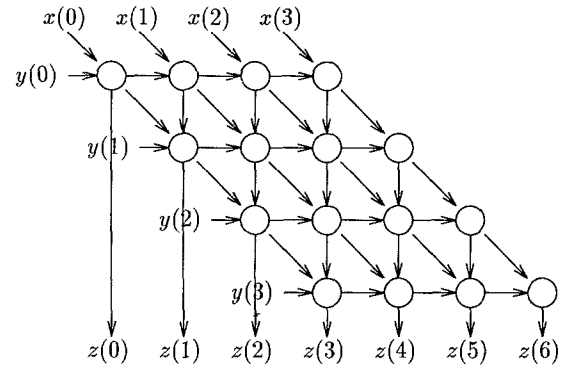

Fig. 8. The dependency graph for convolution.

temporal surfaces that are not hyperplanes. It is now easy to verify the remaining Σ constraints: for every α^j, there exists a non-α computation cell that is in the same equitemporal surface as α^j and is projected to a neighboring PE of PE^j. Figure 9 shows the final, correctly modified dependency graph.

In the original dependency graph every other PE is idle at any given time, and the schedule can use these idle PEs to simulate their neighbors. Under this

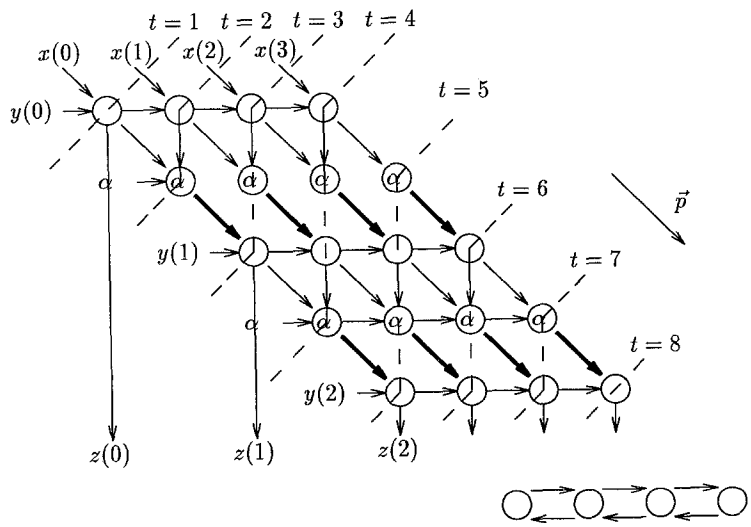

Fig. 9. Modified dependency graph for convolution.

schedule, at most one extra clock period is needed after any number of α symbols are inserted. Although some vertical edges in figure 9 are in equitemporal surfaces, it is still a legal systolic scheduling, since these vertical edges point to α cells and not computation cells. The result in this simple example differs from that of [7] in the following respects: First, our method does not need an extra *PE*. Second, [7] assumes that a *PE* becomes idle at every other cycle, and that every other *PE* is idle at any given time. Our method, however, does not depend on this assumption, but still works when there are no idle *PE*s or no idle cycles are available.

The same general scheduling strategy works for the second approach when $k = 3$. We use idle *PE*s for simulation, and the throughput is reduced by a factor of at most 2 instead of 3.

6. An Example of a Two-Dimensional Working Architecture

In this section, we illustrate how ITRED can be used to incorporate error detection in a two-dimensional systolic mesh for matrix multiplication. Given two n by n matrices A and B, we want to compute $C = AB$. Thus,

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j},$$

where $1 \le i, j \le n$. Writing this as the single assignment statement

$$c_{i,j,k} = c_{i,j,k-1} + a_{i,k} b_{k,j}$$

leads to the three-dimensional dependency graph shown in figure 10, with axes (i, j, k).

We choose the projection vector to be $\vec{p} = (0, 0, 1)$, and the α-hyperplane to be the two-dimensional plane of the input data A, which means that $\vec{\alpha} = (0, 0, 1)$ (see figure 11). The vector $\vec{s}$ can be taken to be (1, 1, 1). It is easy to verify that with these choices the Σ constraints are satisfied, and the correctly modified dependency graph is shown in figure 11.

For the second approach, we can use a two-dimensional hexagonal array to implement majority voting for $k = 3$. A modified dependency graph can be easily obtained from the graph in figure 11 by substituting α_1 for α and adding an α_2 hyperplane above the α_1 hyperplane. When $PE_{i,j}$ receives α_1, it will simulate the computation in $PE_{i,j-1}$, and when $PE_{i+1,j}$ receives α_2, this *PE* will also simulate the computation in $PE_{i,j-1}$. The corresponding two-dimensional hexagonal array representing the working architecture is shown in figure 12.

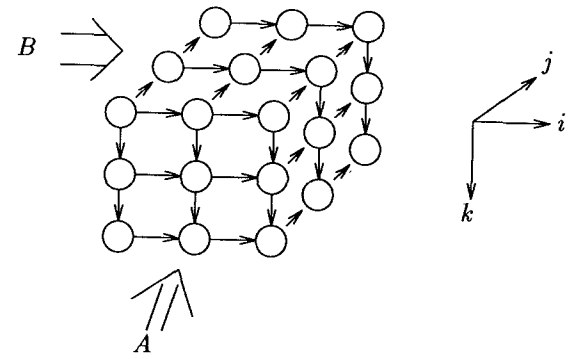

Fig. 10. The dependency graph for matrix multiplication.

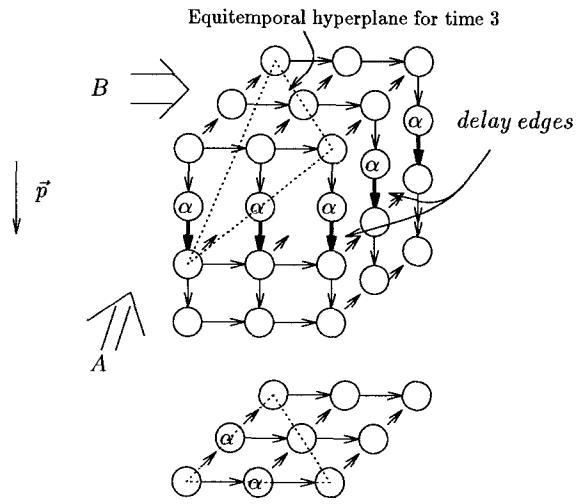

Fig. 11. The modified dependency graph for matrix multiplication.

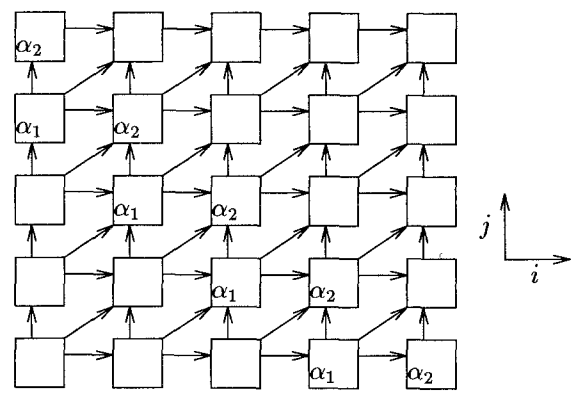

Fig. 12. The hexagonal array for the second approach.

7. Proof of Correctness of ITRED

In this section we prove that ITRED results in a correct design if the Σ constraints are satisfied. We begin with a lemma. We say that a dependency graph is *feasible for ITRED* if α symbols can be inserted at inputs, and each *PE* receiving an α symbol will delay its own computation and simulate the computation of a neighboring *PE*.

LEMMA 7.1. A dependency graph modified according to the Σ constraints will be feasible for ITRED, and no extra *PE*s will be introduced.

Proof. Constraint 1 (there is an α cell on the border where data arrives) ensures that α symbols can be inserted in the input. Constraint 2 (delay edges are parallel to the projection vector) ensures that a *PE* will do its delayed computations later. Constraint 4 (every *PE* is the image of a computation cell) ensures that there are no extra *PE*s. Constraint 5 (there is a non-α computation cell in the same equitemporal surface as α^j that projects to a neighbor of PE^j) ensures that *PE*s neighbor does its normal computation at the same time that PE^j simulates it.

We can now prove our main result. Recall that a strategy for inserting α's is termed *alpha-successful* if it results in all *PE*s being tested at least once, and with correct timing.

THEOREM 7.2. A strategy for ITRED that obeys the Σ constraints is α-successful.

Proof. From lemma 7.1 we know that the modified dependency graph can be used by ITRED. Constraint 3 (every *PE* is the pre-image under projection of an α cell) implies that every *PE* can be tested. It remains to be shown that the timing is correct.

An α cell represents a delay (or null operation) in the modified dependency graph. Recall that from constraint 6 (the α-surface is unicutting) we know that all edges cross the α-surface in the same direction.

Let $i_1, i_2, \ldots, i_k$ be incoming data for one computation of a normal computation cell in the original, unmodified dependency graph. Suppose for a contradiction that after the α's are inserted and the computation graph modified, one of the data items, say i_j, arrives earlier than the other data. Then it was not delayed by an α cell, which contradicts the condition that the α surface is a cutting surface. If it arrives later than the other data items, it crossed the α surface more than once, which contradicts the fact that the dependency graph is acyclic and the α surface is unicutting. Thus the required data items arrive together at the correct time, which finishes the proof.

The proof can be extended easily to the second approach.

8. Diagonal Projection with Modified ITRED

In this section we give an example where a certain choice of a projection vector $\vec{p}$ results in a signal flow graph for which it appears impossible to apply the ITRED method without introducing extra *PE*s. We then show how to modify the ITRED method to handle this case, and how to modify the Σ constraints to reflect this modification. This example is meant to illustrate the flexibility of the approach, and suggest ideas for further applications.

The example is the minimum substring-distance discussed in Section 2. For simplicity, assume that strings S and P both have the same length n. Suppose now that given the dependency graph in figure 1, for some reason the designer chooses the projection vector $\vec{p}$ to be (1, 1), resulting in a diagonal projection. If now the α surface is chosen to be a row (column), α symbols will pass through only the right (left) half of the processors, violating constraint 3 and resulting in a design where not all the processors can be tested. It is clear that we must introduce α's into both rows and columns. Figure 13 shows such an α surface. This satisfies both contstraints 3 and 4: every *PE* is the image under projection of both of an α cell and a normal computation cell.

But now we run into a problem because constraint 1 is violated: there is no α cell on the border at which input data enters. We can in effect generate α symbols from inside the dependency graph by modifying the ITRED method as follows. Each data value that needs to be transmitted between two *PE*s will be in one of the two *states*: *normal*, or α'. If the user wants to test the *PE*s, an input data point is inserted in the α' state; otherwise, the input data is inserted in the normal state. Note that we do not insert special α symbols here. Whenever two data values that are both in the α' state meet at PE_i, that *PE* changes the state of the data values to normal, simulates the same operation as one of its neighbors, and sends α symbols on in accordance with the modified dependency graph. That is, PE_i behaves as if it had received an α symbol, and then

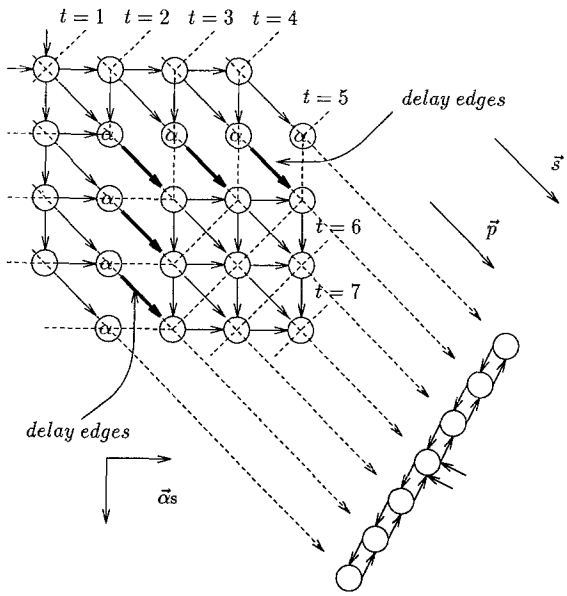

Fig. 13. Modified dependency graph for a bilateral linear array.

generates α symbols for the other processors. In our example, two data values in the α' state are inserted into row and column inputs, and meet in the middle *PE*. At the next clock interval, two α symbols are sent to the left and right neighboring *PE*s respectively (see figure 13).

There is no decrease in throughput with this scheduling. Also, as before, although some vertical and horizontal edges are in an equitemporal surface, the schedule is still systolic because these edges point to α cells. This modified strategy does result in one disadvantage: the last computation cell in the first row and first column cannot be tested. All the other computation cells can be tested, however.

In our example, although there is no α cell on the border at which data arrives, the union of the row and column of α cells forms a unicutting surface in the dependency graph. Thus, if the *PE*s introduce a delay when they receive an α symbol, the timing correctness will be preserved. To take this new method into account, we should change the Σ constraints by substituting the following for constraints 1 and 6:

1'. the union of α cells is a unicutting surface

The proofs of lemma 7.1 and theorem 7.2 then go through with obvious changes for this more general version of ITRED.

9. Conclusions

We proposed a new methodology for run-time error detection in systolic and wavefront arrays. The method is based on modifying the dependency graph to allow special symbols to enter the computation. These special symbols cause error checking to take place. We developed a set of constraints, the Σ constraints, and showed that they are sufficient to ensure that the timing is correct, that every *PE* can be tested, and that no extra *PE*s are introduced. Since the design choices are made at the abstract level of the dependency graph, the approach is very general, and can be applied to a wide variety of arrays in any dimension.

References

1. J.A. Abraham, et al., "Fault tolerance techniques for systolic arrays," *IEEE Computer*, 1987, pp. 65–74.
2. E.S. Manolakos and S.Y. Kung, "CORP—a new recovery procedure for VLSI processor arrays," *IEEE Symp. on the Engin. of Computer Based Medical Systems*, 1988.
3. E.S. Manolakos and S.Y. Kung, "Neighbor assisted recovery in VLSI processor arrays," *European Signal Processing Symposium, EUSIPCO '88*, North Holland, 1988.
4. C.-C. Wu, and T.-S. Wu, "Concurrent error correction in unidirectional linear arithmetic arrays," *Proc. Int. Symp. Fault-Tolerant Computing*, 1987, pp. 136–141.
5. R. Cosentino, "Concurrent error correction in systolic architectures," *IEEE Trans. on Computer-Aided Design*, vol. 7, 1988, pp. 117–125.
6. L. Shombert and D.P. Siewiorek, "Using redundancy for concurrent testing and repairing of systolic arrays," *Proc. Int. Symp. Fault-Tolerant Computing*, 1987, pp. 246-249.
7. Y.H. Choi, S.M. Han, and M. Malek, "Fault diagnosis of reconfigurable systolic arrays," *Proc. Int'l. Conf. Computer Design: VLSI in Computers*, 1984, pp. 451–455.
8. S.Y. Kung, *VLSI Array Processors*, Englewood Cliffs, NJ: Prentice Hall, 1988.
9. H.-H. Liu and K.-S. Fu, "VLSI arrays for minimum-distance classifications," *VLSI for Pattern Recognition and Image Processing*, (King-Sun Fu, ed.), New York: Springer-Verlag, 1984.
10. R.J. Lipton and D. Lopresti, "A systolic array for rapid string comparison," *1985 Chapel Hill Conference on Very Large Scale Integration*, (Henry Fuchs, ed.), Rockville, MD: Computer Science Press, 1985, pp. 363–376.
11. R.J. Lipton and D. Lopresti, "Comparing long strings on a short systolic array," *1986 International Workshop on Systolic Arrays*,, Oxford: University of Oxford, 1986.
12. G.M. Landau and U. Vishkin, "Introducing efficient parallelism into approximate string matching and a new serial algorithm," *ACM STOC*, 1986, pp. 220–230.
13. P.K. Lala, *Fault Tolerance and Fault Testable Hardware Design*, Englewood Cliffs, NJ: Prentice Hall, 1987.
14. J.F. Wakerly, *Error Detecting Codes, Self-checking Circuits and Applications*, New York: North Holland, 1978.

Edwin Hsing-Mean Sha received the B.S.E. degree in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 1986, and the M.A. degree and Ph.D. degree in computer science from Princeton University in 1990 and 1992. He is going to join the faculty of the Department of Computer Science and Engineering at the University of Notre Dame in the fall of 1992. His research interests include fault tolerant computing, testing, VLSI architectures, high-level synthesis in VLSI, and algorithms.

Kenneth Steiglitz received the B.E.E. (magna cum laude), M.E.E., and Eng.Sc.D. degrees from New York University, New York, NY, in 1959, 1960, and 1963, respectively.

Since September 1963 he has been at Princeton University, Princeton, NJ, where he is now Professor of Computer Science, teaching and conducting research on parallel architectures, signal processing, optimization algorithms, and cellular automata. He is the author of *Introduction to Discrete Systems* (New York: Wiley, 1974), and coauthor, with C.H. Papadimitriou, of *Combinatorial Optimization: Algorithms and Complexity* (Englewood Cliffs, NJ: Prentice Hall, 1982).

Dr. Steiglitz served two terms as a member of the IEEE Signal Processing Society's Administrative Committee, as chairman of their Technical Direction Committee, member of their VLSI Committee, their Digital Signal Processing Committee, and as their Awards Chairman. He is an Associate Editor of the journal *Networks*, and is a former Associate Editor of the *Journal of the Association for Computing Machinery*. A member of Eta Kappa Nu, Tau Beta Pi, and Sigma Xi, he was elected Fellow of the IEEE in 1981, received the Technical Achievement Award of the Signal Processing Society in 1981, their Society Award in 1986, and the IEEE Centennial Medal in 1984.