Dynamic Reconfiguration For Fault Tolerance For Critical, Real-Time Processor Arrays

M. D. Derk, School of Computer Science L. S. DeBrunner, School of Electrical Enginnering
University of Oklahoma, Norman, Oklahoma 73019
(405) 325-4852 FAX: (405) 325-7066

Abstract

Real-time digital signal processing for critical applications demands that rapid, successful reconfiguration techniques be employed to increase fault tolerance. To meet this need, we introduce and demonstrate a local area reconfiguration algorithm for a rectangular processor array that is very efficient, does not require a host processor, and will successfully reconfigure for a fault anywhere in the local area if there is an available spare. Further, if all the spares in a local area are used, areas can be combined in a software controlled process, preventing a system failure.

1. Introduction.

Digital signal processing algorithms, such as Discrete Fourier Transform, matrix inversion, and L-U decomposition, are frequently implemented using systolic or wavefront arrays. More general array structures may also be used for DSP applications. Fault tolerance in arrays of processors can be achieved by reconfiguration, a process by which spare processors are substituted for faulty ones [1-15]. Reconfiguration for yield is applied to correct manufacturing faults, while reconfiguration for reliability (dynamic reconfiguration) is applied to correct "on-the-fly" for processors that become faulty during operation.

Real-time applications, like speech processing and radar, require that dynamic reconfiguration must be as rapid as possible, bearing in mind that the processor array may have already been reconfigured for yield. Ideally, dynamic reconfiguration should be self-reconfiguration, not requiring a host processor. Also, for a critical application, or for an application environment for which repair is difficult or impossible, reconfiguration must be successful as long as there is an available spare.

The algorithm presented here meets these criteria. It is extremely efficient (O(N) time complexity), does not require a host processor for implementation, and can be applied to an array already reconfigured for yield by another method.

The algorithm is designed to work within a specified local area of any size, the only constraint being that there is a column of spares on the right edge. This area could be the entire array, and the algorithm will use any and all spares for any arrangement of faults in the area, as long as the number of faults is less than or equal to the number of spares. Other algorithms utilize the local area approach [5-7]; however, our algorithm is unique in the property that there is a companion algorithm than can combine vertically adjacent local areas, making spares in both areas available for faults in both areas.

The paper is organized as follows: After a presentation of definitions and assumptions, the reconfiguration algorithm is presented. Then, implementation and time complexity is discussed, after which the algorithm to combine local areas is presented. Finally, the algorithm is analyzed and compared, and conclusions are drawn.

2. Definition of terms and assumptions.

Processor state, as indicated by the letter "a", "b", "c", "d" or "s", indicates the state as it relates to the reconfiguration algorithm, and not to the state of the computations being performed by the processor. Each non-failed processor knows its own state, and failed processors have no state.

Compensation path is the term used for the series of logical coordinate changes that start at the faulty processor and end at the point that a spare processor is used. Compensation paths are found in "fault-stealing" reconfiguration techniques, such as presented in [2].

Fault Model. We shall assume that faults occur one at a time (within a local area), randomly and independently, to processors that are in use, and that the reconfiguration algorithm completes for one failure before the next testing cycle. We also assume that interconnections and switches are fault-free, and that spares are not faulty before they are used, though they may become faulty after they are included in the logical array.

The reconfiguration algorithm does not require the presence of a host processor for implementation, but a host processor is needed to implement the algorithm to combine local areas. Local areas may be any size, but must be rectangular with one column of spares on the right, and must be vertically adjacent to be combined.

1058-6393/95 $4.00 © 1995 IEEE

3. Local area reconfiguration algorithm.

This algorithm has two phases. Phase I is always used, as it performs the actual reconfiguration along with some state changes. Phase II is invoked in only some cases, and completes the necessary state changes. In the figures for the following example, boxes indicate processors and lines indicate activated communication lines. An "X" in a box indicates a failed processor.

3.1 Example.

This example takes a 4 x 4 local area through several faults that occur dynamically, and the response of the reconfiguration algorithm to each of them. Each local area begins as a fault-free rectangular grid, as shown in Figure 1. There is one column of spares on the right. Initially, each non-spare processor is assigned a state of "a". The row and column indicators are logical coordinates, and may or may not be physical coordinates also, since the array may have been previously reconfigured for yield.

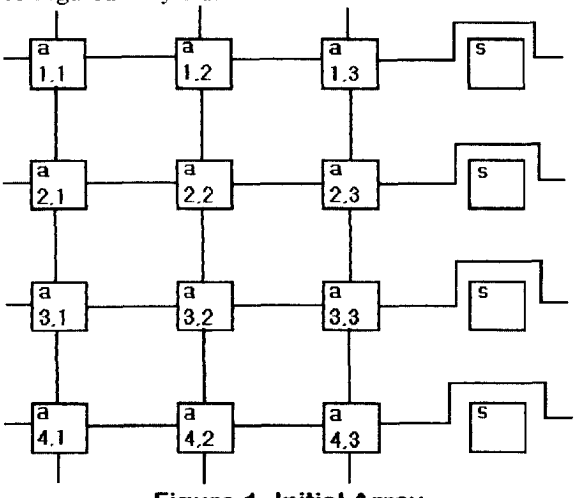

Figure 1. Initial Array.

Assume that the processor at logical coordinate 2,3 fails. Since the processors in row 2 have a state of "a", indicating an initial state, then the closest spare is the one at the end of row 2. Reconfiguration occurs as shown by Figure 2, making use of the spare. Also, the state of each non-faulty processor in row 2 has been changed to "b", indicating that the spare in that row has been used. Figure 3 shows the reconfiguration that occurs should the processor in coordinates 3,1 fail, using the spare at the end of row 3, and changing the states of the processors in row 3 to "b".

If the processor at logical coordinate 3,2 should fail at this point, there is no spare in row 3 to use. Therefore, the processors of logical column 2 are shifted down one logical position in rows 3 and 4. Row 4 processors are

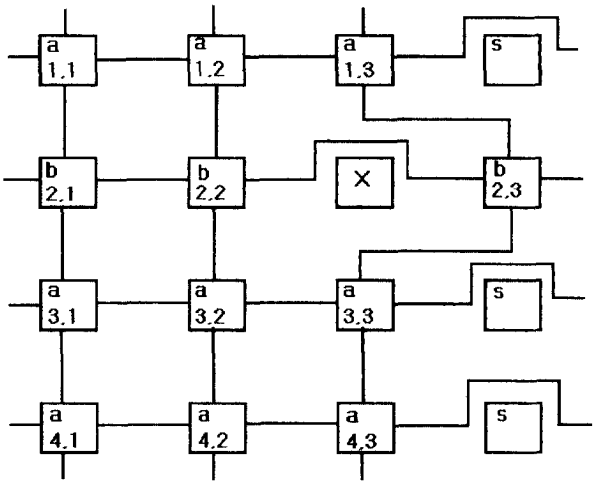

Figure 2. Array after one fault.

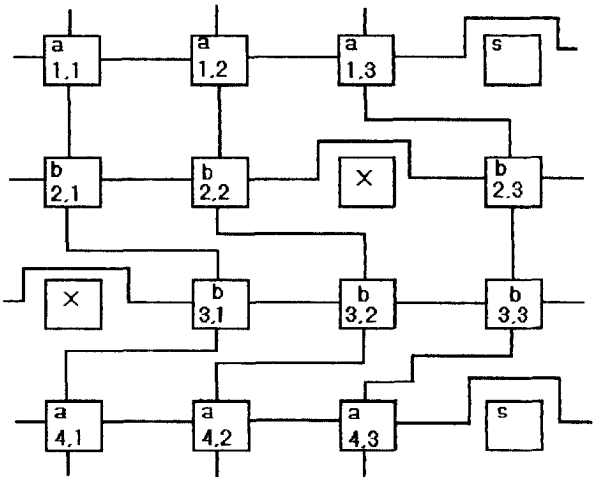

Figure 3. Array after two faults.

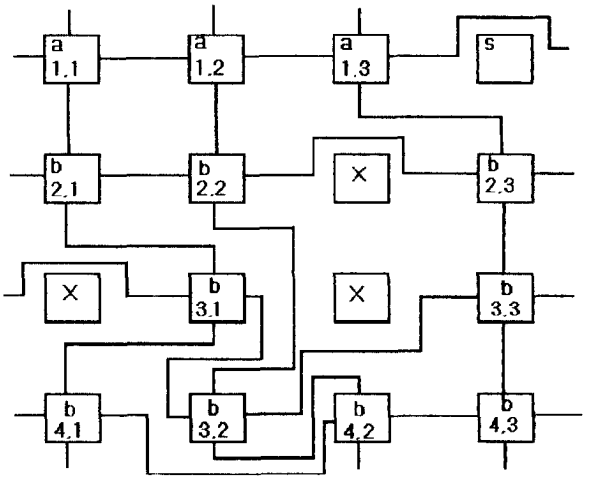

Figure 4. Array after Phase I, third fault.

shifted to the right one logical position between column 2 and the spare, freeing up a processor for use by column 2.The result is in Figure 4. Note that the algorithm has

also changed the states of the processor in logical row 4 to "b", indicating that the spare in row 4 has been used.

Phase II is invoked if the processors in the bottom row have a state of "b" or "c". Since the processors in the lowest row now have a state of "b", this invokes Phase II, which alters the states of the processors in rows 2 through 4 to "c", as shown in Figure 5, indicating that all spares in row twoand the rows below have been used.

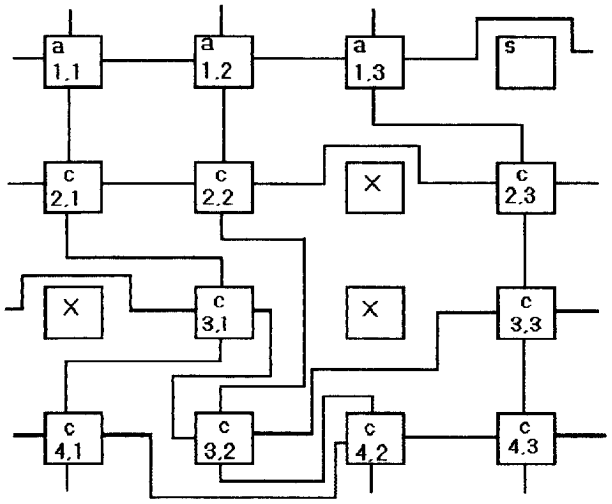

Figure 5. Array after Phase II, third fault.

We now suppose that a failure occurs at coordinates 2,1. Column one is shifted up one logical row. Row one is shifted to the right by one place. Figure 6 shows the state of the array after Phase I. Since the bottom row has a state of "c", Phase II is invoked, which extends the state of "c" to every non-failed processor in the local area. Since the processors in the top row have a state of "c", we know that all spares in the local area have been used. Phase II therefore sweeps each column again, changing the states of all non-failed processors to "d", as shown in Figure 7.

3.2 Phase I.

The first phase is always executed when a fault is detected.

State "a": If a processor with a state of "a" fails, the processors in that row between the failed processor and the spare on the right edge of that row, inclusive, change their logical coordinates so that each one substitutes for the processor to the left. That is, the processor with logical coordinates i,j assumes new logical coordinates i,j-1. Also, all processors in that logical row change their state to "b". The failed processor is excluded from the logical row, and has no state.

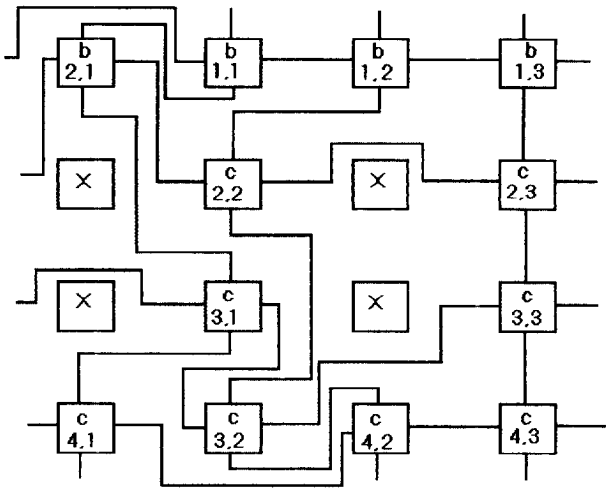

Figure 6. Array after Phase I, fourth fault.

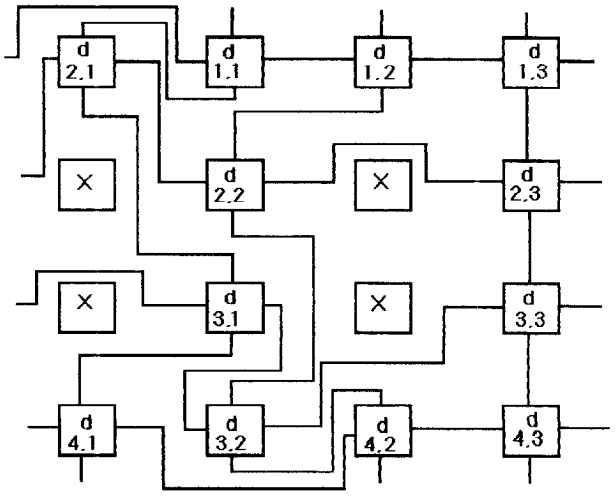

Figure 7. Array after Phase II, fourth fault.

State "b": If a processor with a state of "b" fails, the processors in its logical column between the failed processor and the first processor in that logical column BELOW it with a state of "a" change their logical coordinates so that a processor with logical coordinates i,j has new coordinates of i-1,j. This means they are effectively in one logical row above the one in which they were previously. Therefore, if the first processor in the logical column below the fault with a state of "a" has logical coordinates of x,y, then, that processor will assume new logical coordinates of x-1,y. The processors on logical row "x" from logical column y+1 rightward to the spare change their logical coordinates from x,j to x,j-1, effectively shifting them one place to the right to make room for the downward shifted column. Then, all the processors in logical row "x" change their states to "b".

State "c": If a processor with a state of "c" fails, then a procedure analogous to the one described for state "b" takes place, except that the column is shifted up rather

than down, and all the processors in the logical row that shifted to the right change their states to "c".

State "d": If a processor with a state of "d" fails, then there are no spares available to use to compensate. We must combine local areas or declare a fatal system failure.

3.3 Phase II.

The second phase is executed only if processors on the bottom row of a local area have state "b" or "c".

If a processor is on the bottom row of a local area and its state is "b" or "c", it first changes its own state to "c" (if it is not already in state "c"). Then, it initiates a signal that propagates up through its own logical column. This signal, when received by a processor with a state of "b", results in that processor changing its state to "c" and propagating the signal. If it is received by a processor with a state of "a", there is no action, and the signal does not propagate further. This ensures that if a processor has state "c", then all spares in that row and all rows below have been used and all the available spares are in rows above.

If a processor is on the top row of a local area, and it changes its state to "c", then all spares in the top row and below have been used, i.e., all the spares in the local area have been used. Therefore, it changes its own state to "d", and initiates a signal down its own logical column to the bottom row, which results in all non-faulty processors changing their states to "d", indicating no available spares

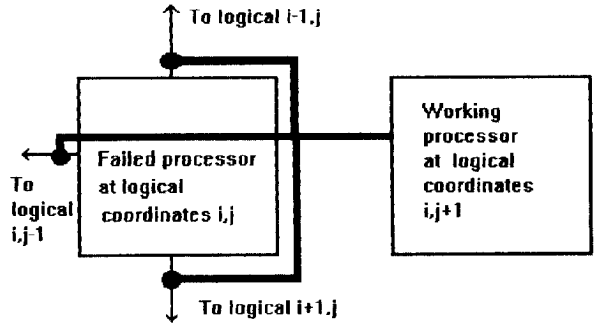

Figure 8. Implementation.

4. Implementation.

A processor failure is detected by its nearest logical horizontal neighbor. Testing by a logical neighbor rather than a physical neighbor guarantees that this neighbor will be a working processor. The fault detection and reconfiguration initiation must be performed by a horizontal neighbor rather than a vertical neighbor, because the horizontal neighbor will have the same state as the processor that failed. This state must be correct for the proper reconfiguration signals to be sent.

To start reconfiguration, the processor that detects the fault sends a signal through the normal communication line that is active because it is a logical neighbor, through bypass circuitry, to the other logical neighbors, as shown by the heavy lines in Figure 8.

5. Time complexity of algorithm.

Within an N x M (N rows and M columns) local area, the reconfiguration signal travels through at most only one logical row and one logical column, which is N+M processors. Each processor signaled executes code only for its particular state and type of signal received. In each case, the signal is first propagated, so most other activity is overlapped with signal propagation. Thus, the total time complexity for the production is M + N + K, where K is a constant amount. This initial production may trigger a second wave of state changes, depending on the fault pattern. However, each logical column performs these state changes in parallel. For the first through N-1st processor failure, the state changes propagate upwards from the bottom, changing fewer than N rows of processors. After the Nth failure a second wave propagates upwards through all N rows, then down through all N rows again. Therefore, the most steps it will take is 2N, making the maximum total steps 3N+M plus a constant, which is O(N+M).

6. Global algorithm to combine local areas.

All rows and columns referred to in this description are logical rows and columns.

First, the algorithm scans the leftmost column of the proposed new, larger local area. If there is at least one processor in this column with a state of "a", then there is at least one unused spare, and area combination will be productive. Otherwise, the algorithm halts.

To combine the areas, the algorithm clears all top and bottom row indicators for rows that will be interior rows of the new area. Then, it scans the states of all processors, changing the states of all processors with states of "c" or "d" to "b". This is done for each logical column in parallel. After that, it tests the state of the processor in the leftmost column of the bottom row. If it is "b", it initiates execution of Phase II of the reconfiguration algorithm so that the processors in the rows below the lowest spare all have states of "c".

Two or more local areas that are adjacent vertically may be combined as needed. Some areas can be combined while others are not; the entire array need not be affected.

7. Comparisons to other algorithms.

This algorithm can make use of every spare in the local area for any fault in the local area, until the number of faults equals the number of spares. The algorithm is efficient, with a time complexity of O(M+N), where M and N are the dimensions of the local area of the array. If a local area is square, of dimensions N X N, then the time complexity is O(N). This reconfiguration algorithm is the only one in the literature to combine high efficiency with high (nearly 100% or 100%) spare utilization, as shown in the table below. (It must be noted here that Kim and Efe [8] have recently published an O(1) algorithm, but it requires a parallel host machine for implementation, and is designed for static, not dynamic reconfiguration.)

Algorithm	Time Complexity	High Spare Utilization?
Varvarigou, et al [6]	$O(N^2)$*	Yes
FUSS [7]	$O(N^2)$	Yes
Bruck, et al [8]	$O(N^2)$	Yes
Chen, et al [9]	$O(N^2)$	Yes
Dutt and Hayes [10]	$O(N^2)$	Yes
Fault-stealing [1]	O(N)	No
MORA [11]	O(NlogN)	No
Spanning Tree [12]	O(NloglogN)	No

*For the number of faults approximating the number of rows.

Table 1. Comparison to other algorithms.

8. Conclusions.

We have described an efficient algorithm for reconfiguration for reliability for rectangular processor arrays. It will use every spare in a local area, and, should every spare be used within a particular area, that area can be combined with a neighboring area to allow access to additional spares.

This algorithm involves only a limited number of processors, making it ideally suited for dynamic reconfiguration. Since the algorithm works with logical rather than physical addresses, it can be implemented on an array that has already been reconfigured for yield during production, by any reconfiguration method.

Several features of this reconfiguration method make it possible for the processor array to self-reconfigure rather than relying on a host processor. By using processor reconfiguration states, each processor knows the nature of the reconfiguration that has already occurred in its immediate locality, and therefore the "direction" of the nearest spare. Communications for implementation are limited to those processors in the compensation path, and the algorithm is small enough to be encoded in the array processors. Also, using logical rows and columns, reconfiguration messages travel along the data paths that have been established by previous reconfigurations, either static or dynamic, which means that the array processors use the same ports for application communications and reconfiguration communications.

References

[1] Sami, M. and Stefanelli, R. Reconfigurable architectures of VLSI processing arrays. Proceedings of the IEEE, vol. 74, no. 5, May 1986, pp. 712-722.

[2] Singh, Adit D. Interstitial redundancy: an area efficient fault tolerance scheme for large area VLSI processor arrays. IEEE Transactions on Computers, vol. 37, no. 11, Nov. 1988, pp. 1398-1410.

[3] Wang, M., Culter, M. and Su, S.Y.H. Reconfiguration of VLSI/WSI mesh array processors with two-level redundancy. IEEE Transactions on Computers, vol. 38, no. 4, April 1989, pp. 547-554.

[4] Tsuda, Nobuo. Hierarchical redundancy for orthogonal arrays. Proceedings. International Conference on Wafer Scale Integration, Jan. 22-24, 1992, pp. 220-229.

[5] Kim, Jung H. and Efe, Kemal. A parallel reconfiguration algorithm for WSI/VLSI processor arrays. Microprocessors and Microsystems, vol. 17, no. 6, July-Aug. 1993, pp. 353-360.

[6] Varvarigou, Theodora A., Roychowdhury, Vwani P., and Kailath, Thomas. A polynomial time algorithm for reconfiguring multiple-track models. IEEE Transactions on Computers, vol. 42, no. 4, April 1993, pp. 385-394.

[7] Chean, Mengly and Fortes, Jose A.B. The full-use-of-suitable-spares (FUSS) Approach to hardware reconfiguration for fault-tolerant processor arrays. IEEE Transactions on Computers, vol. 39, no. 4, April 1990, pp. 564-571.

[8] Bruck, Jehoshua, Cypher, Robert, and Ho, Ching-Tien. Fault-tolerant meshes and hypercubes with minimal numbers of spares. IEEE Transactions on Computers, vol. 42, no. 9, September 1993, pp. 1089-1103.

[9] Chen, Chang, Feng, An, Kikuno, Tohru and Torii, Koji. Reconfiguration algorithm for fault-tolerant arrays with minimum number of dangerous processors. 21st International Symposium on Fault-Tolerant Computing, June, 1991, pp. 452-459.

[10] Dutt, Shantanu and Hayes, John P. Some practical issues in the design of fault-tolerant multiprocessors. 21st International Symposium on Fault Tolerant Computing, 1991, pp. 292-299.

[11] Lombardi, F., Sami, M.G., and Stefanelli, R. Reconfiguration of VLSI arrays: an index mapping approach. Proceedings, First International Conference on Computer Technology, Systems, and Applications, 1987, pp. 60-65.

[12] Lombardi, Fabrizio and Sciuto, Donatella. Reconfiguration in WSI arrays using minimum spanning trees. Proceedings, First International Conference on Computer Technology, Systems, and Applications, 1987, pp. 547-550.