CONCURRENT ERROR DIAGNOSIS IN MESH ARRAY
ARCHITECTURES BASED ON OVERLAPPING H-PROCESSES

E. S. Manolakos D. Dakhil M. Vai

Electrical and Computer Engineering Department
409 Dana Research Building
Northeastern University
Boston, Massachusetts 02115

Abstract

Unlike other methods for Concurrent Error Detection and Location (CED), the one proposed here is not application specific and does not require fault free comparators and custom VLSI design for the processing cells. It is suitable for any algorithm that can be decomposed in block operations of the format \[(a \circ \circ \circ b) \circ \circ \circ (c \circ \circ \circ d)\], where \(a, b, c, d\) are arbitrary operands and \(\circ \circ \circ \circ \circ \circ \) dyadic operators. The process of computing such an operation in a distributed and redundant way on an H-tree shaped sub-array is called an H-process. Many H-processes can overlap providing a general purpose mechanism for run-time fault tolerance in data driven mesh array architectures. Errors can be detected during normal operation. Suspected erroneous results can be masked while location is attempted. There is no need for retries. Diagnosis is achieved "on the fly" without graceful degradation, upon detection.

Introduction-Motivation

During the last decade substantial research effort has been focused on the development of run-time fault tolerant processor arrays. One of their major capabilities is to perform Concurrent Error Detection (CED) along with normal operation, as the first step towards error recovery. The CED techniques proposed in the literature roughly fall into two categories: In the former, the main objective is to locate the error in the Processing Element (PE) where it occurred, by using the concept of Duplication With Comparison (DWC). Either structural [1, 2] or time redundancy [3, 2] may be used for this purpose. In any case, the comparison between multiple versions of the same result is assumed to be fault-free. This approach, although it delivers high coverage and low latency diagnosis, it imposes at least 100% hardware or time overheads and the need for Totally Self Checking (TSC) checkers. The latter, known as Algorithm Based Fault Tolerance (ABFT) [4] attempts CED in a distributed fashion. The detection latency is higher and the data encoding scheme may vary for every linear algebra algorithm often used in digital signal processing applications. Again special purpose design is necessary for the PEs.

In this paper we propose a non-algorithm specific, general purpose scheme for distributed CED, using time redundancy. There is no need for hardware duplication, comparison
is performed in the ALU and need not be perfect. As long as the architectural framework is preserved, many different implementations can be envisioned; from low grain massively parallel Wafer Scale Integration (WSI) to medium grain off-the-self microprocessors based.

The Array Model and Representation

The nucleus is an asynchronous data-driven 4 \times 4 mesh array \cite{5}, where each Processing Element (PE) is connected to its four immediate neighbors by data buses, Figure 1(a). For example, PE_i,j can pass a token to PE_{i+1,j} in one step. Only a single control line, called opinion-exchange, runs diagonally. Each PE is a general purpose cell able to perform the basic arithmetic and logical operations. The generic structure of a PE, shown in Figure 1(b), consists of the Data Bus Interface Unit (DBI), the ALU, the Control Unit, Registers and the Error Handling Logic (EHL).

In wavefront processing it is the sequence of events that guarantees correct operation, not their exact timing \cite{5}. Since PEs are locally synchronized by passing data tokens and not by a global clock reference, it is impossible to "freeze" the array activity and depict it by "snapshots". The array Activities Map (AM) captures the order of steps taken by every PE in a self-explanatory and concise way. During a step a PE may participate in the following possibly concurrent types of activities:

- **A Computation**: If at a given step, PE_i,j is idle due to data unavailability, the corresponding box in the AM is drawn empty. Otherwise it is shaded and labeled by the executed opcode. Only one computation is allowed per step.

- **A Communication**: If PE_i,j sends a token to a near-neighbor, say PE_{i+1,j}, a vertical arrow is drawn from PE_i,j to PE_{i+1,j}. Arrows are labeled by the names of the tokens transmitted, whenever necessary. A link may be used for communication only once during a step.
Example: Suppose that the operation to be performed in a distributed fashion in some part of the array is \( [a \cdot (b + c) \cdot d] \), where \( a, b, c, d \) are arbitrary operands. Figure 2 shows the array Activities Map explaining how PE22, PE32 and PE42 cooperate to produce the result, given that the data are already loaded. Three steps are needed and there is no concurrency between computation and communication activities in this case.

The H-process

A very instrumental notion for concurrent error detection is that of the H-process. It corresponds to a set of 7 allocated PEs organized as an H-tree and assigned to perform an H-operation (hop) in a distributed and redundant fashion. An H-process has the following structural characteristics (Figure 3):

H-CENTER(HC): It is the root PE of the H-tree (also called judge PE.)

ORIENTATION: It can be either Vertical (Figure 3.a) or Horizontal (Figure 3.b). A Vertical H-process can be partitioned into the following subsets: Left-Wing (VLW), H-Center (HC) and Right-Wing (VRW). Every wing is composed of 3 PEs, the middle one is called Parent PE and the others Children PEs. Similarly, a Horizontal H-process is partitioned to: Upper-Wing (HUW), H-Center (HC) and Lower-Wing (HLW). Therefore if PEij is the HC for a vertical H-process (VH) it is true that: 

\[ VH = VLW \cup HC \cup VRW, \]

where \( VLW = \{ PE_{i-1,j-1}, PE_{i-1,j}, PE_{i+1,j-1} \} \) and \( VRW = \{ PE_{i-1,j+1}, PE_{i+1,j-1}, PE_{i+1,j+1} \} \). A horizontal H-process is defined similarly.
H-Process allocation Region: Is the 3 x 3 sub-array where an H-process is placed.

An H-process is employed to perform in a distributed and redundant way an H-operation \((\text{hop})\), that functionally corresponds to the evaluation of a numerical/logical expression with the H-format: \(\{(a \odot b) \odot (c \odot d)\}\), where \(a, b, c\) and \(d\) are arbitrary operands and \(\odot\) any usual dyadic operator; e.g. \(\{(a + b) \cdot (c + d)\}\), \(\{\frac{(a-b)}{(c-d)}\}\). Notice that the dyadic operation \((a \cdot b)\) can be expressed in H-format as \(\{(a \cdot b) + (z \cdot z)\}\) where \(z = 0\) (the trivial operand for multiplication). The H-Center is called the judge, because it decides if the two versions of the same hop performed in the two wings produce identical results. Figure 4 shows the array activities during the execution of a vertical H-process centered at PE33, assigned to perform the hop: \(\{(a \cdot b) + (c \cdot d)\}\). Wing VRW (VLW) computes version hop, \(\{\text{hop}_1\}\) (steps 1, 2 and 3). The two results, \(\{\text{hop}\}\) and \(\{\text{hop}_1\}\), are passed to the judge (PE33) at step 4 and compared at step 5. Comparison is implemented by subtraction in the ALU. It is assumed that the operands are already loaded into all PEs involved.

Overlapping H-processes

Since PEs utilization is less than 100% free PEs can be engaged in another H-process. Therefore, the Allocation Regions of different H-processes working concurrently on different data sets (and possibly performing different hops) may overlap. Figure 5 shows the placement of 4 H-processes (two vertical and two horizontal) into a 4 x 4 array. As it will become clear later on this placement is not the only appropriate one. The two
vertical processes centered at PE_{22} and PE_{33} are called Yellow and Blue respectively. Similarly, the two horizontal processes centered at PE_{23} and PE_{32} are called the Green and Red. Assuming that all operands are available where needed, Figure 6 shows how the four processes are executed concurrently in the 4 × 4 array. Different shadows are used to represent colors, as explained at the bottom of the Figure. Notice that the centers, or judges, of the 4 H-processes are placed in the middle PEs. Every judge PE is associated with the color of the process that it rules, e.g. PE_{22} is the yellow judge.

The life span of four overlapping concurrent H-processes corresponds to a phase. It consists of 3 stages; each stage consists of two consecutive steps (one computation and one communication.) Four hop operations identified by color (hop_r, hop_g, hop_b, and hop_y) are concurrently performed. It takes 3 steps to produce the results and another 3 steps to verify them. The four hop operations, hop_c = [(a, op_r b_c) op_r (c, op_r d_c)], where c is the color index, c ∈ {r, y, g, b}, are performed as follows, assuming that the data are preloaded.

THE OVERLAPPING-HS STAGE:

Step 1 (Computation): In every wing of a process c, one child PE computes partial result A_c and the other partial result B_c, where A_c = (a_c op_r b_c) and B_c = (c, op_r d_c), respectively.

Step 2 (Communication): Children PEs deliver their results to their parents. (Due to the overlapping, a PE may act as child for one process and parent or judge for another, in different steps).

THE FAN STAGE: Now the array activity is contracted to a region that resembles a rotating fan. For the active PEs (shaded) the colors are rotated clock-wise (see bottom of Figure 6).

Step 3 (Computation): Parent PEs in all processes compute, hop_c = A_c op_r B_c, according to their color.

Step 4 (Communication): Parent PEs in all processes deliver their results to their judges.
Figure 6: Concurrent execution of 4 overlapping $H$-processes in a $4 \times 4$ array (phase k).
The Decision Stage: The array activity contracts further and is confined to the four judges.

Step 5 (Comparison): This is a computation step because comparison is done in the ALU. Every judge PE compares the two copies of hop, received at step 4. If the copies are not identical, the judge PE sets a one-bit Error Flag (EF, ← 1). Otherwise EF, is cleared (EF, ← 0).

Step 6 (Opinion Exchange): Every judge PE sends its opinion (Error Flag value) regarding the comparison outcome to the other three judges via the opinion-exchange control lines.

Array Fault Tolerance

The Fault Model

A technology independent functional fault model is adopted because the traditional stuck-at fault model is not sufficient to describe many types of failures especially in CMOS VLSI technology. The underlying assumptions are summarized below:

1. No new PE error occurs in the array before the previous one is detected and located.
2. A faulty PE produces erroneous results in all computations it is involved, until it is detected and located.
3. Internal and external buses are assumed to transmit data tokens reliably.
4. The hard-core of the PE consists of the Opinion-Exchange lines and the Error Handling Logic (EHL).

The adopted assumptions are less strict than those in other proposed schemes such as, RES0 [3], TTCDD [6]. For instance, a judge is allowed to fail while it compares. Comparison is performed in the ALU thus eliminating the need for Totally-Self-Checking checkers and custom VLSI design.

Error Detection, Confinement and Masking

As explained before, every judge will know the set of error flags of all other judges,\( EF^k = \{EF^2_k, EF^3_k, EF^4_k, EF^5_k\} \), after the opinion exchange (step 6) at the end of phase \( k \) (\( k = 1, 2, \ldots, M \) and \( M \) is the number of phases needed to complete a given algorithm.) The following discussion is based on Figure 6 and Table 1. Since the four H-processes partially overlap, if PE2 starts producing wrong results during phase \( k \), only the processes that employ it after it becomes erroneous will be affected. For example, if \( EF^4 = \{0, 1, 0, 0\} \) the red, green and blue judges know that their PEs didn't experience any problem, however since \( EF^4 \) is set, the yellow judge (PE24) knows that an error had occurred in one of the yellow H-process PEs during phase \( k \). Thus, the yellow judge has to do something to mask the incorrect results. Assuming that no error occurred during phase \((k - 1)\) (\( EF^{k-1} = \{0, 0, 0, 0\} \)), the yellow judge recognizes three possibilities at the end of phase \( k \) (see row 2 in Table 1):
Table 1: The Error Flags at the end of phase $k$, $\{EF_k\}$.

<table>
<thead>
<tr>
<th>Possible EFs in phase $(k)$</th>
<th>Connected Effs</th>
<th>Error occurred at</th>
<th>Possible Wounded Wraps (Color-Wrap)</th>
<th>Deliver From</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Step 1</td>
<td>R-FLW</td>
<td>R-FLW</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>PE4, PE8,</td>
<td>Step 2</td>
<td>R-FLW</td>
<td>R-FLW</td>
</tr>
<tr>
<td></td>
<td>PE6, PE22</td>
<td>Step 5</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>PE11, PE13,</td>
<td>Step 1</td>
<td>Y-VLW</td>
<td>Y-VLW</td>
</tr>
<tr>
<td></td>
<td>PE1 or PE2</td>
<td>Step 2</td>
<td>Y-VLW</td>
<td>Y-VLW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Step 5</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>PE14, PE21,</td>
<td>Step 1</td>
<td>G-HUW</td>
<td>G-HUW</td>
</tr>
<tr>
<td></td>
<td>PE1 or PE2</td>
<td>Step 3</td>
<td>G-HUW</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Step 5</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>PE44, PE46,</td>
<td>Step 1</td>
<td>B-VLW</td>
<td>B-VLW</td>
</tr>
<tr>
<td></td>
<td>PE4 or PE32</td>
<td>Step 2</td>
<td>B-VLW</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Step 3</td>
<td>B-VLW</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>PE13 or PE22</td>
<td>Step 1</td>
<td>Y-VLW, G-HLW or Y-VLW</td>
<td>Y-VLW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Step 3</td>
<td>Y-VLW</td>
<td>Y-VLW</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>PE1 or PE2</td>
<td>Step 1</td>
<td>B-VLW, Y-VLW or B-VLW</td>
<td>B-VLW</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>PE4 or PE22</td>
<td>Step 1</td>
<td>B-VLW, R-HLW or B-VLW</td>
<td>B-VLW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Step 3</td>
<td>B-VLW</td>
<td></td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>PE42 or PE22</td>
<td>Step 1</td>
<td>B-VLW, R-HLW or B-VLW</td>
<td>B-VLW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Step 3</td>
<td>B-VLW</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>No error has occurred</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7: The orientation of the 4 overlapping H-processes, in phase $k + 1$. 
1. One of the children PEs (PE_{11} or PE_{31}) in the Yellow Vertical Left Wing (Y-V LW) failed (step 1) and hop_{1} is corrupted. Therefore, the yellow judge discards hop_{1} and delivers hop_{1} (the version computed at the Y-VRW). A wounded wing (Table 1) is one with corrupted results. The other judges deliver results from the default (D) wing that is unspecified for every orientation.

2. The parent PE of Y-V LW (PE_{21}) failed (step 3). Again, hop_{1} is corrupted and results should be delivered from the Y-VRW.

3. The yellow judge (PE_{22}) failed (step 5), while comparing the results received from both wings. In this case, neither hop_{1} nor hop_{2} is corrupted (one PE is allowed to fail at a time) and delivering results from the Y-VRW is again safe.

For a single error that occurred during phase k Table 1 summarizes all possible valid combinations of EF_{k}, the suspected PEs and the deliver from action taken by the judge(s) of affected process(es) to mask the incorrect results. The discussion so far proves that:

**Conclusion:** Under the stated fault model, the overlapping H-processes mechanism is capable of detecting, confining and masking any single error in any PE of the 4 x 4 array.

**Error Location**

The capability of confining an error in one wing is not sufficient if reconfiguration is to take place upon error detection. Location of an erroneous PE can always be achieved by alternating the orientation of all H-processes (vertical/horizontal) between successive phases, while keeping their centers fixed. Figure 7 shows the arrangement of the 4 H-processes for phase k + 1, assuming that Figure 5 corresponds to the same thing for phase k. The original orientation (for k = 1) can be either one, as long as it is known. From now on it is assumed, w.l.o.g. that k is odd.

One phase is enough to detect and mask but not to locate an erroneous PE. To achieve location, the error flags of two consecutive phases are needed. Therefore, every judge has to memorize EF_{k} until the end of phase k + 1, then EF_{k} gets replaced by EF_{k+1} and so on. Let's assume for the moment that a PE fails permanently during phase k. Table 2 provides all possible combinations of EF_{k} and EF_{k+1} based on the fault model adopted. If say EF_{k} = \{0,1,0,0\} the set of suspected PEs is \{PE_{11}, PE_{21}, PE_{31}, PE_{22}\} (the Y-V LW and the Y judge). EF_{k+1} will be used to disambiguate the situation and locate the faulty PE, while maintaining the masking capability. Since the array integrity is not compromised by the presence of a single permanently erroneous PE, phase k + 1 does not need to be a retry but can correspond to a new set of overlapping H-processes. Error location is achieved without graceful degradation at the end of phase k + 1. Actually the following cases are possible (see row 2 in Table 2):

1. EF_{k+1} = \{0,1,0,0\}; since the error is detected by the yellow judge only, the faulty PE is employed only by the Yellow process in phase k + 1. Therefore it should be PE_{22} (see figure 7).

2. EF_{k+1} = \{1,1,0,0\}; since the error is detected by two judges the faulty PE is employed by both the Red and the Yellow processes. Therefore PE_{22} is uniquely
Table 2: The Error Flags for two consequent phases $k$ and $k+1$.
identified as the faulty one. The red judge delivered results from the default wing at the end of phase $k$, because the red process engages $PE_{11}$ only in phase $k + 1$.

3. $EF^{k+1} = \{1, 0, 0, 0\}$; since the error is detected by the red judge only, the faulty PE is employed only by the red process in phase $k + 1$. Therefore it should be $PE_{21}$.

4. $EF^{k+1} = \{0, 1, 1, l\}$; since the error is detected by three judges the faulty PE should be the yellow judge $PE_{22}$, because it is the only one in the suspected set that is used by three processes during phase $k + 1$. It delivers results from the default wing since none of the yellow wings is corrupted.

Although, the overlapping H-processes technique has the capability to deal with some transient errors, it imposes restrictions on their duration and the timing of their disappearance. Transients that occur in phase $k$ and last until the end of phase $k + 1$ can be handled. However short transients that vanish in the middle of either phase although detected they are not necessarily masked. Reconfiguration should be initiated upon location, to prevent corruption from lengthy transients that may disappear in the middle of phase $k + l, l > 1$. For example, a transient error that lasts only for step 1 of phase $k$ in $PE_{13}$ will generate the flag combination, $EF_k = \{0, 1, 0, 0\}$. This set will be interpreted by the yellow judge as if a permanent error occurred in the yellow left wing (Y-V LW). Therefore the result delivered from the Y-VRW, where $PE_{13}$ is residing, will be considered delivered, although $Y$ is corrupted by the short transient.

An Example—Matrix Multiplication

The product of an $m \times l$ matrix $A$ by an $l \times n$ matrix $B$ is the $m \times n$ matrix $C$ whose elements are given by: $c_{ij} = \sum_{l=1}^{l} a_{il} \cdot b_{lj}$, $i=1,2,\ldots,m; j=1,2,\ldots,n$.

Example: For the special case $m = 4, l = 4, n = 2$, the problem can be decomposed in 16 hops (4 per color). A token has two data fields and two opcode fields. Data fields contain the first and second operands consumed by a child PE when the first opcode is fired. The second opcode is delivered to the parent PE along with the child PE's results. A parent PE receives two copies of the second opcode, it uses one and discards the other (if not identical a link failure is detected). The opcode field is redundant in this example, because cells repeat the same operation(s) in a regular fashion. However, it is needed in general when a process is assigned a different type of hop in every phase, like in the LU decomposition (not shown here due to lack of space). A color-token notation is used to show data distribution; e.g. since elements $a_{11}$ and $b_{11}$ are processed by a yellow process, they are grouped as yellow token $Y(a_{11}, b_{11})$. The same notation applies to the outputs of the hops; e.g. $Y_{o1} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21}$ denotes the first result delivered by the yellow judge. Hence the $C$ matrix elements are produced as follows:

\[
\begin{align*}
c_{11} &= (a_{11} \cdot b_{11} + a_{12} \cdot b_{21}) + (a_{13} \cdot b_{31} + a_{14} \cdot b_{41}) = [Y_{o1}] + [Y_{o2}] \\
c_{12} &= (a_{11} \cdot b_{12} + a_{12} \cdot b_{22}) + (a_{13} \cdot b_{32} + a_{14} \cdot b_{42}) = [Y_{o3}] + [Y_{o4}] \\
c_{21} &= (a_{21} \cdot b_{11} + a_{22} \cdot b_{21}) + (a_{23} \cdot b_{31} + a_{24} \cdot b_{41}) = [G_{o1}] + [G_{o2}] \\
c_{22} &= (a_{21} \cdot b_{12} + a_{22} \cdot b_{22}) + (a_{23} \cdot b_{32} + a_{24} \cdot b_{42}) = [G_{o3}] + [G_{o4}] \\
c_{31} &= (a_{31} \cdot b_{11} + a_{32} \cdot b_{21}) + (a_{33} \cdot b_{31} + a_{34} \cdot b_{41}) = [R_{o1}] + [R_{o2}] \\
c_{32} &= (a_{31} \cdot b_{12} + a_{32} \cdot b_{22}) + (a_{33} \cdot b_{32} + a_{34} \cdot b_{42}) = [R_{o3}] + [R_{o4}] \\
c_{41} &= (a_{41} \cdot b_{11} + a_{42} \cdot b_{21}) + (a_{43} \cdot b_{31} + a_{44} \cdot b_{41}) = [B_{o1}] + [B_{o2}] \\
c_{42} &= (a_{41} \cdot b_{12} + a_{42} \cdot b_{22}) + (a_{43} \cdot b_{32} + a_{44} \cdot b_{42}) = [B_{o3}] + [B_{o4}] \\
\end{align*}
\]
Figure 8 shows how the colored tokens are fed from the top (bottom) side of the array.

\[ c_{22} = [a_{21} \cdot b_{22} + a_{22} \cdot b_{22}] + [a_{23} \cdot b_{22} + a_{24} \cdot b_{22}] = [G_{23}] + [G_{24}] \]
\[ c_{33} = [a_{31} \cdot b_{32} + a_{32} \cdot b_{32}] + [a_{33} \cdot b_{32} + a_{34} \cdot b_{32}] = [R_{33}] + [R_{34}] \]
\[ c_{42} = [a_{41} \cdot b_{42} + a_{42} \cdot b_{42}] + [a_{43} \cdot b_{42} + a_{44} \cdot b_{42}] = [B_{43}] + [B_{44}] \]

Figure 8 shows how the colored tokens are fed from the top (bottom) side of the array. Figure 9 shows the Array Activity map, where vertical solid arrows correspond to external colored tokens. Notation \( k.LOAD.\!r \) is used to represent the row of colored tokens needed for phase \( k \), whose destination is row \( r \) from the top (bottom); e.g. step \( 1.LOAD.2 \) corresponds to the loading step of the first row of tokens in Figure 8, needed for phase 1 and to be consumed in row 2 from top (bottom). Horizontal solid arrows correspond to the results being delivered to the left and right sides of the array. Gray arrows correspond to internal results being passed between PEs.

Computations for phase 1 start at step 1.1 (after tokens are loaded during steps 1.LOAD.2 and 1.LOAD.1). Tokens for phase 2 start loading at step 1.2, phase 2 starts computation at step 1.4. Therefore, two consecutive phases partially overlap. It is important to emphasize here that a step is not equivalent to a clock, but to a sequence of concurrent computation and communication actions. While a clock is a fixed and predefined period of time, a step is neither fixed nor predefined. The duration of step \( s \) (\( s = 1, 2, \ldots, 6 \)) differ from phase to phase depending on data availability (data-driven architecture) and on the type of operation under execution (multiplication, addition, comparison or opinion exchange).

If a permanent error occurs at step 3.1 in PE13, it will be detected and confined at step 3.6 (the opinion exchange step of phase 3). The error will be located at step 4.6. Error detection/confinement latency is 6 steps and error location latency is 9 steps, and this is a worst case example leading to maximum latency. If an error occurred in PE22 at step 3.5, the Error Flags will be \( EF^3 = (0, 1, 0, 0) \). Since the error occurred during comparison none of the results is corrupted, thus the \textit{deliver from} action taken by the
Figure 9: The Array Activity Maps for matrix-matrix multiplication example.
yellow judge according to Table 1 will not compromise the data integrity. The error will be detected and masked in the next step and located after 8 steps (5.6). The maximum location latency is small justifying the single error assumption of the fault model.

Conclusions

In this paper a new approach for distributed Concurrent Error Detection and Location for mesh data driven array architectures has been proposed. It is based on the simple concept of overlapping H-processes as a run-time fault tolerance mechanism. The H-processes concurrently perform in a redundant distributed way H-format operations such as \((a \cdot op^1 b) \cdot op^2 (c \cdot op^3 d)\). Since a very broad class of linear algebra and tree based algorithms fit into this formulation the array is general purpose and can be useful in many application domains. A functional realistic fault model was adopted, where there is no need for perfect comparators. The comparison can be performed at the ALU and is allowed to fail. Therefore many different implementations are envisioned using either general purpose microprocessors (such as the Transputers) to simple VLSI processing cells without TSC logic. No retry is necessary for transient fault recovery and reconfiguration should not be immediate. However some short transients cannot be handled.

Improvements under investigation include ways of dealing with transient errors of arbitrary duration and catastrophic communication link failures. Furthermore consideration is given on how to use arrays of this nature as fault tolerant "patches" in a wafer setup.

References


