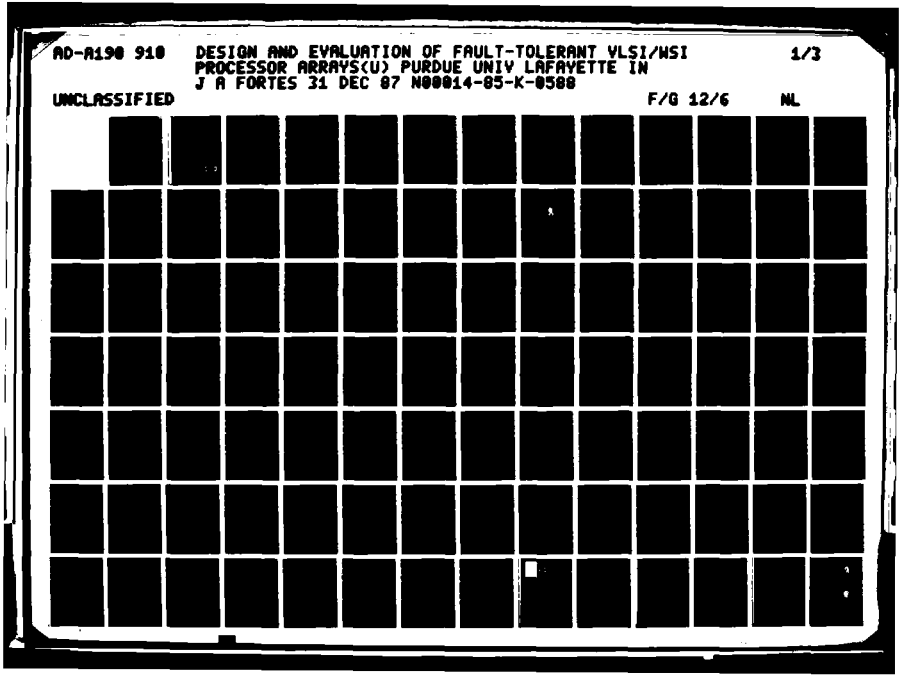
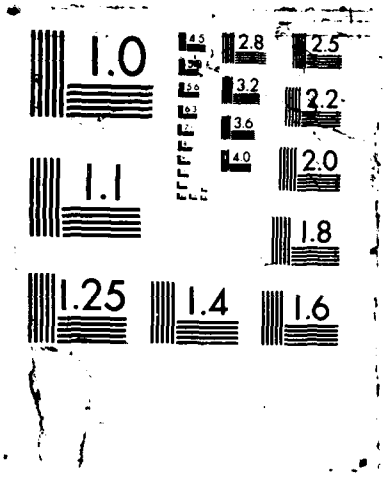
AD-A190 910 DESIGN AND EVALUATION OF FAULT-TOLERANT VLSI/WSI PROCESSOR ARRAYS(U) PURDUE UNIV LAFAYETTE IN J A FORTES 31 DEC 87 N00014-85-K-0588 1/3

UNCLASSIFIED F/G 12/6 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

FINAL TECHNICAL REPORT

for

CONTRACT NUMBER N00014-85-k-0588

and project entitled

DESIGN AND EVALUATION OF FAULT-TOLERANT VLSI/WSI PROCESSOR ARRAYS

Period: 85 Jul 01 to 87 Dec 31

Scientific Officer: Dr. Clifford Lau
ONR, Detachment, Pasadena

Prepared by: Dr. Jose A. B. Fortes
Purdue University

Date: 87 Dec 31

FINAL TECHNICAL REPORT

for

CONTRACT NUMBER N00014-85-k-0588

and project entitled

DESIGN AND EVALUATION OF FAULT-TOLERANT VLSI/WSI PROCESSOR ARRAYS

Period: 85 Jul 01 to 87 Dec 31

Scientific Officer: Dr. Clifford Lau
ONR, Detachment, Pasadena

Prepared by: Dr. Jose A. B. Fortes
Purdue University

Date: 87 Dec 31

Distribution:

Addressee	*Number of Copies*
Dr. Clifford Lau SDIO-IST Scientific/Technical Agent Department of the Navy Detachment, Pasadena 1030 East Green Street Pasadena, CA 91106	1
Administrative Contracting Officer Office of Naval Research Resident Representative 536 South Clark Street, Rm 286 Chicago, IL 60605-1588	1
Director, Naval Research Laboratory ATTN: Code 2627 Washington, D.C. 20375	6
Defense Technical Information Center Building 5, Cameron Station Alexandria, VA 22314	12

Table of Contents

Introductory Remarks

This document is the final report of work performed under the project entitled "Design and Evaluation of Fault-Tolerant VLSI/WSI Processor Arrays" supported by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and administered through the Office of Naval Research under Contract No. 00014-85-k-0588. With the concurrence of Dr. Clifford Lau, the Scientific Officer for this project, this final report consists of reprints of publications reporting work performed under the project. In the attached list of publications, items 1, 2, 3 and 7 are papers where fault-tolerant systems for processor arrays are proposed and studied. Studies on algorithmic and software aspects relevant to systems are reported in items 4, 5, 6, 12 and 13. Research on hardware and reconfigurability issues for fault-tolerant processor arrays is reported in items 8, 9, 10 and 11.

LIST OF PUBLICATIONS

1. Fortes, J. A. B., and Raghavendra, C. S., "Gracefully Degradable Processor Arrays," in IEEE Transactions on Computers, Volume C-34, Number 11, pp. 1033-1045, November 1985. (before SDIO support)

2. Fortes, J. A. B., Milutinovic, V., Dick, R., Helbig, W., and Moyers, W., "A High-Level Systolic Architecture for GaAs," 1986 International Workshop on High-Level Computer Architecture, Proc. of the 19th Hawaii International Conference on System Sciences (HICSS), pp. 253-258, January 1986. (before SDIO support)

3. Fortes, J. A. B., "Algorithm Reconfiguration Techniques for Gracefully Degradable Processor Arrays," International Workshop on Systolic Arrays, July 1986.

   Fortes, J. A. B., "Algorithm Reconfiguration Techniques for Gracefully Degradable Processor Arrays," in "Systolic Arrays," W. Moore, A. McCabe, R. Urquhart, editors, Adam Hilger, pp. 259-268, September 1986.

4. O'Keefe M., and Fortes, J. A. B., "A Comparative Study of Two Systematic Design Methodologies for Systolic Arrays," (Long Version) International Workshop on Parallel Algorithms and Architectures, pp. 313-324, April 1986.

   O'Keefe, M., and Fortes, J. A. B., "A Comparative Study of Two Systematic Design Methodologies for Systolic Arrays," (Short Version) International Conference on Parallel Processing, pp. 672-675, August 1986.

5. Taylor, V. E. and Fortes, J. A. B., "Using RAB to Map Algorithms into Bit-Level Systolic Arrays," 2nd International Conference on Supercomputing, pp. 227-236, May 1987.

6. Carlson, W. W. and Fortes, J. A. B., "On the Performance of Combined Data Flow and Control Flow Systems: Experiments Using Two Iterative Algorithms," International Conference on Parallel Processing, August 1987.

   Carlson, W. W. and Fortes, J. A. B., "On the Performance of Combined Data Flow and Control Flow Systems: Experiments Using Two Iterative Algorithms," Journal of Parallel and Distributed Computing, to appear.

7. Fortes, J. A. B. and Wah, B. W., "Systolic Arrays-From Concept to Implementation (Guest Editor's Introduction)," Computer, pp. 12-17, July 1987.

8. Rau, D., Fortes, J. A. B., Siegel, H. J., "Destination Tag Routing Schemes Based on a State Model for the IADM Network," Technical Report TR-EE 87-39, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907, October 1987. (Submitted to IEEE Transactions on Computers, October 1987).

9. Rau, D. And Fortes, J. A. B., "Partially Augmented Data Manipulator Networks: Minimal Designs and Fault Tolerance," Princeton Workshop on Algorithm, Architecture and Technology Issues for Models of Concurrent Computation, October 1987.

10. Rau, D. and Fortes, J. A. B., "Destination Tag-Controlled Fault Tolerant Interconnection Networks," Symposium on Innovative Science and Technology, January 1988.

11. Wang, Y-X. and Fortes, J. A. B., "Hierarchical Approaches to Fault-Tolerance in Processor Arrays," Symposium on Innovative Science and Technology, January 1988.

12. Fortes, J. A. B., Fu, K. S. and Wah, B. W., "Systematic Design Approaches for Algorithmically Specified Systolic Arrays," in Computer Architecture: Concepts and Systems," Milutinovic, V. M., eds., Elsevier Science Publishing Co., Inc., New York, pp. 448-488, 1988.

13. Fortes, J. A. B. and O'Keefe, M. T., "Current Architectures and a Tool for the Design and Programming of Bit Level Processor Arrays," 1988 IEEE Int. Symposium on Circuits and Systems, June 1988.

COPIES OF PUBLICATIONS

REFERENCE NO. 1

Fortes, J. A. B., and Raghavendra, C. S., "Gracefully Degradable Processor Arrays," in IEEE Transactions on Computers, Volume C-34, Number 11, pp. 1033-1045, November 1985. (before SDIO support)

Gracefully Degradable Processor Arrays

JOSE A. B. FORTES, MEMBER, IEEE, AND C. S. RAGHAVENDRA, MEMBER, IEEE

Abstract — **A new approach to the design of gracefully degradable processor arrays is discussed. Fault tolerance and graceful degradation are achieved by simultaneously reconfiguring the processor array and the algorithm in execution. Two types of algorithm reconfigurability are considered, namely, row reconfigurability (RR) and row–column reconfigurability (RCR). Correspondingly, two array reconfiguration schemes are discussed, i.e., successive row elimination (SRE) and alternate row–column elimination (ARCE). It is shown that the computations of any algorithm executable in a processor array can always be (re)organized so that the resultant algorithm has the RR and/or RCR properties. Upper bounds on the increase in execution time of an algorithm due to reorganization of computations for reconfigurability are derived. Detailed analysis of performance and reliability is done for both SRE and ARCE reconfiguration schemes. These reconfiguration techniques are applicable to any processor array and suitable for VLSI technology.**

Index Terms — **Algorithm transformations, computational availability, dynamic reconfiguration, graceful degradation, performability, processor arrays, reliability.**

I. Introduction

A recurrent theme in the quest for efficient high-speed computing systems is the need for matching the structure of algorithms and the configuration of parallel computers. In these systems, successful fault tolerance and graceful degradation schemes must disturb minimally the conformability of algorithm and architecture. These brief ideas underlie this paper's approach to the design of processor arrays where graceful degradation is achieved by simultaneous reconfiguration of algorithm and architecture. The pervasive consideration in the efficient use of processor arrays [1]–[10] is the careful development of algorithms that allow the allocation or pipelining of data and instructions so that the right data can be made available to the right processor at the right time by using the limited interconnection capabilities of the array. In [1]–[10] the reader can find a sample of problems, solutions, and experience on mapping algorithms into processor arrays. In these works, the prevalent realization is that dynamic reallocation of data and instructions is a difficult and time-consuming task. A duality argument can be used to claim that dynamic reconfiguration of an array with fully rearrangeable interconnections can be just as hard. In fact, at a more abstract level, both problems reduce to dynamically achieving "isomorphism" between algorithm and architecture. Any component failure in a processor array (without fault tolerance) is a potentially disrupting event to this "isomorphism" and may result in severe, if not total, performance loss. Because the large size of processor arrays and their tasks imply a high probability of failure, this may become an important limiting factor to the use of such computational machines.

Redundancy can be used to add fault tolerance to processor arrays, i.e., spare components are added to the system and they can replace faulty units, thus preserving the original computing structure and algorithm mapping. An alternative and equivalent way of thinking about redundancy solutions consists of programming algorithms which are smaller than those requiring the use of the full array and sparing out extra unused processors [30]. A distinct but yet related benefit of redundancy is the possibility of improving VLSI array fabrication yields [14]–[20], and several redundancy techniques used for this purpose are potentially applicable to fault-tolerant array computation. For a critique and appraisal of some of these schemes the reader is referred to [14]. The amount of redundancy used in a system is limited by economical and technological constraints (e.g., in [16] it was observed that yield improvement saturates above 10 percent of redundancy), and the minimization of redundancy for a given fault tolerance level is an important research problem [11]. Limited redundancy has been proposed or used for the MPP [3], Illiac IV [12], CHiP [6], and Diogenes arrays [14], among others. The main observation is that, by definition, redundancy solutions still require a fully operational replica of the original array, i.e., no degradation is possible. Alternative approaches to fault tolerance include error-correction techniques [21] and algorithm rescheduling strategies [22]. The former explores mathematical properties of the algorithm and is specialized in nature. The latter is algorithm dependent and does not explore the possibility of limited array reconfigurability. An algorithm independent approach to fault tolerance oriented towards preserving the connectivity of VLSI multiprocessor systems has also been reported [13].

In this paper we present a novel approach to fault tolerance and graceful degradation in array processors. The main idea is discussed informally in Section II. It consists of using algorithm mapping strategies and simple hardware mechanisms which make it possible to preserve conformability of algorithm and architecture despite the removal of faulty processors. It will become clear that our approach can be used together with previously proposed redundancy solutions, and such "hybrid" schemes (briefly discussed in the last section) would have the advantages of both approaches. Section III describes in a formal setting the theory behind the two main algorithm reconfiguration strategies. Array reconfiguration schemes are discussed in Section IV and their performance is

Manuscript received September 27, 1984, revised March 18, 1985 and April 9, 1985

J. A. B. Fortes is with the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907

C. S. Raghavendra is with the Department of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA 90089

0018-9340/85/1100-1033$01.00

studied in Section V. Reliability, performability, and computational availability studies of our techniques are presented in Section VI. Section VII is dedicated to conclusions.

II. Basic Ideas

We consider two different approaches for achieving limited dynamic reconfigurability of processor arrays and algorithms. The first approach to array reconfiguration consists of logically removing rows containing one or more faulty processors, and is referred to as successive row elimination (SRE). The second approach consists of removing either rows or columns with faulty processors (starting with rows), and is referred to as alternate row column elimination (ARCE). Both schemes require the addition of programmable switches and interconnections to the original array architecture and assume that "peripheral" processors are cyclically connected via "wrap-around" links (like in Illiac IV and MPP) or external memory (which is always possible). In correspondence with the two possible array reconfiguration schemes, we consider two algorithm reconfiguration strategies, namely, row reconfigurability (RR) and row–column reconfigurability (RCR). Interchanging the words "row" and "column" in the definitions of SRE, ARCE, RR, and RCR yields the dual reconfiguration schemes SCE, ACRE, CR, and CRR. Due to this duality, we will not discuss them. The next paragraph introduces informally the basic ideas on algorithm reconfigurability.

Consider an algorithm with $(T_0 \times n_1 \times n_2)$ computations which is executed in time T_0 in an array with $(n_1 \times n_2)$ processors. To each computation associate the time of execution and the coordinates of the processor where it is executed, i.e., index each computation with an integer vector (t, j_1, j_2), $1 \le t \le T_0$, $1 \le j_1 \le n_1$, $1 \le j_2 \le n_2$. The resulting index set of the algorithm is geometrically represented in Fig. 1(a). If, during execution, data move in a direction for which the value of j_1 does not decrease, then we say that the algorithm has the RR property; if the values of j_1 and j_2 do not decrease, then the algorithm satisfies the RCR property. Assume that our algorithm has the RR property and due to a fault we remove the last row of the original $(n_1 \times n_2)$ array. Then, we must also reconfigure the algorithm for execution in an $((n_1 - 1) \times n_2)$ array. This can be done by partitioning the algorithm into two "subalgorithms" or "bands" separated by the plane $j_1 = n_1 - 1$ [Fig. 1(b)]. First, the reduced array executes the subalgorithm for which $1 \le j_1 \le n_1 - 1$ and the RR property ensures that no computations require data from the other band. Next, the second subalgorithm is executed, possibly using data generated in the previous band and recycled through wrap-around or external memory connections. Note that potentially slow external memory communication can be done concurrently with the execution of a band. In fact, data generated in some order by a band will be used by the next band in the same order (i.e., FIFO stacks are suitable memory structures for this purpose). Similarly, if the algorithm has the RCR property and an $((n_1 - 1) \times (n_2 - 1))$ array is used, then the algorithm can be partitioned by the planes $j_1 = n_1 - 1$ and $j_2 = n_2 - 1$ into four bands which can be executed in increasing lexicographical order without violating data dependences [Fig. 1(c)]. It is important to compare this situation with the case when the array has $((n_1 - 2) \times n_2)$ processors for which the algorithm would still be partitioned in only two bands [Fig. 1(d)]. The remaining considerations on data communication are similar for RR and RCR with the exception that RCR may require additional wrap-around connections or external memory.

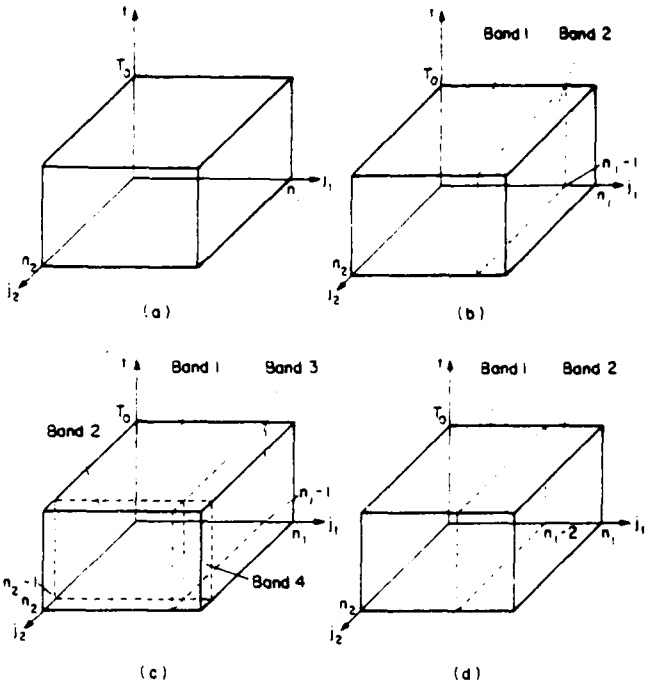


Fig. 1. Partitioned algorithm on processor arrays of four sizes. (a) $(n_1 \times n_2)$ (partitioning not required). (b) $((n_1 - 1) \times n_2)$. (c) $((n_1 - 1) \times (n_2 - 1))$. (d) $((n_1 - 2) \times n_2)$.

The ideas underlying our approach to algorithm reconfigurability are also useful for the problem of partitioning an algorithm for execution in fixed-size VLSI array architectures. From the discussion above, it is clear that RCR is a sufficient condition for algorithm partitionability, ([9], [10], [23]). Similarly, one can also think of RR as a sufficient condition for the partitionability of an algorithm along a single direction.

Not all algorithms executed in processor arrays have the RR or RCR property. However, in the next section we show that for any such algorithm we can always find an equivalent algorithm which satisfies such properties.

III. Algorithm Reconfiguration Schemes (RR and RCR)

We see a processor array as a two-dimensional grid in which each integer point is a vector index of a processor and a set of vectors (the interconnection primitives) which describes the (regular) pattern of interconnections of the array.

Definition 3.1:[1] *A processor array* is tuple (L^2, P) where

[1] Z and I denote the sets of integers and nonnegative integers, and Z^n and I^n denote their respective nth Cartesian powers.

$L^2 \subset Z^2$ is the *index set* of the array and $P \in Z^{2\times r}$ is a matrix of $r \in I$ *interconnection primitives*.

Thus, in a processor array (L^2, P), the processor with index $\bar{l} \in L^2$ is connected to a processor with index $\bar{l}' = \bar{l} + \bar{p}$, $\bar{p} \in P$, if $\bar{l}' \in L^2$, and is connected to an input–output port otherwise. This definition does not account for "wrap-around" external connections, which, however, are assumed to exist between input and output ports.

Example 3.1: The structure of orthogonal arrays like the Illiac IV, MPP, WAP, and others can be described by (L^2, P) where

$$L^2 = \{(l_1, l_2): 0 \le l_1, l_2 \le N - 1\}$$

$$P = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix}$$

where $N = 8$ for the Illiac IV, $N = 128$ for the MPP, and N is variable for VLSI arrays. Fig. 2 shows a (4×4) square orthogonal array. (End of example.)

The execution of an algorithm on a given array can be thought of as an ordered set of instantiations of the array, each of which contains an assignment of computations to processors at a particular time of execution. Consequently, we see an array algorithm as a three-dimensional grid in which each integer point $(j_1 j_2 j_3)^T$ indexes a computation at time j_1 and processor $(j_2 j_3)^T$, and a set of vectors (dependence vectors) which is related to the pattern of generation and use of data in time and space. In other words, if a computation with index $\bar{j}$ generates a value used in computation with index $\bar{j}'$, then $\bar{j}' - \bar{j}$ is a dependence vector. Clearly, the first entry of any dependence vector must be ≥ 1 (i.e., *at least one unit* of time separates generation and use of a variable), and the vector corresponding to the other two entries must correspond to a linear combination of interconnection primitives (i.e., a path connecting the processors where the variable is generated and used). Assuming that communication (over a single interconnection primitive) and execution of a computation take one unit of time,[1] the number of interconnection primitives used to communicate a result from computation with index $\bar{j}$ to computation with index $\bar{j}'$ must also be less than or equal to the first entry of the dependence vector $\bar{j}' - \bar{j}$ (i.e., the interval of time between the computations). These considerations translate into the following definition of array algorithm.

Definition 3.2: Consider an array (L^2, P), $P \in Z^{2\times r}$. An *array algorithm* is a tuple (J^3, D) where $J^3 \subset Z^3$ is the *index set* of the algorithm, and $D \in Z^{3\times m}$ is a matrix of $m \in I$ dependence vectors such that

$$d_{i1} \ge 1 \qquad i = 1, \cdots, m \tag{1}$$

and

$$\begin{bmatrix} d_{i2} \\ d_{i3} \end{bmatrix} = PK \qquad \text{for } K \in I^{r\times m} \text{ such that } \sum_{k=1}^{r} k_{ki} \le d_{i1},$$
$$i = 1, \cdots, m \tag{2}$$

[1] We follow the usual assumption that in one unit of time, a processor can read the output registers of neighboring processors, process data if necessary, and write results into its own output registers.

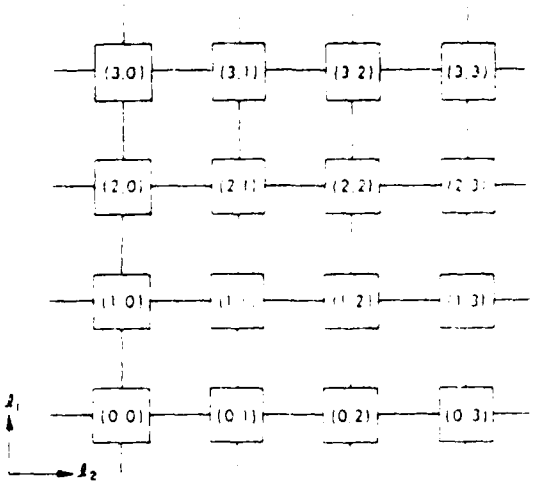


Fig. 2. A (4×4) square orthogonal array.

In this definition of array algorithm we represent only the structure of the algorithm and abstract from the actual computations being performed. This is adequate because we are essentially worried with problems of matching computational structures. Also, input and output data are not explicitly represented because they can be treated as generated data (i.e., for a given processor receiving data from other processors there is no distinction between data generated and data "passed" by those processors). Finally, the description of dependences would be more precise if to a given dependence vector we associate the index point where the dependence is valid. This complication turns out to be unnecessary for the derivation of our main results.

Example 3.2: In [8] the following computation was performed on the MPP as a filtering procedure required to avoid nonlinear instability in the solution of Navier–Stokes equations

$$q^t_{ij} = (q^{t-1}_{i,j+1} - 2q^{t-1}_{ij} + q^{t-1}_{i,j-1})/4$$
$$t = 1, \cdots, 8, \quad 1 \le i \le N, \quad 1 \le j \le N$$

where $N = 128$ = number of processors along one dimension of the MPP. The structure of the MPP is described in Example 3.1. This computation corresponds to an MPP array algorithm because

$$D = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix} (t) = PK$$

$$= \begin{bmatrix} 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

In other words, this algorithm maps trivially into the MPP because the number of computations matches the number of processors and the next-neighbor communication can be performed in one unit of time. (End of example.)

Example 3.3: In [7] it was shown that the following algorithm, which describes a simplified version of a standard relaxation computation, is not amenable to parallel execution (unless transformed as described later in Example 3.4):

$$u_{ijk} = (u_{i-1jk} + u_{ij-1k} + u_{ij-1k-1} + u_{ijk-1})/4$$
$$1 \le i \le L, 1 \le j \le M, 1 \le k \le N.$$

Here we note that, because the first entries of the last two dependence vectors in

$$D = \begin{matrix} \\ (i) \\ (j) \\ (k) \end{matrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

are smaller than 1, this algorithm is not an array algorithm. (End of example.)

As illustrated by the last example, not all algorithms are array algorithms. In other words, they must be transformed, or equivalently, their computations must be reorganized so that an equivalent array algorithm is obtained. The reorganization of an algorithm can be seen as permutation of its index set, and hence, it can be described as linear transformation $T \in Z^{(3\times3)}$ such that T is nonsingular and $T = \begin{bmatrix} \pi \\ S \end{bmatrix}$ where $\pi \in Z^{(1\times3)}$ is referred to as a time transformation and $S \in Z^{(2\times3)}$ is called a space transformation. In other words, T reorganizes the computations so that a computation with index $\bar{j}$ is executed at time $\pi\bar{j}$ and processor $S\bar{j}$. Due to the linearity of this type of transformation, the dependence matrix of a transformed algorithm is simply TD where D is the dependence matrix of the original algorithm. Hence, T can be selected so that the new dependence matrix makes the transformed algorithm an array algorithm. The fact that (1) must hold for the new matrix ensures that data dependences are not violated (i.e., T yields an array algorithm which is equivalent to the original one). This type of transformation was introduced in [27] where T is denoted R and referred to as reindexing transformation. Subsequent work in reindexing transformations is reported in [9], [10], [28], [29] and their references. In this paper, we add to the knowledge of algorithm transformations by showing that there exists always some T which yields an algorithm with RR and RCR properties (Theorem 3.1) and how to derive upper bounds in the execution time of such algorithms (Theorem 3.2). Next, we illustrate how a reindexing transformation can be used to transform an algorithm into an array algorithm.

Example 3.4: Assume $L = M = N = 4$ in the algorithm of Example 3.3, and consider the array shown in Fig. 2. In [7], the transformation

$$T' = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

was used to obtain an equivalent algorithm suitable for parallel computation. The slightly different transformation

$$T = \begin{bmatrix} \pi \\ S_1 \\ S_2 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

transforms that algorithm into an array algorithm because the resulting dependence matrix is

$$TD = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

for which the first row has only positive entries and

$$PK = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Fig. 3 shows three steps of the execution of this new algorithm. Empty squares represent unused processors. The index of the variable generated is shown for each busy processor. Arrows indicate data communication required for the computations, and broken lines identify computational wavefronts. The total execution time is $\pi[L-1, M-1, N-1]^T + 1 = 13$ units of time. (End of example.)

The problems of selecting T for general case algorithms are out of the scope of this paper, and the interested reader is referred to [9]. Here, we concentrate on the problem of selecting T for array algorithms so that the transformed algorithm satisfies the reconfigurability properties, i.e., the RR and RCR property. These properties can now be defined very simply in terms of the dependence matrix of an array algorithm.

Definition 3.3: An array algorithm (J^3, D) has the

RR property — if the entries of the second row of D are nonnegative;

RCR property — if the entries of the last two rows of D are nonnegative.

Example 3.5: The algorithm of Example 3.2 has the RR property, but does not have the RCR property. (End of example.)

Example 3.6: The algorithm of Example 3.4 does not have the RR and RCR properties. However, if T is redefined as

$$T = \begin{bmatrix} \pi \\ S_1 \\ S_2 \end{bmatrix} = \begin{bmatrix} 3 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

the resultant algorithm has the RR property. In fact,

$$TD = \begin{bmatrix} 2 & 2 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix},$$

and it is interesting to note that communication associated with the first two dependences takes two units of time, and for the second one it requires the use of two interconnection primitives, namely, $(1 \quad 0)^t$ and $(0 \quad -1)^t$. Three steps of the execution of this algorithm are shown in Fig. 4 where arrows indicating data communication which takes two units of time are labeled with a (2). The execution time is now $\pi[L-1,$

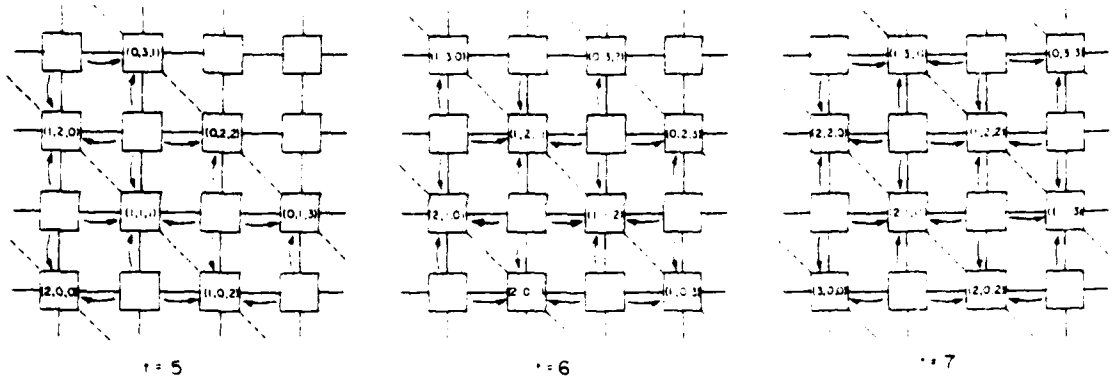


Fig. 3. Three steps of the execution of the algorithm of Example 3.5.

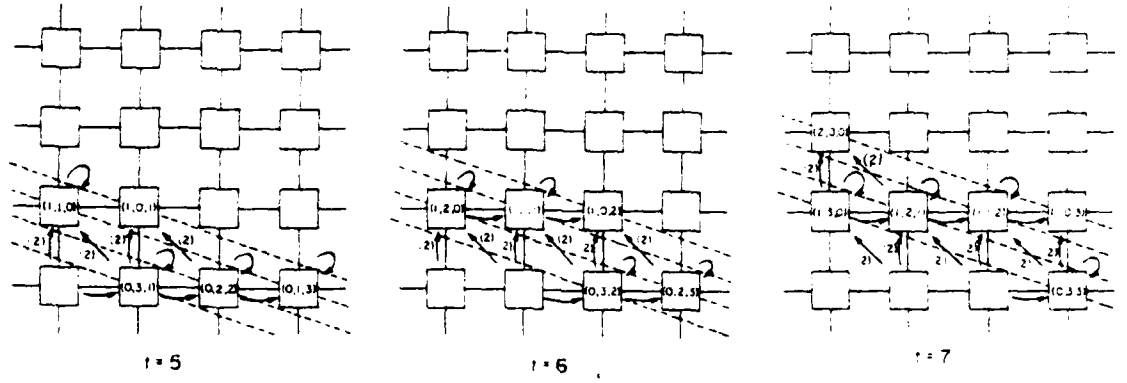


Fig. 4. Three steps of the execution of the algorithm of Example 3.6.

$M-1, N-1]^T + 1 = 16$ units of time. (End of example.)

Next we prove that we can always reorganize the computations of any array algorithm so that the resultant algorithm has the RR and the RCR properties.

Theorem 3.1: Given an array algorithm A' it is always possible to reorganize the computations of A so that the resultant array algorithm has the RR and RCR properties.

Proof: We show that we can always find T such that the new dependence matrix has no negative entries. By definition of array algorithm, the dependence matrix D of A must have only positive entries in the first row. This means that there exists a convex set which contains all dependence vectors of A. Hence, using convex set theory, there exists an infinite number of separating hyperplanes. In other words, there exists an infinite number of vectors $S \in Z^{(1\times 3)}$ such that $S\bar{d} \geq 0$ for all $\bar{d} \in D$. Furthermore, because the subspace outside the convex set is not degenerate, from that set of vectors one can always choose three linearly independent vectors for rows of T so that T is nonsingular. It is also clear that they can always be chosen so that the new dependence matrix satisfies (2) for some K. Hence, the new algorithm is still an array algorithm, and its dependence matrix has no negative entries, thus implying that the RR and RCR properties hold. Q.E.D.

The next theorem provides an upper bound on the execution time of the reconfigured algorithm as a function of the execution time of the original algorithm. This upper bound is valid for arrays with a matrix of interconnection primitives identical to that of Example 3.1. In other words, we consider only the class of square orthogonal arrays. The methodology used can be easily applied to the derivation of similar bounds for other classes of arrays.

Theorem 3.2: For any $(n \times n)$ orthogonal array algorithm with execution time T_0 there exist equivalent orthogonal array algorithms with the RR and the RCR properties and with execution time $T^* < 2T_0^2/n + 3T_0 + n$ and $T^* < 3T_0^3/n^2 + 9T_0^2/n + 6T_0$, respectively.

Proof: Assume that the original orthogonal array algorithm does not have the RR property and consider the worst case possible, i.e., the case when at every instant of time all processors and interconnection primitives are used. This corresponds to an algorithm for which

$$D = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix},$$

and it can be transformed into an equivalent algorithm with the RR property by choosing T as

$$T = \begin{bmatrix} \pi \\ S_1 \\ S_2 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{so that}$$

$$TD = \begin{bmatrix} 2 & 3 & 2 & 1 & 2 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix}.$$

Given that the original algorithm has $(T_0 \times n \times n)$ com-

putations, the new algorithm requires

$$\pi[T_0 \quad n \quad n] - \pi[1 \quad 1 \quad 1] + 1 = 2T_0 + n - 2 \qquad (3)$$

units of time to execute in an array with T_0 rows and n columns (due to the values of S_1 and S_2). Given that the actual array has only n rows, we must partition the algorithm into $\lceil T_0/n \rceil < T_0/n + 1$ bands (i.e., now S allocates computation with index $\bar{j}$ to processor $[(S_1\bar{j} \bmod n)S_2\bar{j}]^T$), and each band takes at most the time given by (3). Hence, the total execution time of the new algorithm is

$$T^* < (2T_0 + n - 2)(T_0/n + 1) < 2T_0^2/n + 3T_0 + n. \qquad (4)$$

Choosing $\pi = [300]$, $S_1 = [110]$, and $S_2 = [101]$ leads to a similar proof for the RCR case.

Q.E.D.

We remark that the upper bounds derived in the last theorem may be too high, and how to find (for any array) a transformation T which yields exact upper bounds for T^* is an open problem. Nevertheless, typically, algorithms executed in $(n \times n)$ arrays have execution time linear in n, which implies that, according to Theorem 3.2, reconfigured algorithms also have linear execution time.

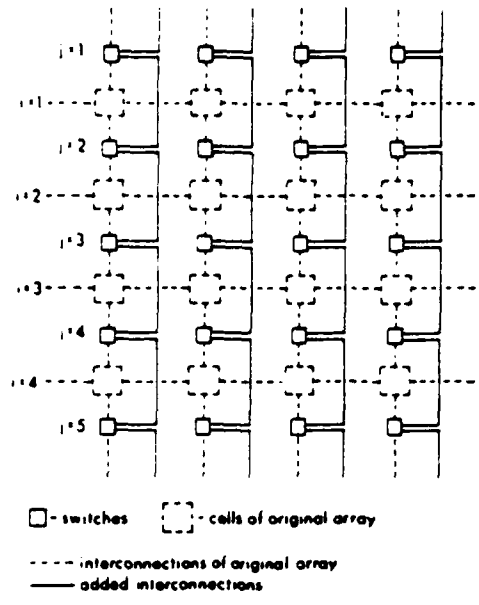
Example 3.7: The algorithm of Example 3.4 does not have the RR and RCR properties and executes in 13 units of time, whereas the equivalent algorithm in Example 3.5 has the RR property and executes in 16 units of time. However, the value of the upper bound given by Theorem 3.2 is $T^* < 2T_0^2/n + 3T_0 + n|_{T_0=13,\,n=4} = 85$. This discrepancy between the values of the upper bound and the actual execution time is also due to the fact that the algorithm of Example 3.4 is not a worst case algorithm like the one considered in the proof of Theorem 3.2. This is easily realized from Fig. 3, which shows that not all processors and interconnection primitives are used every time. (End of example.)

IV. Array Reconfiguration Schemes (SRE and ARCE)

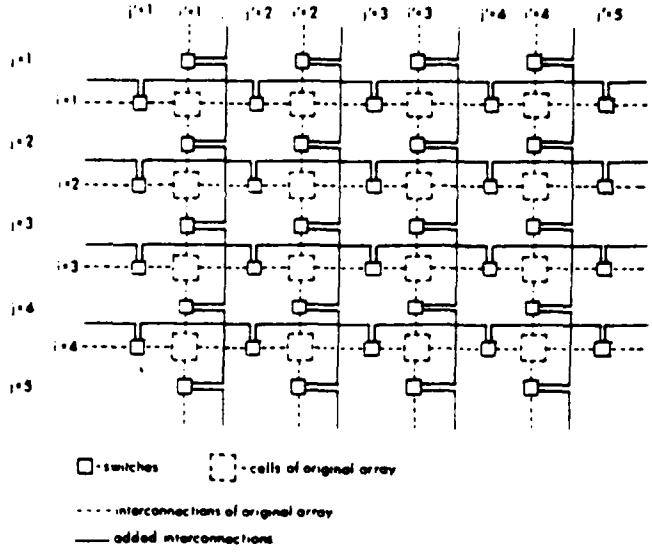
This section describes the architectural features of processor arrays capable of SRE and ARCE reconfiguration. In both cases, limited reconfigurability is achieved by using redundant interconnections and switches. The figures used to describe these architectures display the logic organization and functional characteristics of the architectures and their components. Their physical layout and implementation depend on the technology used and may take different forms, as discussed in [14]–[20], [26].

A. SRE Reconfiguration

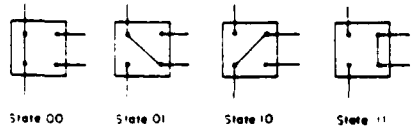
The basic idea of SRE is as follows: if a fault occurs in processor (i, j), then eliminate logically the ith row of the array. The logical elimination of a row is done by setting identical programmable switches to certain states and using redundant interconnections to bypass the eliminated row. Fig. 5(a) shows the additional interconnections and switches for a (4×4) array. The broken lines represent the original hardware. In general, $(n_1 - 1)n_2$ interconnections ($\approx$50 percent interconnection redundancy) and $(n_1 - 1)n_2$ switches are required. The structure and possible states of each switch are shown in Fig. 5(c). We ignore the need for additional external memory due to further partitioning of the algorithm because external memory is usually available or inexpensive to add.



(a)



(b)



(c)

Fig. 5. Array architecture. (a) SRE reconfiguration. (b) ARCE reconfiguration. (c) States and structure of switches used.

The reconfiguration of the array is done by setting the switches to certain states. Let the ith row of switches be such

that it connects the $(i-1)$th row and ith rows of processors. Let $X_i = 1$ when row i of processors is eliminated from the array because it contains at least one faulty processor, and $X_i = 0$ otherwise (i.e., row i is present). The ith row of switches has its state determined by X_i and X_{i-1}, i.e., each and every switch in row i is in state $X_{i-1}X_i$. It is easy to see that the above rule can be implemented with simple logic distributed across every row or every processor. Thus, additional control hardware is minimal and can be ignored for practical purposes. Furthermore, note that complete isolation of faulty modules is provided by the switches and the control rule used.

B. ARCE Reconfiguration

In describing ARCE reconfiguration, we assume without loss of generality that in an $(n_1 \times n_2)$ array we have $n_1 \geq n_2$; all the following discussions remain valid when $n_2 \geq n_1$ if we interchange n_1 and n_2 and replace the word ARCE by ACRE. ARCE removes either the row or the column of the array containing a faulty processor according to the following rule: remove a column if and only if $\lceil n_1/n_2 \rceil$ rows have been eliminated after the last column elimination (if any). Note that for $n_1 = 1$ or $n_2 = 1$, ARCE reduces to SCE and SRE, respectively. As for SRE, logical elimination of rows or columns uses additional switches and interconnections. Fig. 5(b) shows this additional hardware (full lines) and the original array (broken lines) with (4×4) processors. In general, $(n_1 + 1)n_2 + n_1(n_2 + 1)$ additional interconnections (=100 percent redundancy) and $(n_1 + 1)n_2 + n_1(n_2 + 1)$ switches are required. As for SRE, additional external memory and internal control logic can be ignored. The structure and states of the switches are the same as for SRE [Fig. 5(c)]. To reconfigure the array, the scheme used for SRE reconfiguration is also used in ARCE for row elimination. For column elimination the rule is similar except that the word "row" is replaced by the word "column." Thus, simple distributed logic can also be used, and total fault isolation is also guaranteed.

V. Performance Analysis

A consequence of the simultaneous use of SRE with RR and ARCE with RCR reconfiguration is graceful performance degradation as the array size is reduced. In this section we give a lower bound for array performance as a function of the number of rows and columns eliminated. The lower bound is exact in the sense that there are worst case algorithms for which such lower bound performance results. Assuming worst case algorithms for which the RR and RCR properties hold, we also derive exact lower bounds for the performance as a function of the number of faults for the best and worst case fault distributions. The best case fault distributions can also be thought of as the case when an ideal array has the capability of reconfiguring itself so that faulty processors can always be grouped in the same row or column. For simplicity, we only present the reasoning leading to these bounds, and the reader can use simple inductive rules to verify their correctness.

Any algorithm executed in an $(n_1 \times n_2)$ array in time T_0 will perform at most $(T_0 \times n_1 \times n_2)$ computations. Thus, the worst case happens when the algorithm has exactly $(T_0 \times n_1 \times n_2)$ computations, i.e., all processors are always busy. We consider this as the worst case in the sense that any single processor failure results in the largest number of computations that are not performed. Consider the case when this worst case algorithm must be executed in an array of size $(n_1 - m_1) \times (n_2 - m_2)$, $0 \leq m_1 < n_1$, $0 \leq m_2 < n_2$. This smaller array will need at most time T_0 to compute each of the $\lceil n_1/(n_1 - m_1) \rceil \lceil n_2/(n_2 - m_2) \rceil$ partitions of the original algorithm, each of which has at most $T_0 \times (n_1 - m_1) \times (n_2 - m_2)$ computations. Taking computation time as a measure of performance, the performance of any array with m_1 rows and m_2 columns eliminated is

$$T \leq \lceil n_1/(n_1 - m_1) \rceil \lceil n_2/(n_2 - m_2) \rceil T_0 \tag{6}$$

Let T_k denote the computation time for an array with k faulty processors. Let the ratio T_0/T_k be a normalized measure of performance when k faults occur and let P_k denote a lower bound on such ratio (i.e., consider worst case algorithms). Depending on the distribution of faults in the array, P_k can take distinct values for a fixed k. For SRE, k faults cause the elimination of at most k and at least $\lceil k/n_2 \rceil$ rows. From (6) it follows that

$$\frac{1}{\left\lceil \frac{n_1}{n_1 - k} \right\rceil} \leq (P_k)_{\mathrm{SRE}} \leq \frac{1}{\left\lceil \frac{n_1}{n_1 - \left\lceil \frac{k}{n_2} \right\rceil} \right\rceil} \tag{7}$$

For ARCE, in the worst case fault distribution,

$$k - \left\lfloor \frac{k}{\frac{n_1}{n_2} + 1} \right\rfloor \text{ rows and } \left\lfloor \frac{k}{\frac{n_1}{n_2} + 1} \right\rfloor \text{ columns}$$

are removed. In the best case fault distribution, the numbers of rows and columns removed satisfy complex expressions, and we prefer to use simpler conservative estimates. Clearly, the number of rows removed is less than $\lceil k/(n_2) \rceil$, and the number of columns removed is less than $\lfloor k/\lceil n_1/n_2 \rceil n_2 \rfloor$. Hence, from (6) we have

$$\frac{1}{\left\lceil \frac{n_1}{n_1 - k + \left\lceil \frac{k}{\frac{n_1}{n_2} + 1} \right\rceil} \right\rceil \left\lceil \frac{n_2}{n_2 - \left\lfloor \frac{k}{\frac{n_2}{n_1} + 1} \right\rfloor} \right\rceil} \leq (P_k)_{\mathrm{ARCE}}$$

$$\leq \frac{1}{\left\lceil \frac{n_1}{n_1 - \frac{k}{n_2}} \right\rceil \left\lceil \frac{n_2}{n_2 - \left\lceil \frac{k}{\frac{n_1}{n_2} n_2} \right\rceil} \right\rceil} \tag{8}$$

Note that for both SRE and ARCE the lower bound on performance reduction (i.e., for the worst case algorithm and worst case fault distribution) is always less than or equal to 0.5 for $k \geq 1$. Fig. 6 shows these bounds as functions of the number of faults for the case when $n_1 = n_2 = n$.

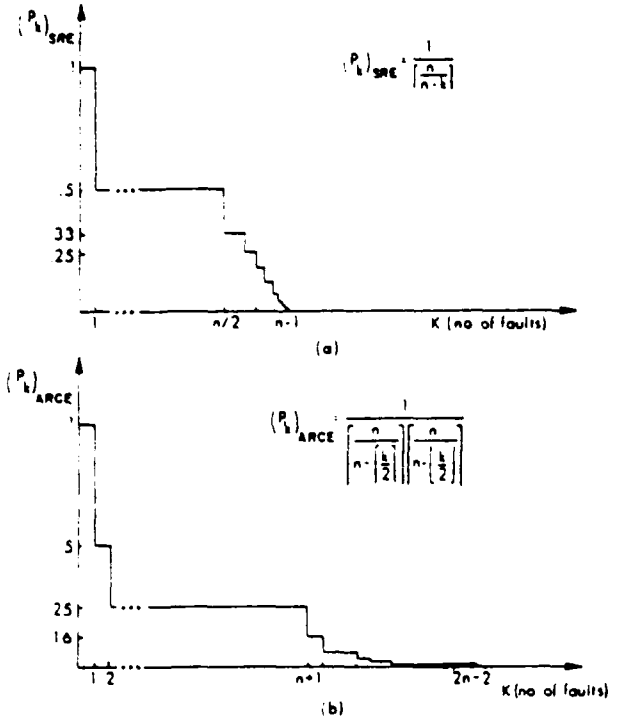


Fig. 6. Worst case performance reduction. (a) SRE. (b) ARCE as a function of the number of faults in an $(n \times n)$ array.

Example 5.2: Assume that, due to the occurrence of faults, the algorithm of Example 3.6 must be executed by a) a (3×4) array, and b) a (2×4) array. Fig. 7 shows three steps of the execution of the algorithm in case b). The total execution time is $2(\pi[1, M-1, N-1]+1) = 20$ units of time where π, M, and N are as in Example 3.6. The normalized execution time or performance is $16/20 = 0.8$, which is higher than the value of 0.5 predicted by (7). The same can be said for case a), and full details can be found in [31]. This example also illustrates the execution of a reconfigured RR algorithm in an SRE reconfigured array. In Fig. 7, the wraparound arrows are labeled with (5) to indicate that communication takes 5 units of time. This illustrates our claim that this scheme allows the potentially time-consuming use of external memory for recycling data. (*End of example.*)

VI. Analysis of Reliability Performability and Computational Availability

In this section we evaluate SRE and ARCE reconfiguration schemes in terms of the following commonly used measures:

1) reliability $R(t)$, i.e., the probability of no array failure in the interval of time $[0, t]$;

2) performability $\mathrm{Perf}(B, t)$ [24], i.e., the probability that the array performs above some performance level B;

3) computational availability $A(t)$ [25], i.e., the expected value of the computational capacity of the array at time t (in this work, computational capacity means the number of processors that have not been removed from the array).

We use the common assumption that the processors of the array have exponentially distributed failures, i.e., the probability that a processor has not failed before time t, given that it was initially functional, is $e^{-\lambda t}$ where λ is the failure rate. In our analysis the unit interval of time $[0, t]$ is such that $\lambda t = 1$, i.e., we use a failure rate that is normalized with respect to the time unit (e.g., $\lambda = 10^{-6}$ failures/h, $t = 10^6$ h). The assumption of exponentially distributed failures is also convenient because it will allow us to use Markov models for arrays with SRE and ARCE reconfiguration (exponential distributions are memoryless). This is justifiable for arrays with independent processor modules. We also consider the possibility of imperfect reconfiguration by using a parameter c, the coverage, which is the conditional probability of a successful array reconfiguration given that a failure has occurred. This parameter incorporates the effectiveness of both the fault-detection and switching mechanisms used and we assume it to be the same for any fault in any processor. Next, we give the number of possible array configurations (also called degradation states, or simply states) for SRE and ARCE schemes. This number also measures the number of faults that such schemes sustain before total array failure (assuming worst case fault distribution).

The number of processor failures tolerated in an $(n_1 \times n_2)$ array with SRE or SCE reconfiguration is, respectively,

$$(n_1 - 1) \le (K)_{\mathrm{SRE}} \le (n_1 - 1)n_2 \tag{9}$$

and

$$(n_2 - 1) \le (K)_{\mathrm{SCE}} \le (n_2 - 1)n_1 . \tag{10}$$

The left- and right-hand sides of (9) correspond to the cases when all faults occur in distinct rows and in the same rows, respectively. This comment also applies to (10) if we replace the word "rows" by the word columns. Clearly, SRE is more fault tolerant than SCE if and only if $n_1 > n_2$. This is a criterion as to when to use SCE or SRE.

For ARCE, the number of processor failures tolerated in an $(n_1 \times n_2)$ array is

$$\left\lfloor \frac{n_1}{\left\lfloor \frac{n_1}{n_2} \right\rfloor} \right\rfloor + n_1 - 2 \le K \le n_1(n_2 - 1) + \left\lfloor \frac{n_1}{\left\lfloor \frac{n_1}{n_2} \right\rfloor} \right\rfloor \left\lfloor \frac{n_1}{n_2} \right\rfloor - 1 \tag{11}$$

where the left- and right-hand side expressions correspond to the case when all faults occur in distinct rows and columns and to the case when they occur in the rows and columns already eliminated, respectively.

Without loss of generality, we consider an $(n \times n)$ array and analyze its reliability, performability, and computational availability when SRE and ARCE reconfiguration is used. The Markov state diagrams for these two cases are shown in Fig. 8. The number of degradation states is given by the left-hand side of (9) and (11) where $n_1 = n_2 = n$, i.e., $n - 1$ for SRE and $2n - 2$ for ARCE. In the same figures, every state is represented by a circle showing the state number and the number of processors in the reduced array for that state; the state transition rates are also indicated. An arrow starting

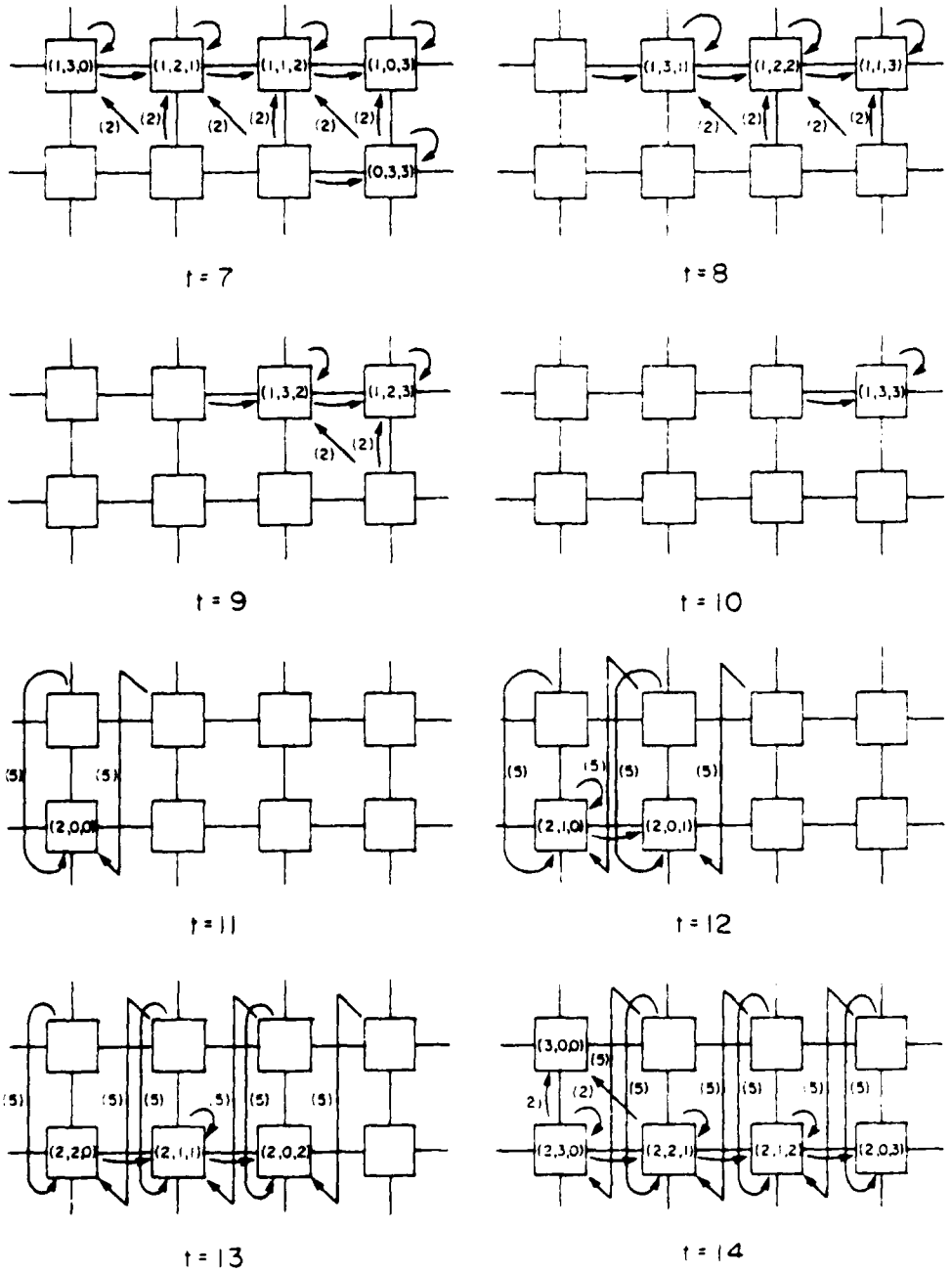


Fig. 7. Six steps of the execution of the algorithm of Example 5.1 (case b) (a (2×4) array). For $t = 5$ and $t = 6$ the computations proceed as shown in Fig. 4.

in state #i goes to state $(i + 1)$ to indicate that a failure of the array in state #i is covered (i.e., detected and recovered successfully). An arrow starting in state #i goes to state F to indicate that the array in state #i failed to recover from failure. Each arrow is labeled with a state transition rate, and this is

- in SRE

$$c\lambda_i = cn(n - i)\lambda \qquad \text{from state } \#i \text{ to state } \#(i + 1)$$

$$(1 - c)\lambda_i = (1 - c)n(n - i)\lambda$$

from state #i to state F, $i = 0, \cdots, n - 2$ (12)

- in ARCE

$$c\lambda_i = c\left(n - \frac{i}{2}\right)\left(n - \frac{i}{2}\right)$$

from state #i to state #$(i + 1)$

$$(1 - c)\lambda_i = (1 - c)\left(n - \frac{i}{2}\right)\left(n - \frac{i}{2}\right)$$

from state #i to state F, $i = 0, \cdots, 2n - 3$ (13)

In both ARCE and SRE there is a single possible transition from the last degradation state to state F. For SRE, the state

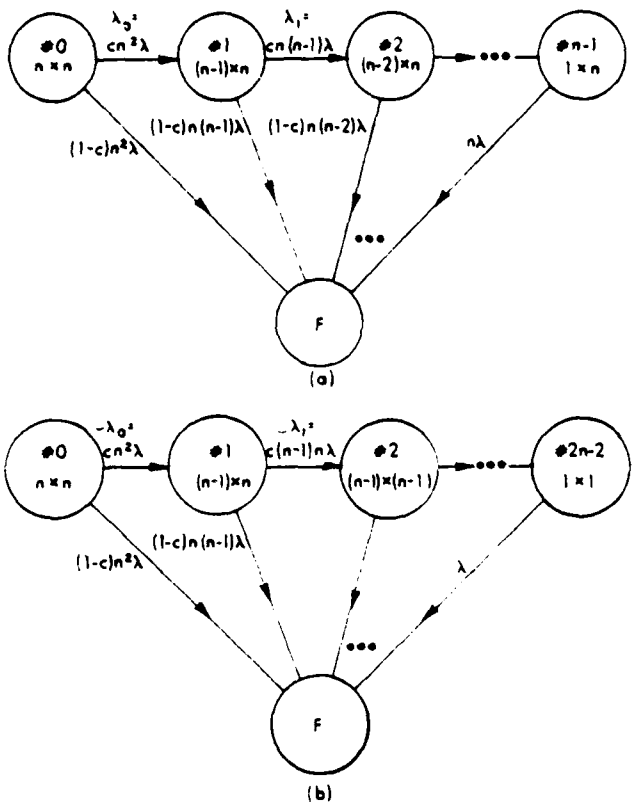


Fig. 8. Markov state diagram. (a) SRE. (b) ARCE reconfiguration.

transition rate is $\lambda_{n-1} = n\lambda$, whereas for ARCE it is $\lambda_{2n-2} = \lambda$. The Markov models for SRE and ARCE differ only in the number of states, and they are described by the differential equations

$$\frac{d\Pr_0(t)}{dt} = -(c\lambda_0 + (1-c)\lambda_0)\Pr_0(t) = -\lambda_0\Pr_0(t)$$

$$\frac{d\Pr_k(t)}{dt} = -\lambda_k\Pr_k(t) + c\lambda_{k-1}\Pr_{k-1}(t), \qquad k = 1, \cdots, D \tag{14}$$

where $D = n - 1$ for SRE and $D = 2n - 2$ for ARCE, $\Pr_k(t)$ denotes the probability of the array being in state k at time t, and the λ's are as in (12) and (13) for SRE and ARCE, respectively.

Assuming the initial conditions $\Pr_0(t) = 1$, $\Pr_k(0) = 0$, $k = 1, \cdots, D$, the solution to (14) can be obtained by using Laplace transforms and partial fraction expansion as

$$\Pr_0(t) = e^{-\lambda_0 t}$$

$$\Pr_k(t) = c^k \sum_{i=0}^{k} \frac{\prod_{j=0}^{k-1} \lambda_j}{\prod_{\substack{j=0 \\ j \neq i}}^{k} (\lambda_j - \lambda_i)} e^{-\lambda_i t}.$$

After evaluating the state probabilities, the reliability, performability, and computational availability can be computed using the following expressions:

- reliability = $R(t) = \sum_{k=0}^{D} \Pr_k(t)$

where the number of degradation states is $D = n - 1$ for SRE and $D = 2n - 2$ for ARCE.

- performability = $\text{Perf}(B, t) = \sum_{k=0}^{j} \Pr_k(t)$

where j is such that the performance $P_k \geq B$ for $k = 0, \cdots, j$;

- computational availability = $A_c(t) = \sum_{k=0}^{D} \Pr_k(t) C_k$

where D is as above and C_k is the number of processors in state k, i.e.,

$$C_k = n(n-k) \qquad \text{for SRE}$$

and

$$C_k = \left(n - \left\lceil \frac{k}{2} \right\rceil\right)\left(n - \left\lfloor \frac{k}{2} \right\rfloor\right) \qquad \text{for ARCE}.$$

We can also compute the reliability improvement factor (RIF), defined as $\text{RIF}(t) = (1 - R_{nr}(t))/(1 - R(t))$, where $R_{nr}(t)$ is the reliability of a nonreconfigurable array, and thus, $R_{nr}(t) = \Pr_0(t)$.

We used the above expressions to evaluate the following arrays using SRE and ARCE: a (5×5) array for the cases when $(c = 1, B = 0.5)$, $(c = 1, B = 0.25)$, and a (10×10) array for the cases when $(c = 1, B = 0.5)$, $(c = 1, B = 0.25)$ $(c = 0.95, B = 0.5)$, $(c = 0.98, B = 0.5)$, and $(c = 0.99, B = 0.5)$. We considered the operation of the array during five intervals of 0.1 units of time starting at $t = 0$. The results summarized in Tables I and II allow us to conclude the following.

- ARCE has better reliability than SCE when fault coverage $c = 1$ (see Table I).
- ARCE and SRE have comparable reliability when fault coverage $c \leq 0.99$ (see Table II).
- For high performance (i.e., large values of B), SRE has better performability than ARCE (see Table I).
- ARCE has better computational availability (see Table I).
- Reliability for both SRE and ARCE degrades significantly as coverage decreases (see Table II); this is particularly drastic for ARCE.

Table III compares SRE and ARCE qualitatively and the following comments complement the information of that table.

- The amount of additional hardware can be measured in terms of the increase in the number of a given type of component of the array or in terms of the chip area taken by that hardware. The first approach may be unrealistic for VLSI arrays, whereas the second is dependent on the technological process and the size of each cell used. We used the first approach to account for the additional number of interconnections and the second to measure the effect of adding switches. We assumed that every switch takes 20 percent of the area of a cell (like in [20]).
- Fault coverage is a critical parameter for the reliability of ARCE and SRE. Two important conclusions can be made. First, if fault coverage is not very good, then the difference in reliability for SRE and ARCE is negligible. This, in turn, means that SRE is preferred to ARCE because it requires less additional hardware. The second conclusion is that the fault detection and recovery schemes used should be as simple and reliable as possible so that fault coverage is very close to unity. The fact that our reconfiguration schemes use only a reduced number of interconnections, switches (which can be

TABLE I
RELIABILITY, PERFORMABILITY FOR PERFORMANCE LEVELS OF $B = 0.5$ AND $B = 0.25$, AND COMPUTATIONAL AVAILABILITY FOR SRE AND ARCE IN (5 × 5) AND (10 × 10) ARRAYS (COVERAGE $c = 1$)

Array Size	Time	SRE				ARCE			
		R(t)	Perf (.5,t)	Perf (.25,t)	$A_c(t)$	R(t)	Perf (.5,t)	Perf (.25,t)	$A_c(t)$
5x5	1	0.990	0.693	0.912	15.1	1.000	0.348	0.980	16.1
	2	0.899	0.263	0.606	9.19	0.999	6.45×10^{-2}	0.818	11.2
	3	0.717	7.72×10^{-2}	0.311	5.57	0.999	1.01×10^{-2}	0.551	8.30
	4	0.516	2.00×10^{-2}	0.138	3.38	0.998	1.49×10^{-3}	0.315	4.36
	5	0.348	4.87×10^{-3}	0.057	2.06	0.996	2.12×10^{-4}	0.162	3.01
10x10	1	0.989	0.230	0.775	38.7	1.000	9.25×10^{-4}	0.983	44.8
	2	0.788	4.92×10^{-3}	0.243	13.5	1.00	0.00	0.595	25.2
	3	0.399	4.8×10^{-5}	1.13×10^{-2}	4.87	0.999	0.00	0.155	16.1
	4	0.168	0.00	6.69×10^{-4}	1.83	0.999	0.00	2.38×10^{-2}	11.1
	5	0.065	0.00	3.50×10^{-5}	0.673	0.999	0.00	2.77×10^{-3}	8.2

TABLE II
RELIABILITY IMPROVEMENT FACTOR FOR SRE AND ARCE IN A (10 × 10) ARRAY FOR COVERAGE $c = 1$, $c = 0.99$, $c = 0.98$, AND $c = 0.95$

Time	RIF(t) for SRE				RIF(t) for ARCE			
	$c=1$	$c=.99$	$c=.98$	$c=.95$	$c=1$	$c=.99$	$c=.98$	$c=.95$
1	98.1	14.1	7.82	3.58	2.21×10^{12}	15.3	7.93	3.47
2	4.28	3.39	2.84	2.01	7.48×10^{7}	10.4	5.44	2.49
3	1.66	1.57	1.50	1.34	4.77×10^{5}	9.74	4.62	2.17
4	1.20	1.18	1.16	1.11	2.18×10^{4}	7.92	4.21	2.01
5	1.08	1.06	1.05	1.0	2.73×10^{3}	7.41	3.96	1.91

TABLE III
COMPARISON OF SRE AND ARCE

	Additional Hardware	Reliability	Sensitivity to coverage	Performability	Computational Availability
SRE	50% (interconn.) 20% (switches)	Good	High	Good for high performance levels	Good
ARCE	100% (interconn.) 40% (switches)	Very good	Very high	Good for low performance levels	Very good

designed conservatively), and a very simple control rule matches this need. This may not be true or feasible in fully reconfigurable arrays.

- SRE has high performability for high performance levels, whereas ARCE has high performability for low performance levels. This suggests that SRE should be used in real-time applications where execution time is critical, whereas ARCE is adequate for applications requiring long periods of operation (e.g., remote systems).
- Computational availability is used here to measure the potential computational capacity of the array. The actual computational capacity depends also on the method used to explore the potential capacity. In other words, there may be other algorithm partitioning techniques or computation-to-processor allocation policies that yield better performability than our techniques. In particular, note that (potential) computational availability is always better for ARCE in contrast to worse performability for high performance levels.

VII. CONCLUSIONS

Any processor array can be made fault tolerant by using the approach described in this paper. However, our reconfiguration schemes assume the existence of complementary testing and fault recovery techniques and adequate technologically feasible hardware implementations. Due to limitations in space we do not elaborate on these issues here, and for a brief discussion on some of them the reader is referred to [31].

In summary, this paper proposed and analyzed two possible approaches to the design of gracefully degradable array processors. They thrive on the generality and simplicity of reconfiguration schemes which make it possible to preserve the conformability of the processor array and the algorithm being executed. We showed that any algorithm can be (re)mapped into a processor array so that it can be partitioned or reconfigured along one or both orthogonal directions of the plane. Array reconfiguration is achieved by logical elimination of rows and/or columns with faulty processors. The switching mechanism isolates failed modules and is extremely simple and cost effective. We analyzed and exemplified the use of our techniques in detailed examples. Closed form expressions were derived for reliability, performability and computational availability, and they were used to evaluate (5 × 5) and (10 × 10) array systems. Besides its simplicity and generality, our approach has another significant advantage over previously proposed solutions, i.e., the possibility of graceful degradation. Our schemes tolerate at least $(n - 1)$ faults in an $(n \times n)$ array, whereas redundancy solutions tolerate a small constant number of faults and require larger amounts of additional hardware. The novelty and superiority of our schemes results from the fact that they explore the characteristics of both the algorithm and the architecture. Clearly, our approach can be used together with other solutions based on the use of redundancy or more complex forms of array reconfiguration. In these hybrid schemes, redundancy could be used to preserve the size and structure of the array as long as possible, followed by progressive simple and fast SRE or ARCE reconfiguration steps interspersed with other complex time-consuming reconfiguration procedures.

REFERENCES

[1] K. Hwang and F. A. Briggs, *Parallel Computer Architecture*. New York: McGraw-Hill, 1984.

[2] L. Uhr, *Computer Arrays and Networks: Algorithm-Structured Parallel Architectures*. New York: Academic, 1982.

[3] K. E. Batcher, "Design of a massively parallel processor," *IEEE Trans. Comput.*, vol. C-29, pp. 836–840, Sept. 1980.

[4] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI processor arrays," in *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds. Reading, MA: Addison-Wesley, Sec. 8.3, 1980.

[5] S.-Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. B. Rao, "Wavefront array processor: Language, architecture and applications," *IEEE Trans. Comput.*, vol. C-31, pp. 1054–1066, Nov. 1982.

[6] L. Snyder, "Introduction to the configurable highly parallel computer," *IEEE Comput.*, vol. 15, pp. 47–56, Jan. 1982.

[7] L. Lamport, "The parallel execution of DO loops," *Commun. ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974.

[8] E. J. Gallopoulos and S. D. McEwan, "Numerical experiments with the massively parallel processor," in *Proc. 1983 Int. Conf. Parallel Processing*, pp. 29–35.

[9] J. A. B. Fortes, "Algorithm transformations for parallel processing and VLSI architecture design," Ph.D. dissertation, Dep. Elec. Eng. Syst., Univ. Southern California, Los Angeles, Dec. 1983.

[10] D. I. Moldovan and J. A. B. Fortes, "Partitioning of algorithms for execution in fixed size VLSI architectures," to be published; see also ——, Dep. Elec. Eng. Syst., Univ. Southern California, Los Angeles, Tech. Rep. PPP-5, 1983.

[11] J. P. Hayes, "A graph model for fault tolerant computing systems," *IEEE Trans. Comput.*, vol. C-25, pp. 875–884, Sept. 1976.
[12] A. I. Baqai and T. Lang, "Reliability aspects of the Illiac IV computer," in *Proc. 1976 Int. Conf. Parallel Processing*, pp. 123–131.
[13] D. K. Pradhan, "Fault-tolerant architectures for multiprocessors and VLSI systems," in *Proc. 13th Fault-Tolerant Comput. Symp.*, 1983, pp. 436–441.
[14] A. L. Rosenberg, "The Diogenes approach to testable fault tolerant arrays of processors," *IEEE Trans. Comput.*, vol. C-32, pp. 902–910, Oct. 1983.
[15] I. Koren and M. A. Breuer, "On area and yield considerations for fault tolerant VLSI processor arrays," Dep. Elec. Eng.-Syst., Univ. Southern California, Los Angeles, Tech. Rep. DISC 82-5, Nov. 1982.
[16] T. E. Mangir and A. Avizienis, "Fault-tolerant design for VLSI: Effect of interconnect requirements on yield improvement of VLSI designs," *IEEE Trans. Comput.*, vol. C-31, pp. 609–615, July 1982.
[17] C. S. Raghavendra and T. E. Mangir, "On the VLSI implementation of fault-tolerant architectures," in *Proc. 1983 Int. Conf. Comput. Des., VLSI Comput.*, pp. 744–747.
[18] D. Fussel and P. Varman, "Fault-tolerant wafer scale architectures for VLSI," in *Proc. 9th Symp. Comput. Architecture*, 1982, pp. 190–198.
[19] J. W. Greene and A. El Gamal, "Area and delay penalties in restructurable wafer scale arrays," in *Proc. 3rd Caltech Conf. VLSI*, 1982.
[20] K. Smith, "Serial convolver to be fault-tolerant," *Electron.*, p. 76, Aug. 1983.
[21] K.-H. K. Kuang and J. A. Abraham, "Low cost schemes for fault tolerance in matrix operations with processor arrays," in *Proc. 9th Symp. Comput. Architecture*, pp. 330–337.
[22] R. H. Kuhn, "Yield enhancement by fault-tolerant systolic array (Summary)," in *USC Workshop VLSI Modern Signal Processing*, Nov. 1982, pp. 145–152.
[23] J. A. B. Fortes and C. S. Raghavendra, "Dynamically reconfigurable fault-tolerant array processors," in *Proc. 14th Fault-Tolerant Comput. Symp.*, 1984.
[24] J. F. Meyer, "On evaluating the performability of degradable computer systems," *IEEE Trans. Comput.*, vol. C-29, pp. 720–731, Aug. 1980.
[25] M. D. Beaudry, "Performance-related reliability measures for computing systems," *IEEE Trans. Comput.*, vol. C-27, pp. 540–547, June 1978.
[26] H. T. Kung and M. S. Lam, "Wafer-scale integration and two-level pipelined implementation of systolic arrays," *J. Parallel Distrib. Comput.*, vol. 1, no. 1, pp. 32–63, 1984.
[27] R. H. Kuhn, "Optimization and interconnection complexity for parallel processors, single stage networks and decision trees," Ph.D. dissertation, Dep. Comput. Sci., Univ. Illinois, Urbana-Champaign, Rep. 80-1009, Feb. 1980.
[28] D. I. Moldovan, "On the analysis and synthesis of VLSI algorithms," *IEEE Trans. Comput.*, vol. C-31, pp. 1000–1009, Dec. 1982.
[29] J. A. B. Fortes and D. I. Moldovan, "Parallelism detection and transformation techniques useful for VLSI algorithms," *J. Parallel Distrib. Comput.*, 1985, to appear.
[30] D. K. Pradhan, "Fault-Tolerant multiprocessor and VLSI-based system communication architectures," in *Fault-Tolerant Computing: Theory and Techniques*. Englewood Cliffs, NJ: Prentice-Hall, ch. 6, to appear.
[31] J. A. B. Fortes and C. S. Raghavendra, "Gracefully degradable array processors," School Elec. Eng., Purdue Univ., West Lafayette, IN, Tech. Rep. TR-EE-84-15, 1984.

Jose A. B. Fortes (S'80–M'84) was born in Luanda, Angola, on August 25, 1954. He received the Licenciatura em Engenharia Electrotecnica degree from the Universidade de Angola in 1978, the M.S. degree in electrical engineering from the Colorado State University, Fort Collins, in 1981, and the Ph.D. degree in electrical engineering from the University of Southern California, Los Angeles, in 1983.

Since 1984 he has been an Assistant Professor in the School of Electrical Engineering, Purdue University, West Lafayette, IN. His research interests include architectures, languages and algorithms for parallel processing, fault-tolerant computing, and design automation.

Dr. Fortes is a member of the Association for Computing Machinery.

C. S. Raghavendra (S'80–M'82), for a photograph and biography, see p. 55 of the January 1985 issue of this TRANSACTIONS.

REFERENCE NO. 2

Fortes, J. A. B., Milutinovic, V., Dick, R., Helbig, W., and Moyers, W., "A High-Level Systolic Architecture for GaAs," 1986 International Workshop on High-Level Computer Architecture, Proc. of the 19th Hawaii International Conference on System Sciences (HICSS), pp. 253-258, January 1986. (before SDIO support)

Presented at the 19th Hawaii Int'l Conference on System Sciences, January, 1986 (recipient of honorary mention)

A High-Level Systolic Architecture for GaAs

J. A. Fortes
V. Milutinović
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

R. J. Dick
W. A. Helbig
W. D. Moyers
Microelectronics Laboratory
Advanced Technology Laboratories
RCA/Aerospace and Defense
Moorestown, New Jersey 08057

I. Introduction

This paper describes the design of a high-level GaAs systolic architecture intended for use in a new generation of advanced communication and radar systems. The purpose of the systolic array is to provide those systems with a real time adaptive filter signal processing capability with very large throughput and short response time. The high performance characteristics of the architecture described here result from the use of fast GaAs technology, an efficient and numerically stable algorithm, and an innovative systolic array architecture. The following sections of this paper describe these design choices and how they interplay and converge into a solution which meets the stringent requirements of communication and radar systems of the 1990's.

Section II summarizes the basic advantages and disadvantages of GaAs. Section III explains in more detail the adaptive filter signal processing problem as it occurs in the targeted applications and describes the algorithm used for its solution. The global systolic architecture is described in section IV, as well as the design of the individual processor elements of the array. Section V is dedicated to considerations on fault-tolerance, modularity and extensibility, and, architectural impact of GaAs technology. Section VI is dedicated to conclusions.

II. Why GaAs?

Gallium Arsenide (GaAs) technology has recently shown rapid increases in maturity [1]. In particular, the advances made in digital chip complexity have been enormous. This progress is especially evident in two types of chips: static RAMs and gate arrays. In 1983, static RAMs containing 1K bits were announced. One year later 4K-bit versions were presented. Several companies were working on 8K-bit designs in 1985. Gate arrays have advanced from a 1000-gate design presented in 1984 to a 2000-gate design announced in 1985. With this enormous progress underway, it is now appropriate to consider the use of this new technology in the implementation of high-performance systolic arrays.

GaAs technology generates high levels of enthusiasm primarily because of two advantages it enjoys over Silicon. These are higher speed and greater resistance to adverse environmental conditions.

GaAs gates switch faster then Silicon bipolar Transistor-Transistor Logic (TTL) gates by at least an order of magnitude [2]. These switching speeds are even faster than those attained by the faster Silicons, CMOS and bipolar ECL but at lower power levels [2] [3]. For this reason, GaAs is seen to have applications in computer designs in several computationally-intensive areas. In fact, it has been reported that the Cray-3 will contain GaAs parts.

GaAs also enjoys greater resistance to radiation and temperature variations than does Silicon. GaAs successfully operates in radiation levels of 10 to 100 million RADs [2]. Its operating temperature range extends from -200 to 200 degrees centigrade [2]. Consequently, GaAs has created great excitement in the military and aerospace markets.

Unfortunately, GaAs is also characterized by some undesirable properties. Two significant areas where GaAs is inferior to Silicon are cost and transistor count capability.

The higher cost of GaAs chips is largely the result of the higher cost of GaAs material itself and the lower yield of GaAs chips. GaAs material is more expensive than Silicon. Also, since GaAs is a compound material, additional processing is required to create it and to verify

Proceedings of the 1986 International Workshop on High-Level Computer Architecture, Honolulu, Hawaii, January 1986.

its composition. The lower GaAs yield is also due to multiple influences. First, although improvements are being made in this area, GaAs is characterized by a higher density of dislocations than Silicon. Second, in order to achieve working devices with adequate noise margins, very fine control of circuit parameters is required, and this is not yet easily achieved [2]. Finally, the high brittleness of GaAs contributed to its high cost due to its increased breakage [4]. Currently, GaAs chips are roughly two orders of magnitude more expensive than their Silicon counterparts, however, this difference should narrow to possibly one order of magnitude or less by the end of this decade.

Transistor count limitations of GaAs are attributed to both yield and power considerations. The relatively low yield of GaAs chips forces designers to consider chips with smaller area (therefore lower transistor count) in order to remain cost-effective. Although GaAs gates require less power than their Silicon counterparts when operating at similar speeds, GaAs gates do consume considerably more power than slower Silicon MOS gates. Because of the thermal management problem this creates, fast GaAs chips cannot match the transistor count potential of Silicon chips.

It is believed that these four GaAs-Silicon differences are not of a temporary nature, but instead result from inherent differences between GaAs and Silicon materials. Conclusions which are based on these four fundamental characteristics will remain valid even as GaAs technology matures.

Because of these GaAs-Silicon differences, it is not sufficient to merely copy existing Silicon designs into GaAs in order to obtain optimal GaAs performance. The GaAs environment presents the computer architecture designer a new set of challenges. However, the rewards of successfully exploiting this new environment are substantial. With the high speeds which characterize GaAs and the recent examples of GaAs chips with VLSI levels of integration (>10,000 transistors), we are presently on the verge of achieving, with a single-chip processor, speeds for scalar operations typical of present-day supercomputers.

III. Applications and Algorithms for Adaptive Filter Signal Processing

Two similar adaptive filter signal processing applications exist for the proposed systolic array processor; adaptive antenna array beamforming and adaptive doppler/spectral filtering. In an adaptive antenna array, the phase and amplitude of the waveform incident upon each antenna element within the receiving aperture are adjusted to control properties of the far field antenna pattern such as maximum gain, low sidelobe levels, narrow mainbeam and pattern nulls in the angular direction of interfering signals. In adaptive doppler/spectral filtering, the phase and amplitude of each waveform sample in the time domain are adjusted to control properties of a filter in the frequency domain such as maximum gain at the desired frequency, low filter sidelobe levels, narrow main filter response and sidelobe nulls at the frequencies of interfering signals. In both cases, the amplitude and phase adjustments (i.e., complex weights) are determined by processing all voltage samples in real time. In either case, an adaptive Discrete Fourier Transform filter operating on a number, N, of complex voltage samples may be used. The N input samples, frequently multiplied by a window weighting function to control filter sidelobe levels in the transform plane, form a complex N-dimensional vector, x. The filter output is formed from the product of the weight vector, w, and the signal vector, x. The optimum weight vector is

$$w = R^{-1}s^* = M^{-1}v^*$$

where s is the N-dimensional steering vector defining the antenna direction or doppler frequency peak response and $R = \overline{x^*x^t}$ is the N by N covariance matrix of the signal, whose ij-th component is $r_{ij} = \overline{x_i^*x_j^t}$. M and v are rescaled versions of R and s, respectively (for full details, the reader is referred to [5]).

The adaptation process requires the inversion of an NxN complex matrix in real time or, equivalently, the solution of a set of simultaneous linear equations. This problem has been one of the main concerns in numerical analysis and control theory for many years. A number of algorithms have been developed and studied with the adaptive array application in mind [5-6]. However, for the size of the future systems for both communications and radar applications, the complexity of solving the associated equations grows rapidly, implying the need for utilizing only the most efficient algorithms. Tradeoffs between hardware complexity and convergence time, maximization of signal-to-noise ratio and minimization of the effect of error sources on the adaptive process must be seriously considered for each application. After analysis and simulation of the candidate algorithms, one of the direct Matrix Square Root (MSR) algorithms proposed in [6] was selected as the most adequate.

The MSR algorithms involve directly updating the sample matrix square root factors U, D that evolve from a Cholesky factorization of the positive semi-definite M matrix.

$$M = UDU^T$$

where U is a lower triangular matrix with unit diagonal elements and D is a diagonal matrix with positive or zero diagonal elements. U and D are defined as

$$M_K = U_K D_K U_K^T = \sum_{i=1}^{K} x_i x_i^T$$

and are recursively updated as

$$U_K D_K U_K^T = U_{K-1} D_{K-1} U_{K-1}^T + x_K b x_K^T$$

where b is a scalar set to one initially. The matrix inversion needed to solve for the optimal weights, w_0, can be reduced to a single back-substitution. Distinct MSR algorithms differ only in the values used for the diagonal elements u_{ii} and/or d_i. The chosen MSR algorithm results from letting $u_{ii} = 1$.

IV. The Basic Systolic Architecture

The algorithm lends itself to a systolic array realization. The triangular structure of the array reflects the matrix triangularization step, characteristic of the algorithm. The array consists of a triangular grid with N(N-1)/2 nodes or elements (N being the size of the original matrix). With regard to the computation involved at each of them, the elements are of two types: the elements along the diagonal and the others.

Three Systolic Waves

Assuming initially a fully systolic realization, the algorithm iteration requires three systolic waves of computation (they are shown for the simple case of N=6 in Figures 1 through 3):

1) Wave 1, covariance matrix updating, starts from the top-left element and propagates toward the right and the bottom; the data vector x enters the array from the top,

2) Wave 2, first step of the back substitution, propagates, as wave 1, from the top left element and generates an intermediate vector 2 used next to compute the weights w,

3) Wave 3, second step of the back substitution, propagates backward from the bottom element of the array, and generates the weights sequentially.

One notes that waves 1 and 2 can be run simultaneously, but wave 3 must wait for the completion of the previous two to start backward. Not only that, wave 3 has to operate on old covariance values: as an example, when wave 3 reaches the last row at the top, it expects to find the U values that were there when the data vector X, for which the weights are now being computed, first entered the array. This requires memory within the array.

Given the array fully populated with processing elements and necessary memory, each algorithm iteration will perform two passes through the array (Figure 4):

- The first pass (including waves 1 and 2) proceeds from the top and left, down and right at a 45 degree angle The data vector is fed in parallel and at a 45 degree angle into the array, i.e., each data element in a same vector enters the array one cycle after the input of the previous data element in the same vector. The process then proceeds at an array clock interval determined by the longest computation time in any array element, which, in this case, happens to be that of the diagonal element. Concurrently, the updated covariance matrix U values are stored into each cell, as in a shift register.

- the second pass (generation of the weights w's) can start at the same time the last diagonal element D is processed (same clock); now the computation proceeds backward also at a 45 degree angle starting from the last column and last row on the right. At each clock a new weight is computed. Note in Figure 4 the uneven but predictable length of the "shift register" in each array element.

From Figure 4, one can assess easily not only the memory requirements but also the latency time (of the order of 2N) between the time a new data vector enters the array and the time the last of the weights w's is released.

Figure 1 also shows the organization of the ICs within the systolic processor. The processor elements are arranged as a right triangle. Calculations within the triangle ripple from top to bottom (root covariance update and first step of back substitution) and from right to left (second step of back substitution). Data flows horizontally, vertically, and downward along the outside diagonal. One important feature of the array is that the

autocovariance values remain stationary within the array, so that no busses are required to transmit them to other parts of the array.

The array is constructed of two cell types, which are designated SAA-1 and SAA-2. The SAA-1 cells perform calculations needed by the root covariance update and the first step of the back substitution.

The SAA-2 cells are involved in both the root covariance updates and both of the steps of back substitution. The root covariance values are kept within the SAA-2 cells. For purposes of pipelining the second step of the back substitution, each SAA-2 cell contains a FIFO register which delays these U-values for the necessary cycles.

V. The Fault-Tolerant, Expandable, GaAs Systolic Array

Fault-tolerance is achieved by periodically testing the array and dynamically reconfiguring it when a fault is detected. To avoid performance degradation, spare columns and rows are provided to allow for the logical removal of faulty processing elements. If a cell in row i fails then both row i and column i are bypassed and logically replaced by the neighbor column and row, respectively. Figure 5 show the basic array augmented with spare rows and columns and figure 6 illustrates the reconfiguration of a (128x128) triangular array with an extra row and one extra column and one faulty processor. In the worst case fault distribution (i.e., all faults occur in different rows and columns), up to K faults can be tolerated if K spare columns and K spare rows are provided. To tolerate K worst-case faults in a system for N degrees of freedom the percentage of additional hardware required is $100 \times \frac{K(2N + K + 1)}{N(N-1)}\%$. For example, for N=12 and K=1, 20% redundancy is required. On-cell multiplexers set by the array control system are used to bypass rows and/or columns of the systolic array. Fault detection is done by interleaving test vectors with the input data and checking the output generated by the array for the test inputs against the expected results.

The occurrence of a fault after all spare rows and columns have been used does not have to cause the crash of the system. Graceful degradation is possible in two ways: (a) by reducing throughput and (b) by eliminating degrees of freedom. If processor in row i and column j fails then (a) the row i can be bypassed and a neighbor row is time-multiplexed to replace row i or (b) row i and column j can be bypassed and the corresponding degree of freedom ignored. Notice that a reduction in throughput may require the system to ignore some samples but the degradation affects all weights instead of simply eliminating one.

An alternative to the use of test vectors and complex diagnostics for the detection and location of faults consists of using actual receiver sampled data and time redundancy. The basic idea is to multiplex the systolic array in time so that the same input samples are processed twice. However, for the second processing cycle, the samples are circularly shifted so that column i of the array receives the same data received by column i-1 in the first processing time (column 1 would receive the same data received by column N in the previous step). Internally, each processor can then compare its result with the result computed by the neighbor processors for the same data.

Figure 7 shows a (6x6) square array module as an extension of the (6x6) basic triangular array. Note that: (1) the diagonal elements are capable of performing as SAA-1 or SAA-2 cells and (2) the upper triangle of SAA-2 elements in the basic array is replicated below the diagonal of square array module. The basic idea underlying the extensibility and universality of this square array module is illustrated in Figure 5. This figure shows how a large triangular array can be generated by replicating the square array module. The replication can be done in time, i.e., by time multiplexing the square array so that it emulates the large triangular array. The replication can also be done in space, i.e., several identical modules are simply tiled together until the large triangle is covered. Partial space and partial time replication is also possible. Thus, the square array module can serve as the building block the systems with different customer requirements and intended for different applications. These basic ideas are similar to those discussed in [7].

VI. Conclusions

We described the design of a high level systolic architecture for adaptive signal processing in high performance advanced communication and radar systems. The main characteristics are extremely high throughput, fast response time and high reliability as result of marrying advanced GaAs technology, a sophisticated algorithm and innovative concepts in computer architecture and fault-tolerance.

References

[1] Milutinović, V., Fura, D., Helbig, W., "Impacts of GaAs on Microprocessor Architecture," *Proceedings of the ICCD 85*, Port Chester, New York, October 1985, pp. 30-40.

[2] Eden, R. C., Livingston, A. R., Welch, B. M., *Integrated circuits: the case for gallium arsenide*, IEEE Spectrum, Vol. 9, No. 12, December 1983, pp. 30-37.

[3] DiLorenco, J. V., Fowlis, D. C., Hewitt, B. S., Hou, T. W., Mogab, L. J., Roman, B. J., "GaAs-Status and Directions," *Proceedings of the ICCD 85*, Port Chester, New York, October 1985, pp 371-383.

[4] Heagerty, W., GaAs Seminar presented at Purdue University, January 1985.

[5] Manzingo, R., and Miller, T., "Introduction to Adaptive Arrays", John Wiley and Sons, New York, 1980.

[6] Bierman, G. J., "Factorization Methods for Discrete Sequential Estimation," Academic Press, 1977.

[7] Fortes, J. A., and Raghavendra, C. S., "Gracefully Degradable Processor Arrays", IEEE Transactions on Computers, November 1985.

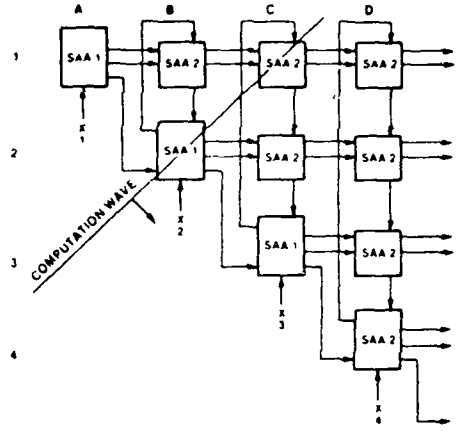


Figure 1. Root covariance update data flow.

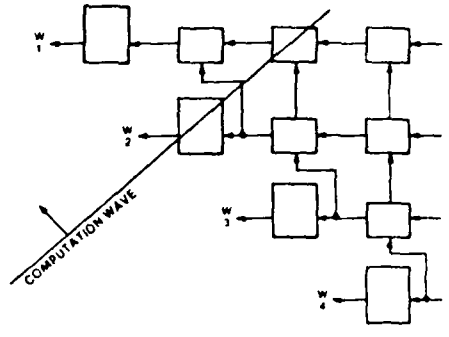


Figure 3. Back substitution, step 2.

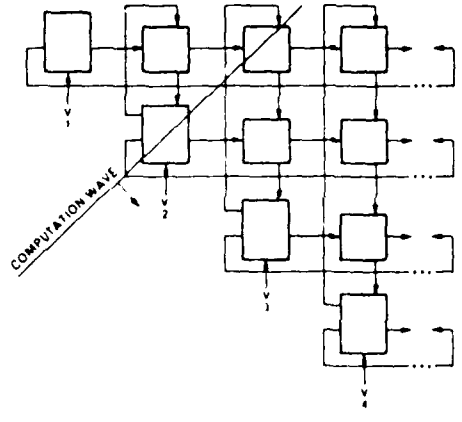


Figure 2. Back substitution, step 1.

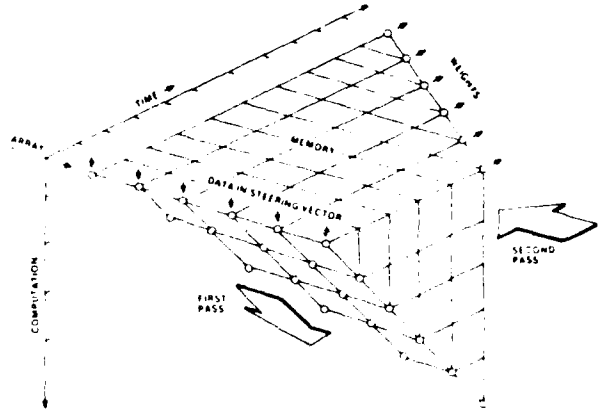


Figure 4. Systolic array operation (3-dimensional representation).

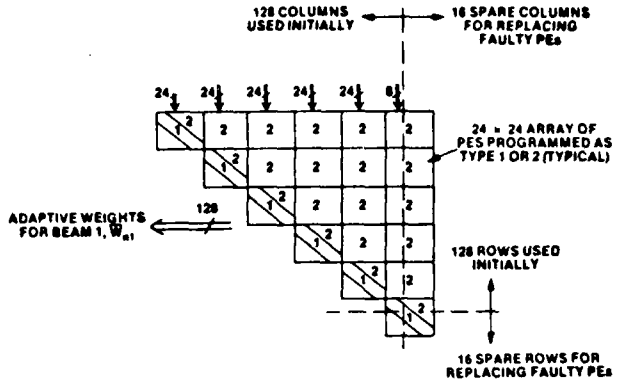


Figure 5. (128x128) basic array augmented with 16 spare rows and 16 spare columns.

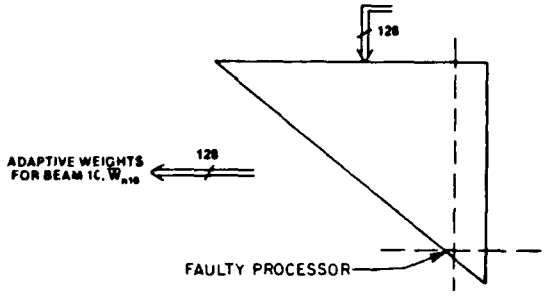


Figure 6. Reconfigured (128x128) array with one spare row and one spare column and a faulty processor.

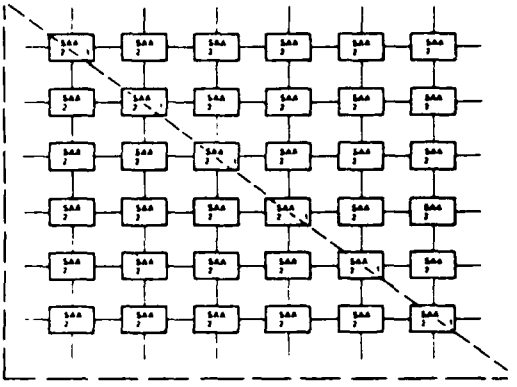


Figure 7. A (6x6) square array module as a result of extending a (6x6) triangular basic array by (a) designing the diagonal elements so that they can behave as either SAA-1 or SAA-2 cells and (b) adding a lower triangle of SAA-2 cells.

REFERENCE NO. 3

Fortes, J. A. B., "Algorithm Reconfiguration Techniques for Gracefully Degradable Processor Arrays," International Workshop on Systolic Arrays, July 1986.

Fortes, J. A. B., "Algorithm Reconfiguration Techniques for Gracefully Degradable Processor Arrays," in "Systolic Arrays," W. Moore, A. McCabe, R. Urquhart, editors, Adam Hilger, pp. 259-268, September 1986.

3.6 ALGORITHM RECONFIGURATION TECHNIQUES FOR GRACEFULLY DEGRADABLE PROCESSOR ARRAYS

Jose Fortes

INTRODUCTION

Operational fault-tolerance in VLSI/WSI processor arrays remains an important obstacle to the widespread use of such architectures. In particular, graceful degradation is hard to achieve, thus implying the need for large amounts of redundancy. Without graceful degradation, after redundancy is exhausted, any additional fault causes the entire system to fail, a unacceptable fact for the very large processor arrays made possible by VLSI/WSI. Some solutions have been proposed for this problem. In these relatively few fault tolerance schemes, graceful degradation is achieved at the cost of large losses in throughput or response time, costly additional interconnect, complex switching mechanisms and/or involved control schemes.

A promising approach to this problem relies on using simple on-line algorithm reconfiguration techniques together with simple hardware reconfiguration mechanisms. In essence, algorithms are reconfigured so that they can execute on the same processor array after the occurrence of faults and possible removal of processing elements.

In the space allowed, this paper shows how rational quasi-affine (RQA) algorithm transformations can be used to devise such reconfiguration schemes. It describes the mathematical framework underlying our techniques, discusses examples and three approaches based on a common RQA transformation which yields optimal graceful degradation and briefly discusses extensions of our approach to 2-dimensional arrays.

MATHEMATICAL FRAMEWORK

In the following discussion we use Z and I to denote the set of integers and the set of nonnegative integers, and use Z^n and I^n to refer to their corresponding nth Cartesian powers. We will consider only q-dimensional processor arrays, where $q = 1,2$. We see a processor array as a finite q-dimensional grid in which each integer point is a vector index of a processor and a set of vectors (the interconnection primitives) which describes the regular pattern of interconnections of the array. The following definition formalizes this view.

Definition 1 - A *processor array* is a tuple (L^q,P) where q is the *dimension* of the array, $L^q \subset Z^q$ is the *index set* and $P \in Z^{(q \times r)}$ is a matrix of $r \in I$ *interconnection primitives*.

Thus, in a processor array (L^q,p), the processor with index $\bar{\ell} \in L^q$ is connected to a processor with index $\bar{\ell}' = \bar{\ell} + \bar{p}$, $\bar{p} \in P$, if $\bar{\ell}' \in L^q$, and it is connected to an input-output port otherwise.

The research was supported in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contract no. 00014-85-k-0588.

Example 1: The linear processor array shown in figure 1(a) can be described by (L^1,P) where $L^1 = \{\ell : 0 \le \ell \le 3\}$ and $P = [0\ 1\ -1]$. The square processor array of figure 1(b) can be described by (L^2,P) where $L^2 = \{(\ell_1,\ell_2) : 0 \le \ell_1, \ell_2 \le 3\}$ and

$$P = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix}$$

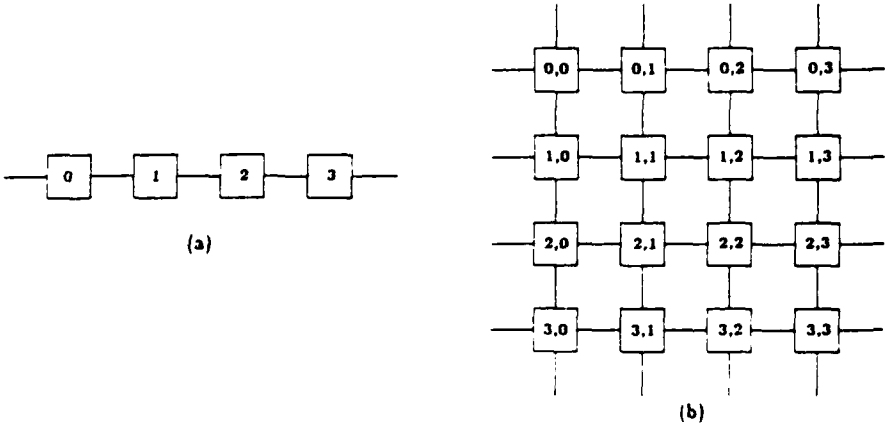


Figure 1 (a) - A linear processor array; (b) - A (4×4) orthogonal processor array

The execution of an algorithm on a given array can be thought of as an ordered set of instantiations of the array, each of which contains an assignment of computations to processors at a particular time of execution. Consequently, we see an array algorithm as a $(q+1)$-dimensional grid in which (a) for $q = 1$, each integer point $(j_1j_2)^T$ indexes a computation at time j_1 and processor j_2 and (b) for $q = 2$, each integer point $(j_1j_2j_3)^T$ indexes a computation at time j_1 and processor $(j_2j_3)^T$. In addition, we associate with each array algorithm a set of vectors (called dependence vectors) which describes the pattern of generation and use of data in time and space. In other words, if a computation with index $\bar{j}$ generates a value used in computation with index $\bar{j}'$, then $\bar{j}' - \bar{j}$ is a dependence vector. Clearly, the first entry of any dependence vector must be ≥ 1 (i.e., at least one unit of time separates generation and use of a variable), and the vector corresponding to the other entries must correspond to a linear combiuation of interconnection primitives (i.e., a path connecting the processors where the variable is generated and used). Assuming that communication (over a single interconnection primitive) and execution of a computation take one unit of time, the number of interconnection primitives used to communicate a result from computation with index $\bar{j}$ to computation with index $\bar{j}'$ must also be less than or equal to the first entry of the dependence vector $\bar{j}' - \bar{j}$ (i.e., the interval of time between the computations). These considerations translate into the following definition of array algorithm.

Definition: Given an array (L^q,P), $P \in Z^{[q\times r]}$. An *array algorithm* is a tuple (J^{q+1},D), where $J^{q+1} \subset Z^{q+1}$ is an *index set* of the algorithm, and $D \in Z^{[(q+1)\times m]}$ is a *dependency matrix* of $m \in I$ dependence vectors such that

$$d_{ii} \ge 1 \quad i = 1,\ldots,m \tag{1}$$

and

$$\left[d_{ab} \times \quad 2,q+1, i = 1,\ldots,m\right] = PK \quad \text{for} \quad K \in I^{(r\times m)} \text{ such that } \sum_{j=1}^{r} k_{ji} \le d_{ii} \quad , \quad i = 1,\ldots,m \tag{2}$$

In this definition of array algorithm we represent only the structure of the algorithm and abstract from the actual computations being performed. This is adequate because we are essentially worried with problems of matching computational structures. Also, input and output data are not

explicitly represented because they can be treated as generated data (i.e., for a given processor receiving data from other processors there is no distinction between data generated and data "passed" by those processors). Finally, the description of dependences would be more precise if to a given dependence vector we associate the index point where the dependence is valid. This complication turns out to be unnecessary for the derivation of our main results.

Example 2: Consider the linear systolic array shown in figure 2 for convolution computation (proposed in [Li & Wah 85]). It computes the recurrence

$$\begin{aligned} x_{i+k-1}^{i,k} &= x_{i+k-1}^{i+1,k-1} \\ a_k^i &= a_k^{i+1} \\ y_i^k &= y_i^{k-1} + a_k^i x_{i+k-1}^{i,k} \quad 0 \le i \le n, \quad 0 \le j \le m \end{aligned} \tag{3}$$

for m=3 and n=5. The algorithm executes in 9 units of time and some processors are idle only during the initial and final phases of the computation. Variables "a" stay always in the same processor and variables "y" and "x" move to the right neighbor processor every one and two units of time, respectively. This array algorithm can be described by (J^2,D) where $J^2 = \{(j_1,j_2)^T \quad 0 \le j_2 \le 3, j_2 \le j_1 \le 5 + j_2\}$, i.e., j_1 = time and j_2 is the processor index, and

$$D = \begin{matrix} \\ (j_1) \\ (j_2) \end{matrix} \begin{matrix} \begin{matrix} 1 & 2 & 1 \end{matrix} \\ \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \\ \begin{matrix} (a) & (x) & (y) \end{matrix} \end{matrix} \tag{4}$$

where the second row corresponds to

$$PK = \begin{bmatrix} 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}. \tag{5}$$

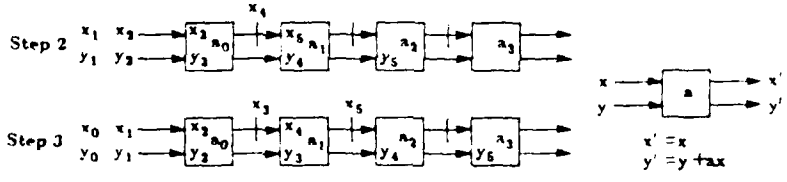


Figure 2 - Systolic array for convolution ($n = 5$, $m = 3$).

Since we are interested in general purpose algorithm reconfiguration schemes, we will consider "worst case" array algorithms. In other words, we will consider algorithms which, at any time during execution, use all processors and all interconnection links of the array. Thus, for a linear array algorithm which takes T units of time to execute on a linear array with N processors we have (J^2,D) where

$$J^2 = \left\{(j_1,j_2)^T : \ 0 \le j_1 \le T-1 \ , \ 0 \le j_2 \le N-1\right\} \quad \text{and} \quad D = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \end{bmatrix}$$

Figure 3(a) illustrates this for T = 3, N = 4. Similarly, for a square orthogonal array algorithm with execution time T and (N×N) processors we have (J^3,D) where

$$J^3 = \left\{(j_1,j_2,j_3)^T : \ 0 \le j_1 \le T-1 \ , \ 0 \le j_2,j_3 \le N-1\right\} \quad \text{and} \quad D = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & -1 & 0 \end{bmatrix}$$

Hereon and unless otherwise stated, we use the term array algorithm to mean a worst case array algorithm. We are interested in the case when, due to the failure of one processor, the original array algorithm must be executed by a smaller processor array. This requires that the algorithm be reconfigured, i.e., that operations initially allocated to faulty processors be remapped into the

operational processing elements. This is equivalent to saying that we need to obtain a new array algorithm by transforming the original array algorithm. Algorithm transformations have been studied extensively and are reported in [Fortes & Raghavendra 85] and [Moldovan & Fortes 86] and the references thereof. In [Fortes & Raghavendra 85] it was shown that a simple transformation can be used to reconfigure array algorithms with unidirectional data movements. It was also shown that any array algorithm can be transformed into an equivalent array algorithm with unidirectional data movements, thus making that scheme generally applicable. One of the disadvantages of this approach is that the equivalent algorithm may be slower than the original one. Another disadvantage is the requirement for "wrap-around" links between processors at the boundaries of the array. To show the impact of this approach on the "degraded" performance of the array with faulty processors we need to discuss this scheme in more detail. Consider the case of a linear array algorithm which executes in T units of time on N processors. Assume that data movements are unidirectional and one processor fails. The remaining operational processors have virtual indices ranging from 0 to N-2. The reconfiguration is done by simply mapping a computation originally performed in processor j_2 at time j_1 (i.e., point (j_1,j_2)) into processor $j_2 \bmod (N-1)$ at time $j_1 + \left\lfloor j_2/(N-2) \right\rfloor T$. This means that response time is doubled and that the average throughput is halved.

The reconfiguration technique proposed in this paper does not suffer from the drawbacks discussed above. It evolved from the theory of linear algorithm transformations ([Fortes & Raghavendra 85], [Moldovan & Fortes 86]) whose basic ideas are explained next. The reorganization of an algorithm corresponds to a permutation of its index set and can be described as a linear transformation $T \in Z^{((q+1)\times(q+1))}$ such that T is nonsingular. The first row of T, denoted π, is referred to as time transformation and the remaining submatrix of T, denoted S, is called a space transformation. In other words, T reorganizes the algorithm so that a computation with index $\bar{j}$ in the original algorithm is executed at time $\pi\bar{j}$ and processor $S\bar{j}$ (i.e., the index in the transformed algorithm is $(\pi\bar{j},S\bar{j})^T$). Due to the linearity of the transformation, the dependence matrix of the transformed algorithm is simply TD, where D is the dependency matrix of the original algorithm. Of course, T must be selected so that the new algorithm is an array algorithm, i.e., (1) and (2) are satisfied. To illustrate this approach the reader can verify that

$$T = \begin{bmatrix} -1 & 1 \\ 0 & 1 \end{bmatrix}$$

can be used to transform the algorithm described by (3), for which

$$D = \begin{bmatrix} -1 & -1 & 0 \\ 0 & 1 & 1 \end{bmatrix},$$

into the array algorithm of figure 3 for which the dependence matrix is (4), i.e., TD.

NEW RECONFIGURATION SCHEMES

In this paper, we consider rational quasi-affine (RQA) transformations of the form $\left\lfloor T\bar{j} + \bar{t} \right\rfloor$, where $T \in Q^{((q+1)\times(q+1))}$, $\bar{t} \in Q^{(q+1)}$ and Q denotes the set of rational numbers. As for linear transformations, T consists of a time transformation π and space transformation S. Time transformations of this type are discussed in [Fortes & Parisi 84] and a full discussion of RQA mappings will appear in a forthcoming paper. Clearly, the class of RQA transformations is a superset of that of linear mappings mentioned in the previous section. However, RQA transformations for which T is nonsingular do not necessarily correspond to one-to-one mappings. Thus, before considering an RQA transformation, one must show that it indeed specifies an injective mapping. In addition, conditions similar to those used to select linear transformations must also be used to choose RQA mappings. As mentioned before, a valid algorithm transformation must yield a new algorithm for which the dependency matrix satisfies (1) and (2). For a linear transformation T, the new matrix is simply TD where D is the dependency matrix of the original algorithm. For RQA transformations, this is not true. However, it is still possible to define conditions which ensure that (1) and (2) are satisfied. Note that for any X, Y and W the value of $\lfloor X/W \rfloor - \lfloor Y/W \rfloor$ is either $\lfloor (X-Y)/W \rfloor$ or $\lceil (X-Y)/W \rceil$. Hence, for the transformation R mentioned above and any dependence $\bar{d} = \bar{j} - \bar{j}'$, we have that the value of

$R(\bar{j}) - R(\bar{j}')$ is $\left\|T\bar{d}\right\|$ where T is as defined above and the notation $\|\ \ \|$ means that any entry in $T\bar{d}$ can be replaced by either its ceiling or floor value. Thus, a valid RQA transformation must be such that $\|TD\|$ satisfy (1) and (2).

We start by discussing the case of linear array algorithms. Hereon, we will only consider RQA transformations of the type introduced in the next theorem. The theorem shows that such transformations are injective and, in addition, reversible in the integers. Afterwards we show that $\|TD\|$ satisfy (1) and (2). We use the symbol $\mathbb{R}$ to denote the set of real numbers and $\mathbb{R}^2 = \mathbb{R} \times \mathbb{R}$.

Theorem 1

Let $J^* \subset \mathbb{R}^2$ and $J = J^* \cap I^2$. Consider the RQA transformation $R : J \to L$ such that

$$R(\bar{j}) = \left\lfloor T\bar{j} + \bar{t} \right\rfloor \tag{6}$$

where

$$T = \begin{bmatrix} \pi \\ S \end{bmatrix} = \frac{1}{a-1} \begin{bmatrix} a & 1 \\ -1 & a-2 \end{bmatrix}, \quad a \in I, \ a \geq 2 \tag{7}$$

and

$$\bar{t} = \frac{1}{a-1} \begin{bmatrix} 0 \\ a-2 \end{bmatrix}$$

and

$$L = L^* \cap I^2, \ L^* = \left\{ \bar{\vartheta}^* : \bar{\vartheta}^* = T\bar{j}^* + \bar{t}, \ \bar{j}^* \in J^* \right\}.$$

The transformation R is a bijection.

Proof

We show that R is both an injection and a surjection and thus it must also be a bijection.

(a) R is an injection - By contradiction. Assume that $\bar{j}, \bar{j}' \in J, \bar{j} \neq \bar{j}'$ and $\bar{\vartheta} = R(\bar{j}) = R(\bar{j}') = \bar{\vartheta}'$. We show that this implies $\bar{j} = \bar{j}'$, i.e., $\bar{\delta} = \bar{j} - \bar{j}' = 0$. $R(\bar{j})$ can be reexpressed as

$$R(\bar{j}) = \bar{j} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \left\lfloor \frac{1}{a-1} \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} j_1 \\ j_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right\rfloor = \begin{bmatrix} \vartheta_1 \\ \vartheta_2 \end{bmatrix}$$

and, because $\lfloor -(x+1)/k \rfloor = -\left\lfloor 1 + \lfloor x/k \rfloor \right\rfloor$ for all x and k, we have

$$\vartheta_1 = j_1 + \left\lfloor (j_1 + j_2)/(a-1) \right\rfloor \quad \text{and} \quad \vartheta_2 = j_2 - \left\lfloor (j_1 + j_2)/(a-1) \right\rfloor. \tag{8}$$

The assumption $\bar{\vartheta} = \bar{\vartheta}'$ implies

$$\begin{cases} \delta_1 + \left\lfloor \dfrac{\delta_1 + j_1 + j_2 + \delta_2}{a-1} \right\rfloor - \left\lfloor \dfrac{\delta_1 + j_2}{a-1} \right\rfloor = 0 \\ \delta_2 - \left\lfloor \dfrac{\delta_1 + j_1 + j_2 + \delta_2}{a-1} \right\rfloor + \left\lfloor \dfrac{\delta_1 + j_2}{a-1} \right\rfloor = 0 \end{cases} \Longrightarrow \delta_1 + \delta_2 = 0$$

Substituting for $\delta_1 + \delta_2$ in the floor functions of the above equations implies $\delta_1 = \delta_2 = 0$, i.e., $\bar{\delta} = \bar{0}$.

(b) R is a surjection - Since $|T| = 1$, L^* has the same area of J^*. Thus, L^* cannot contain more integer points than J^* does. Since R is an injection it must also be a surjection (pigeonhole principle). Q E D

Now we show that $\{|TD|\}$ satisfies (1) and (2). In fact we have

$$TD = \frac{1}{a-1}\begin{bmatrix} a & 1 \\ -1 & a-2 \end{bmatrix}\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \end{bmatrix} = \frac{1}{a-1}\begin{bmatrix} a & a+1 & a-1 \\ -1 & a-3 & -(a-1) \end{bmatrix} \tag{9}$$

The worst case occurs when we take the floor of the first row entries and the ceiling of the absolute values in the second row. This corresponds to the case when the least time is available for data communications to take place. The resulting matrix is

$$\begin{bmatrix} 1 & 1 & 1 \\ -1 & 1 & -1 \end{bmatrix} \tag{10}$$

which clearly satisfies (1) and (2). Another possible matrix resulting from $\{|TD|\}$ corresponds to the case when the ceiling and floor functions are applied to the first and second rows, respectively. The resulting matrix is

$$\begin{bmatrix} 2 & 2 & 1 \\ 0 & 0 & -1 \end{bmatrix} \tag{11}$$

which, expectably, also satisfies (1) and (2). It also becomes obvious that, at different execution times, the same variables could move every unit of time, or move every two units of time or stay in the same processor for two units of time. This may suggest that data timing and movement are hard to predict. Fortunately this is not the case and much information can be derived from our formalism. For example it can be shown that for any dependence $\bar{d}$, the only values of $\left\{\left|T\bar{d}\right|\right\}$ which occur in the new algorithm correspond to those where the ceiling function is applied to one entry of $\bar{d}$ and the floor function is applied to the other entry (a consequence of $\vartheta_1+\vartheta_2 = j_1+j_2$ from (8)). In other words, the new dependencies correspond to vectors present in the matrices

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 2 & 2 & 1 \\ -1 & 0 & -1 \end{bmatrix}.$$

An important implication is the fact that buffering (local memory) is required (for the first two columns of the second matrix). As another example, consider the first 2 entries of the second row of (9) and consider the question of finding out when, for a given processor, the corresponding variable moves or remains in the processor (i.e., when the ceiling and floor values are valid). It is possible to show that the ceiling function is valid (a-2) out of every (a-1) times for the first entry and (a-3) out of every (a-1) times for the second entry (in a periodic manner). Similar deductions can be done with respect to timing and data movement for other individual processors and variables.

As for linear transformations, graphical representations of RQA mappings are quite insightful. For linear transformations, the locus of $\pi\bar{j}$ = constant in the index set of the original algorithm corresponds to a plane or line which describes a computational wavefront (i.e., the execution of computations whose indices belong to the wavefront takes place at the same time). Likewise, the locus of $S\bar{j} = \bar{p}_0$ contains the indices of computations executed by processor $\bar{p}_0$. For the case of RQA mappings, the locus of $\left\lfloor \pi\bar{j} + t_1 \right\rfloor$ = constant corresponds to several consecutive wavefronts which are computed simultaneously. The locus of $\left\lfloor S\bar{j} + t_2 \right\rfloor = \bar{p}_0$ contains all indices of computations executed by processor $\bar{p}_0$.

The transformation R given by (6) is the basis for three reconfiguration schemes to be described later in this paper. In general, when reconfiguring an algorithm for execution on a linear array with N processors, one of which is faulty, we assign the value N to the parameter a in R. In order to illustrate the concepts introduced above, we now discuss an example for which N = a = 5. Thus, we have, from (6),

$$R(\bar{j}) = \left\lfloor \frac{1}{4}\begin{bmatrix} 5 & 1 \\ -1 & 3 \end{bmatrix}\bar{j} + \frac{1}{4}\begin{bmatrix} 0 \\ 3 \end{bmatrix} \right\rfloor \quad \text{and} \quad TD = \frac{1}{4}\begin{bmatrix} 5 & 6 & 4 \\ -1 & 2 & 4 \end{bmatrix}.$$

Figure 3 illustrates several ideas discussed before. Figure 3(a) shows the original worst case algorithm before any fault occurred. The dots represent computation indices and the arrows depict

data movement. We assume an execution time of a-1 = 4 units of time (the reason will become clear later). Figure 3(b) depicts the computational wavefronts (full lines) and the time when they are executed as determined by R (the symbol τ denotes *execution time in the new algorithm*). The broken lines contain indices of computations executed by the same processor (we use the symbol ρ for processor indices). Figure 3(c) shows the same information as figure 3(b) in different form together with the movement of data. The "crosshatched" bars contain indices of computations executed at the same time. The "dotted" bars contain indices of computations executed by the same processor. The following properties of the reconfigured algorithm discussed previously are now readily apparent. First, notice that only N-1 = 4 processors are used, i.e. the faulty processor is not required. Second no arrow crosses a dotted bar, i.e., all communication occurs between operational neighboring processors (we will comment on necessary reconfiguration hardware later). Third, some data must be buffered in each processor for two units of time. This occurs whenever an arrow crosses a crosshatched bar. As predicted, stationary data in the original algorithm remains in the same processor three out of four steps in the new algorithm (i.e., (a-2)/(a-1) = 3/4), data shifting to the right stays in the same processor two out of four steps (i.e., (a-3)/(a-4) = 2/4) and data shifting left moves in every step. Finally, from figure 3(b), it is clear that each time and "space" wavefront contains a single index, i.e., each computation belongs to a different wavefront. Since the smallest entry in the first row of the matrix given by (9) is a-1 = 4, we know that at most that many computations can occur simultaneously ([Fortes & Parisi 84]). Hence, only that many processors are needed.

Note that this reconfiguration scheme is optimal since no operational processor is ever idle, the execution time is increased the least possible, i.e., by a factor given by the ratio of the number of all processors over the number of operational processors (5/4 in this case). It is interesting to note that a necessary condition for a transformation to preserve the product of the number of processors by execution time is that it be reversible in the integers. Theorem 1 proved that R satisfies this condition. With respect to throughput, assume that the original algorithm accepted a new input and generated a new output every unit of time. The new algorithm accepts 4 new inputs and generates 4 new outputs every 5 units of time, which is also optimal.

Now let us consider the general case when the original array algorithm has execution time larger than a-1 = 4 units of time. The algorithm which results from applying R has the same characteristics as before including the fact that at most 4 processors are used at any time. However, the indices of these processors are not restricted to range from 0 to 3 and more than 4 processors would be required. To solve this problem, we consider three possible schemes based on the transformation R. We describe them next and discuss their characteristics afterwards. Let (τ,ρ) denote the image in the reconfigured algorithm of the index $\bar{j} = (j_1, j_2)$ in the original array algorithm. The three possible schemes are as follows:

Scheme 1 - $(\tau,\rho) = (\theta_1, \theta_2 \bmod(a-1))$ where $(\theta_1, \theta_2) = \bar{\theta} = R(\bar{j})$

Scheme 2 - $(\tau,\rho) = (\theta_1 + a\left\lfloor j_1/(a-1)\right\rfloor, \theta_2)$ where $(\theta_1,\theta_2) = \bar{\theta} = R(j_1 \bmod(a-1), j_2)$

Scheme 3 - $(\tau,\rho) = (\theta_1 + a\left\lfloor j_1/(a-1)\right\rfloor, \theta_2)$ where

$$(\theta_1,\theta_2) = \bar{\theta} = \begin{cases} R(j_1 \bmod(a-1), j_2) & \text{if } \left\lfloor j_1/(a-1)\right\rfloor \bmod 2 = 0 \\ R'(j_1 \bmod(a-1), j_2) & \text{otherwise} \end{cases}$$

where R' is such that

$$R'(\bar{j}) = \left\lfloor \frac{1}{a-1}\begin{bmatrix} a & -1 \\ 1 & a-2 \end{bmatrix}\bar{j} + \frac{1}{a-1}\begin{bmatrix} a-1 \\ 0 \end{bmatrix} \right\rfloor$$

Though it may be possible to analyze the three schemes mathematically, it is easier to explain them by referring to figure 3. For execution times of the original algorithm larger than a-1 (a=5 for the example), scheme 1 essentially replicates figure 3 every additional a-1 units of time. However, for every replica, in addition to changes in the time indices, the processor indices are increased by

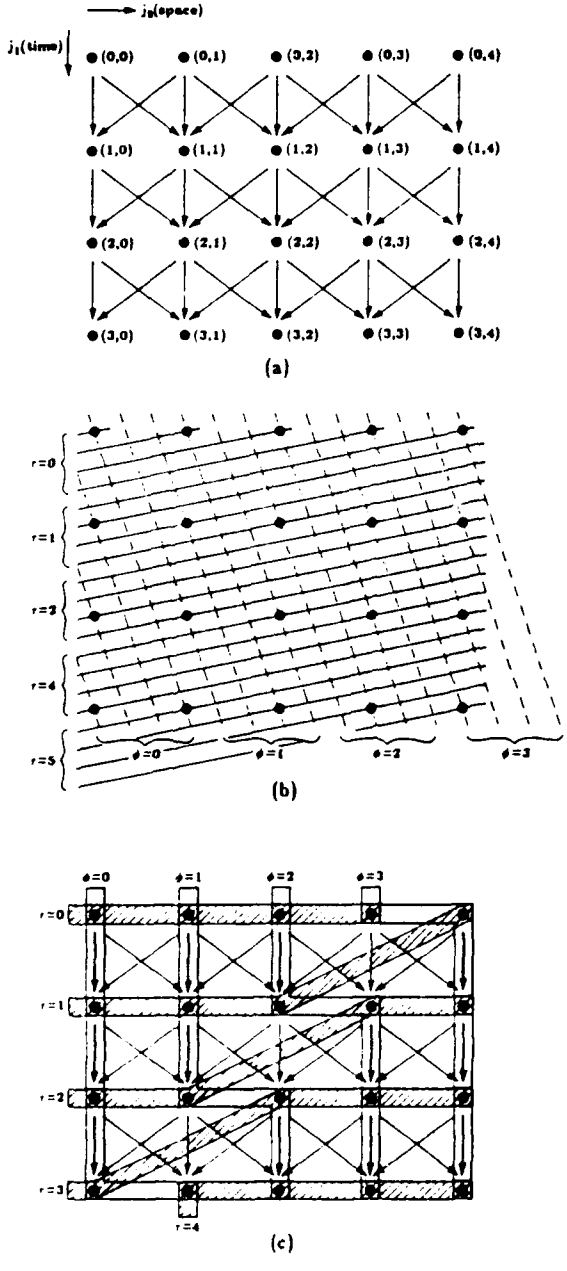


Figure 3 – (a) Original worst case algorithm
(b) Time and space wavefronts defined by R
(c) Timing, processor allocation and data movement in the reconfigured algorithm.

$\lfloor r/(a-1) \rfloor \bmod(a-1)$. For example, computations with indices (a-1,0) through (a-1,a-2) are executed at time $r=5$ by processors with indices 3, 0, 1 and 2 in this order. Computation at index (a-1,a-1) is executed at time 6 by processor 2. It is easy to see that this scheme will require a "wrap-around" link between the first and last processor of the array.

Scheme 2 also generates a replica of figure 3 every a-1 units of time. In this case the replica is exact for the processor indices. However, at the interface between consecutive replicas, the movement of data associated with the original dependence $(1\ 1)^T$ does not occur between adjacent operational processors. If such dependence is not present (e.g., in an algorithm which is not a worst case computation) then this scheme is acceptable. For example, computations with indices (a-1,0) through (a-1,a-2) are executed at time $r = 5$ by processors with indices 0 through 3 and computation with index (a-1,a-1) is executed at time $r = 6$ by processor 3.

Scheme 3 also generates a replica of figure 3 every a-1 units of time. However, successive replicas are mirror images of each other (except, of course, for the arrows depicting data movement). This scheme does not have any of the disadvantages of previous approaches. Note that R' is obtained from R by changing the sign of the off-diagonal entries of T and changing the value of $\bar{t}$ from $(0, (a-2)/(a-1))^T$ to $((a-1)/(a-1), 0)^T = (1\ 0)^T$. Graphically, the net result of these changes is the reversal of the sign of the slope of the wavefronts shown in figure 3(b). This, in turn, yields the mirror image of the figure 3(c) generated by R. For example, computations with indices (a-1,1) through (a-1,a-1) are executed at time $r = 5$ by processors 0 through 3, and, computation with index (a-1,0) is executed by processor 0 a time $r = 6$.

It remains to discuss the case when there are X faulty processors where X can be larger than one. The solution is simple and consists of recursively applying the proposed scheme(s) X times. Clearly, N-1 faults can be tolerated with minimal performance degradation in a linear processor with N processing elements.

HARDWARE REQUIREMENTS

Simple additional hardware is required to support the algorithm reconfiguration schemes discussed here. It must be possible to bypass each faulty processor. Thus, switching hardware is minimal. Also additional local memory is required for each processor in an amount proportional to the number of faults to be tolerated. The constant factor is rather small and the reader can easily verify that for the systolic array of example 2 this constant is 1. In fact, if we reverse the sign of the coordinate j_1 of the index set for that example, this constant is 2 for the resulting algorithm. Finally, one must also consider the implications of implementing on-line the reconfiguration schemes on the complexity of the control and host interface hardware. It must be possible to implement R in real-time. As mentioned in the proof of theorem 1, $\bar{j} = (j_1, j_2) = R(\bar{i})$ implies that $i_1 = j_1 + \left\lfloor (j_1+j_2)/(a-1) \right\rfloor$ and $i_2 = j_2 - \left\lfloor (j_1+j_2)/(a-1) \right\rfloor$. Thus, $R(\bar{i})$ can be computed with at most two adders, one divider and one subtracter. Note that the floor functions and the modulo operations which are also required for each scheme are easily done by discarding or masking bits of a number. In addition to the computation of R, scheme 3 also requires the computation of R'. It is relatively easy to show that $\bar{j} = (j_1, j_2) = R'(\bar{i})$ implies that $i_1 = j_1 + 1 + \left\lfloor (j_1-j_2)/(a-1) \right\rfloor$ and $i_2 = j_2 + \left\lfloor (j_1-j_2)/(a-1) \right\rfloor$. Thus, the same hardware can be used to compute R and R'. Hence hardware requirements are rather small.

TWO-DIMENSIONAL ARRAYS

Since our formalism and basic ideas are applicable to 2-dimensional arrays, RQA transformations can also be used to devise reconfiguration schemes for these arrays. Several types of RQA transformations are useful depending on the degree of hardware reconfigurability assumed. In general, optimal graceful degradation is harder to achieve than for linear arrays, unless relatively complex reconfiguration hardware is used. This seems to be inherent to the nature of the interconnection structure of 2-dimensional arrays. When considering very simple forms of hardware reconfiguration,

it may be necessary to logically remove operational as well as faulty processors. We have studied several schemes with advantages over previous approaches which require hardware mechanisms of comparable complexity. Due to space limitations, a discussion of these schemes and their relative merits is not done here and will appear in a forthcoming paper.

CONCLUSIONS

This paper described three related algorithm reconfiguration schemes which, together with simple reconfiguration hardware, can be used to achieve optimal graceful degradation in linear processor arrays. These schemes are based on a class of RQA transformations, a new type of algorithm transformations introduced in this paper. While our results have general applicability, their practical advantages will undoubtedly depend also on the nature of the implementation and intended application of the processor array. The general approach described here can also be applied to 2-dimensional processor arrays and is also useful for mapping arbitrarily large algorithms into arrays of fixed size.

References

Fortes, J. A. B. and Raghavendra, C. S., 1985 "Gracefully Degradable Processor Arrays," IEEE Transactions on Computers, Vol. C-34, No. 11, pp. 1033-1044.

Li, G.-J. and Wah, B. W., 1985 "The Design of Optimal Systolic Arrays," IEEE Transactions on Computers, Vol. C-34, No. 1, pp. 66-77.

Moldovan, D. I. and Fortes, J. A. B., 1986 "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," IEEE Transactions on Computers, Vol. C-35, No. 1, pp. 1-12.

Fortes, J. A. B. and Parisi-Presicce, F., 1984 "Optimal Linear Schedules for the Parallel Execution of Algorithms," Proceedings of the 1984 International Conference on Parallel Processing, pp. 322-329.

REFERENCE NO. 4

O'Keefe M., and Fortes, J. A. B., "A Comparative Study of Two Systematic Design Methodologies for Systolic Arrays," (Long Version) International Workshop on Parallel Algorithms and Architectures, pp. 313-324, April 1986.

O'Keefe, M., and Fortes, J. A. B., "A Comparative Study of Two Systematic Design Methodologies for Systolic Arrays," (Short Version) International Conference on Parallel Processing, pp. 672-675, August 1986.

PARALLEL ALGORITHMS & ARCHITECTURES
M. Cosnard et al. (editors)
© Elsevier Science Publishers B.V. (North-Holland), 1986

A COMPARATIVE STUDY OF TWO SYSTEMATIC DESIGN METHODOLOGIES FOR SYSTOLIC ARRAYS

M. T. O'Keefe and J. A. B. Fortes *

School of Electrical Engineering, Purdue University, West Lafayette, IN 47907

This paper identifies equivalences between two systematic methodologies for the design of systolic arrays and illustrates the benefits of understanding these relationships. The methods are the parameter method of Li and Wah and the dependency method of Moldovan and Fortes. After a review of the core ideas, models and parameters of each method, mathematical relations between them are derived. The usefulness of these relations is illustrated by showing how (1) optimization procedures for the parameter method suggest similar procedures for the dependency method, (2) systolic designs for convolution and deconvolution, obtained through different methods, can be mathematically proven to be identical and (3) new systolic equations for the parameter method result from the knowledge of equivalent equations in the dependency method.

I - Introduction

In this paper, two proposed methodologies for the systematic design of systolic arrays are comparatively studied. They are the data dependency method of Moldovan and Fortes, [1]-[8], and the parameter method of Li and Wah, [9], [10]. We find and expose the recondite relationships and equivalences between the two methodologies and use this information to improve them and verify similar designs.

Section II provides a short description of both methodologies. Section III establishes equivalences between the mathematical expressions used to systematically design systolic arrays in the two methods. The equivalences of section III are used in section IV to propose optimization procedures and improvements for both methodologies. Additionally, the two methods are used to obtain systolic arrays for the convolution algorithm and the resulting designs are mathematically proven to be equivalent. Section V is dedicated to conclusions.

II - Introduction to the Parameter and Data Dependency Methods

2.1 Parameter Method [9],[10]

This methodology considers the design of optimal pure planar systolic arrays for a class of linear recurrences which take the general form

$$z_i^k = f\left[z_i^{k-\delta}, x_{x(i,k)}, y_{y(k,j)}\right], \quad \delta = \pm 1 \tag{2.1.1}$$

where f is the function to be executed by each cell of the array and $x(i,k)$, $y(k,j)$ are linear indexing functions for the two-dimensional input variables **X** and **Y**. In the following presentation, the coefficients of i, j, k are either 1 or -1. One-dimensional recurrences have the general form

$$z_i^k = f\left[z_i^{k-\delta}, x_{x(i,k)}, a_{a(k)}\right], \qquad \delta = \pm 1 \tag{2.1.2}$$

More general recurrences can also be considered. In the interest of brevity we do not consider them here. However, all results of this paper can be extended to include those cases, as reported in [12].

---

* This work was supported in part by the National Science Foundation under Grant DMC-8419745 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contract no. 00014-85-k-0588.

Three sets of parameters are used to characterize a systolic array: velocities of data flow, data distributions, and periods of computation. The velocity of a datum x is the directional distance passed by that datum in one clock cycle and is denoted by $\vec{x}_d$. The distance between two PEs is defined to be one. Thus, $\vec{x}_d$ must be less than or equal to one because broadcasting is not allowed in pure systolic arrays.

Data distributions are defined using row and column displacements. For two-dimensional input and output matrices, the elements along a row or column are arranged in a straight line and the distance between adjacent elements in a row or column remains constant as the data flows through the array. To define the row displacement of array $\mathbf{X}$, suppose that the row and column indices of $\mathbf{X}$ are i and j, respectively. The row displacement of $\mathbf{X}$ is the directional distance between $x_{\vec{x}(i,j)}$ and $x_{\vec{x}(i+1,j)}$ and is written as $\vec{x}_{is}$. Similarly, the column displacement is the distance between $x_{\vec{x}(i,j)}$ and $x_{\vec{x}(i,j+1)}$ and is written as $\vec{x}_{js}$.

Periods of computation are described using two functions, τ_c and τ_a. τ_c is defined as the time at which a computation is performed, whereas τ_a defines the time at which a variable is accessed. The periods of i and j for two-dimensional outputs are defined as

$$t_i = \tau_c(z_{i+1,j}^{k}) - \tau_c(z_{i,j}^{k}) \tag{2.1.3}$$

$$t_j = \tau_c(z_{i,j+1}^{k}) - \tau_c(z_{i,j}^{k}) \tag{2.1.4}$$

$$t_k = \tau_c(z_{i,j}^{k+1}) - \tau_c(z_{i,j}^{k}) . \tag{2.1.5}$$

It will be assumed that t_k is positive. If this is not true for a given recurrence, the recurrence can be rewritten to satisfy this condition. In computing $z_{i,j}$, $x_{\vec{x}(i,k)}$ and $y_{\vec{y}(k,j)}$ are accessed and two additional periods can be included to describe this interaction. They are

$$t_{kx} = \tau_a(x_{\vec{x}(i,k+1)}) - \tau_a(x_{\vec{x}(i,k)}) \tag{2.1.6}$$

$$t_{ky} = \tau_a(y_{\vec{y}(k+1,j)}) - \tau_a(y_{\vec{y}(k,j)}) \tag{2.1.7}$$

Depending on the order of access, t_{kx} and t_{ky} may be negative. Since operands to be used in a computation must arrive at a PE simultaneously, the magnitude of the periods must equal t_k, i.e. ,it must be true that

$$t_k = |t_{kx}| = |t_{ky}| \tag{2.1.8}$$

The periods are independent of the indices i, j, and k, and they must be greater than or equal to one to prevent broadcasting.

These parameters (velocity, data distribution, and periods) can be combined into a set of equations which describe the operations of a systolic array. These equations, for the two-dimensional case, are

$$t_{kx}\vec{x}_d + \vec{x}_{ks} = t_{kx}\vec{z}_d \tag{2.1.9}$$

$$t_{ky}\vec{y}_d + \vec{y}_{ks} = t_{ky}\vec{z}_d \tag{2.1.10}$$

$$t_i\vec{x}_d + \vec{x}_{is} = t_i\vec{y}_d \tag{2.1.11}$$

$$t_i\vec{z}_d + \vec{z}_{is} = t_i\vec{y}_d \tag{2.1.12}$$

$$t_j\vec{y}_d + \vec{y}_{js} = t_j\vec{x}_d \tag{2.1.13}$$

$$t_j\vec{z}_d + \vec{z}_{js} = t_j\vec{x}_d \tag{2.1.14}$$

The parameters and equations described previously can be used to formulate the design process as an optimization problem. In the following equations, the expression $|p|$ represents the magnitude of the vector quantity $\vec{p}$. The design problem is formulated as follows:

$$\text{minimize} \quad \#\mathbf{PE} \cdot \mathbf{T}^2 \quad \text{or} \quad \#\mathbf{PE} \cdot \mathbf{T} \quad \text{or} \quad \mathbf{T} \tag{2.1.15}$$

$$\text{subject to (2.1.9 - 2.1.14)}$$

and

$$\frac{1}{t_{imax}} \leq |\dot{x}_i| \leq 1 \quad \text{or} \quad |\dot{x}_i| = 0 \tag{2.1.16}$$

$$\frac{1}{t_{jmax}} \leq |\dot{y}_i| \leq 1 \quad \text{or} \quad |\dot{y}_i| = 0 \tag{2.1.17}$$

$$\frac{1}{t_{kmax}} \leq |\dot{z}_i| \leq 1 \quad \text{or} \quad |\dot{z}_i| = 0 \tag{2.1.18}$$

$$1 \leq |t_k| \leq t_{kmax} \; ; \; 1 \leq |t_i| \leq t_{imax} \; ; \; 1 \leq |t_j| \leq t_{jmax} \tag{2.1.19}$$

$$|t_k|\,|\dot{z}_i| = k_1 \leq t_{kmax} \; ; \; |t_i|\,|\dot{y}_i| = k_2 \leq t_{imax} \; ; \; |t_j|\,|\dot{x}_i| = k_3 \leq t_{jmax} \tag{2.1.20}$$

$$|\dot{x}_{is}| \neq 0 \; ; \; |\dot{x}_{ks}| \neq 0 \; ; \; |\dot{y}_{ks}| \neq 0 \tag{2.1.21}$$

$$|\dot{y}_{js}| \neq 0 \; ; \; |\dot{z}_{is}| \neq 0 \; ; \; |\dot{z}_{js}| \neq 0 \tag{2.1.22}$$

$$t_k = |t_{kx}| = |t_{ky}| \tag{2.1.23}$$

and the recurrence determines the relative signs of t_{kx} and t_{ky}.

Recall that t_k describes the number of clock cycles which elapse between two consecutive computations using variable z, and that $\dot{z}_i$ represents the directional distance traversed by datum z in one clock cycle. Thus, k_1 in (2.1.20) represents the magnitude of the directional distance traversed by a datum z between its use in two consecutive computations. Similarly k_2 and k_3 represent this same distance for variables y and x, respectively. k_1, k_2, and k_3 describe the spatial distance covered by a datum between its use in two consecutive computations. The t_{imax}, t_{jmax}, and t_{kmax} values are the maximum period values considered in the optimization. Periods equal to or greater than these maximum values result in completion times which are equal to or greater than the serial processing time. Constraint equations (2.1.21) and (2.1.22) prohibit multiple inputs from entering a PE in one cycle. Clearly, if a data distribution vector is equal to zero, two or more data elements are separated by zero distance and must enter a PE simultaneously. Thus, the formulation of the design problem as an optimization problem as given in equations (2.1.9-2.1.14) and (2.1.15-2.1.23) ensures that the resulting array satisfies the constraints of systolic processing.

The optimal systolic array for a given recurrence can be found by systematically enumerating the possible solutions using a search order that guarantees that the first feasible solution found is, in fact, the optimal one. Consider optimizing $\mathbf{T}$, the total time needed to complete the computation. First set $k_1 = k_2 = k_3 = 1$ or, if a particular variable p is to remain in the same PE, set the associated $k_p = 0$. Then, set the magnitudes of the periods t_i, t_j, and t_k equal to one and determine if a feasible solution exists. If a feasible solution is found it is the optimal solution for $\mathbf{T}$ because $\mathbf{T}$ is a linear function of the periods that increases monotonically with increases in the magnitude of these periods. If no feasible solution is found with $t_i = t_j = t_k = 1$, one of the periods is increased by one and the search for a feasible solution repeats. If no feasible solution can be found with $k_1 = k_2 = k_3 = 1$, one of the k_i, $1 \leq i \leq 3$, is increased by one and the search begins again. A flowchart describing this optimality procedure is shown in [12].

Consider optimizing $\mathbf{AT}^2$. First, use the above procedure to find a solution with optimal execution time $\mathbf{T}_1$ which uses $\mathbf{P}_1$ processing elements. Then, let the largest dimension of the input or output matrix be $(n \times n)$ and assume that the smallest number of PEs that can be used in a solution is n^2. Then, solutions with completion time $\mathbf{T}_2$ such that $n^2\mathbf{T}_2^2 \geq \mathbf{P}_1\mathbf{T}_1^2$ i.e.,

$$\mathbf{T}_2 \geq \frac{\sqrt{\mathbf{P}_1}}{n}\,\mathbf{T}_1 \tag{2.1.24}$$

need not be considered and the search is carried out only for values smaller than $\mathbf{T}_2$. The reason for ignoring designs which have completion times greater than $\mathbf{T}_2$ is that, if $\mathbf{T}_2$ multiplied by the minimum possible number of PEs is greater than the $\mathbf{AT}^2$ measure for the $\mathbf{T}_1$, then the $\mathbf{T}_2$ solution cannot have a smaller $\mathbf{AT}^2$ measure. When all possible designs with

execution time between $\mathbf{T}_1$ and $\mathbf{T}_2$ have been found, the $\mathbf{AT}^2$ measure is compared to find the minimum solution.

This optimization procedure has been applied to find optimal systolic arrays for matrix multiplication, FIR filtering, discrete Fourier transform and other algorithms, [9].

2.2 Data Dependency Method [1]-[8]

Let Z^n denote the nth cartesian power of Z, the set of nonnegative integers. To describe an algorithm **A**, a five tuple $\mathbf{A} = (J^n, C, D, X, Y)$ is used where $J^n \subset Z^n$ is the index set, **C** is the set of computations, **D** is the set of dependence vectors, **X** is the set of input variables, and **Y** is the set of output variables. The data dependencies describe the structure of the algorithm and are given as a set of triples $(\bar{d},v,\bar{j})$ such that the computation indexed by $\bar{j}$ requires the variable v, generated at index $\bar{j} - \bar{d}$, as an operand.

As an example, consider the two-dimensional recurrence $a(j_1,j_2) = f[\,a(j_1-1,j_2+1),\, a(j_1-1,j_2-1)\,]$, $0 \le j_1 \le 4$, $0 \le j_2 \le 4$, where f is some function. We can describe it as $\mathbf{A} = (J^n, C, D, X, Y)$ where $J^2 = \{\, \bar{j} : 0 \le j_i \le 4,\ i = 1,2 \,\}$. **C** is the set of all computations on the right-hand side of the recurrence equation, i.e., $\mathbf{C} = \{\, f[\,a(\bar{j}_1-1,\bar{j}_2+1),\, a(\bar{j}_1-1,\bar{j}_2-1)\,] : (j_1,j_2)^T \in J^2 \,\}$, and **D**, a set of triples $(\bar{d},v,\bar{j})$, can be described by a matrix whose columns correspond to the first element of each triple and v, $\bar{j}$ need not have an explicit representation. Thus, the columns of **D** correspond to the vector difference between $(j_1,j_2)^T$ and the indices of the references to a on the right-hand side of the recurrence, $(j_1-1,j_2+1)^T$ and $(j_1-1,j_2-1)^T$, which yields

$$\mathbf{D} = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}.$$

X is the set of input variables and **Y** is the set of output variables, i.e., $\mathbf{X} = \{a(-1,j_1) : -1 \le j_2 \le 5\} \cup \{a(j_1,j_2) : 0 \le j_1 \le 3, j_2 = -1,5\}$, $\mathbf{Y} = \{a(j_1,j_2) : 0 \le j_1,j_2 \le 4\}$.

Linear indexing functions [1] describe how variables are referenced. A linear indexing function $\bar{\mathbf{F}}: \mathbf{J}^n \rightarrow \mathbf{Z}^m$ is defined by an equation of the form $\bar{\mathbf{F}}(\bar{j}) = \bar{C}_0 + \mathbf{C}\bar{j}$ where $\bar{C}_0 \in Z^{(m\times 1)}$ is called the index displacement and $\mathbf{C} \in \mathbf{Z}^{(m\times n)}$ is called the indexing matrix. For example, the variable $a(j_1 - j_2, j_2 + j_3 - 1, j_3 - j_1)$ has a linear indexing function for which

$$\mathbf{C} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \bar{C}_0 = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}.$$

Data broadcasting is not allowed in systolic arrays and the data dependency method can detect, remove or reduce broadcasts from algorithms to be implemented on systolic arrays [4]. During the execution of an algorithm, a variable needs to be broadcasted if and only if both of the following conditions are satisfied: (1) at least two computations use the variable and (2) such computations are scheduled for execution at the same instant of time. To determine if the first condition is satisfied, it is clear that a variable with indexing function $\bar{\mathbf{F}}$ is used by computations indexed by $\bar{j}$ and $\bar{j}'$ if and only if

$$\bar{\mathbf{F}}(\bar{j}) = \bar{\mathbf{F}}(\bar{j}'), \text{ i.e., } \bar{\mathbf{F}}(\bar{\Gamma}) = \bar{0} \tag{2.2.1}$$

where $\bar{\Gamma} = \bar{j} - \bar{j}'$. From the definition of $\bar{\mathbf{F}}$ equ. (2.2.1) can be rewritten as

$$\mathbf{C}\bar{\Gamma} = \bar{0}. \tag{2.2.2}$$

In essence, the dependency method finds a reindexing transformation which, when applied to the original algorithm, yields a new r_1-equivalent algorithm which maps easily into a systolic array. A transformation matrix **T** *can be used to describe a linear bijection which transforms the dependency matrix and index set of an algorithm so that it can be executed in a VLSI array.* **T** *can be partitioned into two matrices,* π *and* **S**:

$$\mathbf{T} = \begin{bmatrix} \pi \\ \mathbf{S} \end{bmatrix}$$

The π *matrix defines the time transformation whereas* **S** *defines the space transformation to be applied to the dependence matrix and index set of an algorithm. The time at which a computation indexed by* $\bar{j}$ *is executed is determined by* $\pi\bar{j}$, *while* $\mathbf{S}\bar{j}$ *specifies which processor is to execute this computation. In other words, the transformed equivalent algorithm is such*

that the first coordinate of the index of any computation determines its execution time and the remaining coordinates determine which processor is to be used. *Of course*, π and S must satisfy certain conditions if they are to be considered valid transformations. Let m be the number of columns in the dependency matrix. Time transformations must satisfy $\pi\bar{d}_i > 0$, $i=1,2,\ldots,m$, where $\bar{d}_i$ is a column vector in the dependence matrix. This constraint results from the requirement that a variable must be generated before it is used in a computation. The time of execution of a computation with index $\bar{j}$ is given by

$$t_\pi(\bar{j}) = \left\lfloor \frac{\pi\bar{j} - \min\{\pi\bar{j} \mid \bar{j} \in J^n\} + 1}{\mathrm{disp}\pi} \right\rfloor$$

where $\mathrm{disp}\pi$, the displacement of the ordering determined by π, must satisfy $\mathrm{disp}\pi \leq \min(\pi\bar{d}_i : i=1,\ldots,m)$. Intuitively, the displacement describes the number of parallel wavefronts that simultaneously sweep over the index set to complete the computation. In this paper, unless otherwise stated, the displacement is considered to be one, since the parameter method considers only this case.

The space transformation S maps the computation indexed by $\bar{j}$ into processor $S\bar{j}$. This assumes a processor array model consisting of a grid which has the dimensionality of the array. Each point of the grid corresponds to a processor and the coordinates of the point are the index of the processor. Certain restrictions must be placed on possible solutions for S due to the limited interconnections available in VLSI arrays. These restrictions can be embodied in the **P** and **K** matrices. The **P** matrix describes the interconnection primitives available within an array, i.e., the vector differences between indices of connected processors. For example, a square array with only north-south, east-west, nearest-neighbor connections would have the following **P** matrix:

$$\mathbf{P} = \begin{bmatrix} 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \end{bmatrix}$$

where each column of **P** describes one interconnection primitive to be used to send or receive data from a next-neighbor processing element. The primitive with all zero entries indicates that a variable can also be stored in the processor. The utilization matrix **K** describes the interconnections used by the transformed algorithm during execution. *The relationship between* **K**, **P**, **S**, and **D** is

$$\mathbf{SD} = \mathbf{PK} \tag{2.2.3}$$

where the entries of **K** must satisfy the following constraint

$$\sum_{j=1}^{r} k_{ij} \leq \pi\bar{d}_i, \quad i=1,\ldots,m \tag{2.2.4}$$

This last constraint requires that the time between the generation and use of a variable must be greater than or equal to the number of interconnection primitives needed by the datum to travel from the PE in which it was generated to the PE in which it will be used. In fact, the inequality in (2.2.4) can be replaced by equality. If the number of primitives is less than the time allowed for communication, the datum must be stored for the remaining time, thus using the all zeros primitive. An additional constraint can be added that reflects the limited control available within the simple PEs. This means that, in general, data must travel along the same direction as it flows through the array. Thus, only one entry in each column of the **K** matrix can be nonzero. These restrictions can be relaxed to reflect advances in VLSI technology.

The design problem in the data dependency method can be formulated as follows: find a suitable π, which then defines possible solutions for **K** (2.2.4). Examine the solution (or solutions) for S corresponding to each **K** and determine which S requires the smallest number of PEs. A procedure is available for finding the optimal π in terms of the smallest execution time [5]. There is no guarantee that a solution for S in the equation **SD** = **PK** exists for the matrices **K** associated with the optimal π. If no solution for S can be found for the optimal π, certain heuristics [3] can be applied to find a suboptimal π so that a space transformation S exists. However, note that these results refer to a space of solutions where $\mathrm{disp}\pi$ may be equal to or larger than unity. This greatly complicates the optimization procedure [5].

III - Equivalences between the Parameter and Data Dependency Methods

The two methods discussed in this paper each contain sets of equations which describe the flow of data in a systolic array. The data dependency and parameter methods have, respectively, the space equations (2.2.3) and the systolic processing equations (2.1.9-2.1.14). In the following analysis, the relationships between the two sets of equations will be established. *Lemmas* 1–3 provide equivalences between the different parameters of the two methods, while *Lemma* 4 describes the form of the dependency matrices for algorithms considered in the data dependency method. These lemmas are then applied in *Theorem* 1 to show that the space equations and systolic processing equations are equivalent. The proofs are omitted here but can be found in [12].

The first lemma gives expressions for the data distribution and velocity vectors of the parameter method in terms of the transformations and indexing matrices of the data dependency method.

Lemma 1

Let S, π be as defined previously in section 2.2, and let v be any of the variables x, y, z as given for the parameter method. Also, let C^v represent the indexing matrix for variable v. Then the following relationships hold for the two-dimensional case:

$$S\begin{bmatrix} C^v \\ \pi \end{bmatrix}^{-1}\begin{bmatrix} \pm 1 \\ 0 \\ 0 \end{bmatrix} = \vec{v}_x \quad , \quad S\begin{bmatrix} C^v \\ \pi \end{bmatrix}^{-1}\begin{bmatrix} 0 \\ \pm 1 \\ 0 \end{bmatrix} = \vec{v}_y \quad , \quad S\begin{bmatrix} C^v \\ \pi \end{bmatrix}^{-1}\begin{bmatrix} 0 \\ 0 \\ \pm 1 \end{bmatrix} = \vec{v}_z$$

For the one-dimensional case, the following relationships apply:

$$S\begin{bmatrix} C^v \\ \pi \end{bmatrix}^{-1}\begin{bmatrix} \pm 1 \\ 0 \end{bmatrix} = \vec{v}_x \quad , \quad S\begin{bmatrix} C^v \\ \pi \end{bmatrix}^{-1}\begin{bmatrix} 0 \\ \pm 1 \end{bmatrix} = \vec{v}_y$$

The next lemma describes the relationship between the π vector of the data dependency method and the periods t_i, t_j, t_k of the parameter method. The relationship will prove to be remarkably simple.

Lemma 2

$$\pi = \begin{bmatrix} t_i & t_j & t_k \end{bmatrix}$$

Thus, the periods of the parameter method are the elements of the π matrix. The next lemma relates the elements of the data dependency method's $\mathbf{K}$ matrix and the constants, k_i $(1 \le i \le 3)$, as defined in equation (2.1.20); i.e., $|t_k|\,|\vec{z}_d| = k_1$, $|t_i|\,|\vec{y}_d| = k_2$, $|t_j|\,|\vec{x}_d| = k_3$. Let $\bar{k}_i$ be the single nonzero entry of the ith column of $\mathbf{K}$.

Lemma 3

$$\bar{k}_i = k_i \qquad 1 \le i \le 3$$

The next lemma describes the form of the dependency matrices for the class of recurrences considered in the parameter method.

Lemma 4

The dependency matrices for the class of recurrences considered in the parameter method have the following structure:

Two-dimensional Recurrence:

$$\mathbf{D} = \left[\begin{array}{ccc|c} \pm 1 & 0 & 0 & \\ 0 & \pm 1 & 0 & \vec{d}_1 \ldots \vec{d}_r \\ 0 & 0 & \pm 1 & \end{array}\right]$$

One-dimensional Recurrence:

$$\mathbf{D} = \left[\begin{array}{cc|c|c} \pm 1 & 0 & c_1 & \\ 0 & \pm 1 & c_2 & \bar{d}_1 \ldots \bar{d}_r \end{array}\right], \quad |c_1| = |c_2| = 1$$

where $\bar{d}_1,\ldots,\bar{d}_r$ are dependency vectors which are a function of the recurrence, as is r, the total number of these additional dependencies.

The following theorem shows that the equations used in both methods to describe the operation of a systolic array are equivalent.

Theorem 1

The constraint equations (2.1.9 - 2.1.14) of the parameter method are equivalent to the space equations, **SD = PK**, of the data dependency method.

IV - Optimisation Procedures and Examples

Optimisation Procedures

Optimization procedures for the parameter method were discussed previously in section II. By directly translating the parameters and constraints of this method into the corresponding elements of the dependency method, we can devise a similar procedure which is applicable to the recurrences considered in [9]. However, by using a slightly different approach, it is possible to propose a related optimization procedure applicable to all cases for which $\mathrm{disp}\pi = 1$ in the dependency method. It differs from that proposed for the parameter method in that it checks *all possible values of K before considering longer* execution times (i.e., different π's). The flowchart of Figure 2 describes the new optimization procedure. In words, it starts by finding all transformations π which minimize execution time. This is relatively easy, since only the case $\mathrm{disp}\pi = 1$ is considered and execution time is therefore a monotonic function of the entries of π. Hence, one can start with all entries of π being zero and progressively increase their absolute values considering all possible combinations of signs and magnitudes (while, of course, checking for the validity of each π). Possible π's, which might result from further increases in the absolute value of the entries of a particular π for which execution time is larger than the known minimum, need not be considered due to monotonicity property mentioned above. Thus, the search space is finite, and, in fact, rather small for most cases. Once the set of π's is known, it is necessary to check if there exists a solution to the equation $SD = PK$ for at least one of the possible values of K. If a solution is found, then the corresponding π (as *well* as *the* design determined by π and S) is optimal with respect to execution time. Otherwise, a new set of π's must be found which increase execution time by the least amount and the process is repeated again. The procedure always terminates, since, in the worst case, serial execution is reached as a feasible solution.

A similar reasoning can be used to optimize measures combining area and execution time, e.g., AT or AT^2. Figure 3 illustrates such a procedure. It differs from that of figure 2 in that the search space is reduced to the set of π's which result in execution time bounded above by T_2 as given by (2.1.24). In this finite space, all valid values of π and S are considered and those which optimize the combined measure of area and time determine the optimal solution. This is exactly the same approach used in the parameter method. The key idea consists of limiting the search space by choosing bounds for π and, thus, for the execution time. Many different criteria can be used to choose the bounds. For example, in [11], the same approach is used and π is bounded by limiting the values of $\pi\bar{d}$ (which can be thought of as the number of buffers for the data associated with the dependence $\bar{d}$) for all dependencies $\bar{d}$ in the matrix D.

Examples - Systolic Designs for the Convolution Algorithm

Convolution can be expressed as the following recurrence equation

$$y_i^0 = 0 \quad 1 \le i \le n$$

$$y_i^k = y_i^{k-1} + a_k x_{i+k-1} \quad 1 \le i \le n,\ 1 \le k \le m,\ x_j = 0 \text{ for } j > n \qquad (4.1)$$

Another possible description with the order of access of the input terms reversed is

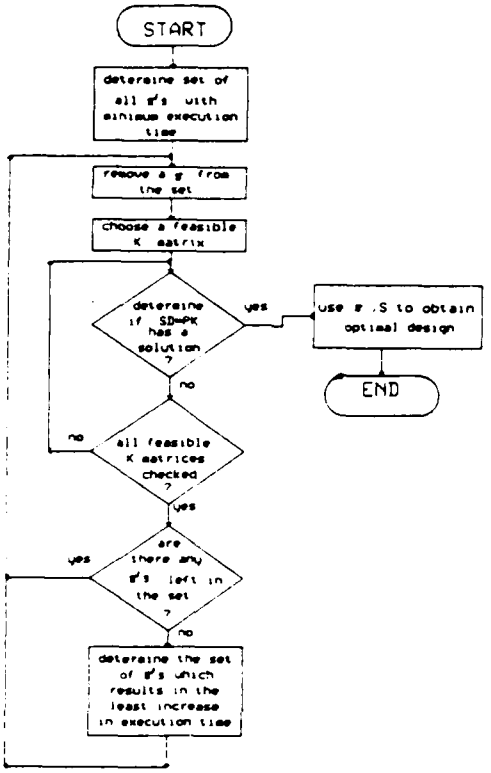


Figure 1 – Optimization procedure for obtaining a systolic array design with minimum execution time (T) using the dependency method.

$$y_i^k = y_i^{k-1} + a_{m-k+1}x_{m-k+i} \quad 1 \le i \le n,\ 1 \le k \le m,\ n > m \tag{4.2}$$

Design using the parameter method

Two cases were considered for the convolution design problem [12], but only one is shown here. Letting $m=4$, $n=6$, the first case will have periods $t_i = 1$ and $t_{ka} = t_{kx} = -1$. Substituting these values into the systolic equations results in four equations in 6 unknowns. For FIR-filtering applications, the a_is are constants that can be loaded into or fixed in the PEs before the computation begins. Thus $\vec{a}_d = 0$ and from the systolic equations $\vec{a}_v = t_{ka}\vec{y}_d = -\vec{y}_d$. To achieve the fastest solution $\vec{y}_d$ can be set to 1 or -1. A possible solution is shown in Figure 3. Four PEs are required; $m + n-1=9$ time units are needed for computation and the preloading of values $x_1, \ldots, x_m$. The time to completion is therefore $2m + n-1=13$ time units.

Design using the dependency method

Now consider the problem in terms of the data dependency method. To do so, the dependency matrices that are valid for recurrences (4.1) and (4.2) must be found. Pipelining the variables in (4.1) and (4.2), we have, respectively,

$$x_{i+k-1}^{i\,k} = x_{i+k-1}^{i+1\,k+1} \qquad x_{m-k+i}^{i\,k} = x_{m-k+i}^{i+1\,k+1}$$

$$a_k^i = a_k^{i+1} \qquad \text{and} \qquad a_{m-k+1}^i = a_{m-k+1}^{i+1}$$

$$y_i^k = y_i^{k-1} + a_k^i x_{i+k-1}^{i\,k} \qquad y_i^k = y_i^{k-1} + a_{m-k+1}^i x_{m-k+1}^{i\,k}$$

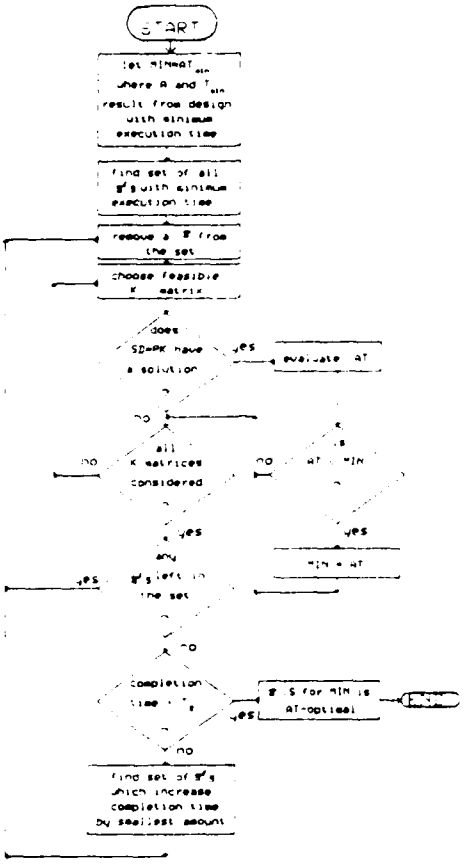


Figure 2 – Optimization procedure for obtaining a systolic array design with *minimum* area.execution time (AT) using the dependency method.

which yields the following form for allowable dependency matrices, respectively.

$$\mathbf{D}_1 = \begin{bmatrix} \pm 1 & 0 & \pm 1 \\ 0 & 1 & \mp 1 \end{bmatrix} \quad \text{and} \quad \mathbf{D}_2 = \begin{bmatrix} \pm 1 & 0 & \pm 1 \\ 0 & 1 & \pm 1 \end{bmatrix}$$

Note that the only difference between $\mathbf{D}_1$ and $\mathbf{D}_2$ is that elements in the dependency vector for z must have different signs for $\mathbf{D}_1$ and the same signs for $\mathbf{D}_2$. According to the dependency method, the first π to be considered is $\pi = [1 \quad 1]$. Multiplying π by $\mathbf{D}_1$ and $\mathbf{D}_2$ yields $\pi\mathbf{D}_1 = [1\ 1\ 0]$, $\pi\mathbf{D}_2 = [1\ 1\ 2]$. The zero entry in $\pi\mathbf{D}_1$ indicates the π selected violates the dependencies of recurrence (4.1) as it is necessary to provide broadcasts. Thus, the recurrence (4.2) is selected and the space transformation corresponding to the systolic array of figure 3 is $\mathbf{S}_2 = [0 \quad -1]$ which yields $\mathbf{S}_2\mathbf{D}_2 = [\ 0 \quad -1 \quad -1 \quad]$.

Verification that both methods yield the same design

Lemma 2 can be easily verified, i.e., $\pi = [t_1 \quad t_R] = [1 \quad 1]$. To verify that the space transformation $\mathbf{S}_2$ corresponds to the same systolic array of figure 3 and, thus, to the velocities and data distributions of the corresponding solution in the parameter method, *Lemma 1* can be verified as follows

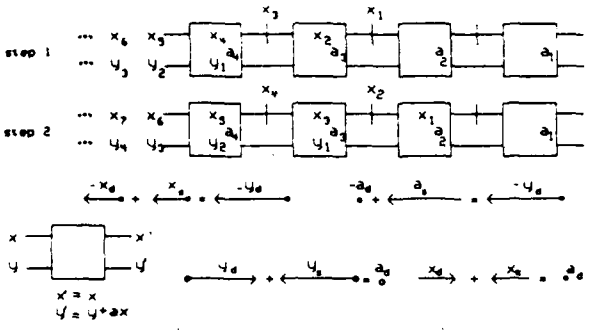


Figure 3 – Systolic array for convolution - [9].

$$S_2\begin{bmatrix}C^y\\ \pi\end{bmatrix}^{-1}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0 & -1\end{bmatrix}\begin{bmatrix}1 & 0\\ -1 & 1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = 1 = \vec{y}_s \quad S_2\begin{bmatrix}C^y\\ \pi\end{bmatrix}^{-1}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0 & -1\end{bmatrix}\begin{bmatrix}1 & 0\\ -1 & 1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = -1 = \vec{y}_d$$

$$S_2\begin{bmatrix}C^a\\ \pi\end{bmatrix}^{-1}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0 & -1\end{bmatrix}\begin{bmatrix}1 & 1\\ -1 & 0\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = 1 = \vec{a}_s \quad S_2\begin{bmatrix}C^a\\ \pi\end{bmatrix}^{-1}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0 & -1\end{bmatrix}\begin{bmatrix}1 & 1\\ -1 & 0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = 0 = \vec{a}_d$$

$$S_2\begin{bmatrix}C^x\\ \pi\end{bmatrix}^{-1}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0 & -1\end{bmatrix}\begin{bmatrix}\frac{1}{2} & \frac{1}{2}\\ \frac{-1}{2} & \frac{1}{2}\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \frac{1}{2} = \vec{x}_s \quad S_2\begin{bmatrix}C^x\\ \pi\end{bmatrix}^{-1}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0 & -1\end{bmatrix}\begin{bmatrix}\frac{1}{2} & \frac{1}{2}\\ \frac{-1}{2} & \frac{1}{2}\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = \frac{-1}{2} = \vec{x}_d$$

Systolic Design for the Deconvolution Algorithm

Deconvolution is the inverse of FIR filtering and can be expressed [9] as the following recurrence with temporary variable z_i

$$z_i^0 = \qquad 1 \le i \le n$$

$$z_i^k = z_i^{k-1} - a_{m-k+1}x_{i+m-k}$$

$$1 \le k \le m-1,\ 1 \le i \le n,\ x_j = 0 \text{ for } j > n$$

$$x_i = \frac{z_i^{m-1}}{a_1} \qquad 1 \le i \le n$$

Design Using the Parameter Method

The parameter method was applied to develop a systolic array which performs deconvolution [9]. The array must perform division to obtain x_1, and x_i's are used in the computation of the z_i's. The division operation may take more time than multiplication, and this fact should be considered in the design process. Assume the delay of a division PE is w and the delay of other PEs is 1. This yields the equation $|t_i| = w + 1$. Analysis of the feedback condition of datum x_i yields an additional systolic equation

$$\vec{x}_d = w\vec{z}_d - \vec{z}_s$$

These two equations must be included in the optimization; letting $w = 2$, $\vec{a}_d = 0$, $\vec{a}_s = -1$, $t_i = -3$, and observing that $t_{ks} = t_{ka}$, a possible solution to the constraint equations yields $t_k = -3/2$, $\vec{z}_d = 2/3$, $\vec{z}_s = 2$, $\vec{x}_d = -2/3$, and $\vec{x}_s = -2$. Note that the velocities of data flow have been averaged over three clocks cycles. A systolic array corresponding to these parameters, with $m = 4$, $n = 5$, is shown in Fig. 4(a).

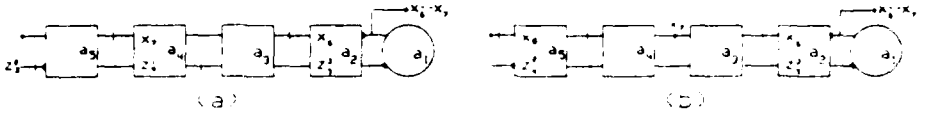


Figure 4 – Systolic array for deconvolution, m=4, n=5. (a) Parameter method solution, $t_1 = -3$, $t_k = 3/2$. (b) Dependency method solution, $\pi = [-3 \quad 1]$ and $S = [0 \quad 1]$

Design Using the Dependency Method

The following dependency matrix can be derived from the recurrence equations for deconvolution

$$D = \begin{bmatrix} -1 & 0 & -1 & k-m \\ 0 & 1 & -1 & k-m+1 \end{bmatrix}$$

Examining the recurrence, the critical dependence occurs between the generation of z_i^{m-1} and the use of x_{i1}; this occurs with $k=m-1$, yielding $\begin{bmatrix} -1 \\ 0 \end{bmatrix}$ as the dependence vector. The usual broadcasting analysis is then applied to pipeline variable x_{ij}, resulting in dependence vector $\begin{bmatrix} -1 \\ -1 \end{bmatrix}$. Since division requires w time units, $\pi\begin{bmatrix} -1 \\ 0 \end{bmatrix} = w+1$ and with $w=2$, $\pi_1 = -3$. Using the proposed optimization procedure, $\pi_2 = 1$, the smallest possible value; possible S values include $S = [0 \quad 1]$, which optimizes space. This means that the generation and usage of z_i^{m-1} and x_{ij}, respectively, occur in the same PE. This array is optimal with respect to completion time T and #PE x T. It could be developed using the parameter method, where, from Lemma 2, $t_1 = -3$ and $t_k = 1$. A systolic array which conforms to this π and S for $m = 4$, $n = 5$, is shown in Fig. 4(b); the completion time of this array is $|\pi_1|(n-1) + |\pi_2|(m-1+w)$. The deconvolution array of Fig. 4(a) has $\pi = [-3 \quad 3/2]$ and $S = [0 \quad 1]$. The same process of verification used for convolution can be applied to show equivalence between the deconvolution arrays designed using the different methods.

New systolic equations

Theorem 1 showed that the systolic equations of the parameter method are equivalent to the space equations, $\mathbf{SD} = \mathbf{PK}$, of the data dependency method. Systolic equations for the one-dimensional case are equivalent to $\mathbf{S}\bar{\mathbf{d}}_y = \mathbf{P}\bar{\mathbf{k}}_y$ and $\mathbf{S}\bar{\mathbf{d}}_x = \mathbf{P}\bar{\mathbf{k}}_x$, respectively. The subscripts on the vectors $\bar{\mathbf{k}}$ and $\bar{\mathbf{d}}$ indicate which variable is associated with a particular vector. Thus, the $\mathbf{S}\bar{\mathbf{d}}_x = \mathbf{P}\bar{\mathbf{k}}_x$ space equation is not contained within the systolic equations of the parameter method for the one-dimensional case.

The systolic equations for the x dependency will take the general form

$$f_x\vec{a}_d + \vec{a}_s = f_x\vec{x}_d \tag{4.13}$$

$$f_y\vec{y}_d + \vec{y}_s = f_y\vec{x}_d \tag{4.14}$$

where f_x and f_y are linear functions of t_1 and t_k. To determine the functions f_x, f_y, the equivalences defined in *Lemmas* 1 and 2 are applied to (4.13-4.14) resulting in

$$f_x S\begin{bmatrix} C^x \\ \pi \end{bmatrix}^{-1}\begin{bmatrix} 0 \\ 1 \end{bmatrix} + S\begin{bmatrix} C^x \\ \pi \end{bmatrix}^{-1}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = f_x S\begin{bmatrix} C^x \\ \pi \end{bmatrix}^{-1}\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$f_y S\begin{bmatrix} C^y \\ \pi \end{bmatrix}^{-1}\begin{bmatrix} 0 \\ 1 \end{bmatrix} + S\begin{bmatrix} C^y \\ \pi \end{bmatrix}^{-1}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = f_y S\begin{bmatrix} C^x \\ \pi \end{bmatrix}^{-1}\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Factoring out the S and simplifying the above equations yields

$$\begin{bmatrix} \mathbf{C}^{\mathbf{x}} \\ \pi \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ f_x \end{bmatrix} = \begin{bmatrix} \mathbf{C}^{\mathbf{x}} \\ \pi \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ f_x \end{bmatrix} \tag{4.15}$$

$$\begin{bmatrix} \mathbf{C}^{\mathbf{y}} \\ \pi \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ f_y \end{bmatrix} = \begin{bmatrix} \mathbf{C}^{\mathbf{x}} \\ \pi \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ f_y \end{bmatrix} \tag{4.16}$$

Selecting recurrence (4.2) yields $\mathbf{C}^{\mathbf{x}} = [1 \ -1]$, $\mathbf{C}^{\mathbf{s}} = [0 \ -1]$, and $\mathbf{C}^{\mathbf{y}} = [1 \ 0]$, then,

$$\begin{bmatrix} \mathbf{C}^{\mathbf{x}} \\ \pi \end{bmatrix}^{-1} = \frac{\begin{bmatrix} t_k & 1 \\ -t_i & 1 \end{bmatrix}}{(t_k + t_i)} \begin{bmatrix} 0 \\ f(t_i, t_k) \end{bmatrix}, \quad \begin{bmatrix} \mathbf{C}^{\mathbf{s}} \\ \pi \end{bmatrix}^{-1} = \begin{bmatrix} 0 & -1 \\ t_i & t_k \end{bmatrix}^{-1} = \begin{bmatrix} \frac{t_k}{t_i} & \frac{1}{t_i} \\ -1 & 0 \end{bmatrix},$$

$$\begin{bmatrix} \mathbf{C}^{\mathbf{y}} \\ \pi \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 \\ t_i & t_k \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 \\ \frac{-t_i}{t_k} & \frac{1}{t_k} \end{bmatrix}$$

Using these inverses in (4.15-4.16) gives $f_x = -(t_i + t_k)$, $f_y = (t_i + t_k)$ and equations (4.13-4.14) become

$$-(t_i + t_k)\vec{s}_d + \vec{s}_s = -(t_i + t_k)\vec{x}_d$$

$$(t_i + t_k)\vec{y}_d + \vec{y}_s = (t_k + t_i)\vec{x}_d$$

The above analysis was applied to the recurrence of equation (4.2). A similar analysis of the recurrence expressed in equation (4.1) results in $f_x = (t_k - t_i)$ and $f_y = -(t_k - t_i)$ in equations (4.13) and (4.14). This new set of systolic equations, derived from the data dependency method through the equivalences described in Section III, can be added to the set of systolic equations. From this new set, only four equations are needed to provide equivalent solutions to those derived from the original four equations.

References

[1] D.I. Moldovan, "On the Analysis and Synthesis of VLSI Algorithms," IEEE Trans. Comp., Vol. C-31, pp. 1121-1126, Nov. 1982.

[2] D.I. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays," Proc. IEEE, Vol. 71, No. 1, Jan. 1983.

[3] J.A.B. Fortes, "Algorithm Transformation for Parallel Processing and VLSI Architecture Design," Ph.D. dissertation, University of Southern California, L.A., CA, Dec. 1983.

[4] J.A.B. Fortes and D.I. Moldovan, "Data Broadcasting in Linearly Scheduled Array Processors," 11th Ann. Int. Symp. on Computer Architecture, June, 1984.

[5] J.A.B. Fortes and F. Parisi-Presicce, "Optimal Linear Schedules for the Parallel Execution of Algorithms," 1984 Int. Conf. Parallel Processing, 1984.

[6] J.A.B. Fortes and D.I. Moldovan, "Parallelism Detection and Transformation Techniques Useful for VLSI Algorithms," J. of Par. Dist. Computing, August, 1985.

[7] J.A.B. Fortes, B.W. Wah, K.S. Fu, "Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays," ICASSP '85.

[8] D.I. Moldovan and J.A.B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," IEEE Trans. Comp., Vol. C-35, No. 1, Jan. 1986.

[9] G. J. Li and B. W. Wah, "The Design of Optimal Systolic Arrays," IEEE Transactions on Computers, Vol. C-34, pp.66-77, Jan. 1985.

[10] G.J. Li and B.W. Wah, "Optimal Design of Systolic Arrays for Image Processing, "Workshop on Com. Arch. for Pat. Analysis and Image Database Mgnt., Oct. 1983.

[11] Y. Wong and J. Delosme, "Optimal Systolic Implementations of N-dimensional Recurrences," ICCD '85.

[12] M. O'Keefe, "A Comparative Study of Two Systematic Design Methodologies for Systolic Arrays," MSEE Thesis, School of Electrical Engineering, Purdue University, 1986.

REFERENCE NO. 5

Taylor, V. E. and Fortes, J. A. B., "Using RAB to Map Algorithms into Bit-Level Systolic Arrays," 2nd International Conference on Supercomputing, pp. 227-236, May 1987.

Presented at the Second International Conference on Supercomputing, May 1987.

Using RAB to Map Algorithms into Bit-Level Systolic Arrays

Valerie E. Taylor and Jose A.B. Fortes

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

ABSTRACT

RAB, a Reconfiguration Algorithm for Bit-level code, is a large program which systematically maps a class of numerical algorithms into bit-level processor arrays. This paper explains the purpose of RAB, outlines its overall organization, presents the underlying ideas and techniques of the main components of RAB, and discusses some implementation details. The input to RAB consists of C programs with word-level computations. Each arithmetic operation in these computations is first replaced by several bitwise operations (i.e a bit-level expansion) which implement that operation. Dependencies are then detected in the bit-level code and represented as a dependence matrix which is used in the synthesis phase of RAB to generate an algorithm transformation. In the final mapping phase of RAB, each bit level operation in the transformed algorithm is replaced by a corresponding microprogram (i.e., a microcode expansion). This microcode is also optimized in this phase to produce the output of RAB, an algorithm executable on the processor array. Currently, prototype processor arrays composed of several NCR Geometric Arithmetic Parallel Processor (GAPP)† chips are the targets for the output of RAB.

I. INTRODUCTION

RAB is a program which maps a class of numerical algorithms programmed in C into bit-level processor arrays. It can be used to derive a full design specification for an algorithmically-defined processor array as well as to identify full (partial) mappings of an algorithm into an existing processor array of fixed (variable) size. This paper explains the purpose of RAB, outlines its overall organization, presents the underlying ideas and techniques of the main components of RAB, and discusses some implementation details. In order to illustrate the concepts and operation of RAB, we show how two algorithms for convolution are mapped into a variable size processor array composed of NCR GAPP chips (i.e., each chip is a (12 × 6) processor array [DaTh84]).

This work was supported in part by the National Science Foundation under Grant DMC-8419745 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered throught the Office of Naval Research under contract no. 00014-85-k-0588.

†GAPP is a trademark of NCR Corporation.

Processor arrays generally consist of a collection of processing elements (PE's) with a regular interconection scheme. Systolic arrays, as characterized by Kung [Kung82], are a special case of processor arrays in which data flows from one PE to another in a regular and synchronous fashion. Generally, a systolic array is easy to implement and extend because of its regularity and modularity. Due to simplicity of local processor design and advances in VLSI technology, relatively general purpose bit-level arrays are becoming common (i.e., GAPP [DaTh84], MPP [Batc80], DAP [Redd79], CLIP [DuWa75] and others). As compared to word-level arrays, bit-level arrays require simple processing elements (e.g., processing elements composed of a full adder, some simple logic, and a number of registers) and provide high throughput rates (i.e., bit rates). These characteristics also make bit-level processor arrays very attractive for special-purpose applications, e.g., digital signal processing ([McMc82], [Mcetal84], [Mcetal85]).

Despite being ideally suited for various applications and VLSI implementation, processor arrays can be difficult to program (in the case of an existing general purpose architecture) or design (in the case of a special purpose architecture). This is a particularly acute problem for bit-level systolic arrays where the goal is to implement high-level computations (e.g., matrix computations, convolution, etc) using bitwise operations. In order to solve this problem, it is desirable to develop methodologies and tools which enable the systematic mapping of algorithms into processor arrays. In the past, several research efforts have been pursued in this direction and a survey can be found in [Foetal85]. Many of those methodologies, which were intended for word-level processor arrays, are applicable to bit-level arrays. However, besides some of the limitations that still characterize those methodologies, systematic bit-level designs present additional problems. RAB represents an attempt to develop an automated tool for the design and programming of bit-level arrays and to understand and solve the open questions and problems involved in this process.

In practice, potential users of processor arrays are given an algorithm and must devise a means for its execution using one of the following options: (1) to use an existing processor array, (2) to design a special purpose processor array, or (3) to design an array that uses a number of existing "smaller" processor array modules as the basic components. Option (1) requires mapping of the algorithm into an existing array taking into consideration size limitations, fixed interconnection schemes, and predesigned processing elements. In this

option, which we refer to as full mapping, the programming decisions are totally subordinated to the characteristics of the array. Option (2) allows the user to design the hardware taking into consideration only the characteristics of the algorithm and perhaps some rather general VLSI design constraints (i.e., planarity, limited pinout, etc). This option is referred to as full design. Option (3) is a compromise between full mapping and full design, where the designer can decide the overall organization (i.e., shape, size, interfaces) of the array, but uses given basic blocks which are themselves fully defined "small" processor arrays. We refer to this option as partial mapping/design.

The input to RAB consists of C programs which implement word-level algorithms. In section II of this paper we characterize the class of algorithms for which RAB is intended, present the algorithm model, and describe the representation of dependencies in an algorithm. RAB first expands the computations in the input program into bit-level operations as shown in figure 1. This expansion phase, which is described in section III, replaces word-level computations with a bit-level implementation of the arithmetic operations. This phase is followed by data dependence/broadcast analysis which uses techniques discussed in section IV. The results of this analysis can be used to generate an algorithm transformation which yields a full design of an algorithmically-defined array or full (partial) mapping for a fixed (variable) size array corresponding to the third level of modules in figure 1. In section V we present the methodology for the generation of a partial mapping and discuss how a full mapping or full design can be obtained. The last two modules in figure 1, microcode expansion and microcode optimization, comprise the mapping phase which is discussed in section VI. In section VII we review the status of the implementation effort and present some concluding remarks about the project.

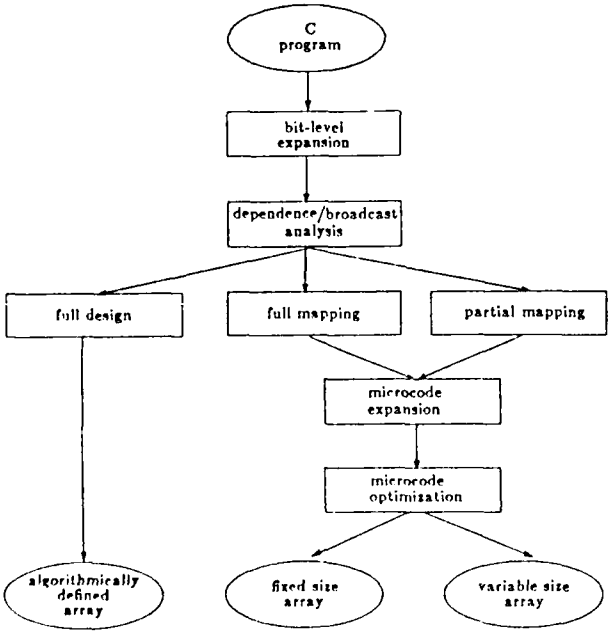


Figure 1. Flow diagram of RAB.

II. ALGORITHM MODEL AND REPRESENTATION

Algorithm Representation

RAB accepts as input a program which uses a subset of C constructs. Since algorithms that run efficiently on a processor array are likely to have a repetitive and regular structure, the input to RAB consists of programs which typically contain loops. For this reason, RAB is capable of efficiently analyzing loop-like programs with static behavior. In addition to the fact that pointers and function calls cannot be used, the structure of the loop-like programs accepted by RAB must exhibit the following characteristics:

- the lower and upper bounds of the outermost loop must be integer constants.
- the bounds of the nested loops must be linear expressions of the outer loop indices or integer constants.
- the step of each loop must be one.
- no two loops can have the same nesting level.
- arrays of any dimensions are allowed; the range of each dimension must be an integer constant.
- the boolean expression of a conditional statement must be a linear expression of the outer loop indices.
- all subscript expressions used when referencing elements of arrays must be linear expressions of the outer loop indices.

Example 2.1

The following convolution algorithm is an example of a program which satisfies the criteria of the algorithm representation.

```
for(j1 = 1;  j1 <= N1; j1++){
  for(j2 = 1;  j2 <= N2; j2++){
    y[j1] = y[j1] + w[j2] * x[j1+j2-1]
  }
}
```

where $w[j_2]$ is the sequence of weights, $x[j_1+j_2 - 1]$ is the sequence of inputs and $y[j_1]$ is the result sequence.

End of example.

Other programs such as matrix-matrix multiplication, matrix-vector multiplication, and FIR and IIR filtering satisfy the constraints of our algorithm representation. A broader list of suitable programs can be found in the concluding remarks of [Kung82]. Many programs which fall outside of this class can be transformed to satisfy the above constraints, using such techniques as normalization or loop fusion [Wolf82]. Conceivably, such techniques could be easily implemented in a preprocessing step for RAB. However, it is assumed that input programs have been normalized and loop fusion is not needed. The next subsection presents the formal definitions of dependencies.

Modeling Dependencies in Algorithms

The parallel execution of independent operations requires knowledge about the existing dependencies in an algorithm in order to preserve its semantics. There are two types of dependencies that can occur in an algorithm: machine dependencies and algorithm dependencies [Kuetal81]. Machine dependences result from the limitations of the particular architecture used

for execution of the algorithm; algorithm dependencies result from the structure of the algorithm. The first category of dependence, machine dependence, also called resource dependence, is defined as follows.

Definition 2.1 (machine dependence)

Statement S_j, denoted as the head of the dependence, is *machine dependent* on statement S_i, denoted as the tail of the dependence, if and only if

1. statement S_i precedes statement S_j and
2. $res(S_i) \cap res(S_j) \neq \emptyset$

where $res(S_i)$ denotes the set of resources needed to execute statement S_i.

Machine dependencies can be divided into two categories: explicit machine dependence and implicit machine dependence. Explicit machine dependencies result from the apparent limitations of the architecture. For example, statement S_j is explicitly machine dependent on statement S_i if both statements require a write to two different memory (RAM) locations and the given architecture only has one RAM port. Implicit resource dependencies are inherent in the semantics of the instructions. For example, in a GAPP array, the arithmetic and logic unit (ALU) of each PE always executes a "full add" operation every clock cycle, regardless of the instruction being executed. As a consequence, the architecture of each PE exhibits implicit resource dependencies with the use of the calculated variables sm, bw, and cy (which denote sum, borrow, and carry respectively). Thus, if a statement explicitly uses a calculated variable, it will always depend on the previous statement.

The second category of dependencies, algorithm dependence, consists of the three classical dependencies: output dependence, data dependence, and anti-dependence. These dependencies are defined in the following definition.

Definition 2.2 (algorithm dependence)

Statement S_j, denoted as the head of the dependence, is *algorithm dependent* on statement S_i, denoted as the tail of the dependence, if and only if

1. statement S_i precedes statement S_j and
2. one of the following conditions is satisfied:
   i. $out(S_i) \cap out(S_j) \neq \emptyset$
   ii. $out(S_i) \cap in(S_j) \neq \emptyset$
   iii. $in(S_i) \cap out(S_j) \neq \emptyset$

where $out(S_i)$ denotes the set of output variables of statement S_i, and $in(S_i)$, the set of input variables.

It is assumed that the reader is knowledgeable about algorithm dependencies for which an example would be redundant. In RAB, only algorithm data dependencies are detected in the dependence analysis for use in the generation of the algorithm transformation. The reason for the detection of only data dependencies will become evident in the discussion on the algorithm transformations. However machine dependencies and all three algorithm dependencies are detected in the microcode optimization module to be discussed later in this paper.

Distance or dependence vectors provide a particularly convenient way of representing algorithm dependencies between statements referencing arrays. We define the distance vector as the vector difference between the index of the computation where a variable is used and the index of the computation where the same variable is generated. These vectors can be placed in a dependence matrix which is representative of the algorithm dependencies in a program. This matrix and other algorithm parameters are essential features that are represented in the algorithm model defined below.

Definition 2.3 (algorithm model)

An *algorithm* is a 5-tuple, $<J^n, C, D, I_v, O_v>$ where $J^n \in Z^n$ is the index set (Z represents the set of all integers), C is the set of computations, D is the set of dependencies represented by distance vectors, I_v is the set of input variables for the algorithm, and O_v is the set of output variables for the algorithm.

An example of the algorithm model is given in section IV.

The dependencies represented in the dependence matrix, D, must be satisfied by the execution ordering of an algorithm defined below.

Definition 3.4 (execution ordering)

A partial ordering is an *execution ordering* if all distance vectors are positive in the sense of that ordering.

In other words, the execution ordering of an algorithm restricts the generation of a variable to always precede the usage. RAB replaces an execution ordering which is total by an execution ordering which is partial. Thus the original distance vectors represented in the matrix D must be positive in the sense of the lexicographical ordering.

Now that we have presented our algorithm model and representation, we will describe the various modules shown in figure 1. The next section discusses the bit-level expansion of the word-level computations.

III. BIT-LEVEL EXPANSION

The first phase of RAB systematically replaces the word-level computations with bit-level implementations of the arithmetic operations. These bit-level implementations are hereafter referred to as expansions. The actual expansion for a given arithmetic operation is not unique. For example, there are several expansions for the multiplication operation, e.g., Booth's algorithm [Boot51] or the shift-add algorithm [Hwan79]. The bit-level arithmetic expansions used with RAB were chosen due to simplicity since RAB is relatively new and in the initial stages of testing. Conceivably other expansions can be used with RAB to investigate the optimality of different bit-level algorithms. RAB currently implements the bit-level expansions for addition, multiplication, division, subtraction, and all possible pairwise combinations of these operations. Actually, RAB provides two types of expansions for each operation or operation pair. The first type is used for running the expanded algorithm as a conventional C program. This provides the user with a means for gathering test data. In these expansions, statements are included which explicitly convert the initial data values to their bit-level representations. These conversion statements are not contained in the second type of expansion which is used in the dependence analysis module of RAB. In order to facilitate the analysis, the second type of expansion eliminates output and anti-dependencies using such techniques as renaming and expansion ([Gaetal81], [Kuetal81]). Two different bit-level implementations of the convolution algorithm (given in example 2.1) are shown in figure 2. These two algorithms result from the

```
for(j1 = 1; j1 <= N1; j1++){
 for(j2 = 1; j2 <= N2; j2++){
  for(j3 = 1; j3 <= N3; j3++){
   for(j4 = 1; j4 <= N4; j4++){
    if(j4 == 1){
     cy2[j1][j2][j3][0] = (cy2[j1][j2][j3-1][0] & sum1[j1][j2][j3-1])|
          (cy2[j1][j2][j3-1][0] & sum2[j1][j2-1][j3-1])|
          (sum1[j1][j2][j3-1] & sum2[j1][j2-1][j3-1]);
     sum2[j1][j2][j3-1] = cy2[j1][j2][j3-1][0] ^ sum1[j1][j2][j3-1] ^
          sum2[j1][j2-1][j3-1];
    }
    else
     if((j4 > 1) && (j3 < N3)){
      cy1[j1][j2][j3][j4-1] = (sum1[j1][j2][j3+j4-2] & (w[j2][j3] & x[j1+j2-1][j4-1]))|
          (sum1[j1][j2][j3+j4-2] & cy1[j1][j2][j3][j4-2]) |
          ((w[j2][j3] & x[j1+j2-1][j4-1]) & cy1[j1][j2][j3][j4-2]);
      sum1[j1][j2][j3+j4-2] = sum1[j1][j2][j3+j4-2] ^ cy1[j1][j2][j3][j4-2] ^
          (w[j2][j3] & x[j1+j2-1][j4-1]);
     }
     else
      if((j4 > 1) && ( j3 == N3)){
       cy2[j1][j2][j3][j4-1] = (cy2[j1][j2][j3][j4-2] & sum1[j1][j2][j3+j4-2])|
          (cy2[j1][j2][j3][j4-2] & sum2[j1][j2-1][j3+j4-2])|
          (sum1[j1][j2][j3+j4-2] & sum2[j1][j2-1][j3+j4-2]);
       sum2[j1][j2][j3+j4-2] = cy2[j1][j2][j3][j4-2] ^ sum1[j1][j2][j3+j4-2] ^
          sum2[j1][j2-1][j3+j4-2];
      }
    }
   }
  }
 }
}
```

Figure 2a. A bit-level expansion of the convolution algorithm.

use of two distinct expansions of the operation pair (+, *).

The algorithm presented in figure 2b is currently used in the expansion phase of RAB. However, the user is not required to input an algorithm into the expansion module; facilities are provided whereby a user may bypass this module and input a bit-level algorithm different from the one generated in the expansion module. This is the case with the algorithm presented in figure 2a. Both algorithms correspond to the second type of expansion mentioned above and can serve as inputs to the dependence analysis module discussed in the following section.

IV. DEPENDENCE/BROADCAST ANALYSIS

The dependence analysis module detects dependencies between statements referencing arrays. In order for a dependence to exist between two statements referencing arrays, the following conditions must be satisfied:

1. the array references in the two statements must have the same name.
2. given that condition 1 is satisfied, the functions which specify the subscripts of the array references must have the same value for some index value(s).
3. the index value(s) for which condition 2 is satisfied must belong to the iteration space.

This module is invoked when the parser detects that condition 1 has been satisfied. Kuhn's Dependence Arc Set Analysis (DASA) technique [Kuhn80] is used with RAB to verify conditions 2 and 3.

Dependence Detection

DASA utilizes five relations represented as convex sets to gather information about the possible dependencies and to determine whether conditions 2 and 3 are satisfied. Dependencies are considered in relation to the Cartesian product of the loop indices and the nesting level of the statements involved in the possible dependencies. Two of the five relations, T and H, define the control structure of the loops surrounding the tail statement (i.e., the point where the data is generated) and head statement (i.e., the point where the data is used) of the possible dependence, respectively. Two

```
for(j1 = 1; j1 <= N1; j1++){
  for(j2 = 1; j2 <= N2; j2++){
    for(j3 = 1; j3 <= N3; j3++){
      for(j4 = 1; j4 <= N4; j4++){
        cy[j1][j2][j3][j4] = (w[j2][j3] & sum[j1][j3+j4-1]) | (cy[j1][j2][j3][j4-1] &
                             (w[j2][j3] | sum[j1][j3+j4-1]))
        sum[j1][j3+j4-1] = ( w[j2][j3] ^ sum[j1][j3+j4-1] ^ cy[j1][j2][j3][j4-1] &
                             x[j1+j2-1][j4]) | (sum[j1][j3+j4-1] & ~x[j1+j2-1][j4])
      }
    }
  }
}
```

Figure 2b. The bit-level expansion of the convolution algorithm used in the expansion phase of RAB.

other relations, S_g and S_u, respectively define the indexing function of the generated and used arrays referenced in the tail and head statements. The fifth relation, F_{th}, represents the forward relation which is used to test different conditions of the loop indices for the existence of a dependence. These relations are represented as convex sets in matrix format that is easily implemented and manipulated in software. If a solution space results from the convex analysis of the intersection of the relations, T, H, F_{th}, and S_g composed with S_u^{-1}, then a dependence exists for the conditions defined by the forward relation, F_{th}. The inequalities of the solution space of a dependence are then ordered to form an upper bound matrix, U, and a lower bound matrix, L, to be used in generating distance vectors. Further details about DASA can be found in [Kuhn80] and [Tayl86].

Broadcast Analysis

An analysis scheme similar to the one used with DASA is used to discover when a broadcast exists. Since only one relation is required for the broadcast analysis we will elaborate on this concept to introduce the reader to the representation of relations as convex sets and the analysis scheme also used with DASA for dependence detection.

A data item requires a broadcast if and only if the datum is needed to simultaneously execute two or more computations in distinct processors. In [Fort84], sufficient and necessary conditions for a broadcast are provided in relation to an array index function, $F(\bar{j}) = C\bar{j} + C_0$, where C is the indexing matrix and C_0 is the index displacement. In order to remain consistent with the representation of the relations involved in DASA, we represent the array indexing function, $F(\bar{j})$, by the subscript relation, S_i, which is defined below.

Definition 4.1 (subscript relation)

Let the *subscript relation* of the β dimensional array A be represented by the relation

$$S_i \begin{bmatrix} \bar{j} \\ \bar{\beta} \end{bmatrix} = [S_i' \quad -I] \begin{bmatrix} \bar{j} \\ \bar{\beta} \end{bmatrix} = \bar{\sigma}_i$$

where $\bar{\beta}$ represents the array subscripts, $S_i' \in Z^{(\beta \cdot n)}$, is called the subscript matrix, I is the identity matrix, $\sigma_i \in Z^{(\beta \cdot 1)}$ is called the constant vector, and n is the dimension of the iteration space.

The subscript matrix, S_i', is equivalent to the indexing matrix, C, and the constant vector, σ_i, is equivalent to the index displacement, C_0. For example, the representation of the subscript relation for the input variable $x[j_1+j_2-1][j_4-1]$ is given by the following:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} j_1 \\ j_2 \\ j_3 \\ j_4 \\ \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\text{where } S_i' = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } \bar{\sigma}_i = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

According to Theorem 3.1 in [Fort84], if the $rank(S_i')$ (or $rank(C) = n-1$) then broadcasting can by eliminated be including the distance vectors for the input variables in the dependence matrix used to generate an algorithm transformation. These vectors for the input variables are hereafter referred to as buffering vectors. The buffering vector is defined as the vector difference between the two points of the iteration space using the same variable. This vector can be generated in the following manner.

Two computations indexed by $\bar{j}'$ and $\bar{j}''$ use the same variable if and only if the value of the array subscripts are the same. This condition is represented by the equation $S_i'\bar{j}' - \bar{\sigma}_i = S_i'\bar{j}'' - \bar{\sigma}_i$ or

$$S_i'(\bar{j}' - \bar{j}'') = 0 \tag{4.1}$$

where $\bar{j}' - \bar{j}'' = \bar{d}$ is the buffering vector. Equation 4.1 can be represented by the following convex set:

$$\begin{bmatrix} S_i' & -S_i' \\ -S_i' & S_i' \end{bmatrix} \begin{bmatrix} \bar{j}' \\ \bar{j}'' \end{bmatrix} \leq \begin{bmatrix} \bar{0} \\ \bar{0} \end{bmatrix}. \tag{4.2}$$

The intersection of the five relations used in DASA is represented in a similar manner. The solution to the convex set in (4.2) is found using an analysis procedure similar to the one used with DASA. First the variables are eliminated using a reduction procedure. If a consistent solution results from the elimination step, the variables are projected onto the space $Z^n \times Z^n$. A detailed description of the reduction and projection procedures can be found in [Kuhn80] and [Tayl86]. In the projection step the inequalities defining the solution space are ordered to form the L and U matrices mentioned in the previous subsection. The results of the broadcast analysis for the input variable, $x[j_1+j_2-1][j_4-1]$ are given below.

The resultant L matrix is shown as the coefficient matrix in the following convex set which specifies the lower bounds of the solution space:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} j_1' \\ j_2' \\ j_3' \\ j_4' \\ j_1'' \\ j_2'' \\ j_3'' \\ j_4'' \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Similarly, for the upper bounds of the solution space, the U matrix is as shown in the following convex set:

$$\begin{bmatrix} -1 & -1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} j_1' \\ j_2' \\ j_3' \\ j_4' \\ j_1'' \\ j_2'' \\ j_3'' \\ j_4'' \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Similar matrices result from the dependence analysis (however, typically, $L \neq -U$). The generation of the distance vectors for the input variables and the generated variables involved in a dependence are discussed in the following subsection.

Distance Vector Generation

The distance vectors can be extracted from the L and U matrices by inspection and enumeration. The enumeration step is only necessary when the elements of the distance vector are functions of the outer loop indices instead of constants. This step consists of substituting each point in the solution space of the dependence (defined by the L and U matrices) into the given functions and keeping only the distance vectors with integer elements (fractional entries cannot result from the difference of integer vectors).

The buffering vectors for the input variables are extracted in a similar manner. The L and U matrices resulting from the broadcast analysis of the input variable $x[j_1+j_2-1][j_4-1]$ represent the equations $(j_1' - j_1'') - (j_2' - j_2'') = 0$ and $j_4' - j_4'' = 0$. The number of buffering vectors resulting from the broadcast analysis is equal to $n-\text{rank}(S_i')$ (which for our example is 2 since $n=4$ and $\text{rank}(S_i') = 2$) corresponding to the number of free variables. For the equation above, the buffering vectors are the transpose of the vectors $[1\ -1\ 0\ 0]$ and $[0\ 0\ 1\ 0]$ corresponding to the case when $(j_1'-j_1'') = 1$, $(j_3'-j_3'') = 0$ and $(j_1'-j_1'') = 0$, $(j_3'-j_3'') = 1$, respectively. These vectors are not unique; the value ± 1 may be used for each of the free variables. However, this is not the case with the dependence analysis. The distance vectors resulting from DASA are unique since the control structure of the tail and head statements are represented in the analysis.

The relations defining the control structure of a dependence in DASA correspond to a subset of the iteration space or a subset of J^n. For the algorithm given in figure 2a, the iteration space is the set of the points $(j_1, j_2, j_3, j_4) \in J^n$ for $1 \leq j_i \leq N_i$. The results of the intersection of the control structure with the relations S_g, S_u, and F_{th}, along with the results of the broadcast analysis are combined to form the following dependence matrix (for the algorithm given in figure 2a).

$$D = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & -1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Column 1 of the dependence matrix corresponds to the used variables $cy2[j_1][j_2][j_3-1][0]$ and the input variables $x[j_1+j_2-1][j_4-1]$, column 2 corresponds to the used variables $sum1[j_1][j_2][j_3+j_4-2]$, column 3 corresponds to the used variables $sum2[j_1][j_2-1][j_3+j_4-2]$, column 4 corresponds to the used variables $cy2[j_1][j_2][j_3][j_4-1]$ and $cy1[j_1][j_2][j_3][j_4-2]$ and the input variable $w[j_2][j_3]$, and the last two columns correspond to the input variables $w[j_2][j_3]$ and $x[j_1+j_2-1][j_4-1]$. The set of computations, C, used in the algorithm in figure 2a correspond to full add operations. The set of output variables for this algorithm consists of the results of summing the products that is stored in the array sum2. The set of input variables consists of the weights, inputs, and the sums and carries which were never generated.

The algorithm model parameters for the algorithm given in figure 2b are similar to the ones presented above with the exception of the dependence matrix, D, given below.

$$D = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 1 & -j_3 & 0 & 0 & 1 \\ 1 & -1 & j_3 & 0 & 0 & 0 \end{bmatrix}, \quad j_3 = 1,\ldots,N_3 \text{ given } N_3 < N_4.$$

Column 1 of this dependence matrix corresponds to the used variables $cy[j_1][j_2][j_3][j_4-1]$ and the input variables $w[j_2][j_3]$, columns 2 and 3 correspond to the used variables $sum[j_1][j_3+j_4-1]$, and the last three columns corresponds to the input variables $w[j_2][j_3]$ and $x[j_1+j_2-1][j_4]$.

The distance vectors for the generated data items are used in the synthesis phase to preserve the semantics of the program; the buffering vectors for the input data items are included in the dependence matrix in an attempt to schedule different execution times for the computations requiring the same variable. The next section describes the methodology used to generate an algorithm transformation for a variable size array.

V. TRANSFORMATIONS

The synthesis phase of RAB utilizes a well known transformation methodology hereafter referred to as Linear Algorithm Transformation (LAT). This methodology, which is described in ([FoPa84], [FoMo85], and references therein) generates a transformation matrix, $T = \begin{bmatrix} \pi \\ S \end{bmatrix}$, which maps the index points of the bit-level algorithm into the space-time domain. The LAT methodology uses the dependence matrix, D, to insure that generated and input data are available for usage by the scheduled PE at the scheduled time of execution for a given computation. Due to this fact, only distance vectors for data dependencies and buffering vectors for input variables are extracted in the

dependence/broadcast analysis module. The first component of T, $\pi \in Z^{(1 \times n)}$, corresponds to the time transformation; the second component, $S \in Z^{((n-1) \times n)}$, corresponds to the the space transformation. These components are described in the following subsections.

Time Transformations

The linear time transformation, $\pi \in Z^{(1 \times n)}$ maps the index set of the algorithm into the unidimensional time space, $\pi: J^n \rightarrow t$. Given the time transformation, π, the time of execution of a computation indexed by j is given by:

$$t(\bar{j}) = \left\lfloor \frac{\pi \bar{j} + O}{\text{disp } \pi} \right\rfloor \qquad (5.1)$$

where disp $\pi = \min\{\pi \bar{d}_i, \bar{d}_i \in D\}$ ($\bar{d}_i$ corresponds to the ith column vector in D) and $O = -\min\{\pi j: j \in J^n\} + 1$. The constant O forces the first computations to be executed at time t=1. The parameter disp π represents the maximum number of parallel arithmetic computations executed in each processing element. We restrict the value of disp π to one. This restriction is representative of the systolic array used with RAB (GAPP) and some other available architectures (i.e., MPP, DAP, CLIP). Given disp $\pi = 1$, the total execution time of an algorithm is represented by the expression

$$\left\{1 + \sum_{i=1}^{i=n} |\pi_i| (N_i - L_i)\right\} \delta \qquad (5.2)$$

where N_i and L_i correspond to the upper and lower bounds of the loop variable j_i respectively and δ represents the number of clock cycles needed for the execution of the arithmetic computations. To insure that the ordering determined by π is an execution ordering, we impose the restriction that $\pi d_i > 0$ for all $d_i \in D$.

The time transformation, π, is found by trying to minimize the function (5.2) which is monotonic in terms of the entries of π. Due to the monotonicity of the function, we use a heuristic approach to generate π, similar to the one presented in [OKFo86]. We start with all entries of π being zero and progressively increase the sum of the absolute value of the entries of each π. All possible combinations of signs of each π are considered with the exception of those obtained by negating previously generated π's. We then check the validity of each of the π's. The valid time transformations, i.e. those for which $\pi d_i > 0$ for all $d_i \in D$, are ordered according to the execution time (5.2). Possible π's, which might result from further increases in the absolute value of the entries of a particular π, for which execution time is larger than the known minimum, need not be considered due to the monotonicity property mentioned above. The ordered list of π's is used to generate the space transformations.

Space Transformation

The space transformation, S, determines the spatial mapping of the algorithm into the systolic array. This mapping requires knowledge of the essential characteristics of the given architecture. These characteristics are represented in the architecture model defined in the following definition.

Definition 5.1

The *systolic architecture* is a four tuple $<L^q, P, R, T>$ where L^q is the index set of the processor array, P is the matrix of interconnection primitives, R is the set of resources available in each PE, and T is the local execution time of a computation.

Each point $\bar{\ell} \in L^q$ corresponds to the relative location of a processing element in the systolic array. The matrix of interconnection primitives is such that if $\bar{p} \in P$ then for any $\bar{\ell} \in L^q$, $\bar{\ell}$ is connected to $\bar{\ell}' = \bar{\ell} + \bar{p}$ if $\bar{\ell}' \in L^q$ and $\bar{\ell}$ is connected to an input-output port if $\bar{\ell}' \notin L^q$. The set consisting of the resources available in each PE, R, is used in the microcode optimization phase to detect machine dependencies. If two instructions require resources beyond those given in R (i.e., the case when two RAM ports are required and only one port is given in R) a machine dependence exists between the two statements. The local execution time, T, represents the worse case time for the execution of an instruction. This value is used to calculate δ in (5.2) to determine the worse case execution time.

The two parameters of the architecture model, L^q and P, define the global topology of the systolic architecture. The other two parameters, R and T, define the local architecture of each PE comprising the systolic array. The parameters of the systolic architecture for the GAPP array (shown in figure 3) are given below.

Example 5.1

The GAPP array shown in figure 3 is a systolic architecture which can be represented by the four-tuple $<L^q, P, R, T>$ where the index set of one chip is given by

$$L^2 = \left\{(\ell_1, \ell_2): 1 \le \ell_1 \le 12,\ 1 \le \ell_2 \le 8\right\}$$

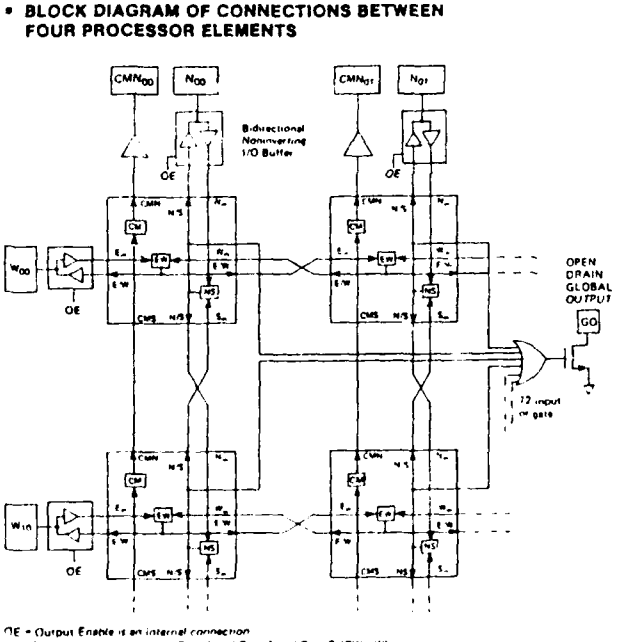


Figure 3. Representation of GAPP interconnection scheme [DaTh84].

The matrix of interconnection primitives is

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & -1 & 0 \end{bmatrix}.$$

The set of shared resources available in each processing element is given by

$$R = \left\{ \text{cm reg., ns reg., ew reg., c reg., ram port, ALU} \right\}$$

and the worse case execution time of a computation, T, is assumed to be three clock cycles (one clock cycle to place data in proper registers and execute a "full add" operation, and two clock cycles to place data in the proper register for shifting purposes).

end of example.

In mapping an algorithm into a systolic array, the main goal is to insure that the data communication between processors can be accomplished using the given interconnection primitives. In other words, if a computation performed by processor $\vec{p}_1$ at time t_1 depends on data generated by processor $\vec{p}_2$ at time t_2, then there must be a composition of interconnection primitives that connects $\vec{p}_2$ to $\vec{p}_1$ in time $t_1 - t_2$. The composition of interconnection primitives is given by the matrix $K \in Z^{(r \times m)}$. To insure that a direct path is taken for the movement of data, we restrict all entries in a column of K to have the same sign. Given these parameters, the spatial transformation, S, must satisfy the following set of diophantine equations

$$SD = PK \tag{5.3}$$

where $S \in Z^{(q \times n)}$, $D \in Z^{(n \times m)}$, $P \in Z^{(q \times r)}$, and $K \in Z^{(r \times m)}$.

The sum of the absolute value of the entries of column i of the K matrix represents the total number of data movements for the corresponding data item associated with column i of the dependence matrix. This sum is upper bounded by $\pi\bar{d}_i$, the upper bound on the propagation time. We require the column sum to equal $\pi\bar{d}_i$ since we include the buffering primitive, $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$, in our set of interconnection primitives. Only one interconnection primitive for each unique data link is included in the P matrix, i.e., even if a data link is bi-directional, we include only one primitive corresponding to one of the directions. Consequently, the matrix of interconnection primitives used with RAB for the GAPP architecture contains only the first three columns of the P matrix given in example 5.1.

If no solution exists to (5.3), we select another π from the ordered list with minimal increase in execution time. If solutions exists to (5.3), we order the transformation matrices (composed of an S and the corresponding π) according to the AT (area × time) criteria. We then choose the first transformation matrix in the ordered list for which a conflict does not occur. A conflict occurs when two or more computations are mapped into the same PE to be executed at the same time given that only 1 ALU is available in each PE. In other words given two computations indexed by $\bar{j}'$ and $\bar{j}''$, a conflict occurs when $T\bar{j}' - T\bar{j}'' = 0$ or

$$T(\bar{j}' - \bar{j}'') = 0 \tag{5.4}$$

where $\bar{j}' - \bar{j}''$ represents the conflict vector. The conflict vectors are generated using an analysis scheme similar to the one used with the generation of buffering vectors for input variables. If the conflict vector exists within the given iteration space, we disregard the corresponding T and check the next transformation matrix in the ordered list. We continue this procedure until a conflict-free algorithm transformation can be found for the partial mapping of the bit-level algorithm into the variable size array. Further details about the conflict vector can be found in [Tayl86]. The conflict-free transformation matrix for the two convolution expansion are given below.

Example 5.2

For the algorithm given in figure 2a, the conflict-free transformation matrix for iteration space defined as $\{1 \le j_1 \le 3,\ 1 \le j_2 \le 3,\ 1 \le j_3 \le 3,\ 1 \le j_4 \le 5\}$ is

$$T = \begin{bmatrix} 3 & 1 & 2 & 1 \\ -1 & -1 & 0 & 0 \\ -2 & 0 & 1 & 0 \end{bmatrix}$$

with execution time of 17δ and spatial requirements of 1 GAPP chip.

The conflict-free transformation matrix for the algorithm in figure 2b with the same iteration space is

$$T = \begin{bmatrix} 5 & 4 & 2 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

with execution time of 27δ and spatial requirements of 1 GAPP chip. Both transformations optimize the measure A × T, where A corresponds to the number of GAPP chips.

end of example.

A full design of an algorithmically-defined array can be specified by generating a transformation matrix using the interconnection primitives for a planar array and modifying the local systolic architecture parameters, R and T, to model a general processing element. The transformation matrix for a full mapping can be generated using the same techniques described for a partial mapping, with the exception of the selection of an S which satisfies the given spatial constraints of the fixed size array. For the case where an S cannot be found which satisfies these constraints, algorithm partitioning is required. The next section discusses the last phase of RAB-mapping.

VI. MAPPING

The mapping phase of RAB consists of the last 2 modules of the flow diagram shown in figure 1, microcode expansion and microcode optimization. Microcode expansion consists of the replacement of the given transformed computations with GAPP code (or the code unique to the architecture used for execution). This code is then optimized using a modified version of a technique developed by Ramamoorthy ([Rama66], [RaGo69]) known as Precedence Partitioning. The straight-line microcode is parsed in a sequential manner placing used and generated variables in a symbol table. If a used variable is encountered, the optimization function checks the symbol table to see if this variable has been generated in a previous statement resulting in a data dependence. The same applies for the other two algorithm dependencies. For the statements which are

algorithm independent, we pairwise check the resources required for the parallel execution of two statements. If the required resources exceed the resources available in R, then a machine dependence exists between the two statements. The algorithm and machine dependencies are represented in a ((v-1) × v) connectivity matrix, where v is the number of statements in the straight-line code. The element c_{ij} has value 1 if statement j is dependent on statement i and it has value 0 otherwise. The precedence partitioning algorithm uses this matrix to partition the set of computations into independent groups by locating columns containing zeros and deleting the row corresponding to the partitioned statement. The partitions are executed serially but the statements within the partitions are executed in parallel. An example of the precedence partition for straight line code is given below. An example using GAPP instructions would require detailed knowledge about the GAPP architecture, which is beyond the scope of this paper.

Example 6.1

For the following straight line code

(1) A = B + C
(2) D = A + E
(3) F = D + E
(4) G = H + I

the connectivity matrix is given by

$$C = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

The following partitions result from this matrix.
{ 1, 4 }, { 2 }, { 3 }.

end of example.

VII. CONCLUDING REMARKS

In this paper we presented the overall organization of RAB and discussed the concepts necessary for the mapping of a class of numerical algorithms into bit-level systolic arrays. We also presented a method for identifying and possibly eliminating the occurrence of a conflict. A conflict is more likely to occur with bit-level algorithms, since bit-level expansions usually result in the addition of 2 or 3 nestings of loops to the original algorithm. Thus the iteration space of the bit-level algorithm with dimension greater than 3 is mapped into the space-time domain consisting of 3-dimensional space. This mapping of n-dimensional space (where $n > 3$) into 3-dimensional space generally results in serializing some of the loops of the iteration space. With the use of the conflict analysis, we search for a transformation matrix in which a conflict occurs outside of the given iteration space resulting in a bijective mapping.

RAB currently maps numerical algorithms into variable size arrays composed of GAPP chips. However, this tool can be used to investigate the optimality (in terms of spatial requirements and total execution time) of different expansions of the same task. The results of these investigations can be used to efficiently design algorithms for parallel execution. In this paper, two different expansions for convolution were transformed via RAB into algorithms for parallel execution. The expansion given in figure 2a resulted in an execution time of 17δ and the second expansion given in figure 2b resulted in an execution time of 27δ. Thus, even though both expansions performed the same task, one expansion was more suitable for parallel execution as evident by the total execution time (both expansion required only 1 GAPP chip).

ACKNOWLEDGEMENTS

The authors would like to acknowledge Weijia Shang for introducing the conflict concept and coining the term "conflict vector". We are indebted to Steve Hand, Timothy Guinter, Daniel Wong, Wei-Min Lin, and M. David Fields for their assistance with the implementation of RAB.

REFERENCES

[Batc80] Batcher, K.E., "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, Vol. C-29, September 1980, pp. 836-840.

[Boot51] Booth, A.D., "A Signed Binary Multiplication Technique," *Quart. Journal of Mechanical and Applied Mathematics*, Vol. 4, 1951, pp. 236-240.

[DaTh84] Davis, R and D. Thomas, "Systolic Array Chip Matches the Pace of High-Speed Processing," *Electronic Design*, October 1984, pp. 207-218.

[DuWa75] Duff, M.J.B. and D.M. Watson, "CLIP3: A Cellular Logic Image Processor," in *New Concepts and Parallel Information Processing*, Noordhoff, Leyden, pp. 75-86.

[FoPa84] Fortes, J.A.B. and F. Parisi-Presicce, "Optimal Linear Schedules for the Parallel Execution of Algorithms," *International Conference on Parallel Processing*, 1984, pp. 322-329.

[FoMo84] Fortes, J.A.B. and D.I. Moldovan, "Data Broadcasting in Linearly Scheduled Array Processors," *11th Annual International Symposium on Computer Architecture*, June 1984, pp. 224-231.

[FoMo85] Fortes, J.A.B. and D.I. Moldovan, "Parallelism Detection and Transformation Techniques," *Journal of Parallel and Distributed Computing*, August 1985, pp. 277-301.

[Foetal85] Fortes, J.A.B., Wah, B.W., Fu, K.S., "Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays," *IEEE International Conference on Acoustics, Speech and Signal Processing*, 1985, pp. 300-303.

[Gaetal81] Gajski, D.D., Kuck, D.J., Padua, D.A., "Dependence Driven Computation," COMCON Spring 1981, pp. 168-172.

[Hwan79] Hwang, Kai, *Computer Arithmetic: Principles, Architecture, and Design,* John Wiley & Sons, Inc., New York, 1979.

[Kuetal74] Kuck, D.J., Budnik, P.P., Chen, S., Lawrie, D.H., Strebendt, R.E, Davis, E.W., Han, J., Kraska, P.W., Muraoka, Y., "Measurements of Parallelism in Ordinary Fortran Programs," *Computer,* January 1974, pp. 37-45.

[Kuetal81] Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B., Wolfe, M., "Dependence Graphs and Compiler Optimizations," *Proc. 8th ACM Symp. on Principles Programming Languages,* January 1981, pp. 207-218.

[Kuhn80] Kuhn, R.H., "*Optimization and Interconnection Complexity for: Parallel Processors, Single Stage Networks, and Decision Trees,*" PhD dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

[Kung82] Kung, H.T., "Why Systolic Architectures?" *IEEE Computer,* January 1982, pp. 37-46.

[Mcetal84] McCanny, J.V., McWhirter, J.G., Wood, K., "Optimised bit level systolic array for convolution," *IEE Proceedings,* Vol. 131, Pt.F, October 1984, pp. 632-637.

[Mcetal85] McWhirter, J.G., Wood, D., Wood, K., Evans, R.A., McCanny, J.V., McCabe, A.P.H., "Multibit Convolution Using a Bit Level Systolic Array," *IEEE Transactions on Circuits and Systems,* Vol. CAS-32, January 1985, pp. 95-99.

[McMc82] McCanny, J.V. and J.G. McWhirter, "Implementation of Signal Processing Functions Using 1-Bit Systolic Arrays," *Electronics Letters,* Vol. 18, March 1982, pp. 241-243.

[Mold82] Moldovan, D.I., "On the Analysis and Synthesis of VLSI Algorithms," *IEEE Transactions on Computers,* Vol. C-31, November 1982, pp. 1121-1126.

[OKFo86] O'Keefe, M.T., Fortes, J.A.B., "A Comparative Study of Two Systematic Design Methodologies," *1986 International Conference on Parallel Processing,* pp. 672-675.

[Rama65] Ramamoorthy, C.V., "Connectivity Considerations of Graphs Representing Discrete Sequential Systems," *IEEE Transactions on Electronic Computers,* October 1965, pp. 724-727.

[RaGo69] Ramamoorthy, C.V. and Gonzalez, "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs," *1969 Fall Joint Computer Conference,* Vol. 35, Montvale, NJ: AFIPS Press, pp. 1-15.

[Redd79] Reddaway, S.F., "The DAP Approach," in *Infotech State of the Art Report on Supercomputers,* Infotech Ltd., Maidenhead, 1979, Vol. 2, pp. 309-329.

[Wolf82] Wolfe, M.J., "*Optimizing Supercompilers for Supercomputers,*" PhD dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, October, 1982.

REFERENCE NO. 6

Carlson, W. W. and Fortes, J. A. B., "On the Performance of Combined Data Flow and Control Flow Systems: Experiments Using Two Iterative Algorithms," International Conference on Parallel Processing, August 1987.

Carlson, W. W. and Fortes, J. A. B., "On the Performance of Combined Data Flow and Control Flow Systems: Experiments Using Two Iterative Algorithms," Journal of Parallel and Distributed Computing, to appear.

On the Performance of Combined Data Flow and Control Flow Systems: Experiments Using Two Iterative Algorithms

William W. Carlson and Jose A.B. Fortes*
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

Abstract – Improved multiprocessor performance can be attained by combining data flow and control flow concepts. This type of combined architecture is characterized and several examples of previously proposed machines are given. A new model is presented that permits the analysis of such systems and performance measures are defined. This model is then used to analyze the performance of the algorithms under a wide variety of combined systems. The results of these experiments show that partition size is a major factor in the performance of such systems and an optimal size may be found for given system parameters.

1. Introduction

This paper investigates the performance of architectures that combine concepts of both data flow and control flow computers. In particular the relationships between the granularity of program partitions and the architecture's performance in terms of execution space and time requirements are examined. These relationships are determined by studying the performance of two iterative algorithms. It is shown that, for these algorithms, partition size has a major effect on the performance of combined architectures and that an optimal partition size may be found.

Recently, there has been considerable interest in combining some of the concepts associated with data flow computers with some of those from the realm of more conventional control flow multiprocessors. This interest seems well founded. While data flow concepts offer the promise of much increased execution speed by removing artificial sequencing constraints, their advancement is slowed by seemingly insurmountable problems [GaP82]. Meanwhile a large portion of the thirty years work devoted to the study of control flow methods does not appear directly applicable to data flow computers.

This paper proceeds by quickly reviewing some previous work in combined systems, describing both existing and proposed systems. Next, a model to facilitate the performance evaluation of multiprocessor systems is described. With this background the paper describes several experiments performed to illustrate relationships between granularity and performance. Finally, several conclusions based on the results of these experiments are given.

* This work was supported in part by the National Science Foundation under Grant DCI-8419745 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contract No. 00014-85-k-0588.

2. Combined Architectures

We define data flow and control flow as schemes to determine the ordering of computational steps in parallel programs. A *pure data flow* scheme sequences operations based only on the availability of their operands and adequate computational resources. In this sense pure data flow is a fully decentralised system. Conversely, a *pure control flow* scheme is based on a schedule independent of the availability of an operation's operands. In this sense pure control flow is a fully centralised system. Of course a "good" control flow scheme will generate a sequence of computational steps that guarantees data availability. A combined system is simply a mix of these two models of computation.

While most parallel systems are not "pure" in their ordering scheme, we will be studying only those systems that combine wide variations in their ordering scheme. In such systems there is, in general, a division of labor between various nearly pure ordering schemes, with the division based on the granularity of partitioning. A graphical illustration of this combination is the *ordering scheme graph,* shown in Figure 1. The ordinate defines the level of granularity, with smaller values representing smaller granules of space and time. The abscissa represents the degree to which ordering is decentralised. The range on this axis is arbitrarily set from zero to one, with zero representing a pure control flow ordering scheme and one representing pure data flow. The resulting graph is a set of coordinate points showing the level of centralisation at each level of sequencing.

2.1. Examples of Combined Systems

Recent research efforts have produced numerous proposals that combine the concepts of data flow and control flow. This section describes several, showing an ordering scheme graph for each. The purpose of this section is to point out the variety of current proposals, not to discuss their relative merits.

The Piecewise Data Flow Architecture [ReM83] uses a two level approach, with distinction occurring at the granularity level of the basic block. A basic block, a term commonly used in compiler theory, is a sequential program section that has only one entry point and one or more exit points. Internal to a basic block a data flow scheme sequences operations, while the collection of basic blocks that make up a program are executed sequentially, with possible overlap between two blocks. In addition to the data flow scalar processors, an SIMD processor is included to allow fast execution of vector operations. This makes this machine an example of a truly combined architecture, with the combination being segregated into data flow and control flow sections. The goal of the architecture is to allow sequential portions of scientific programs to enjoy the speedup that vector portions already receive on systems like the Cray-1.

The ordering scheme graph for the Piecewise Data Flow architecture shows all low granularity operations, up to the level of the basic block, have a high level of decentralisation as they are sequenced using data flow concepts. Operations with larger granularity would be almost completely centralised, as they are sequenced serially. A possible ordering scheme graph for this architecture is shown in Figure 2a. As a basic block can have a range of granularities, the

transition between data flow and control flow is shown as a region instead of a point and symbolised by the dotted area on the graph. When considering this system, there are several important features that our study of combined systems must consider. The first is obviously the combination of data flow and control flow at different levels of granularity. Another important feature is the level of granularity at which the transition is made. This will have an important effect on the performance of such a combined system. Finally this architecture limits concurrency by allowing only one block (perhaps overlapped with the setup of one other) to execute at any given time.

The Cedar project [GaL84] proposes another split level control scheme, with division at the compound function level. The granularity of compound functions [GaK81] is slightly greater than basic blocks, with operations like array primitives, linear recurrences, FORALL loops, pipeline loops, block assignment statements, and compound conditional expressions. The architecture consists of a global control unit and several processor clusters. The global control unit sequences compound functions according to data flow principles. Processor clusters are assigned compound functions to execute according to the principles of control flow. Thus, in this sense, Cedar is the mirror image of Piecewise, as Cedar uses data flow to sequence large granularity items and control flow to sequence low level operations, while Piecewise does the opposite.

A scheme graph for Cedar shows all levels of granularity below that of the compound function with a low level of decentralisation. Operations above this granularity would have higher levels of decentralisation. Figure 2b shows a possible ordering scheme graph for Cedar. Important points about Cedar are similar to those observed for Piecewise, namely the change in ordering scheme is directly related to the granularity of operations, and the level of granularity of where this change occurs. Finally, this architecture's parallelism between compound functions is limited only by the parallelism available between them and the availability of processor clusters. Parallelism within a compound function may be limited by the control flow scheme used by the processor cluster, although the compound functions have regularly structured parallelism that may be easily exploited.

Remps [HwX85] has the same goal of both Cedar and Piecewise, i.e. scientific computation. The global structure of the architecture is similar to Cedar: a collection of interconnected processors and a global controller. The key difference is that Remps allows reconfiguration of interconnection and control to emulate a variety of architectures. On a global level the machine is a data flow computer and at the low level each processor is a reconfigurable control flow computer. The level of change in ordering scheme is the granularity of a task. While the term task is nebulous, it seems to describe a level of granularity slightly larger than a compound function.

Drawing a scheme graph for Remps requires the understanding that, as in Cedar, all low granularity items show low levels of distribution. Large granularity objects may be sequenced centrally (called macro-pipelining), or sequenced in a distributed fashion (called macro-data flowing). A possible scheme graph is shown in Figure 2c. The major interesting feature of this architecture is the two schemes that exist simultaneously for large granularity items, although only one is used in the sequencing of a single set of operations.

The Rediflow multiprocessor [KeL84] contains a complex combination of data flow, control flow and reduction concepts. Reduction is another decentralized ordering scheme in which the *demand for the result of a computation causes its execution.* Rediflow consists of interconnected *Xputers* that combine processor, memory, and packet switch elements. The Xputers function under a reduction ordering scheme. As with the previously described architectures, Rediflow exhibits a change in sequencing at some level of granularity. Here the granularity is medium or function-level. This level is taken to be about the same as basic blocks. Higher granularity items are sequenced by either reduction, data flow or control flow. Data flow provides efficient pipelining, while reduction may be more adaptable to programs requiring unpredictable buffering. In addition, control flow sequencing is available by so called "von Neumann processes".

In drawing an ordering scheme graph for Rediflow, reduction presents a new issue to be represented. The basic ordering scheme graph is augmented with the abscissa extending from -1 to −1, with +1 representing pure data flow as before. The negative range represents demand *driven schemes.* This extension shows the degree of decentralization by the absolute value of the abscissa, while the sign determines if the operations are executed on demand (negative) or availability (positive). This is a proper extension of the ordering scheme graph in that a "pure" demand driven system is also fully decentralized (represented by -1) and any given operation will be executed under either demand or data flow, never both. Figure 2d shows a possible scheme graph for Rediflow. All operations below a medium granularity are given a high level of decentralization in the negative portion of the graph to show demand driven computation. Above this level, three possible schemes exist, resulting in the levels for data flow, reduction, and von Neumann processes. When seen in this light, the sequencing characteristics of Rediflow appear somewhat similar to Remps as large granules may be sequenced in one of several methods. Obviously low level sequencing is totally different. This brief survey can be concluded by reiterating that the ordering scheme graphs show a wide diversity in the approaches used in combining data flow and control flow concepts. Many other combinations are possible, conceivably as many as possible ordering scheme graphs.

2.2. A Variable Combined Architecture

This study does not investigate previously proposed combined systems, but concentrates on one extremely flexible hypothetical architecture. This hypothetical system consists of interconnected processing elements each capable of communicating and controlling each other. An equal delay and infinite capacity communication path exists between each pair of processing elements. The ordering scheme for this system is a variable, two level approach. Larger granules are sequenced according to either data flow or control flow principles, while smaller granules are sequenced by the opposite approach. The size at which the switch occurs, as well as the relative costs of performing various operations are left as variables in the experiments.

This approach has several distinct advantages over analyzing specific systems, and a few shortcomings. The greatest advantage is the availability of the complete range of systems between data flow and control flow, approaching these by either increasing or decreasing

partition size. In addition, this approach avoids the problems associated with comparing two distinct systems, concentrating instead on the underlying differences between ordering schemes. Finally, this approach limits the problem by ignoring, at this point, such issues as network topology. Of course, this advantage can also be a shortcoming when these particular issues play a dominant role in the system. This topic is currently left for our further research.

3. The COSMIC Performance Evaluation Model

To analyze the performance of combined data flow and control flow systems we have developed *COSMIC,* the Combined Ordering Scheme Model with Isolated Components. COSMIC consists of both formal parameters describing a multiprocessor system and the algorithm it executes, and analysis techniques producing performance measures. The underlying principles of this model are the isolation of individual performance issues and the study of systems under conditions close to those encountered when a system is performing useful calculations.

Previous work in modeling multiprocessors has centered in several distinct areas. Program behavior models endeavor to model the fundamental properties of a program without regard for hardware considerations or performance measurement. They center on the important areas of investigating such problems as the determinacy, boundedness, and termination of programs. Models fitting this category include Petri nets [Pet66] and Parallel Program Schemata [KaM66]. Petri nets have been augmented with the notion of time in either the deterministic [RaH80, Ram74] or stochastic [Mol85] sense. The second major category of current models we call machine behavior models as they describe the behavior of machines in their execution of programs as opposed to the behavior of programs themselves. Examples of this class include Turing Machines, Functional Programming Systems, and the von Neumann Model [Bac78]. Classification models describe the configuration and operation of multiprocessors, including Flynn's Model [Fly66], Handler's Classification System [Han77], and the "essential issues" of Gajski and Peir [GaP85].

COSMIC builds on these previous efforts, but is fundamentally different from them in that it combines both program and machine descriptions, as well as performance measures. The usefulness of this model is in this combination, allowing the study of complete systems under varied conditions. This section briefly describes COSMIC and its operation. A more complete description is available in [CaF87].

3.1. COSMIC Parameters

The parameters of a system S include O, the system's organization; G_1, a dependence graph describing a specific algorithm; and OS, the ordering scheme used to execute algorithms on the organization. Included in a system's organization parameter are such features as the number and capacity of processing elements, the amount and organisation of memory, and the interconnection amongst processors and memory. The dependence graph is simply an operation level precedence graph for a certain algorithm. This graph includes only algorithmic

constraints, not those induced by operation sequencing or programming languages. Finally, the ordering scheme describes how algorithms are executed on the organisation. The ordering scheme is further segregated into descriptions of a system's mechanisms for partitioning, sequencing, resource allocation, and memory utilisation.

3.1.1. Organisation (O)

The organization represents the arrangement of hardware elements in a system. Every multiprocessor has three basic components: processing elements (PE), memory locations, and the interconnections between them. Input and output devices are simply treated as specialized processing or memory elements. Consequently, our model for organization is represented by the triple $O = (P, M, I)$.

- P -- A set of processing elements. Each processing element has a set of instructions that it can execute and a relative speed.
- M -- A set of memory locations.
- I -- An interconnection function $M \times P \rightarrow M \times P$. This function defines the possible interconnections, and with each outcome there is a related cost function that describes the cost of traversing that connection. Local memory on a PE can be modeled by a low cost (perhaps zero) of traversing the connection. Inaccessible memory (some other PE's local memory) can be modeled by a partial function.

3.1.2. Data Dependency Graph (G_d)

The data dependence graph is an arc and vertex weighted directed graph in which vertices represent operations and arcs represent data dependencies between operations. The weight of a vertex represents the relative time that it will consume when executed. The weight of an arc represents the size of the data needing transfer to satisfy the dependency. These weights can also be viewed as the number of "atomic operations" required to complete the computational or transfer operation. This graph is acyclic, as any loops in a program are unfolded in creating the dependency graph. Currently data dependent behavior is not considered, but will be a topic for future research.

3.1.3. Ordering Scheme Function (OS)

The ordering scheme for any system is a function mapping the dependency graph into an ordering net, based on the organization parameters. An ordering net is a timed Petri net [Ram74] which depicts ordering constraints placed on the execution of operations, as well as the cost of each operation in the modeled system. The ordering scheme for an organisation, O can be defined as:

$$OS(O): G \rightarrow N,$$

where G is the set of all possible dependency graphs and N is the set of all possible ordering nets. This function is composed of several smaller, more easily defined functions. Thus the

ordering scheme function,

$$OS(O) = \gamma(O) \circ \mu \circ \lambda \circ \phi(O) \circ \tau(O),$$

where the usual composition notation implies that $(f \circ g)(x)$ is equivalent to $f(g(x))$, contains the component functions:

$\tau: G_A \rightarrow \mathcal{N}$	Creates an ordering net from G_A,
$\phi(O): \mathcal{N} \rightarrow \mathcal{N}$	Adds partitioning constraints,
$\lambda: \mathcal{N} \rightarrow \mathcal{N}$	Adds sequencing constraints,
$\mu: \mathcal{N} \rightarrow \mathcal{N}$	Adds memory access constraints and time,
$\gamma(O): \mathcal{N} \rightarrow \mathcal{N}$	Adds resource constraints.

The next sections briefly describe each function.

Computation Function (τ)

The computation function creates an ordering net from a data dependency graph. Its sole purpose is to change domains from data dependency graphs to ordering nets and is scheme independent. Scheme independence implies the function itself never changes over all possible ordering schemes. Formally the computation function is

$$N_c = \tau\left(G_A\right),$$

where N_c is the computation ordering net for a given G_A produced by τ. The process used is to transform vertices in G_A to transitions in N_c, and arcs in G_A to places in N_c. Connections in N_c are created to preserve the structure of the dependency graph. Finally, appropriate weights are assigned to places and transitions, based on the speed of processing elements as defined by the organisation parameter.

Partitioning Function (ϕ)

Partitioning is the process of dividing a program into segments to allow their execution on possibly distinct execution units. This division requires the addition of explicit synchronization operations between segments to preserve data dependencies. The partitioning function creates a new ordering net based on these added synchronisation requirements:

$$N_{part} = \phi\left(O, N_c\right).$$

Unlike the computation function, the partitioning function is scheme dependent but always follows a similar form. First the net is divided into segments by a scheme dependent algorithm. Next, any transition connected via a single place to a transition in another segment is synchronised by a place-transition-place sequence between the two transitions. The transition models the computation required to complete the synchronisation, while the places model data transfer required to perform this function. Finally, weights are assigned to the newly created places and transitions, commensurate with the cost of synchronisation on the system. This function may

be applied recursively to segments to model multi-level ordering schemes.

Sequencing Function (λ)

The sequencing function is responsible for adding constraints to the model induced by the sequencing of segments and operations within those segments. This function causes the interpretation of either control flow, data flow, or combined schemes. Formally, the sequencing function produces a new ordering net from its input:

$$N_{seq} = \lambda\left(N_{part}\right).$$

Again, this function is scheme dependent, and must be specifically defined for each ordering scheme. Several levels of sequencing strategy may be modeled, based on the recursions in the partitioning function. At the lowest level, place-transition-place sequences are inserted between transitions within a single segment to model their sequencing. In a data flow scheme these will be in parallel with each place, as sequencing occurs on each data transfer. In a control flow scheme, they are placed between transitions along some execution trace.

At higher levels, segments are sequenced by creating place-transition-place sequences between segments. The details of this placement are dependent on the scheduling strategy being modeled. Again, appropriate weights are assigned to all places and transitions added by this function.

Memory Access Function (μ)

Recall that in a data dependence graph an arc was weighed in accordance with the amount of information transfer required over that arc. These weights were transferred to places in the ordering net. The memory access function produces a new ordering net reflecting the added costs of memory access and interconnection network traversal:

$$N_{ma} = \mu\left(N_{seq}\right).$$

This function is scheme independent and simply replaces each non-zero weighted place with a place-transition-place sequence. The new transition is given a weight equal to the weight of the place it replaces.

Resource Allocation Function (γ)

Resource allocation is the process of assigning sets of transitions to sets of resources (processing elements and memory locations). This function produces a new ordering net limiting concurrency within these sets:

$$N_{ra} = \gamma\left(O\ ,\ N_{ma}\right)$$

The resource allocation function is organisation dependent and must be defined for each system modeled. In general, a resource allocation function will assign groups of transitions to resource pools modeled by the addition of places to the ordering net. This allows concurrency between a

set of transitions to be limited by the availability of a limited resource. It should be noted that this also allows the modeling of resource contention for memory devices.

3.2. Analysis and Measures

After a system has been described by the parameters of COSMIC, it is analyzed to determine several performance measures. This analysis involves the determination of the time between the firing of the initial and final transitions in an ordering net. Computerized analysis tools aid in this determination. The analysis begins by creating ordering nets using a high-level description language that enables the specification of parameterized nets. Generally, these parameters include the problem size and relative costs for computation, sequencing, and synchronization. A compiler then fixes values for these parameters and produces a set of interconnected places and transitions. Next, a net analyzer determines the various measures by firing the net following the rules of timed Petri nets [Ram74]. Finally the results of many analyses are gathered into a database for further off-line studies. The entire system is capable of analyzing nets up to about 50,000 places and transitions while consuming reasonable amounts of computational resources. This enables the analysis of moderately large problems.

Three values are associated with each performance measure: the serial time, the critical path time, and the number of resources required to achieve the critical path time. These values describe both the time and space requirements of the modeled system for a given configuration. Two classes of measures are used: *primary measures* represent consumption of resources directly related to the algori hmic requirements of the system, and *overhead measures* show the consumption of resources unrelated to any algorithmic requirements. The analysis consists of the application of two analysis functions. The *serial analysis* function is:

$$AN_{Serial} : N \rightarrow \boldsymbol{R},$$

where $\boldsymbol{R}$ represents the set of real numbers and N the set of ordering nets. It computes the time required to fire all nodes in an ordering net, with the added constraint that no two transitions may fire simultaneously. The *critical path analysis* function is:

$$AN_{Crit} : N \rightarrow \boldsymbol{R} \times \boldsymbol{N},$$

where $\boldsymbol{R}$ represents the set of real numbers, $\boldsymbol{N}$ the set of non-negative integers, N the set of ordering nets, and $\times$ the cross product. It computes the time required to fire all nodes in an ordering net, with only the constraints presented by the net, as well as the number of resources required to achieve that level of performance. Finally the general analysis function,

$$AN : N \rightarrow \boldsymbol{R} \times \boldsymbol{R} \times \boldsymbol{N},$$

simply combines of the two previous functions, the result of which is a triple of values: (Serial Time, Critical Path Time, Critical Path Space).

If we let M be such a triple, the total execution measure for a model with organization O, ordering scheme OS, and data dependency graph G_i is:

$$M_{execution} = AN\Big[OS\big[G_i, O\big]\Big].$$

The execution measure is also, by definition, the sum of the five previously defined measures:

$$M_{execution} = M_c + M_{part} + M_{seq} + M_{ma} + M_{ra},$$

where M_c is the computation measure, M_{part} the partitioning costs measure, M_{seq} the sequencing costs measure, M_{ma} is the memory access measure, and M_{ra} is the resource allocation costs measure. All the submeasures are also triples, the addition of which is defined in the usual manner by adding corresponding entries. These measures represent the analysis of an ordering net resulting from the application of a subset of the ordering scheme function. M_c is the primary measure, while the others are overhead measures. Overhead measures may contain negative entries for Critical Path Space: when the critical path time grows, the space required to achieve that performance may decrease.

4. Experiments and Results

This section presents results gauging the effect of partition size on combined system performance. The performance of two simple algorithms is studied in the environment of a hypothetical architecture capable of executing instructions under the control of a variety of ordering schemes. The organisation, data dependency graphs, and the various functions of the ordering scheme that manipulate them are described. Numerical results from experiments are presented graphically and as polynomial equations. As a compromise between the infinite variability of our hypothetical architecture and the availability of computational resources to analyse our systems, several restrictions are imposed on the experiments. Specifically, the scope of analysis is limited by assuming that resource allocation and memory access constraints are ignored. This will lead to resource allocation and memory access functions being set equal to the identity function. The resource allocation limitations can be justified by assuming equally fair and efficient implementations on all systems. The memory access limitation can be justified similarly, although different numbers of memory accesses may be required by the various ordering schemes. However, these factors may effect system performance and ongoing research is aimed at eliminating these restrictions.

4.1. The Organisation

First, the organisation parameters for our hypothetical architecture are described. As memory access or resource allocation are not considered, the organizational parameters of consequence are the number and speed of the processing elements. Both are treated as variables in these experiments. It is also assumed that all processing elements are interconnected, with a communication cost of zero from any source to any destination. Future research is planned to investigate the effects of interconnection topology and cost on combined system performance.

4.2. The Dependency Graphs

The first algorithm studied is for matrix-vector multiplication using the algorithm shown in Figure 3a, in which the matrix has size ($SIZE \times SIZE$). In forming the data dependency graph for this algorithm, note the central operations in the algorithm are the multiplication of two

numbers and the addition of the result to a running sum. This central operation will occur $SIZE^2$ times in the dependency graph. Therefore a base structure consisting of two vertices connected by a directed arc is created. The vertex at the tail of the arc represents the multiplication operation, while the vertex at the head represents the addition. Two arcs enter the multiplication vertex, representing the matrix/vector input values, and one additional arc enters the addition vertex to represent the previous value of the running sum. The addition vertex has a single output arc. Therefore, creating a dependency graph for the algorithm involves replicating this structure $SIZE^2$ times and interconnecting appropriately. Added to this graph are $SIZE$ vertices representing the input vector and $SIZE^2$ vertices representing the input array. Figure 3b shows such a graph for the case when $SIZE = 3$. In this figure the computational vertices are represented by circles and the input matrix/vector vertices by squares. Note that the input vertices are connected to the multiplication operation and the addition operations are chained to form the complete dot product operation.

The second algorithm studied computes a 4-point iterative relaxation function, using the algorithm shown in Figure 4a, in which the matrix has size ($SIZE \times SIZE$) and $ITER$ iterations are computed. When all loops are unfolded into their basic components, a central computational block repeats many times throughout the algorithm. Here, the computational block consists of three additions and a division, therefore resulting in a 4 vertex graph with 4 inputs and one output. This basic graph is repeated $SIZE^2 \times ITER$ times and appropriate interconnections are made. As the dependency graph for the complete algorithm is complex, Figure 4b shows only the central computational block. In this algorithm out-of-range indices in array subscripts "wrap-around" using the modulus function, and for simplicity initial input arcs are ignored.

4.3. The Ordering Schemes

Our experiments investigate two classes of ordering schemes. Both two level combined approaches require an ordering net partitioned into segments, with the number of segments being a variable for experimentation. The first ordering scheme, denoted *Cpart*, sequences partitions using a control flow ordering scheme, while individual operations within a partition are sequenced using data flow concepts. The other ordering scheme, denoted *Dpart*, sequences partitions using a data flow ordering scheme, while individual operations within a partition are sequenced using control flow concepts. This section discusses the specifics of the partitioning and sequencing functions for each case. The computation function, τ, assigns firing times to the transitions it creates based on a parameter of the experiments called the *computation time* Again note that γ and μ, the resource allocation and memory access functions, are the identity function.

The same partitioning function, ϕ, is used for both the Cpart and Dpart ordering schemes. As both algorithms have a grid structure, the ordering net is partitioned first by columns in that grid of operations, and then if required by rows. For example, if 3 partitions were to be created from a matrix example with $SIZE = 3$, each column in the grid of operations (see Figure 3a) would be placed in its own partition. If six partitions were required, then each of the original partitions would be divided in two. This strategy keeps operations that communicate most

often in the same partition whenever possible. Synchronisation operations are then placed between each pair of connected computational vertices that reside in different partitions. The firing time of the additional transitions is the variable called *synchronization time*.

The sequencing function for the Cpart ordering scheme, λ_C, places a data flow sequencing operation in parallel with each unsynchronised place. This enforces a low level data flow scheme. When more partitions are created than columns in the grid structure of operations, groups of "number of columns" partitions must be sequenced by "plies". To this end λ_C also forces each ply of partitions to complete execution before the next is started. This is enforced by adding a single transition between the plies. In effect this function implements the control flow synchronization strategy called "Barrier Synchronization". The firing times of the additional transitions are set to the *sequencing time* variable.

The sequencing function for the Dpart ordering scheme, λ_D, places a control flow sequencing operation between operations within partitions to assure that no concurrency will take place within a partition (i.e. a single trace of operations is executed serially). The previously added synchronisation operations already ensure data flow sequencing among partitions. The firing time of these additional transitions is again called the *sequencing time*.

4.4. The Experiments

Experiments were conducted to determine the system's sensitivity to changes in problem size and the relative time required to execute computational, synchronisation, and sequencing operations. In each experiment, the measures that comprise the triple $M_{execution}$ were determined. Each experiment consisted of setting the problem size and various time requirements constant and varying the number of partitions over the range of "uniform" sizes (i.e. those in which each partition had an equal number of operations to perform). The results of numerous experiments of varying time requirements and problem size were combined to understand the interdependence of all these factors on the performance of the systems.

After numerical results from the experiments were obtained, those related to the critical path execution time were fit to polynomial curves based on the number of partitions. Except for a few "off by one" errors at extreme partition sizes, all cases exhibit a piecewise linear relationship between the number of partitions and the critical path performance of the algorithm. Next, several equations from experiments corresponding to variations of the cost variables were combined to obtain polynomial equations for each measure based on both the number of partitions and the cost variables (e.g. sequencing time). Again all equations could be combined in a piecewise linear fashion. At this point in the analysis several equations represent each measure, one in terms of each cost variable. These equations were then unified to a single equation for each measure in terms of all the cost variables and the number of partitions. These equations can be verified by substituting appropriate constant for the cost variables to obtain the component equations. Finally, the results of experiments on different problems sizes were combined to obtain the final critical path equations for each measure.

The critical path measurement equations are shown in Tables 1 and 2 for the matrix multiplication and iterative relaxation algorithms respectively. In these tables (and the remainder of this paper), N represents the number of partitions; $SIZE$ the problem size; T_c the computation time; T_{sync} the synchronisation time; and T_{seq} the sequencing time. Also, note the ceiling function $\lceil x \rceil$ represents the smallest integer $\geq x$ and θ represents the unit step function:

$$\theta(x) = \begin{cases} 0 & \text{if } x<0 \\ 1 & \text{if } x \geq 0. \end{cases}$$

Figures 5 through 8 are graphical representations of the equations for the execution time measures (obtained by summing appropriate submeasures). The figures contain three graphs, each varying one of the cost variables. Figures 5 and 6 show matrix multiplications results while Figures 7 and 8 are from the iterative relaxation experiments. Figures 5 and 7 are for the Cpart ordering scheme while Figures 6 and 8 represent the Dpart ordering scheme. In all cases the problem size and default values of the time parameters are taken to be 8. Circles on the graphs indicate function values at uniform partition sizes.

Examination of the measure equations yields a good understanding of the performance of these two algorithms. While space limitations prevent complete analysis of these functions, available in [CaF87], this paper endeavors to provide both the flavor and some interesting results from the analysis. The matrix multiplication algorithm's computation measure is $\left(SIZE + 1\right) T_c$, which is easily explained by examining Figure 3b. The length of a critical path is one greater than the size of the problem, and each computation requires T_c to complete.

This algorithm's partitioning measure contains three components. The first two indicate that two synchronisation operations will enter the critical path when $N < SIZE$. This number increases with N after N exceeds the problem size. Two initial operations result from the synchronisations required to start and end each segment. The increasing factor that exists when there are more partitions than columns of computations ($SIZE$) results from added synchronisations needed between serial partitions. Note also the special case of one segment requiring only one synchronization operation. This increase produces the staircase nature of Figure 7 and results when a single partition is added to a "uniform" number causing critical path length to increase. The final factor results from a synchronisation operation in parallel with a computation operation becoming more dominant, within a range of partition sizes.

The Cpart ordering scheme's sequencing measure, obviously the most complex, consists of four parts. The first indicates a sequencing operation and an additional computational vertex per "layer" will enter the critical path, while a synchronization operation leaves. The next component indicates $SIZE - 1$ additional sequencing operations are in the critical path, one between each stage of the computation. The final factor is similar to the final factor of the partitioning measure, adjusting which operation are in the critical path as their costs vary.

The Dpart ordering scheme's sequencing measure also consists of four parts. The first factor indicates that indicates that as the number of partitions increases, added computational operations will drop out of the critical path, to the point where no additional operations are

placed in the path by the sequencing function when $N = SIZE^2$. The second factor shows the same trend and that there are two sequencing operations associated with each computation, with sequencing operations dropping out of the critical path as the number of partitions increases. The final factor are similar to that in the Cpart ordering scheme.

Now consider the iterative relaxation experiments. In these experiments, three iterations of the algorithm were run (i.e. $ITER = 3$) which indicates that the critical path (using a wavefront strategy) will be four times the size of the problem, minus 1. Since the critical path through a single operation in 3 operations long, the resultant computation measure is just that given. The partitioning measure indicates that when $N < SIZE$ three synchronisations are required for each partition: between each stage of the wavefront. Again the "ply" oriented synchronisations exist above this level. Again as in the matrix multiply algorithm, there is a special case when $N = 1$ with 2 fewer synchronisation operations required.

The Cpart sequencing measure consists of three components, one for each cost variable. The first component indicates that as sequencing constraints are added to the model more of the computational operations fall along the critical path. When there are fewer partitions than the problem size this is a constant factor, and above this number a linear increase is seen. The second component shows the sequencing operations that fall along the critical path, which has similar form to the added computational operations. Finally we again see that several synchronisation operations are removed from the critical path.

The Dpart sequencing measure is similar in form to the Cpart measure, except that the weight of the computational and sequencing terms decreases linearly above $SIZE$ partitions instead of increasing. These factors are also responsible for the discontinuities that exist at $SIZE$ partitions. The final two terms of this expression indicate the removal of synchronisation operations is limited, as in the matrix multiplication sequencing measures.

The following observations result from the outcomes of our experiments, as depicted in Tables 1 and 2 and Figures 5-8.

- The relationship between granularity and execution time.

  Figures 5 through 8 show that granularity has a noticeable effect on the execution time performance of these algorithms in the combined environment. In Figure 5 we see that, as N increases, the execution time increases. This is a logical outcome for the Cpart scheme, as parallelism is restricted when the partition size drops below the size containing a complete column of the calculation. Figure 6, however, shows decreasing execution time with increasing N. Again, this is logical as the Dpart scheme restricts parallelism when there are many calculations in a single partition. Interestingly, we see that analogous general trends hold in the relaxation algorithm, as illustrated by Figures 7 and 8. Tables 1 and 2 confirm these results.

- The effects of changing the relative costs of computation, synchronisation, and sequencing.

  Tables 1 and 2 show the relationships between execution time and T_c, T_{seq}, and T_{sync} are all linear for a given problem size and number of partitions.

- The dominant costs in the performance of these algorithms.

  Figures 5 through 8 show that computation and sequencing time are the dominant factors in the performance of these algorithms. The effect of increasing or decreasing their cost by a constant term increases or decreases the execution time by a factor at least three times the effect of changing the partition size by the same amount. Tables 1 and 2 confirm these results as we see larger factors associated with the T_c and T_{seq} terms than the T_{sync} terms.

- The optimal number of partitions.

  In each experiment, the optimal number of partitions varies and is dependent on the relative costs of computation, synchronisation, and sequencing operations.

  - Matrix Multiplication, Cpart Ordering Scheme -- Figure 5 shows the optimal number in all cases is a single partition.
  - Matrix Multiplication, Dpart Ordering Scheme -- Figure 6b shows that as synchronization time increases the optimal number of partitions changes from 64 ($SIZE^2$) to one.
  - Iterative Relaxation, Cpart Ordering Scheme -- Figure 7c shows that as sequencing time becomes dominant, the optimal number of partitions is 8 ($SIZE$), while Figure 7b shows that when the synchronisation time becomes dominant the optimal number is one.
  - Iterative Relaxation, Dpart Ordering Scheme -- Figure 8b illustrates that as synchronisation time becomes dominant, the optimal partition size moves from 64 ($SIZE^2$) to 1.

- The effect of changing problem size.

  Examining Tables 1 and 2 we see that problem size plays two roles in the performance of these algorithms. The first is the linearly increasing critical path execution time with increasing problems size, which is the critical path performance of these algorithms. The second role is the determination of the "uniform" number of partitions as evidenced by the $\frac{N}{SIZE}$ terms throughout these tables.

5. Conclusions and Further Work

COSMIC has been used to study combined systems, and was illustrated by studying the impact of partition size on a system's performance. This allowed the identification the optimal partition size in relation to given system parameters. While these results apply directly only to two iterative algorithms (differing mainly in their interconnectivity), they provided hints to what factors effect the performance of combined systems. Future work will focus on efforts to generalise these results to other algorithms and include the effect of memory accessing and resource allocation.

6. References

[Bac78] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM,* Vol. 21, August 1978, pp. 613-641.

[CaF87] W.W. Carlson and J.A.B. Fortes, *COSMIC: A Model for Multiprocessor Performance Analysis,* TR-EE 87-13, School of Electrical Engineering, Purdue University, 1987.

[Fly66] M.J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE,* Vol. 54, December 1966, pp. 1901-1909.

[GaK81] D.D. Gajski, D.J. Kuck, and D.A. Padua, "Dependence Driven Computation," *Proc. Compcon Spring,* February 1981, pp. 168-172.

[GaL84] D.D. Gajski, D.H. Lawrie, D.J. Kuck, and A.H. Sameh, "CEDAR," *COMPCOM Proceedings,* Spring 1984, pp. 306-309.

[GaP82] D.D. Gajski, D.A. Padua, D.J. Kuck, and R.H. Kuhn, "A Second Opinion on Data Flow Machines and Languages," *IEEE Computer,* February 1982, pp. 58-69.

[GaP85] D.D. Gajski and J-K. Peir, "Essential Issues in Multiprocessor Systems," *IEEE Computer,* Vol. 18, June 1985, pp. 9-27.

[Han77] W. Handler, "The Impact of Classification Schemes on Computer Architecture," *1977 Int'l. Conf. on Parallel Processing,* August 1977, pp. 7-13.

[HwX85] K. Hwang and Z. Xu, "Remps: A Reconfigurable Multiprocessor for Scientific Supercomputing," *1985 Int'l. Conf. on Parallel Processing,* August 1985, pp. 102-111.

[KaM66] R.M. Karp and R.E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM J of App. Math,* Vol. 14, November 1966, pp. 1390-1411.

[KeL84] R.M. Keller and C.H. Lin, "Simulated performance of a Reduction-Based Multiprocessor," *IEEE Computer,* Vol. 17, July 1984, pp. 70-82.

[Mol85] M.K. Molloy, "Discrete Time Stochastic Petri nets," *IEEE Transactions on Software Engineering,* Vol. SE-11, April 1985, pp. 417-423.

[Pet66] C.A. Petri, *Communication with Automata,* Supplement to RAD C-TR-65-337, Graffis Air Force Base (translated from Kommunikation mit Automatin, Univ. Bonn, Bonn, Germany, 1962), 1966.

[RaH80] C.V. Ramamoorthy and G.S. Ho, "Performance Evaluation af Asynchronous Concurrent Systems Using Petri Nets," *IEEE Transactions on Software Engineering,* Vol. SE-5, September 1980, pp. 440-449.

[Ram74] C. Ramchandani, *Analysis of Asynchronous Concurrent Systems by Timed Perti Nets,* MAC-TR-120, Project MAC, MIT, 1974.

[ReM83] J.E. Requa and J.R. McGraw, "The Piecewise Data Flow Architecture: Architectural Concepts," *IEEE Transactions on Computers,* Vol. C-32, May 1983, pp. 425-438.

Table 1

Matrix Multiplication Critical Path Measures

Measure	Equation
Computation	$\left(SIZE + 1\right) T_c$
Partitioning	$T_{sync} + \theta\left(N - 2\right)\left\lceil \frac{N}{SIZE} \right\rceil T_{sync}$ $+ \theta\left(N - SIZE\right)\theta\left(SIZE^2 - N\right)\theta\left(T_{sync} - T_c\right)\left(T_{sync} - T_c\right)$
Sequencing (CPART)	$\left\lceil \frac{N}{SIZE} \right\rceil \left(T_{seq} + T_c - T_{sync}\right) + \left(SIZE - 1\right) T_{seq}$ $+ \theta\left(N - SIZE\right)\theta\left(SIZE^2 - N\right)$ $\left\{\theta\left(T_{sync} - 2T_c - T_{seq}\right)\left(T_{sync} - 2T_c - T_{seq}\right)\right.$ $\left. - \theta\left(T_{sync} - T_c\right)\left(T_{sync} - T_c\right)\right\}$
Sequencing (DPART)	$\left\{SIZE - \left\lceil \frac{N}{SIZE} \right\rceil\right\} T_c + 2\left\{SIZE + 1 - \left\lceil \frac{N}{SIZE} \right\rceil\right\} T_{seq}$ $+ \theta\left(T_{seq} - T_{sync}\right)\left(T_{seq} - T_{sync}\right)$ $- \theta\left(N - SIZE\right)\theta\left(SIZE^2 - N\right)\theta\left(T_{sync} - T_c\right)\left(T_{sync} - T_c\right)$

$$\theta(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0. \end{cases}$$

$$\left\lceil x \right\rceil = \text{Smallest integer} \geq x.$$

Table 2

Iterative Relaxation Critical Path Measures

Measure	Equation
Computation	$\left(12\ SIZE - 3\right) T_c$
Partitioning	$\min\left\{\left(3N-3\right),\ \left\{\frac{N}{SIZE} + 3\ SIZE - 4\right\}\right\} T_{sync} + \theta\left(N-2\right) 2T_{sync}$
Sequencing (CPART)	$\left(9SIZE - 9\left\lceil\frac{N}{SIZE}\right\rceil - 2SIZE + 6\right) T_c$ $+ \max\left\{\left(18SIZE - 6 - 3N\right),\ \left(9\ SIZE + \left\lceil\frac{N}{SIZE}\right\rceil\left(6SIZE - 6\right)\right)\right\} T_{seq}$ $+ \left\{\left\lceil\frac{N}{SIZE}\right\rceil\left(3SIZE - 4\right) - 20 - \theta\left(N-2\right)2\right\} T_{sync}$
Sequencing (DPART)	$\left\{3\ SIZE - 3\left\lceil\frac{N}{SIZE}\right\rceil + 2\theta\left(SIZE - N\right)\right\} T_c$ $+ \left\{14\ SIZE - 1 - 6\left\lceil\frac{N}{SIZE}\right\rceil + 4\ \theta\left(SIZE - N\right)\right\} T_{seq}$ $+ \left\{2SIZE - 2 - 2\left\lceil\frac{N}{SIZE}\right\rceil - \theta\left(SIZE - N\right)\left(2SIZE - 2 - 2N\right) + 2\theta\left(N-2\right)\right\} T_{syn}$

$$\theta(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0. \end{cases}$$

$\lceil x \rceil$ = Smallest integer $\geq x$.

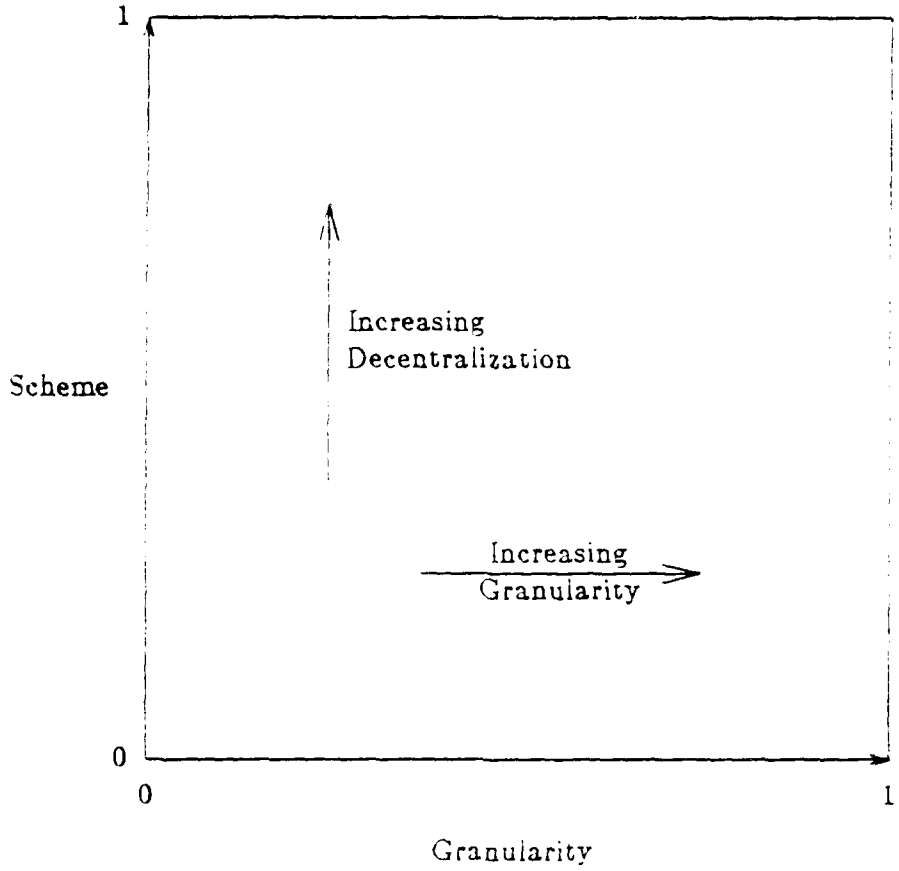


Figure 1
Ordering Scheme Graph

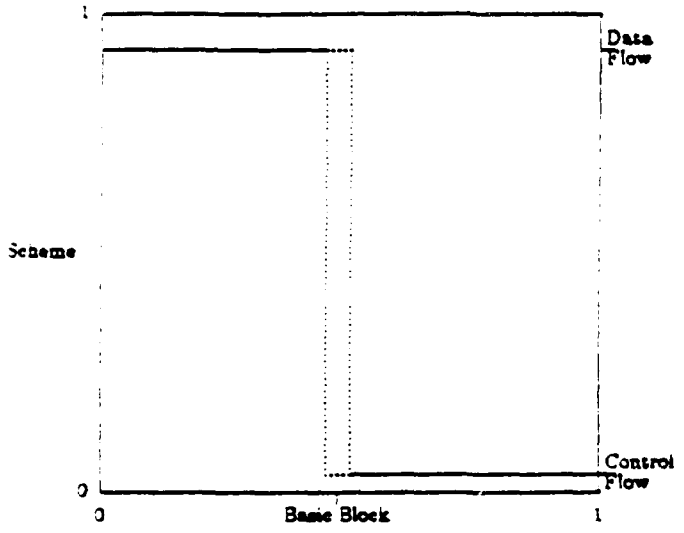


Figure 2a
Ordering Scheme Graph for the Piecewise Dataflow Architecture

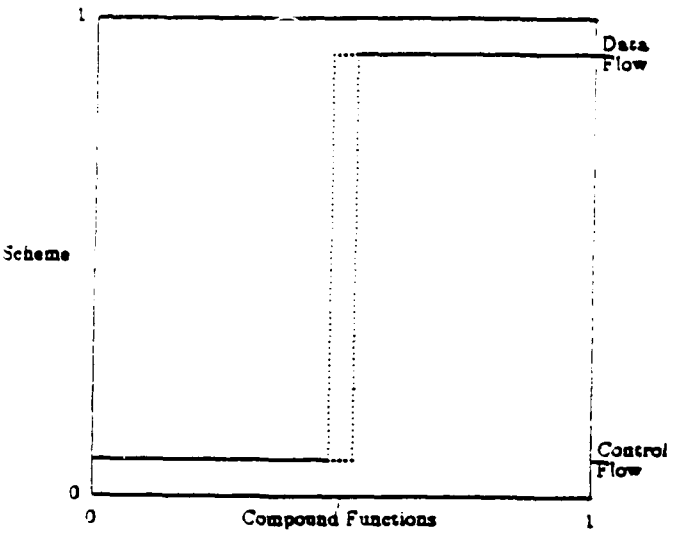


Figure 2b
Ordering Scheme Graph for the Cedar Architecture

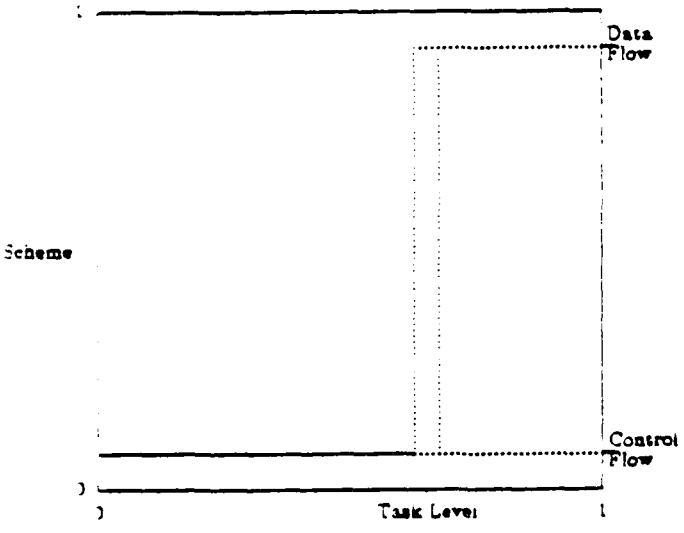


Figure 2c
Ordering Scheme Graph for the Rempe Architecture

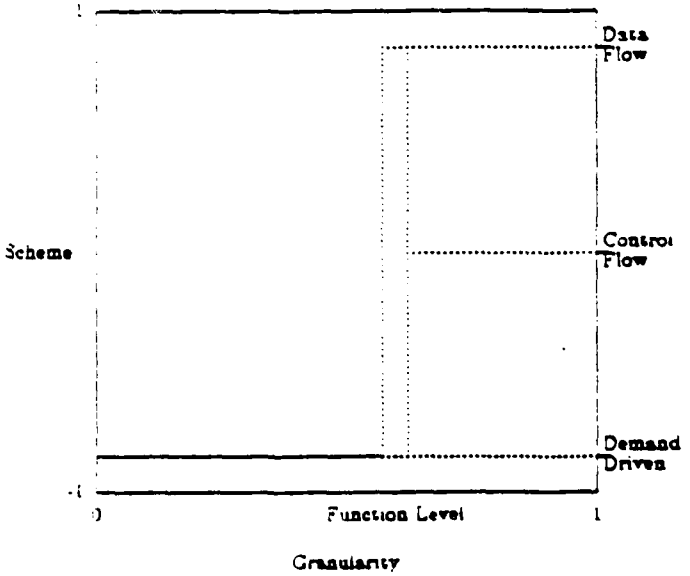


Figure 2d
Ordering Scheme Graph for the Rediflow Architecture

```
For i From 1 To SIZE Do
      For j From 1 To SIZE Do
            result[i] = result[i] - a[i,j] * b[j];
      EndDo
EndDo
```

Figure 3a
Matrix-Vector Multiply Algorithm

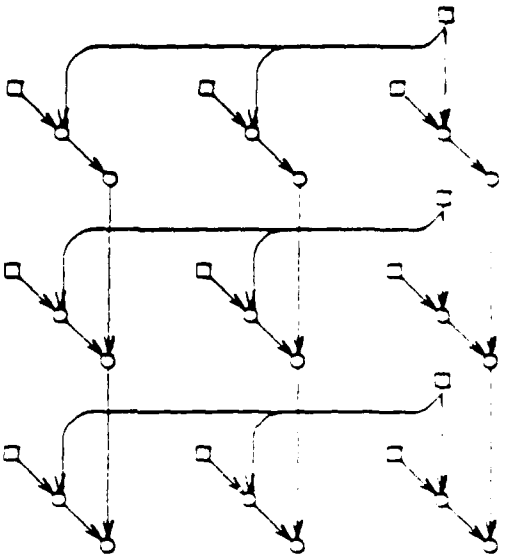


Figure 3b
Matrix-Vector Multiply Data Dependency Graph

```
For r From 1 To ITER Do
      For i From 1 To SIZE Do
            For j From 1 To SIZE Do
                  a[i,j] = (a[i-1,j] + a[i+1,j] + a[i,j-1] + a[i,j+1])/4
            EndDo
      EndDo
EndDo
```

Figure 4a
Iterative Relaxation Algorithm

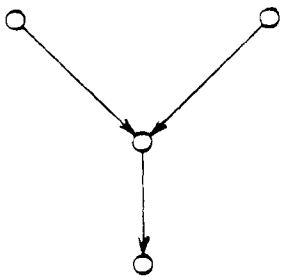
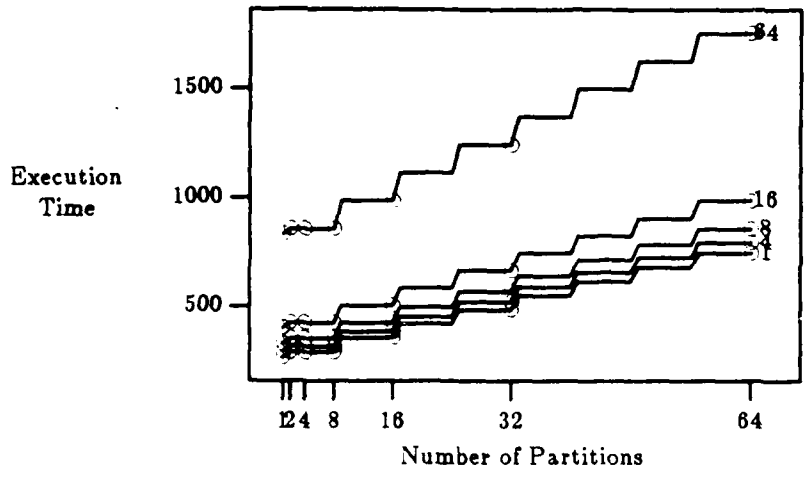
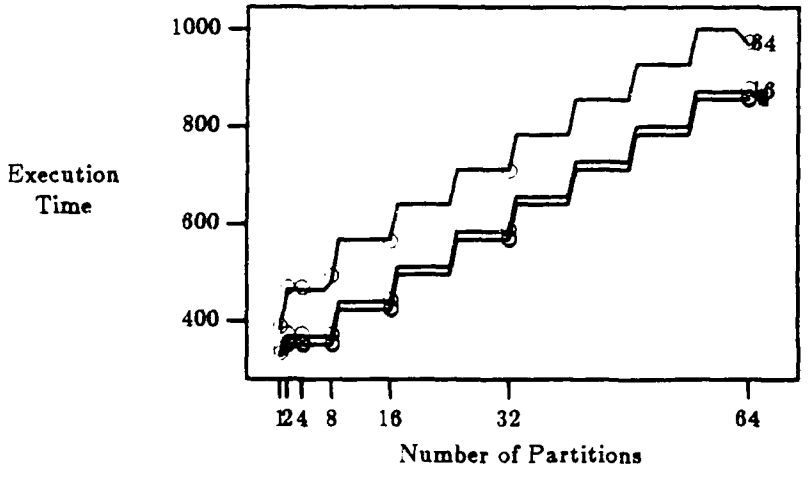


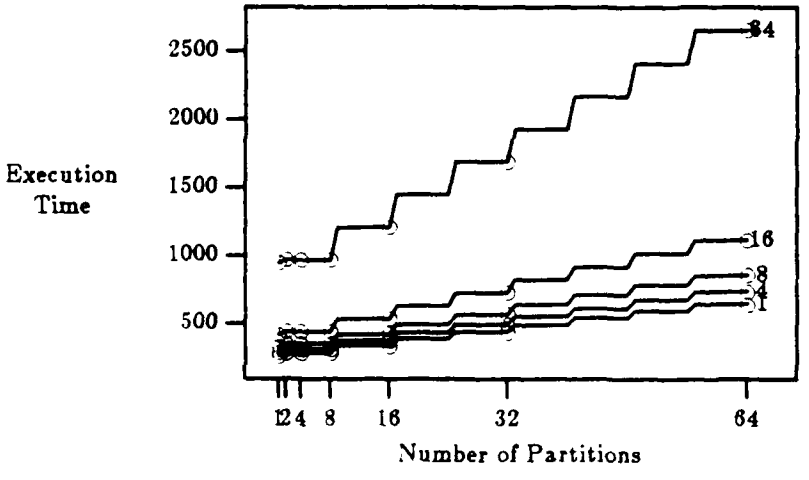
Figure 4b
Iterative Relaxation Data Dependency Graph Fragment



(a) Computation Time Experiment

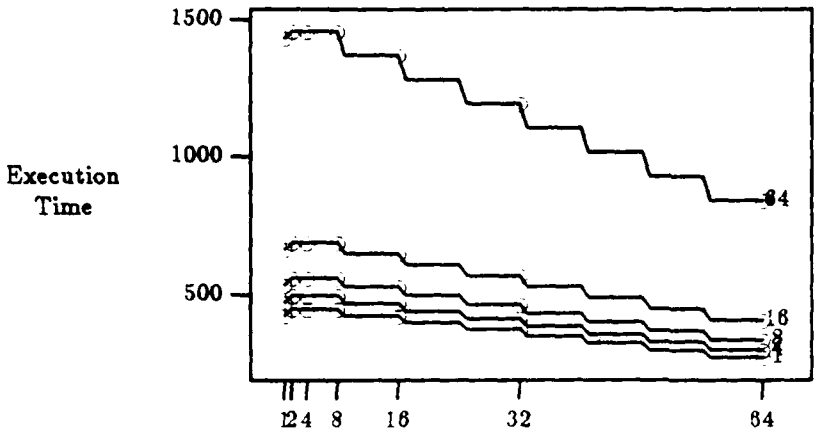


(a) Synchronisation Time Experiment

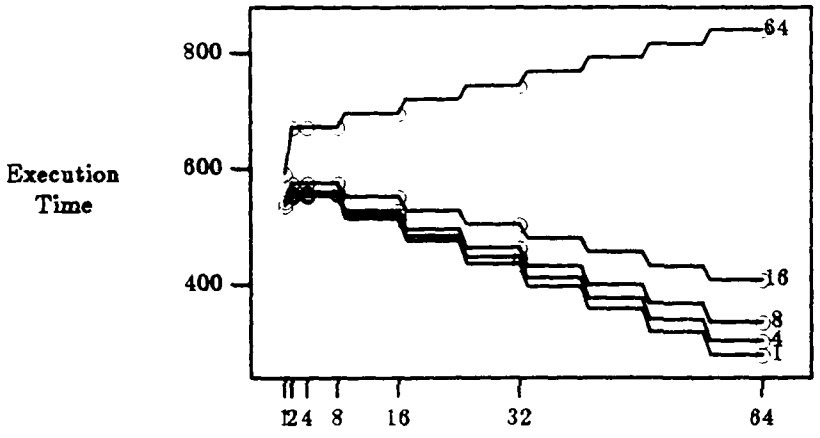


(a) Sequencing Time Experiment

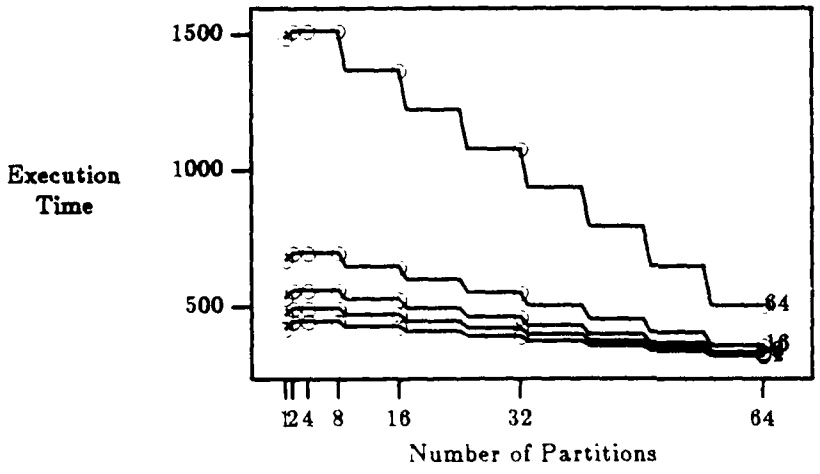
Figure 5
CPART Matrix Multiplication Critical Path Execution Time



(a) Computation Time Experiment

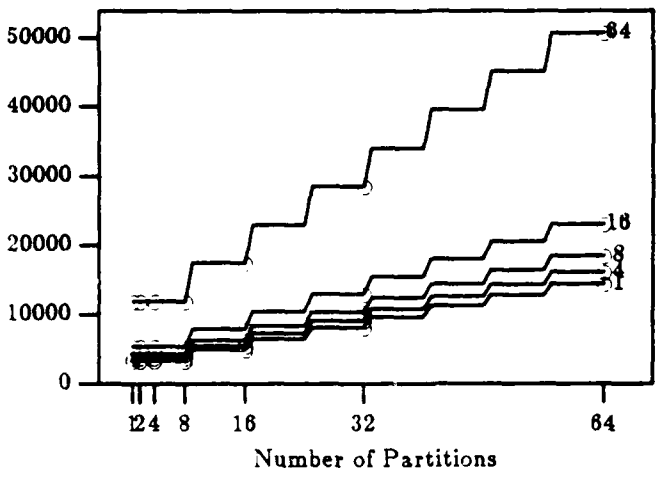


(a) Synchronisation Time Experiment



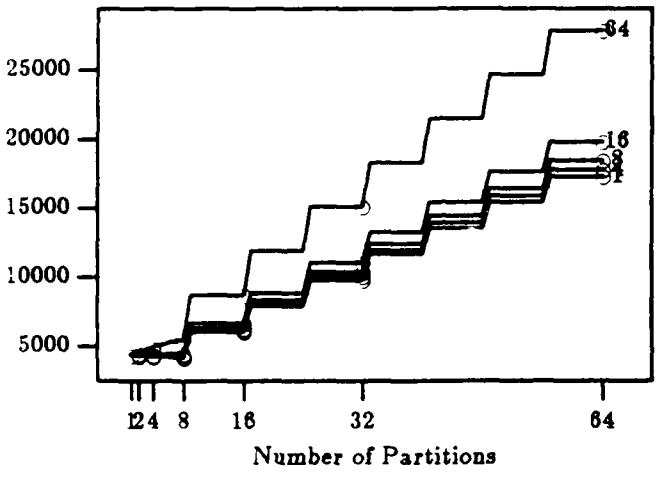
(a) Sequencing Time Experiment

Figure 6
DPART Matrix Multiplication Critical Path Execution Time



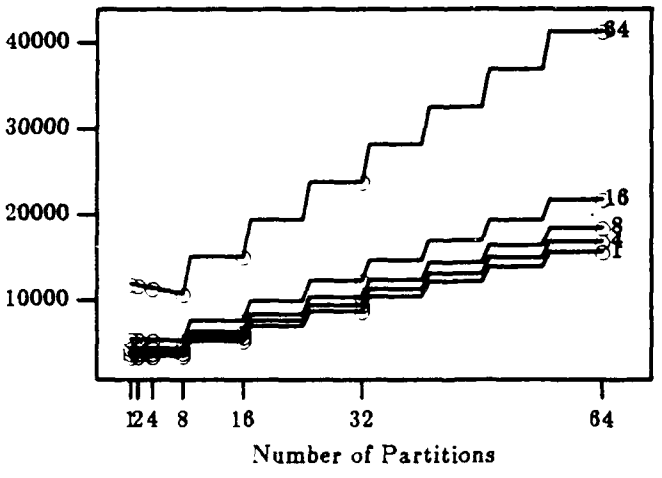
Execution Time

(a) Computation Time Experiment



Execution Time

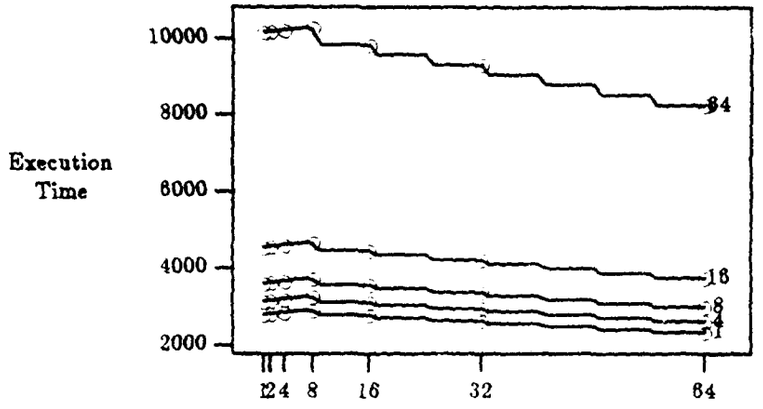
(a) Synchronisation Time Experiment



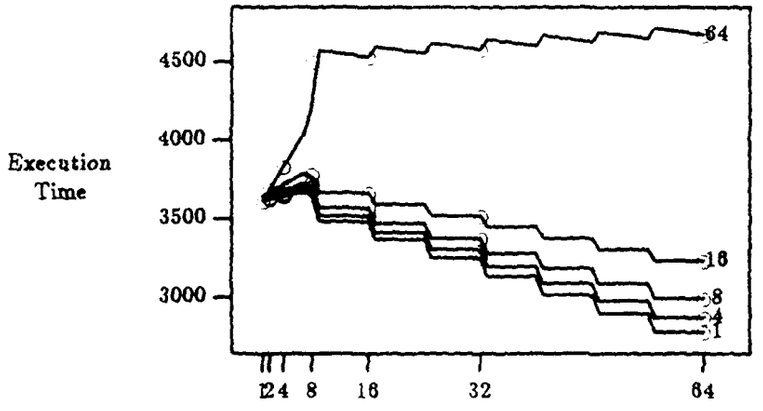
Execution Time

(a) Sequencing Time Experiment

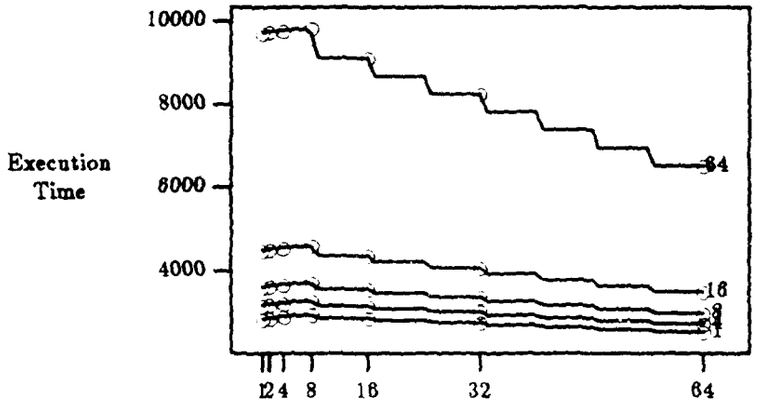
Figure 7
CPART Iterative Relaxation Critical Path Execution Time



(a) Computation Time Experiment



(a) Synchronization Time Experiment

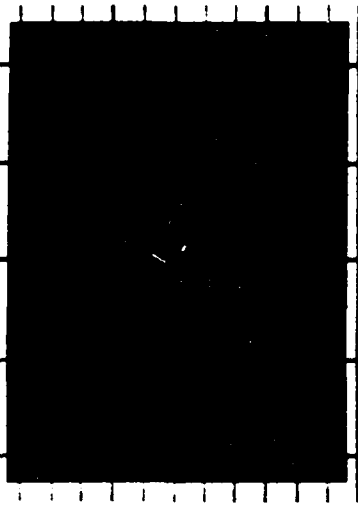


(a) Sequencing Time Experiment

Figure 8
DPART Iterative Relaxation Critical Path Execution Time

REFERENCE NO. 7

Fortes, J. A. B. and Wah, B. W., "Systolic Arrays-From Concept to Implementation (Guest Editor's Introduction)," Computer, pp. 12-17, July 1987.



Guest Editors' Introduction

Systolic Arrays—From Concept to Implementation

José A.B. Fortes
Purdue University

Benjamin W. Wah
University of Illinois at Urbana-Champaign

Systolic arrays have regular and modular structures that match the computational requirements of many algorithms. Their implementation requires that a wealth of subsumed concepts and engineering solutions be mastered and understood.

Systolic arrays are the result of advances in semiconductor technology and of applications that require extensive throughput. Their realization requires human ingenuity combined with techniques and tools for algorithm development, architecture design, and hardware implementation.

Invariably, the first reaction of people who are exposed to the systolic-array concept is one of admiration for the concept's elegance and for its potential for high performance. However, those who next attempt to implement a systolic array for a specific application soon realize that a wealth of subsumed concepts and engineering solutions must be mastered and understood. This special issue attempts to provide insights into the implementation process and to illustrate the different techniques and theories that contribute to the design of systolic arrays.

Characteristics of systolic arrays

Since 1978, when H.T. Kung and C.E. Leiserson[1] introduced the term "systolic array" and the concept behind the term, much research has been done and much has been written about the design of algorithms and architectures suitable for such structures. Today, the idea of a systolic array is as familiar to many computer scientists and engineers as that of a compiler or a microprocessor.

The term "array" originates in the systolic array's resemblance to a grid in which each point corresponds to a processor and a line corresponds to a link between processors. As regards this structure, systolic arrays are descendants of array-like architectures such as iterative arrays,[2] cellular automata,[3] and processor arrays.[4] These architectures capitalize on regular and modular structures that match the computational requirements of many algorithms. Table 1 is a list of applications for which systolic designs are available. Systolic arrays belong to the generation of VLSI/WSI (Very Large Scale Integration/Wafer Scale Integration) architectures for which regularity and modularity are important to area-efficient layouts.

Although the array structure characterizes the interconnections in systolic arrays, it is the term "systolic" that captures the innovative and distinctive behavior of

0018-9162/87/0700-0012$01.00 1987IEEE

these systems. "Systolic" in this context means that pipelined computations take place along all dimensions of the array and result in very high computational throughput. In other words, systolic algorithms schedule computations in such a way that a data item is not only used when it is input but also is *reused* as it moves through the pipelines in the array. This results in balancing the processing and input/output bandwidths, especially in compute-bound problems that have more computations to be performed than they have inputs and outputs. Conventional processor designs are often limited by the mismatch of input bandwidth and output bandwidth, which occurs because data items are read/written every time they are referenced.

One reason for choosing "systolic" as part of the term "systolic array" was to draw an analogy with the human circulatory system, in which the heart sends and receives a large amount of blood as a result of the frequent and rhythmic pumping of small amounts of that fluid through the arteries and veins. In this analogy, the heart corresponds to a source and destination of data, such as a global memory, and the network of veins is equivalent to the array of processors and links. Another explanation of the term is that in many of the first proposed systolic architectures, processing elements alternated between cycles of "admission" and "expulsion" of data—much in the same way that the heart behaves with respect to the pumping of blood.

In the article "Why Systolic Architectures?"[5] H.T. Kung presents an excellent introduction to the basic ideas, the advantages, and the open problems of systolic arrays. Today, this article is still essential reading for those interested in learning the fundamentals of systolic arrays. Our introduction endeavors neither to replace nor to repeat the contents of that pioneering article. However, it is appropriate to elaborate briefly on the three factors that characterize systolic arrays as they were originally proposed, namely *technology*, *parallel/pipelined processing*, and *applications*. These factors also identify the reasons for the success of the concept, namely cost-effectiveness, high performance, and the abundance of applications for which systolic arrays can be used.

Technology and cost-effectiveness. Nowadays, mature VLSI/WSI technology permits the manufacture of circuits whose layouts have minimum feature sizes of 1 to 3 micron. The effective yields of VLSI/WSI fabrication processes make possible the implementation of circuits with up to half a million transistors at reasonable cost—even for relatively small production quantities. However, the advantages of this technology are not fully realized unless simple, regular, and modular layouts are used. Systolic arrays attempt to meet these topological constraints by using simple processing elements that, together with a simple interconnection pattern, are replicated along one or more dimensions. Cost, regularity, and modularity are factors leading to the design and optimization of individual processing elements and their respective interconnections. Consideration of these three factors indicates that processor arrays are cost-effective engineering solutions to the problem of building systems with many processing elements.

The main difference between the design of systolic arrays and that of other integrated systems of comparable complexity is illustrated in a general way in Figure 1. The Y-chart shown in the figure is a convenient and succinct description of the different phases of the process of designing VLSI systems.[6,7] The axes of the Y-chart correspond to orthogonal forms of system representation, and the arrows represent design procedures that translate one representation into another. A top-down design procedure (that is, one that progresses from more complex components to simpler subcomponents) can also be indicated—by arrows drawn along each axis and pointed toward the origin. While many different design approaches and—their corresponding Y-charts—are possible, design is typically carried out through *successive refinements*. In this process, a component's functional specification is translated first into a structural representation and then into a geometrical description in terms of smaller subcomponents; the functional description of each of these subcomponents must then be translated into structural and geometrical descriptions in terms of even smaller parts, and so on. The line arrows shown in the figure are intended to convey, in a general way, the flow of this process for systolic arrays versus more conventional systems. Since a systolic array consists of a large number of a few types of modules, the process of refining the overall system and designing every subcomponent is faster and simpler than it is in systems with the same size but a much larger number of module types. This is conveyed graphically in Figure 1 by means of large arrows showing that in the design of a systolic array, one can proceed faster and more directly to the design of lower-level components of the system than in traditional design.

Commercially available systolic-array chips with 10 to 100 simple, 1-bit proces-

Table 1. Applications for which systolic designs are available.

Signal and Image Processing and Pattern Recognition
FIR, IIR filtering, and 1D convolution
2D convolution and correlation
Discrete Fourier Transform
Interpolation
1D and 2D median filtering
Geometric warping
Feature extraction
Order statistics
Minimum-distance classification
Covariance matrix computation
Template matching
Seismic signal classification
Cluster analysis
Syntactic pattern recognition
Radar signal processing
Curve detection
Dynamic scene analysis
Image resampling
Scene matching
Matrix Arithmetic
Matrix-matrix multiplication
Matrix triangularization
QR decomposition
Sparse-matrix operations
Solution of triangular linear systems
Non-Numeric Applications
Data structures—stacks and queues, sorting
Graph algorithms—transitive closure, minimum spanning trees
Connected components
Language recognition
Dynamic programming
Arithmetic arrays
Relational database operations
Algebra

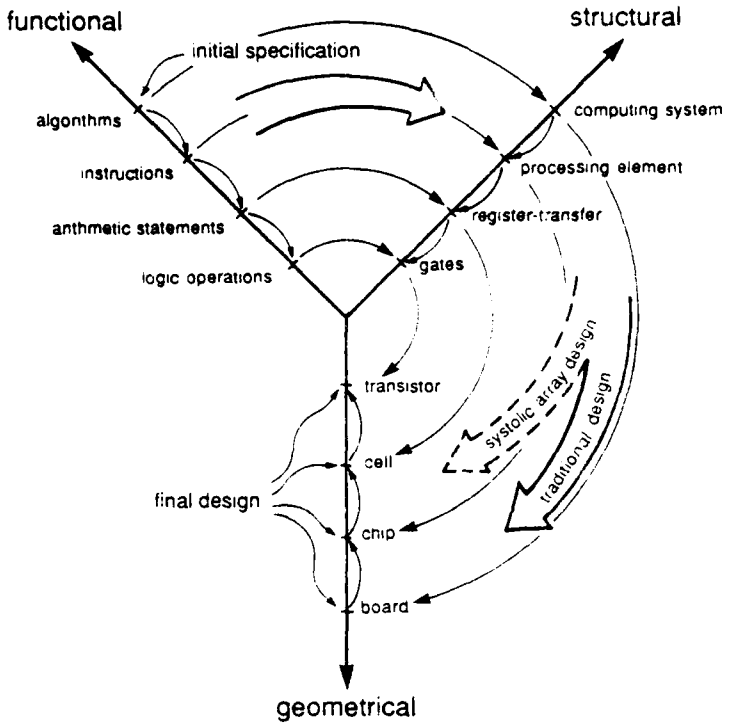


Figure 1. A Y-chart that shows the process of designing algorithmically specified VLSI digital systems.

sors exist; these chips sell for less than one hundred dollars apiece. Other chips, including microprocessors and digital-processing chips, both of which can be used as building blocks in systolic arrays, are also available—at even lower cost. Systolic arrays with thousands of processors can be built by assembling many such building blocks (chips) at total prices that range from ten thousand to a hundred thousand dollars and depend on the complexity of each processor.

Parallel/pipelined processing. Systolic arrays derive their computational efficiency from multiprocessing and pipelining. *Multiprocessing* is a natural consequence of the activities going on simultaneously in various processing elements of the array. *Pipelining* can be thought of as a form of multiprocessing that optimizes resource utilization and takes advantage of dependencies among computations. In systolic arrays, data pipelining reduces the input/output-bandwidth requirements by allowing a data item to be reused once it enters the array. Typically, inputs enter the array through peripheral processing elements and are propagated to neighboring processing elements for further processing. These movements of data through the array take place both along a fixed direction in which a link exists between neighboring processing elements and in a periodic manner.

In addition to data pipelining, systolic arrays are also characterized by *computational pipelining,* in which information flows from one processing element to another in a prespecified order. This information can be interpreted by the receiver as data, control, or a combination of both. Each output is computed by the execution—at different times and in a predetermined sequence—of several operations in a number of processing elements; the execution is performed in such a way that the output generated by one processing element is used as an input by a neighboring processing element. While operations can occur as data flows through each processor, the overall computation is not a dataflow computation, since the operations are executed according to a schedule determined by the systolic-array design. After a processing element generates an intermediate output and sends this output to the element's neighboring processing elements, the element computes another intermediate output. As a result, processing resources are utilized efficiently. In the general case, each processing element can be constructed as a pipelined processor. Such construction results in the so-called *two-level pipelined systolic array* and in even higher throughputs.

Applications and algorithms. Algorithms suitable for implementation in systolic arrays can be found in many applications, such as digital signal and image processing, linear algebra, pattern recognition, linear and dynamic programming, and graph problems. In fact, most of the algorithms in the listed applications are computationally intensive and require systolic architectures for their implementations when used in real-time environments. The acceptance of this fact is evidenced by the existence of prototype and production systolic arrays for modern real-time digital signal processing systems. The manufacturers of these arrays include, among others, companies such as ESL-TRW, Hughes, NCR, GE, Hazeltine, and Motorola. When systolic arrays were first proposed, they were intended for applications with two important sets of characteristics. First, these applications require high throughput and large processing bandwidth, possibly at the cost of increased response time. In other words, it is more important to keep up with the flow of data than to generate a set of outputs for a given set of inputs as quickly as possible. Second, these applications can be efficiently supported by algorithms that can be implemented on arrays consisting of a few types of simple processing elements; the arrays have simple controls and input/output ports in the peripheral processing elements. These algorithms are characterized by repeated computations of a few types of relatively simple operations that are common to many input data items. Often the algorithms can be described by programs with nested loops or by recurrence equations that describe computations performed on indexed data. In addition, the pattern of generation and usage of data by different operations displays some regularity and uniformity, which means that the resulting communication requirements can be met by the localized interconnections.

Implementation issues

Given the technical and economic principles that assure the soundness of the systolic-array concept, one needs to consider the issues involved in implementing a system for a specific application. Some of these issues are briefly discussed here.

General-purpose and special-purpose systolic systems. Typically, a systolic array can be thought of as an algorithmically specialized system in the sense that its design reflects the requirements of a specific algorithm. However, it may be desirable to design systolic arrays that are capable of efficiently executing more than one algorithm for one application or more. Two approaches are possible in designing these "large-purpose" systems, and a compromise between the two is often found in many actual implementations. One approach is based on adding hardware mechanisms so as to reconfigure the topology and interconnection pattern of the systolic array and to emulate the requirements of a specialized design. A concrete example of this approach is the Configurable Highly Parallel computer (CHiP),[8] which has a programmable lattice of switches for reconfiguration purposes. The other approach uses software to map different algorithms into a fixed-array architecture. As is the case with the approach behind other general-purpose parallel computers, this approach may require the use of programming languages capable of expressing parallel computations, as well as the development of translators, operating systems, and programming aids. These requirements apply, for example, in the case of Warp,[9] a systolic array developed at Carnegie Mellon University. For each algorithm, the designer needs to identify the efficient systolic designs and mappings and the appropriate techniques to use. The issue of appropriate techniques is of great importance, since the final performance, cost, and correctness of the design are governed by these techniques.

Design and mapping techniques. To synthesize a systolic array from the description of an algorithm, a designer needs a thorough understanding of and familiarity with the principles behind four things: systolic computing, the application, the algorithm, and the technology. Such skilled designers can provide excellent heuristic designs for important algorithms. However, the process is slow and error prone and may require extensive simulations, and the resulting designs are not guaranteed to be optimal or correct. Progress has been made in the development of systematic design techniques to automate this process.[10] These techniques are unlikely to replace the designers completely; instead, they will provide tools and formal concepts to assist designers in searching for diverse and desirable designs for a given application. Most of these techniques are concerned with the derivation of a relatively high-level specification of the array architecture from a description of the algorithm. Typically, such a specification includes the size and topology of the array, the operations performed by each processing element, the order and timing of data communication, and inputs and outputs. To a limited extent, these techniques can take into account technological factors and the relationship of the systolic array itself to the rest of the system. However, they are not complete; they can only be used at the specification level—and only in an indirect manner there. Until more is learned about design techniques that can be used conveniently for detailed integration of system and technology, such integration problems will continue to be left for the designer to solve.

Many specialized arrays can be seen as hardware implementations of a given algorithm.

Granularity. The basic operation performed in each cycle by each processing element in the various systolic arrays can range from a simple bit-wise operation, to word-level multiplication and addition, and even to execution of a complete program. The choice of granularity is determined by the application, or the technology, or both. For example, applications that use algorithms with basic bit-level operators and data structures naturally suggest that processing elements be of a corresponding complexity. The same choice of processing elements might, however, result from considerations such as input/output-pin restrictions and the technology that may be used. In programmable systolic arrays, the granularity may also be determined by trade-offs between the desired degree and level of programmability. The Saxpy Matrix-1[11] is an example of a programmable systolic computer with large granularity, whereas bit-level systolic arrays, like those discussed by J.V. McCanny and J.G. McWhirter,[6] are special-purpose designs with low granularity.

Extensibility. Many specialized systolic arrays can be regarded as hardware implementations of a given algorithm. This view holds when there is a direct correspondence between the operations and variables of the algorithm and, respectively, the processing elements and wire links of the systolic array. In such a case, the systolic processor can execute only a given algorithm that is designed for a problem of a specific size. If one wishes to execute the same algorithm for a problem of a larger size, then either a larger array must be built or the problem must be partitioned. The first approach is easy to conceptualize and simply requires that more processing elements be used to construct an enlarged version of the original array. However, as regards implementation, one must remember that there may be factors that do not affect performance in small arrays but might affect it in larger systems. These factors include clock synchronization, reliability, power requirements, chip-size limitations, and input/output-pin constraints.

Clock synchronization. In large synchronous systolic arrays, clock lines of different lengths can introduce clock skews and may require that a slower clock be used. Possible approaches that avoid this problem of clock skews include designing systolic arrays that do not allow data to flow in opposite directions and using efficient layouts of the clock distribution network.[12] An alternative to the design of a globally synchronous array is to achieve a self-timed system through the use of asynchronous handshaking mechanisms established between neighboring processing elements. These self-timed implementations are commonly referred to as *wavefront arrays*.[13]

Reliability. Simple laws of probability can be used to explain why increasingly large arrays are decreasingly reliable unless redundancy is incorporated and fault-tolerance mechanisms are available. In fact, the reliability of an array of processors is equal to that of a processor raised to a power of the number of processors in

the array. Since the reliability of a processor is a value less than one, the reliability of the global array quickly approaches zero as the number of processors increases. Fault tolerance requires that faults be detected and located so that faulty processing elements can be replaced by operational spares through an appropriate reconfiguration scheme. A fault-tolerant systolic array may need additional hardware to meet these requirements. In addition, if time redundancy is used or system operation needs to be suspended for testing purposes, the fault-tolerant array can be slower than the original one. A good fault-tolerant design has as its goal maximizing reliability while minimizing the corresponding overhead. In systolic arrays, possible approaches to fault tolerance include simple extensions of well-known techniques used in conventional digital systems. However, these techniques do not take advantage of the characteristics of either systolic arrays or the algorithms they execute. Novel and successful, though general, fault-tolerance schemes[14] that take advantage of these characteristics have been proposed for systolic arrays.

Partitioning of large problems. When it is necessary to execute a large problem without building a large systolic array, the problem must be partitioned so that the same algorithm can be used to solve the smaller problem and so that an array of small, fixed size can be used. The main concerns are to avoid rendering the partitioned algorithm incorrect and to avoid increasing the complexity of the design significantly. One approach identifies algorithm partitions and an order of execution of these partitions such that correctness is preserved and the original array can be used to execute each partition.[15] The perceived result of this approach is that the array "travels" through the set of computations of the algorithm in the right order until it "covers" all the computations. Another approach attempts to restate the problem to be solved so that the problem becomes a collection of smaller problems that is similar to the original one and that can be solved by the given systolic array.[16] While this second approach has less generality and is harder to automate than the first approach, it may have better performance when it is applicable.

Automated design tools. The processing elements and module libraries play an important role in making the process of designing special-purpose arrays of processing elements faster and more cost-effective. In addition to the many existing tools for designing VLSI and WSI systems that can be readily used in this process, the regularity and algorithmic nature of systolic arrays permits the use of high-level silicon compilers.[17] At this time, the development process is not fully automated; the process will depend on future progress in design automation and computer-aided design tools.

Universal building blocks. Systolic arrays cost less to implement than other arrays because of their extensive replication of a small number of simple, basic modules and because of their highly dense and efficient layouts. It is worthwhile for the simple building blocks to be carefully designed and optimized, since the costs involved are amortized over a large number of replicated circuits. The modular design of systolic arrays allows designers who want rapid prototyping of their ideas to use off-the-shelf devices, such as microprocessors, floating-point arithmetic units, and memory chips. However, these parts may not be designed for implementing systolic arrays and may therefore be inadequate to meet the design requirements. This has led to the development of "universal building blocks"—chips that can be used for many systolic arrays. The cost of such development is, therefore, amortized over replicated modules in many arrays rather than concentrated in simply one array. Commercially available chips that are worthy of consideration as basic modules include the INMOS Transputer, the TI TMS32010 and TMS32020, the NEC dataflow chip μPD 7281, Analog Devices' ADSP2100, the Fujitsu MB8764, and the National LM32900. Problems involved in the use of programmable building blocks include developing programming tools to aid designers and providing support for flexible interconnections.

Integrating systolic arrays into existing systems may be nontrivial because of I/O bandwidth.

Integration into existing systems. Although systolic arrays provide extensive throughput, their integration into existing systems may be nontrivial because of the extensive input/output bandwidth involved, especially when a problem has to be partitioned and input data have to be accessed repeatedly. Additional problems that have to be solved for systems with a large number of systolic arrays include the interconnections with the host, the memory subsystem to support the systolic arrays, the buffering and access of data to meet the special input/output data distributions, and the multiplexing and demultiplexing of data when there are insufficient input/output ports. The problems that must be faced are exemplified by Mosaic,[7] a project being carried out at ESL. The system consists of a statically scheduled crossbar switch that connects multiple Warp processors, each with local memory modules, into a macropipeline. The local memory modules are used to store input data and restructure them into the required input format.

The future

By the year 2000, it will be possible to build integrated circuits with one billion transistors—more than one thousand times the number of devices available in today's densest integrated circuits.[18] These incredibly large circuits will use 0.1—micron geometries made possible by advanced optical, electron-beam, ion-beam, or X-ray lithography. While the high cost of setting up integrated-circuit factories that can handle these technologies will certainly impact the initial cost per chip, the main manufacturing limitations will be in the design, verification, testing, and packaging of such large circuits. In addition, the percentage of the chip area dedicated to interconnections could increase to more than 80 percent. Systolic arrays will take advantage of submicron technologies without suffering from the problems just mentioned, since they are modular, have regular interconnections, and are extensible. By the year 2000, mature design and programming tools and extensive knowledge of suitable applications and algorithms will probably render systolic arrays the architecture of choice for submicron circuits designed for digital signal processing, fast arithmetic, symbolic processing, and intelligent databases.

Systolic arrays have triggered extensive related work and research in the areas of processor-array architecture, algorithm

design and analysis, and parallel programming. These areas are often identified as *systolic architecture, systolic algorithms,* and *systolic computing,* respectively. As a consequence, the principles behind systolic arrays have gained an enlarged scope. That is, systolic architectures are not necessarily arrays of processors; systolic algorithms may be very complex and may not necessarily be executed in simple processing elements; and systolic computing can take place in computers other than systolic architectures. The prominent features of systolic arrays are the processing elements, which implement processes, and the regular interconnection of multiple processing elements. The processing elements and the interconnection of processing elements can be implemented in software, general-purpose microprocessors, or specialized hardware. Because of this variety of implementation possibilities, systolic arrays have, since the late seventies, evolved to become cellular computing at the algorithmic, programming, architectural, and hardware levels. We are, therefore, witnessing a trend in which systolic computing is becoming a pervasive form of multiprocessing.□

Acknowledgments

We would like to thank the authors and reviewers for helping to make this special issue a reality. We are also grateful to Bruce Shriver, the editor-in-chief of *Computer,* for his guidance, directions, and help in preparing this special issue.

Despite the large number of articles in this special issue and despite our efforts to solicit manuscripts on major systolic-array projects for it, we were not able to cover all major projects because of page limitations and the tight schedule involved in preparing the issue. We realize that there are many researchers, too numerous to mention individually, who have made notable contributions to the development of systolic-array research. We apologize for any inadvertent omissions, and we would like to acknowledge their efforts here.

Last but not least, we would like to acknowledge the pioneering study on systolic arrays by H.T. Kung and C.E. Leiserson. Without their study, this special issue would not exist.

This project was supported by National Science Foundation Grants DCI 84-19745 and DCI 85-19649, as well as by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization under Office of Naval Research Grant 00014-85-k-0588.

References

1. H.T. Kung and C.E. Leiserson, "Systolic Arrays (for VLSI)," *Sparse Matrix Proc. 1978,* 1979, Academic Press, Orlando, Fla., pp. 256-282; also in "Algorithms for VLSI Processor Arrays," which is Section 8.3 of *Introduction to VLSI Systems,* C. Mead and L. Conway, eds., 1980, Addison-Wesley, Reading, Mass., pp. 271-292.
2. F.C. Hennie, *Iterative Arrays of Logical Circuits,* 1961, MIT Press, Cambridge, Mass.
3. J. von Neumann, "The General Logical Theory of Automata," in *Cerebral Mechanisms in Behavior—The Hixon Symposium,* L.A. Jeffries, ed., 1951, John Wiley & Sons, New York; a more recent work on cellular automata is *Modern Cellular Automata Theory and Applications* by K. Preston, Jr., and M.J.B. Duff, 1984, Plenum Press, New York.
4. D.L. Slotnick, W.C. Borck, and R.C. McReynolds, "The Solomon Computer," *Proc. AFIPS Fall Joint Computer Conf.,* 1962, Spartan Books, Washington, DC, pp. 97-107.
5. H.T. Kung, "Why Systolic Architectures?" *Computer,* Vol. 15, No. 1, Jan. 1982, pp. 37-46.
6. J.V. McCanny and J.G. McWhirter, "Some Systolic Array Developments in the United Kingdom," *Computer,* Vol. 20, No. 7, July 1987 (this issue).
7. *Computer* (special issue on new VLSI tools), Vol. 16, No. 12, Dec. 1983.
8. L. Snyder, "Introduction to the Configurable, Highly Parallel Computer," *Computer,* Vol. 15, No. 1, Jan. 1982, pp. 47-64.
9. M. Annaratone et al., "Warp Architecture and Implementation," *Proc. 13th Int'l Symp. Computer Architecture,* June 1986, Computer Society Press, Silver Spring, Md., pp. 346-356.
10. J.A.B. Fortes, K.S. Fu, and B.W. Wah, "Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays," *Proc. 1985 Int'l Conf. Acoustics, Speech, and Signal Processing,* 1985, IEEE, Piscataway, N.J., pp. 8.9.1-8.9.5.
11. D.E. Foulser and R. Schreiber, "The Saxpy Matrix-1: A General-Purpose Systolic Computer," *Computer,* Vol. 20, No. 7, July 1987 (this issue).
12. A.L. Fisher and H.T. Kung, "Synchronizing Large VLSI Processor Arrays," *IEEE Trans. Computers,* Vol. C-34, No. 8, Aug. 1985, pp. 734-740.
13. S.Y. Kung et al., "Wavefront Array Processors: From Concept to Implementation," *Computer,* Vol. 20, No. 7, July 1987 (this issue).
14. J.A. Abraham et al., "Fault Tolerance Techniques for Systolic Arrays," *Computer,* Vol. 20, No. 7, July 1987 (this issue).
15. D.I. Moldovan and J.A.B. Fortes, "Partitioning and Mapping Algorithms Into Fixed-Size Systolic Arrays," *IEEE Trans. Computers,* Vol. C-35, No. 1, Jan. 1986, pp. 1-12.
16. J.J. Navarro, J.M. Llaberia, and M. Valero, "Partitioning: An Essential Step in Mapping Algorithms Into Systolic Array Processors," *Computer,* Vol. 20, No. 7, July 1987 (this issue).
17. F.C. Lin et al., "MOSAIC: A Heterogeneous Architecture for Signal Processors," *Proc. 12th DARPA Strategic Systems Symp.,* Oct. 1986.
18. B.C. Cole, "Here Comes the Billion-Transistor IC," *Electronics,* Vol. 60, No. 7, Apr. 2, 1987, pp. 81-85.

José A.B. Fortes has been with the faculty of Purdue University's School of Electrical Engineering since 1984.

He is interested in all aspects of parallel processing, including the systematic design of algorithmically specialized processor-array architectures, parallel programming languages, automatic parallelism detection and exploitation techniques, and fault-tolerant computing.

Fortes has published over 20 technical papers in journals and conference proceedings in the areas of parallel processing, fault-tolerant computing, and VLSI architectures. He has worked on several projects in these areas in cooperation with or with funding from NSF, ONR, AT&T, RCA, and NCR.

Fortes is a member of IEEE and ACM.

He received his MSEE and PhD EE degrees from Colorado State University and the University of Southern California in 1981 and 1983, respectively.

Benjamin W. Wah is an associate professor in the Dept. of Electrical and Computer Engineering and in the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign.

He was on the faculty of the School of Electrical Engineering at Purdue University between 1979 and 1985.

His current research activities include parallel computer architectures, artificial intelligence, distributed databases, computer networks, and theory of algorithms.

Wah was a Computer Society Distinguished Visitor between 1983 and 1986.

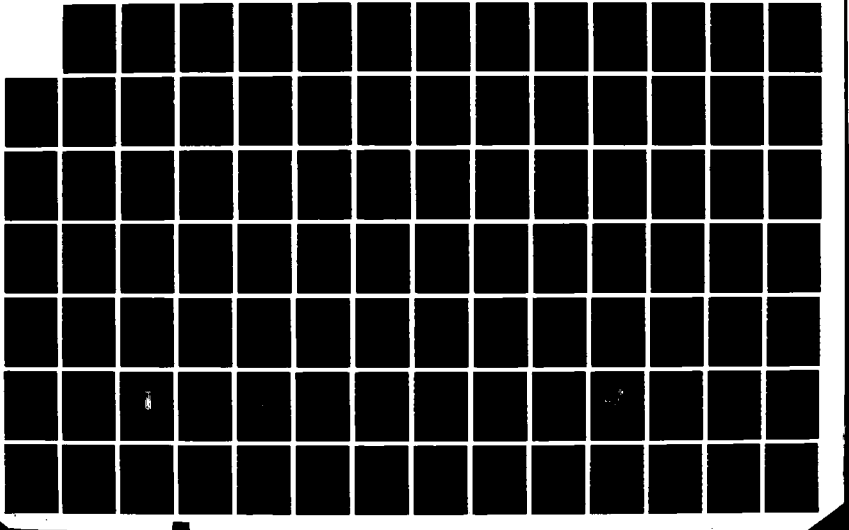
He is an editor of the *IEEE Transactions on Software Engineering,* and the *Journal of Parallel and Distributed Computing.*

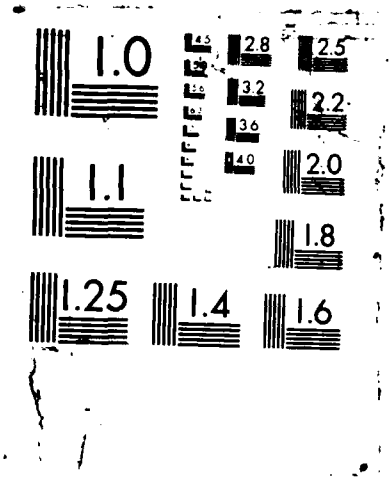
He received the PhD in computer science from the University of California at Berkeley in 1979.

Readers may write for information about this special issue to Jose Fortes, Purdue University School of Electrical Engineering, West Lafayette, IN 47907 or to Benjamin Wah, University of Illinois at Urbana-Champaign, Coordinated Science Laboratory, 1101 W. Springfield Ave., Urbana, IL 61801.

AD-A190 910 DESIGN AND EVALUATION OF FAULT-TOLERANT VLSI/WSI PROCESSOR ARRAYS(U) PURDUE UNIV LAFAYETTE IN J A FORTES 31 DEC 87 N00014-85-K-0500 2/3

UNCLASSIFIED F/G 12/6 NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

REFERENCE NO. 8

Rau, D., Fortes, J. A. B., Siegel, H. J., "Destination Tag Routing Schemes Based on a State Model for the IADM Network," Technical Report TR-EE 87-39, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907, October 1987. (Submitted to IEEE Transactions on Computers, October 1987).

Destination Tag Routing Techniques Based on a State Model for the IADM Network[1]

Darwen Rau and Jose A. B. Fortes
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

Howard Jay Siegel
Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

ABSTRACT

A "state model" is proposed for solving the problem of routing and rerouting messages in the Inverse Augmented Data Manipulator (IADM) network. Using this model, necessary and sufficient conditions for the reroutability of messages are established, and then destination tag schemes are derived. These schemes are simpler, more efficient and require less complex hardware than previously proposed routing schemes. Two destination tag schemes are proposed. For one of the schemes, rerouting is totally transparent to the sender of the message and any blocked link of a given type can be avoided. Compared with previous works that deal with the same type of blockage, the time×space complexity is reduced from $O(\log N)$ to $O(1)$. For the other scheme, rerouting is possible for any type of link blockage. A universal rerouting algorithm is constructed based on the second scheme, which finds a blockage-free path for any combination of multiple blockages if there exists such a path, and indicates absence of such a path if there exists none. In addition, the state model is used to derive constructively a lower bound on the number of subgraphs which are isomorphic to the Indirect Binary N-Cube network in the IADM network. This knowledge can be used to characterize properties of the IADM networks and for permutation routing in the IADM networks.

Index terms - cube network, data manipulator network, destination-tag routing, fault tolerance, interconnection network, multiprocessor, parallel processing, state model.

---

[1]This research was supported in part by the National Science Foundation under Grant DCI-8419745, by the Innovative Science and Technology Office of the Strategic Defense Initiative Organisation and was administered through the Office of Naval Research under contract No 00014-85-k-0588, and by the Supercomputing Research Center under contract MDA904 85 C 5027

1. Introduction

This paper discusses novel and efficient techniques for routing and rerouting messages in the Inverse Augmented Data Manipulator (IADM) network [9]. These results are based on a new approach, the "state model," which characterizes and correlates the topologies of the IADM and Indirect binary n-cube networks, and leads to efficient exploitation of the redundancy available in the IADM network.

Considerable research has been dedicated to the design of multistage interconnection networks for multiprocessor systems. The class of data manipulator networks, introduced in [3], includes, among others, the Augmented Data Manipulator (ADM) network [17], the IADM network [9] and the Gamma network [13][14]. The IADM network and the ADM network differ only in that the input side of one of them corresponds to the output side of the other and vice versa. The Gamma and the IADM networks are topologically equivalent; however, they use switches of different types. Each 3×3 crossbar switch used in the Gamma network can connect simultaneously all three inputs to all three outputs whereas each switch used in the IADM network can connect only one of its three inputs to one or more of its three outputs. The main interest of this paper is the study of the IADM network; both the one-to-one and permutation routings are considered. The schemes proposed for routing and rerouting messages in the IADM network are also applicable to the Gamma network.

Perhaps the most popular class of multistage networks is the multistage *cube-type* networks such as the Indirect Binary N-Cube [15], Omega [6], Baseline [20], Generalized Cube [18], STARAN flip [2] and a special case of SW-Banyan [4] networks. Among the main advantages of these networks are their very efficient destination tag routing schemes, partitionability, $O(N\log_2 N)$ cost and

ability to pass useful permutations [16]. Some results of this paper are based on characteristics of the Indirect Binary N-Cube network (hereon referred to as the *ICube network*). Since the cube-type networks mentioned above are all topologically equivalent [16][17][20][21], the results in this paper are also relevant to any of them.

The ICube network is composed of $n = \log N$ stages labeled from 0 to $n-1$. Each stage consists of $2N$ connection links and N interchange (switches) boxes. The structure of the network is such that two input links of an interchange box differ only in the i-*th* bit of their labels; the upper links have a "0" in the i-*th* bit and the lower links have a "1." Figure 1 illustrates an ICube network of size N=8 and two possible states of an interchange box, "straight" and "exchange." Since this paper considers only one-to-one and permutation routing, broadcast states are not shown.

The *IADM* network is composed of n stages labeled from 0 to $n-1$. Each stage consists of $3N$ connection links and N switching elements. An extra column of switches is appended at the end of the last stage as the output switches and is referred to as stage n. Each switch j at stage i has three output links to switches $(j-2^i)$ *mod* N, j and $(j+2^i)$ *mod* N of the succeeding stage. Each switch selects one of its input links and connects it to one or more output links. Figure 2 illustrates an IADM network of size N=8.

In a multistage interconnection network, the path connecting the source of a message to its destination is determined by a routing scheme that specifies the switching state of each switch in the path. Routing schemes are considerably simpler for the cube-type networks than for the data manipulator-type networks. In cube-type networks, the interchange box at stage i needs to examine the i-*th* bit of the binary representation of the destination address of an

incoming message. If the *i-th* bit is 0, then the upper output of the box is taken. If the *i-th* bit is 1, the lower output of the box is taken. These schemes are known as *destination tag routing schemes* [6] and are extremely efficient and simple to implement. Unlike cube-type networks, in the IADM and other data manipulator-type networks there are several paths between any source s and destination d $(s \neq d)$ and each switching element has at least three switching states. Previously proposed routing schemes [9][10][13] for the IADM network can be thought of as *distance tag schemes*; that is, they require calculation of the distance from source to destination in order to generate routing and rerouting tags. The rerouting schemes in these works are basically finding an alternate representation, which specifies an alternate routing path, for the distance.

McMillen and Siegel [9] proposed three dynamic rerouting techniques for the IADM network for avoiding faulty or blocked $\pm 2^i$ (nonstraight) links. The first and the second schemes require that switches be capable of performing two's complement and $+2^i$ addition operations, respectively. The third scheme requires one extra tag bit which is dynamically updated as the message propagates toward the destination. In [10], the work of [9] was expanded, and a single-stage look-ahead scheme was proposed to avoid certain type of straight link faults. This improved scheme also requires two's complement operations.

Parker and Raghavendra [13] used redundant number representation and proposed an algorithm capable of finding all routing paths, which, effectively, are the redundant number representations for the distance between the source and the destination. Because of the complexity of the algorithm, the cost of computation is prohibitively large so that it is infeasible to implement the algorithm in order to achieve dynamic routing [19]. In addition, although the algorithm can generate all routing tags for any distance, there is no specific work on

rerouting schemes in [13][14].

Lee and Lee [7] proposed signed bit difference tag and destination tag local control algorithms for the ADM and IADM networks that require no computation for the distance between the source and the destination. But their local control algorithms can only find one routing path for each source and destination pair. If the need for rerouting arises, they still resort to the distance tag schemes to find alternate paths.

Past research has shown interesting relationships between data manipulator and cube-type networks. For example, because it is possible to embed the Generalized Cube network in the ADM network [1][17], the set of interconnections implementable by the ADM network is a superset of that of the Generalized Cube network. This fact and the existence of multiple paths between any source s and destination d $(s \neq d)$ in the ADM network suggests that the ADM network can be thought of as a fault-tolerant Generalized Cube network. Analogously, the IADM network can be regarded as a fault-tolerant ICube network[2]. Since the permutations realizable by cube-type networks are well studied, the identification of possible embeddings of the ICube network in the IADM network can help characterize the permutation capabilities of this network. A contribution to the precise understanding of these notions is made in this paper; it consists of the identification of a large number of distinct subgraphs of the IADM network that are isomorphic to the ICube network.

Section 2 of this paper introduces a state model to describe and correlate topologies of the ICube network and the IADM network. Necessary and

[2] While topologically equivalent, the ICube and Generalized Cube I/O ports are addressed so that their inter-relationship is the same as that of the IADM and ADM network, i.e. the input and output sides are interchanged.

sufficient conditions to perform rerouting in the IADM network are derived in Section 3 . In Section 4 two routing and rerouting schemes are proposed based on the theory developed in Section 3, together with a discussion of their merits and implementation considerations. A universal rerouting algorithm is proposed in Section 4, which can deal with any combination of multiple link blockages. A class of subgraphs in the IADM network that are isomorphic to the ICube network are identified in Section 6, and it is shown how to reconfigure the IADM network under certain link faults to pass the cube-admissible permutations. Finally, Section 7 summarizes the results presented in this paper.

2. State Model Descriptions for the ICube and IADM Networks

Multistage networks can be modeled as graphs by treating interchange boxes (also called switching elements) and links of the network as nodes and edges of the graph, respectively. Another equivalent graph model [1][8] results if interchange boxes are associated with edges, and links with nodes. Both models are exemplified in Figures 1 and 3 for the ICube network. The IADM network is shown in Figure 2 according to the first model. The design of switches based on both models is discussed in [11]. Clearly, the ICube network in Figure 3 can be regarded as being a subgraph of the IADM network in Figure 2. Henceforth, the second model is always assumed when referring to the ICube network (i.e. Figure 2) and the first model is assumed when dealing with the IADM network.

With respect to these graph models, the nodes and the edges of the graph refer to the switches and the links of the networks, respectively. The number of switches at each stage of a network is denoted N and $n = \log_2 N$ refers to the number of stages. The switches of each stage are labeled from 0 to $N-1$ from the top to the bottom. Any integer j has a binary representation

$j_0 j_1 \cdots j_{n-1}$, where j_{n-1} is the most significant bit and n denotes the number of bits. The notation $j_{p/q}$ means the bits of j starting at j_p and ending at j_q, where $p \leq q$. Bit $\bar{j}_i$ is 1's complement of bit j_i. Throughout this paper, j and $j+a$, where a is some constant, are reserved to represent labels of switches. Also modulo N arithmetic is assumed, e.g. $j+a$ implies $(j+a)\ mod\ N$. The notation $j \in S_i$ is used to indicate that a switch j belongs to stage i and $(j' \in S_i\ ,\ j'' \in S_{i+1})$ is used to represent a link at stage i joining $j' \in S_i$ and $j'' \in S_{i+1}$. A sequence of switches of contiguous stages $(j' \in S_i\ ,\ j'' \in S_{i+1}\ ,\ \cdots\ ,\ j''' \in S_{i+k})$ is used to represent a path from $j' \in S_i$ to $j''' \in S_{i+k}$.

Notation and terminology required for the characterization of network topologies and destination tag routing schemes are introduced next. A switch j of stage i is an $even_i$ switch if $j_i = 0$ and an odd_i switch if $j_i = 1$. Figure 2 identifies $even_i$ and odd_i switches at different stages of the IADM network of size N=8. Define the functions ΔC_i and $\Delta \bar{C}_i$ that represent connection links at stage i as

$$\Delta C_i(j,t_i) = \begin{cases} 0 & \text{if } j \text{ is an } even_i \text{ switch and } t_i=0, \\ & \text{or if } j \text{ is an } odd_i \text{ switch and } t_i=1 \\ -2^i & \text{if } j \text{ is an } odd_i \text{ switch and } t_i=0 \\ +2^i & \text{if } j \text{ is an } even_i \text{ switch and } t_i=1 \end{cases}$$

$$\Delta \bar{C}_i(j,t_i) = -\Delta C_i(j,t_i)$$

Also, define the functions $C_i(j,t_i) = j + \Delta C_i(j,t_i)$ and $\bar{C}_i(j,t_i) = j + \Delta \bar{C}_i(j,t_i)$. These definitions imply the following lemma of fundamental importance to the results of this paper.

Lemma 2.1

$$C_i(j,t_i) = j_{0/i-1}\, t_i\, j_{i+1/n-1}$$

$$\overline{C}_i(j,t_i) = j_{0/i-1}\, t_i\, q_{i+1/n-1}$$

for some value of $q_{i+1/n-1}$ which depends on j and t_i.

Proof: If j is an $even_i$ switch and $t_i = 0$, then $C_i(j,t_i) = \overline{C}_i(j,t_i) = j$. If j is an odd_i switch and $t_i = 1$, then $C_i(j,t_i) = \overline{C}_i(j,t_i) = j$. If j is an odd_i switch and $t_i = 0$, then $C_i(j,t_i)$ results from subtracting 1 from j_i. Since j is an odd_i switch, $j_i = 1$, no borrow is generated and all remaining bits of j are unchanged; however, $\overline{C}_i(j,t_i)$ adds 1 to j_i, changing the i-th bit to 0 and altering some of the bits in positions $i+1, \ldots, n-1$ due to carry propagation. Similar reasoning applies when j is an $even_i$ switch and $t_i = 1$. □

The notation and terminology just introduced can now be used to describe the networks of interest in this paper. The following description for a network in terms of ΔC_i, $\Delta \overline{C}_i$, C_i and $\overline{C}_i$ is called the *network state model.*

The ICube network is composed of n stages labeled from 0 to $n-1$. Each stage consists of $2N$ links and N switches. An extra column of switches is appended at the end of the last stage as the output switches (Figure 3) and is denoted S_n. A switch $j \in S_i$ is connected to switches $C_i(j,t_i) \in S_{i+1}$, for $0 \le i \le n-1$, $0 \le j \le N-1$, and $t_i = 0$ or $t_i = 1$. When using destination tags, switch $j_i \in S_i$ routes a message to switch $C_i(j,d_i) \in S_{i+1}$ where d_i is the i-th bit of the address of the message destination.

The IADM network is composed of n stages labeled from 0 to $n-1$. Each stage consists of a column of N switches and $3N$ connection links. An extra column of switches is appended at the end of the last stage as the output switches and is denoted S_n. A switch $j \in S_i$ is connected to switches $C_i(j,t_i) \in S_{i+1}$ and $\overline{C}_i(j,t_i) \in S_{i+1}$ for $0 \le i \le n-1$, $0 \le j \le N-1$, and $t_i = 0$ or $t_i = 1$. In other words, three links connect a switch $j \in S_i$ to the switches $(j-2^i)$,

j and $(j+2^i)$ at stage $i+1$. Sometimes $+2^i$ and -2^i are used to represent links $(j \in S_i\ ,\ (j+2^i) \in S_{i+1})$ and $(j \in S_i\ ,\ (j-2^i) \in S_{i+1})$, respectively. The terms *a straight link* refers to link $(j \in S_i\ ,\ j \in S_{i+1})$ and *a nonstraight link* refers to links $\pm 2^i$.

According to the model, two types of switches, $even_i$ and odd_i, are required in the IADM and ICube networks. Figure 4 illustrates the connection links of a pair of $even_i$ and odd_i switches for an ICube and an IADM network of size N=8. The ΔC_i function describes the ICube connections. For the IADM network, the connection links can be described by the union of the functions ΔC_i and $\Delta \overline{C}_i$. In practice, $even_i$ and odd_i switches can be identical and easily programmed (at power-up or system configuration time) to behave differently.

There are two possible routing behaviors (or states) for each switch in an IADM network. A switch is said to be in *state C* if the routing is decided in accordance with the function $C_i(j,t_i)$ and it is in the *state* $\overline{C}$ if the function $\overline{C}_i(j,t_i)$ applies. On the whole, the link on which a message is routed depends on whether the switch is an $even_i$ or odd_i switch, in state C or $\overline{C}$, and the value of tag bit t_i. Also the term *state of the network* is used to denote collectively the states of all switches in the network.

The notion of switch state is only conceptual; it can be implemented by designing the switches with actual logic states as well as by using tags with n added bits specifying the states of the switches on the routing path. In Section 4, these and other aspects of the actual implementation of the proposed schemes are discussed in detail.

3. Theory behind the State-Based Destination Tag Routing Schemes

Based on the framework developed in Section 2, routing problems in the IADM network are now examined. It is clear that when every switch in the IADM network is in state C, the IADM network behaves like an ICube network and, therefore, the destination address $d_{0/n-1}$ can be used as a routing tag, i.e. $t_i = d_i$. More generally, the following theorem can be proven.

Theorem 3.1 Let $d = d_{0/n-1}$ be the destination in the IADM network to which a message is to be sent. Then $t = d_{0/n-1}$ is the unique destination routing tag to the destination d regardless of state of the IADM network.

Proof: Consider an arbitrary tag $f_{0/n-1}$ and assume that the IADM network is in an arbitrary state. Let $t_{0/n-1} = f_{0/n-1}$. Then each switch will route the incoming message to either $C_i(j,f_i)$ or $\overline{C}_i(j,f_i)$. From Lemma 2.1, it can be reasoned by induction that, at stage i, $(C_i(j,f_i))_{0/i} = (\overline{C}_i(j,f_i))_{0/i} = f_{0/i}$; at the last stage, $C_{n-1}(j,f_{n-1}) = \overline{C}_{n-1}(j,f_{n-1}) = f_{0/n-1}$. Thus the address of the destination of the message is the same as the routing tag. This proves both the validity and the uniqueness of $d_{0/n-1}$ as a routing tag. □

It is implicit in the reasoning underlying Theorem 3.1 that any link on a given path results from the appropriate choice of the state of the corresponding switch, i.e. the use of "link" $\Delta C_i(j,t_i)$ results from setting $j \in S_i$ to state C and the use of "link" $\Delta\overline{C}_i(j,t_i)$ results from setting $j \in S_i$ to state $\overline{C}$. Thus, given a path to the destination d, there is at least one network state for which the use of d as the destination tag results in the routing of a message through that path.

The implication of Theorem 3.1 is that the use of a state model for the IADM network reduces the problem of finding alternate routing paths to that of

controlling the states of the switches in the network. Capitalizing on this idea, the following theorems show how alternate routing paths can be found in order to evade blockages in the network. A *straight link blockage* occurs if a straight link on the routing path is faulty or busy. A *nonstraight link blockage* is defined analogously. The third type of blockage, called *double nonstraight link blockage*, occurs if both nonstraight output links of a switch in the routing path are faulty or busy. A *switch blockage* occurs if the switch itself is busy or faulty. A switch blockage has the same effect as blocking all of the switch's input links and can be transformed into a link blockages problem accordingly. The discussion on rerouting in this paper is concerned only with link blockages.

Theorem 3.2 In the IADM network, a change of the state of switch $j \in S_i$ results in a different routing path to a destination d if and only if a nonstraight output link of j is used on the original routing path to d. Moreover, the other nonstraight output link of j is used on the new path.

Proof: Changing the state of j implies that the "link" $\Delta C_i(j,t_i)$ is used instead of $\Delta \bar{C}_i(j,t_i)$ or vice versa. However, if $\Delta C_i(j,t_i) = 0$ then $\Delta \bar{C}_i(j,t_i) = 0$ (i.e. both use a straight link) and vice versa. □

With regard to the rerouting schemes proposed in this paper, the implications of Theorem 3.2 are twofold. First, the "if" part of the theorem implies that dynamic rerouting for a nonstraight link blockage can be achieved by changing the state of the switch whose output is the nonstraight link, which is equivalent to rerouting the message through the oppositely signed nonstraight link connected to the same switch. Thus, the same subset of destinations is reachable from the two switches whose input links are the two oppositely signed nonstraight links. Second, the "only if" part of the theorem implies that dynamic rerouting for a straight link blockage is impossible. This is true in

general since every routing path in the IADM network can be the result of setting the network to some state. Moreover, if a path from stage i' to stage i'' consists of all straight links connecting $j \in S_i$ and $j \in S_{i+1}$, $i' \leq i < i''$, then there exist no alternate routing paths from $j \in S_{i'}$ to $j \in S_{i''}$ for otherwise there would exist an alternate routing path branching from $j \in S_{i'}$ and ending at the destination. The only resort, if any at all, to bypass the straight link blockage is to backtrack to a switch connected to a nonstraight link on the routing path at some preceding stage and to reroute from that switch. It remains to show that an alternate routing path always exists, provided that such a nonstraight link exists. In fact, the existence of an alternate routing path partly results from Theorem 3.2, as stated in the next theorem. Figure 5 illustrates the situation in Theorem 3.3.

Theorem 3.3 Consider a routing path in the IADM network to a destination d that contains a blocked straight link at stage i. There exists at least one network state which results in an alternate routing path that avoids the same straight link blockage at stage i if and only if the original routing path to d contains a nonstraight link at stage $i-k$ for some k, $i \geq k > 0$.

Proof: See Appendix A1. □

Previous work [7][9][13] implies only the "if" part of the theorem, i.e. the possibility of using nonstraight link of opposite sign in order to reroute a message in the case of a nonstraight link failure. However, the "only if" part of the theorem also implies that, in addition, it is not possible to devise a new rerouting scheme capable of avoiding a backtracking (or look-ahead) mechanism in order to deal with straight link blockages.

From Theorem 3.2, (for a given source/destination pair) if the straight output link of a switch is on some routing path, both nonstraight output links of

the switch cannot be used for routing; if one of the nonstraight output links of a switch is on some routing path, the other nonstraight link of the switch is also on another routing path and the straight link of the switch cannot be used for routing. So for a given switch, the output link blockages that affect paths from a given source to a given destination can only be (a) a nonstraight link blockage, (b) a straight link blockage or (c) the double nonstraight link blockage.[3] Theorem 3.2 can be used to avoid case (a) a nonstraight link blockage and Theorem 3.3, case (b), a straight link blockage. If case (c) occurs, then Theorem 3.2 cannot be used to find a rerouting path. A backtracking scheme proposed later in Corollary 4.2 based on Theorem 3.3 can be adapted to overcome this type of blockage. The adapted backtracking scheme is based on Theorem 3.4, which is illustrated in Figure 6.

Theorem 3.4 Consider a routing path in the IADM network to a destination d that contains a switch at stage i whose both nonstraight output links are blocked. There exists at least one network state which results in an alternate routing path that avoids the same blocked nonstraight links at stage i if and only if the original routing path to d contains a nonstraight link at stage $i-k$ for some k, $i \geq k > 0$.

Proof: See Appendix A1. □

[3]Physically it is possible to have any combination of blockages of the output links of a given switch. However, the possible routing paths for a given source/destination pair can be affected by either a straight link blockage or a double nonstraight link blockage in a given switch but never both types of blockage.

4. State-Based Routing and Rerouting Schemes

In this section, routing and rerouting schemes are discussed based on the theory developed in Section 3. As mentioned earlier, the novelty of the ideas in this paper lies in the state model of the routing behavior of each switch. In previously proposed approaches, routing is determined solely by tag bits. According to the state model, the switching action of each network element is conceptually determined by its relative position (i.e. an $even_i$ or odd_i switch), its state (i.e. C or $\overline{C}$) and a destination tag bit (i.e. 0 or 1) (Figure 4). This conceptual separation of routing information makes it possible to devise the simple routing schemes described in this section.

In the first scheme, each switch is initially set up to behave as an odd_i or $even_i$ switch. In addition, each switch can dynamically be set to one of the logical states C or $\overline{C}$. In other words, this scheme corresponds to a direct implementation of the conceptual view of switch states. Destination tags are used and, according to Theorem 3.1, the state of the network is transparent to the sender of the message since it only affects the path of the message and not its destination. Consequently, rerouting is also transparent in the sense that it results from a change in the network state. In practice, the implementation can be such that, for instance, state C (or $\overline{C}$) is used as the default state for each switch in the IADM network and the switch regards the other nonstraight link as a spare link for rerouting; if a nonstraight blockage is detected, then the switch changes state to $\overline{C}$ (or C) so that the spare link is used instead. This scheme is called the *Self-Repairing State-Based Destination Tag (SSDT)* scheme.

Rerouting is useful not only when one nonstraight link in a switch is faulty or busy, but also if both nonstraight links are busy. For example, when considering a packet switching environment, rerouting may be desirable as a means of

balancing the message load throughout the network. The scheme proposed here is well suited for this purpose. Assume that each nonstraight link has an associated buffer (queue). When both nonstraight links are busy due to message traffic congestion, a switch can choose which nonstraight buffer to assign a message to (i.e. which state to associate with that queued message), based on the number of messages present in the buffers in order to evenly distribute the message load to the nonstraight links.

The proposed SSDT scheme has the advantages that it uses simple n-bit destination tags and is capable of rerouting messages when blockages occur in nonstraight links. In addition, rerouting of a message is transparent to its sender since the path of the message is determined by the state of the network. For a given destination tag, the routing behavior of each switch on a possible path is determined by the state of the switch, i.e. the SSDT scheme is fully distributed and rerouting is done dynamically. Each switch requires a negligible amount of extra hardware for the detection of blocked links and the representation of two possible states.

The second scheme is called the *Two-Bit State-Based Destination Tag* (*TSDT*) scheme and it uses $2n$-bit routing tags, which specify both the destination of the message and the states of switches on the corresponding path. The TSDT scheme has the advantage that rerouting is possible when blockages occur for straight as well as nonstraight links.

As with the first scheme, the TSDT scheme assumes that each switch is appropriately initialized to behave as an odd_i or $even_i$ switch. Each "digit" of the routing tag is represented by two bits b_{n+i} and b_i, called the *state bit* and the *destination bit*, respectively. For this scheme, the state of a switch of stage i is specified by b_{n+i} : if b_{n+i}=0, the switch is in state C and if b_{n+i}=1 , the

switch is in state $\bar{C}$. For all i, $0 \leq i \leq n-1$, $b_i = d_i$. In general, if j is an $even_i$ switch, $b_i b_{n+i}$=00 and $b_i b_{n+i}$=01 direct the message through a straight link, $b_i b_{n+i}$=10 through link $+2^i$ and $b_i b_{n+i}$=11 through link -2^i; if j is an odd_i switch, $b_i b_{n+i}$=10 and $b_i b_{n+i}$=11 directs the message through a straight link, $b_i b_{n+i}$=01 through link $+2^i$ and $b_i b_{n+i}$=00 through link -2^i. In general, given a switch, the destination bit specifies use of a straight link or a non-straight link while the state bit determines the choice of the positive or the negative link (if the chosen link is a nonstraight link). Since state information is carried by the routing tag, switches are not required to determine and remember their own states, i.e. the design of the switches does not need to implement the logic states C and $\bar{C}$.

From Theorem 3.2, a nonstraight link blockage at stage i can be bypassed conveniently by complementing the i-*th* state bit while the destination bits remain unchanged. For convenience of reference, this is restated in terms of the TSDT scheme as Corollary 4.1 below.

Corollary 4.1 Let $b_{n/2n-1}$ and $b'_{n/2n-1}$ be the state bits of the routing tag and the rerouting tag, respectively, for the IADM network. In order to bypass a nonstraight link blockage at stage i, state bit b_{n+i} needs to be changed to $\bar{b}_{n+i}$. That is, $b'_{n/2n-1} = b_{n/n+i-1}\bar{b}_{n+i}b_{n+i+1/2n-1}$. □

Figure 7 illustrates an example of routing from $s = 1$ to $d = 0$ in an IADM network of size $N = 8$. Let $b_{0/5} = 000000$ be the routing tag and $b'_{0/5}$ and $b''_{0/5}$ denote the rerouting tags. The original tag $b_{0/5} = 000000$ specifies the path $(1 \in S_0, 0 \in S_1, 0 \in S_2, 0 \in S_3)$. If $(1 \in S_0, 0 \in S_1)$ is blocked, the rerouting tag $b'_{0/5} = 000100$ is obtained by complementing b_3, and link $(1 \in S_0, 2 \in S_1)$ is used for rerouting. This tag specifies the path $(1 \in S_0, 2 \in S_1, 0 \in S_2, 0 \in S_3)$. If $(2 \in S_1, 0 \in S_2)$ is also blocked, the rerouting tag $b''_{0/5} = 000110$ results from

complementing b_4', and link $(2 \in S_1 , 4 \in S_2)$ is used for rerouting. This tag specifies the path $(1 \in S_0 , 2 \in S_1 , 4 \in S_2 , 0 \in S_3)$.

As discussed in Section 3, a straight link blockage and a double nonstraight link blockage cannot be overcome easily; implementing a backtracking (or look-ahead) mechanism is a must in order to evade these types of blockages. Since all links in the routing path from stage $i-k+1$ to stage i consist of only straight links, backtracking of at least k stages is required to find the switch from which an alternate routing path branches. That is, at least k state bits need to be considered for change. Due to the similarity between Theorems 3.3 and 3.4, the TSDT schemes for finding the rerouting paths from Theorems 3.3 and 3.4 are exactly the same, which is stated as Corollary 4.2.

Corollary 4.2 Let $b_{n/2n-1}$ and $b'_{n/2n-1}$ be the state bits of the routing tag and the rerouting tag, respectively, for a source/destination pair in the IADM network. Let $i-k$ be the largest stage number for $i \geq k > 0$ such that a switch at stage $i-k$ is connected to a nonstraight link on the routing path. In order to bypass a straight link blockage or a double nonstraight link blockage at stage i, only state bits $b_{n+(i-k)/n+i-1}$ need to be changed; (i) $b'_{n/n+(i-1)} = b_{n/n+(i-k)-1}\bar{d}_{i-k/i-1}$ if the nonstraight link at stage $i-k$ of the original path is link -2^{i-k}, and (ii) $b'_{n/n+(i-1)} = b_{n/n+(i-k)-1}d_{i-k/i-1}$ if the nonstraight link at stage $i-k$ of the original path is link $+2^{i-k}$. The state bits $b'_{n+i/2n-1}$ have arbitrary values in both cases.

Proof: See Appendix A1. □

The example in Figure 7 can be used to illustrate the TSDT scheme for (a) a straight link blockage and (b) a double nonstraight link blockage. (a) Again the tag $b_{0/5} = 000000$ specifies a path $(1 \in S_0 , 0 \in S_1 , 0 \in S_2 , 0 \in S_3)$. If the straight link $(0 \in S_1 , 0 \in S_2)$ is blocked, the rerouting tag can be 000110 which

specifies path $(1 \in S_0, 2 \in S_1, 4 \in S_2, 0 \in S_3)$ by having $b'_{3+0}b'_{3+1}b'_{3+2} = \bar{d}_0\bar{d}_1 b_{3+2} = 110$. Since state bits $b'_{3+1}b'_{3+2}$ can be arbitrary, 000100, for example, is also a valid rerouting tag; it specifies path $(1 \in S_0, 2 \in S_1, 0 \in S_2, 0 \in S_3)$. (b) Let the tag $b_{0/5} = 000110$ specifies a path $(1 \in S_0, 2 \in S_1, 4 \in S_2, 0 \in S_3)$. If both nonstraight output links of $4 \in S_2$ are blocked, the rerouting tag $b'_{0/5}$ can be 000100 which specifies path $(1 \in S_0, 2 \in S_1, 0 \in S_2, 0 \in S_3)$ by having $b'_{3+0}b'_{3+1}b'_{3+2} = b_{3+0}d_1d_2$. Since state bits b'_{3+2} can be arbitrary, 000101 is also a valid rerouting tag which also specifies the same path.

The rerouting path computed from Corollary 4.2 is blockage-free from stage 0 to stage i. While the rerouting path is different from the original routing path from stage $i-k$ to stage i, the routing path from stage 0 to $i-k-1$ remains the same. This results from the fact that backtracking always proceeds backward along the original path until it stops at stage $i-k$, and the rerouting path only changes course from stage $i-k$ onwards. Although state bits $b_{n+i/2n-1}$ remain unchanged, the routing path from stage i to $n-1$ may still be altered due to the changes from stage $i-k$ to i. For example, in Figure 5, the switch on the original routing path at stage $i+1$ is $j \in S_{i+1}$ whereas the switch on the rerouting path at stage $i+1$ may be $(j+2^{i+1}) \in S_{i+1}$, which may further induce changes at higher-order stages.

In the TSDT scheme, the tag can be computed by the message sender which is assumed to know the location of faulty links and switches in the network. Thus, rerouting is transparent to the switches in the sense that the tag computed by the sender of the message simply avoids the usage of faulty links and switches. Therefore switches do not require any extra hardware for rerouting purposes. An alternative is to implement dynamic rerouting for the TSDT

scheme. Since backtracking is indispensable for avoiding a straight link blockage, it is required that each switch can detect the inaccessibility of any output port (connected to a switch at the next stage) and signal the presence of the blockage back to the switches of previous stages [10][12]. Whether rerouting is done by the sender or dynamically is an implementation decision which depends on how many stages of backtracking are allowed. When the sender computes the tag, it must be able to identify and track the switches and links on the corresponding routing and rerouting paths (the next paragraphs explain how this is done). If any of the switches or links in the path is known to the sender as being faulty, then the sender computes another tag by changing the state bits as described in Section 5.

Locating the switches on the routing path is straightforward. For a given source s and a destination d, the initial routing path can be specified by setting state bits $b_{n/2n-1} = 0_{n/2n-1}$ (a *string of* n 0's), equivalent to setting every switch in the IADM network to state C. Then every switch on the original path has label $d_{0/i-1}s_{i/n-1} \in S_i$, $0 \leq i \leq n-1$, since now the IADM network functions like an ICube network [6][15].

To find the switches on the rerouting path, let $j \in S_i$ be the switch whose output link is blocked. First consider the case where the blocked link is a non-straight link. It may be an (a) positive or (b) negative link. In case (a) the switch at stage $i+1$ reached by the positive link is $(j+2^i) \in S_{i+1}$ and, from Corollary 4.1, rerouting can done through switch $(j-2^i) \in S_{i+1}$. In case (b) the switch at stage $i+1$ reached by the negative link is $(j-2^i) \in S_{i+1}$ and, from Corollary 4.1, rerouting can done through switch $(j+2^i) \in S_{i+1}$. Let the switch at stage $i+1$ on the rerouting path be $w_{0/n-1}$. The state bits $b_{n+(i+1)/n-1}$ remain intact (equal to 0's) because it corresponds to having every switch from

stage $i+1$ to $n-1$ remain in state C so that the IADM network from stage $i+1$ to $n-1$ can emulate the ICube network from stage $i+1$ to $n-1$. Thus, the bits l, $i+1 \leq l \leq n-1$, of the label of a switch on the rerouting path are $w_{l/n-1}$. From Lemma 2.1, bits 0 to $l-1$, $1 \leq l \leq i+1$, of the label of a switch on a path to destination $d_{0/n-1}$ must be $d_{0/l-1}$. Hence the switch on the rerouting path from stage $i+1$ to $n-1$ has label $d_{0/l-1}w_{l/n-1}$, $i+1 \leq l \leq n-1$.

Next consider the case where the blockage of $j \in S_i$ is a straight link blockage or a double nonstraight link blockage so that backtracking is necessary. There are two sub-cases for each type of blockage: (i) the nonstraight link found in backtracking is a negative link and (ii) it is a positive link. Here only sub-case (i) of the straight link blockage is considered; the other cases can be dealt with similarly. From the proof of Corollary 4.2 (case (i) only), the switch on the rerouting path is $(j+2^l) \in S_l$, $i-k \leq l \leq i$. The switch of stage $i+1$ on the rerouting path is $j \in S_{i+1}$ if $b'_{n-i-1} = 0$ and $j \in S_{i+1}$ is an odd_i switch or if $b'_{n-i-1} = 1$ and $j \in S_{i+1}$ is an $even_i$ switch, and is $(j+2^{i+1}) \in S_{i+1}$ if $b'_{n-i-1} = 0$ and $j \in S_{i+1}$ is an $even_i$ switch or if $b'_{n-i-1} = 1$ and $j \in S_{i+1}$ is an odd_i switch. The identification of switches on the rerouting path from stage $i+1$ to $n-1$ is done as in the case of a nonstraight link blockage described above.

The blocked link can be represented by the two switches joined by the link. Since every switch on the original routing path and the rerouting paths can be easily identified as described above, it can be readily determined whether or not the blocked link is on the current path.

In summary, for both SDT schemes, the binary representation of the destination address can be used directly as the routing tag. In the SSDT scheme, rerouting tags are not needed and in the TSDT scheme, rerouting tags result from simple bit complementing operations. In terms of complexity of the

computation for a rerouting tag, the SSDT scheme and the TSDT scheme for one instance of nonstraight link blockage require time×space complexity O(1); an improvement over previous proposed schemes [9] dealing with rerouting for a nonstraight link blockage that require time×space complexity O(log$N$). In [10] a single-stage look-ahead scheme for rerouting of a straight link blockage was proposed; it requires use of two's complement to compute the positive and negative dominant tags so that the scheme has time×space complexity of O(log$N$). Note that the single-stage look-ahead rerouting scheme is valid only for some cases of the straight link blockage; it cannot be applied to any case of the straight link blockage. From Corollary 4.2, k-stage backtracking is needed for a straight link blockage and k bits of the state bits needs to be changed; thus the complexity of the TSDT scheme for a nonstraight link is O(k). If only single-stage backtracking (corresponds to single-stage look-ahead) is necessary, rerouting can be done dynamically and the complexity is O(1), an improvement over the scheme in [10].

5. A Universal Rerouting Algorithm for Multiple Blockages

The TSDT scheme can be applied to not only one instance of some blockage, but also can be applied repetitively each time a new blockage is encountered as the message propagates along. This section considers the derivation of an algorithm to deal with any case of multiple blockages. The backtracking schemes proposed in Corollary 4.2 find a rerouting path for a straight link blockage and a double nonstraight link blockage. Nevertheless, it is possible that blockages also exist on the rerouting path; then further backtracking to a lower-order stage is needed. Since this phenomenon can recur, repeated backtracking may be necessary due to blockages on the rerouting paths. The

algorithm BACKTRACK described next performs iterated backtracking to find an alternate routing path. It underlies a universal rerouting algorithm (called REROUTE) to be shown later that can find a routing path, if there exists any, to bypass multiple blockages in the network.

The inputs to algorithm BACKTRACK are the current routing path P, the stage number i where a blockage occurs, and state bits $b'_{n/2n-1}$ representing path P. The algorithm returns updated values of the state bits $b'_{n/2n-1}$ which specify a rerouting path that is blockage-free from stage 0 to stage i if such a rerouting path exists, or returns FAIL if the blockages on the current routing path and the rerouting paths eliminate the possibility of communication between the source and the destination. It is assumed that the blockage on the original routing path at stage i is a straight link blockage or a double non-straight link blockage and $j \in S_i$ is the switch whose output links are the blocked links. Informal explanations for the algorithm will be given following the algorithm and the correctness proof of this algorithm can be found in Appendix A2.

Algorithm BACKTRACK (and REROUTE) presumes existence of the knowledge of all blockages in the network. The network controller is responsible for collecting this information and maintaining a global map of blockages, which is accessible to every sender of the messages in order to compute a path to avoid the blockages. In addition, since it may take several iterations before a blockage-free path can be found or it can be concluded that no blockage-free paths exist, the sender of the message needs to maintain and update the locations of switches on the rerouting path in each iteration.

Algorithm BACKTRACK (P, i, $b'_{n/2n-1}$)

0: q = stage number where a blockage occur.

$q \leftarrow i$.

1: P = the current routing path.

Backtrack on path P from stage q to find a nonstraight link. If no nonstraight link exists at any preceding stage, return(FAIL); otherwise assign to r the stage number where the first nonstraight output link is found.

2: If the nonstraight link at stage r on the routing path is $+2^r$, assign flag *linkfound* value 0; if it is -2^r, assign *linkfound* value 1.

3: If *linkfound* $= 0$, $b'_{n/2n-1} \leftarrow b'_{n/n+r-1} d_{r/q-1} b'_{n+q/2n-1}$; if *linkfound* $= 1$, $b'_{n/2n-1} \leftarrow b'_{n/n+r-1} \bar{d}_{r/q-1} b'_{n+q/2n-1}$.

4a: This step applies only when the blockage at stage q on path P is a straight link blockage.

If *linkfound* $= 0$, set $b'_{n+q} = d_q$; if $((j-2^q) \in S_q , (j-2^{q+1}) \in S_{q+1})$ is blocked, change b'_{n+q} to $\bar{d}_q$; furthermore, if $((j-2^q) \in S_q , j \in S_{q+1})$ is also blocked, return(FAIL). If *linkfound* $= 1$, set $b'_{n+q} = \bar{d}_q$; if $((j+2^q) \in S_q , (j+2^{q+1}) \in S_{q+1})$ is blocked, change b'_{n+q} to d_q; furthermore, if $((j+2^q) \in S_q , j \in S_{q+1})$ is also blocked, return(FAIL).

4b: This step applies only when the blockage at stage q on path P is a double nonstraight link blockage.

If $((j-2^q) \in S_q , (j-2^q) \in S_{q+1})$ is blocked for *linkfound* $= 0$, or $((j+2^q) \in S_q , (j+2^q) \in S_{q+1})$ is blocked for *linkfound* $= 1$, return(FAIL).

5: Let $\hat{Q}$ denotes the part of the rerouting path (specified by the tag in step 3) from stage $r+1$ to q from step 3.

If *linkfound* $= 0$, $\hat{Q} = ((j-2^{r+1}) \in S_{r+1} , \cdots , (j-2^{q-1}) \in S_{q-1} , (j-2^q) \in S_q)$; if *linkfound* $= 1$,

$\hat{Q} = ((j+2^{r+1}) \in S_{r+1}, \cdots, (j+2^{q-1}) \in S_{q-1}, (j+2^{q}) \in S_{q})$.

If a blockage occurs on path $\hat{Q}$, return(FAIL).

6: If *linkfound* $= 0$ and $((j-2^{r}) \in S_{r}, (j-2^{r+1}) \in S_{r+1})$ is blocked, or if *linkfound* $= 1$ and $((j+2^{r}) \in S_{r}, (j+2^{r+1}) \in S_{r+1})$ is blocked, go to step 7; else return($b'_{n/2n-1}$).

7: $j \leftarrow j+2^{r}$, $q \leftarrow r$.

8: Backtrack on path P from stage q to find a nonstraight link. If no nonstraight link exists at any preceding stage, return(FAIL); otherwise assign to r the stage number where the first nonstraight output link is found.

9: If *linkfound* $= 0$ and the nonstraight link at stage r is -2^{r}, or if *linkfound* $= 1$ and the nonstraight link at stage r is $+2^{r}$, return(FAIL).

10: If *linkfound* $= 0$, $b'_{n/2n-1} \leftarrow b'_{n/n+r-1} d_{r/q-1} b'_{n+q/2n-1}$; if *linkfound* $= 1$, $b'_{n/2n-1} \leftarrow b'_{n/n+r-1} \bar{d}_{r/q-1} b'_{n+q/2n-1}$. Go to step 4b.

Step 0 is the initialization step. From Theorems 3.3 and 3.4, an alternate path exists for avoiding a straight link blockage or a double nonstraight link blockage if and only if there exists a nonstraight link at some stage preceding stage r; step 1 of the algorithm searches backward for such a nonstraight link. If not found, it results in premature termination of the algorithm, reflecting the fact that no alternate paths for rerouting exist. Step 2 is used to differentiate the cases when the nonstraight link at stage r found in the first backtracking is a positive link and when it is a negative link; flag *linkfound* is assigned 0 for the former and 1 for the latter. If a nonstraight link exists at some stage preceding the blockages, in step 3, Corollary 4.2 is applied to find the stage bits specifying the rerouting path; cases (i) and (ii) in Corollary 4.2 correspond to *linkfound* $= 1$ and *linkfound* $= 0$, respectively, and q and r correspond to i

and $i-k$, respectively.

Steps 4a and 4b deal with the link blockage at stage q on the rerouting path computed in step 3. If the blockage of a switch at stage q on path P is a straight link, the possible rerouting links at stage q are two nonstraight links. In step 4a the default link is negative link if $linefound = 0$ and a positive link if $linkfound = 1$. If the default link is blocked, step 4a attempts to reroute the message through the other nonstraight link. If both nonstraight links are blocked, there exist no blockage-free paths. Step 4b applies if the blockage of a switch at stage q on path P is a double nonstraight link blockage. The rerouting path must use a straight link at stage q. If it is also blocked, no blockage-free path exists.

Step 5 checks blockages from stage $r+1$ to stage $q-1$ on the rerouting path; if any blockage falls on $\hat{Q}$, there exists no blockage-free path. In step 6, if the blockage falls in the link of stage r on the rerouting path, further backtracking is necessary. Otherwise (no blockages on the rerouting path), the algorithm terminates with the state bits specifying the rerouting path. Step 7 updates the stage number q and the switch label j where a blockage on the rerouting path occurs, initiating a new iteration of backtracking. Step 8 is the same as step 1, searching backward at lower-order stages again for a nonstraight link. Step 9 of the algorithm dictates that if the encountered nonstraight link in the first iteration of backtracking is a positive (or negative) link, the nonstraight link found in each subsequent iteration of backtracking must be also a positive (or negative) link; otherwise no blockage-free paths exist. If the condition in step 9 is satisfied, step 10, which is the same as step 3, computes a rerouting path. After the rerouting path is found, the algorithm returns to step 4b, to check for further blockages on the rerouting path.

For each source/destination pair, a link on some routing path for the source/destination pair is called a *participating* link. As a direct result of Theorem 3.2, the set of participating output links of a switch is composed of either its straight output link or both of its nonstraight output links, but never all of them. So the output link blockages of a switch, for a given source/destination pair, can only be a straight link blockage, a nonstraight link blockage, or a double nonstraight link blockage. Algorithm BACKTRACK deals with the first and third kind of blockages, and the second kind of blockage can be overcome by applying Corollary 4.1. Algorithm BACKTRACK and Corollary 4.1 can be used to form a universal algorithm capable of rerouting messages when multiple blockages exist in the IADM network. This algorithm, called REROUTE, returns state bits $b'_{n/2n-1}$ specifying a blockage-free rerouting path if one exists, or returns FAIL otherwise.

Algorithm REROUTE $(P, b'_{n/2n-1})$

0: P= the original routing path.

$b_{n/2n-1}$= the routing tag specifying the original routing path.

$b'_{n/2n-1}$= the rerouting tag specifying the rerouting path.

$b'_{n/2n-1} \leftarrow b_{n/2n-1}$.

1: Let i be the smallest stage number such that there exists a blockage at stage i on path P. If no blockages occur on path P, return$(b'_{n/2n-1})$.

2: If the blockage at stage i on path P is a nonstraight link blockage and the other nonstraight link is not blocked, apply Corollary 4.1 to find state bits $b'_{n/2n-1}$ and go to step 4.

3: $b'_{n/2n-1} \leftarrow$ BACKTRACK$(P, i, b'_{n/2n-1})$.

4: Q = the rerouting path specified by state bits $b'_{n/2n-1}$.

$P \leftarrow Q$ and go to step 1.

Step 0 is the initialization step. At the end of each iteration, a blockage-free path from stage 0 to stage i is found. Then a new iteration starts and i is given a new value in order to find a path avoiding the blockages at a higher-order stage. The only terminating conditions for algorithm REROUTE are that a return of FAIL from step 3 indicating that no blockage-free paths exist and the return from step 1 indicating a blockage-free path is found. Algorithm REROUTE is executed iteratively to evade blockages from lower-order to higher-order stages. The correctness of this algorithm follows from the correctness of algorithm BACKTRACK and Corollary 4.1.

6. Permutation Routing and Cube Subgraphs of the IADM Network

The results discussed so far are a consequence of the existence of spare nonstraight links in addition to the ICube network embedded in the IADM network. This section pursues this issue further by showing that there exist multiple distinct subgraphs in the IADM network, each called a *cube subgraph*, that are isomorphic to the ICube network. Two cube subgraphs are considered to be distinct if they differ in at least one link. As mentioned in the introduction of this paper, the cube-type networks have been studied extensively in the literature and shown to be topologically equivalent. Together with results from these studies, the knowledge of how to identify cube subgraphs can help the understanding of the capabilities of the IADM network and be useful for permutation routing in the IADM network.

Since each switch can be in state C or $\bar{C}$, there are as many as $2^{N \cdot n}$ $(= N^N)$ network states, although each does not necessarily generate a unique

permutation. Setting a switch to a certain state indicates that one of its non-straight output links can be used for routing (i.e. it is *active*) while the other cannot. Thus, each network state can be associated with a subgraph of the IADM network which contains only the active links. When all switches in the IADM network are set to state C, the IADM network functions as an ICube network; this network state corresponds a cube subgraph. The constructive derivation of a lower bound for the number of cube subgraphs of the IADM network uses the two basic ideas discussed in the next paragraphs.

Since $+2^{n-1} \equiv -2^{n-1} \ mod \ N$, $C_{n-1}(j,t_{n-1}) = \bar{C}_{n-1}(j,t_{n-1})$, i.e. the state of each switch of stage $n-1$ is irrelevant in the sense that any switch at stage $n-1$ is always connected to the same two switches at stage n. Consequently, given any cube subgraph, there exist (2^N-1) subgraphs isomorphic to it which differ only in their choices of the nonstraight link $+2^{n-1}$ or -2^{n-1} at stage $n-1$. Therefore, the total number of distinct cube subgraphs is given by the product of 2^N and the number of distinct subgraphs of the IADM network from stage 0 to stage $n-2$ that are isomorphic to the same stages in the ICube network.

The calculation of the number of subgraphs in the first $n-1$ stages uses an idea similar to that proposed in [5] for reconfiguring the DR network so that it performs as a Generalized Cube network. All switches of the IADM network are logically relabeled by adding a constant x, $0 \leq x \leq N-1$ to the original labels, i.e. switch j becomes $j' = j + x$. By setting each switch to be an $even_i$ or odd_i switch according to its new label and having all switches be in state C, a cube subgraph results for each relabeling. However, of the N possible subgraphs, only $\frac{N}{2}$ are distinct as far as the first $n-1$ stages are concerned. This result is stated in Theorem 6.1. A graphical interpretation of cube subgraph isomorphism for an IADM network of size $N=8$ is illustrated in Figure 8. In

Figure 8, each physical switch j acts as a logical switch $j' \equiv (j+1)\ mod\ 8$. The isomorphism to the ICube network can be easily visualized by moving switch 7 to the top of each stage as shown in the figure. Notice that setting some switch to state C according to its logical label may be equivalent to setting the switch to state $\bar{C}$ according its original label. For instance, switch $0 \in S_0$ (logical label 1) is set to state $\bar{C}$ in Figure 8.

Theorem 6.1 There exist at least $\frac{N}{2} \cdot 2^N$ distinct cube subgraphs in the IADM network.

Proof: See Appendix A1. □

In order to reconfigure the IADM network to one of its cube subgraphs, each switch of stage i, for $0 \leq i \leq n-2$, needs to know the i-*th* bit of its logical label. This can be done by sending the same logical label to every switch in the same row at system reconfiguration time. Each switch is set as being an odd_i or $even_i$ switch by examining the i-*th* bit of the logical label. All switches operate in state C according to its logical label with the exception of those at stage $n-1$ for which different states correspond to different subgraphs.

The results of this section can be used in different ways. One usage is in characterizing a class of permutations performable by the IADM network. Permutations passable by the ICube network are discussed in [15] and adaptable from [6]. Thus, the IADM network can perform all of these permutations plus the same set of permutations with a given x added to both the same source and destination labels, $0 \leq x < \frac{N}{2}$. Another use of the results of this section is that the IADM network can pass the permutations performable by the ICube network when the ICube network embedded in the IADM network experiences nonstraight link failures. This is done by incorporating a reconfiguration

function in the system that reassigns each switch j to $(j+r)$ and reconfiguring the IADM network to a corresponding cube subgraph which does not include the faulty nonstraight links. In [21] it is shown that any of the cube-type networks can pass the permutations performable by the others by incorporating appropriate reconfiguration functions. By the same token, the IADM network with a nonstraight link fault can also pass the permutations performable by the cube-type networks by including these reconfiguration functions in the system.

7. Concluding Remarks

One of the main contributions of this paper is the identification of destination tag routing schemes for the IADM network. They are simpler and more efficient than previously known approaches, thus requiring less complex switches and reducing message communication delays due to routing overhead. In the SSDT scheme rerouting can be done when nonstraight links fail and in the TSDT scheme both the straight and double nonstraight link blockages can be avoided. As for the SSDT scheme, routing and rerouting are transparent to the source and only negligible hardware and time are used by each switch for routing and rerouting purpose. These are considerable advantages over previously proposed schemes which do not use destination tags and require extra hardware or delays of $O(\log N)$ complexity instead of $O(1)$. In addition, previous works all deal only with certain types of bloc[illegible]. Based on the TSDT scheme, a universal rerouting algorithm is deri[illegible], which is capable of avoiding any combination of multiple blockages if there exist a blockage-free path and indicating absence of such a path if there exists none. The rerouting capabilities of the new schemes can be readily used for fault-tolerance and load balancing purposes since they adequately exploit the redundancy available in the IADM

network.

Another contribution of this paper is the constructive derivation of a lower bound on the number of cube subgraphs of the IADM network. While it was previously known that the ICube network is a subgraph of the IADM network, this paper shows that there exist at least $\frac{N}{2} \cdot 2^N$ distinct cube subgraphs. This, combined with previous multistage cube network studies, can help characterize some of the permutations performable by the IADM network. As other use of the subgraph analysis, it is shown how to reconfigure the IADM network under nonstraight link faults to pass the cube-admissible permutations.

Perhaps the most fundamental contribution of this paper is that of the network state model used for the IADM and the ICube networks. The essence of this model is in the recognition that the routing action of each switch is conceptually dependent on its position in the network (topological information), its state (functional information), and the destination of the message (routing information). Topological information is fixed and, when using destination tags, the same can be said of routing information for a given message destination. Consequently, the routing path is solely determined by the state of the network. These basic concepts are applicable to networks other than those considered in this paper; the state model can help devise new designs, solve routing problems, and understand relationships among networks.

References

[1] D. Agrawal, "Graph Theoretical Analysis and Design of Multistage Interconnection Networks," *IEEE Trans. Computers*, Vol. C-32, No. 7, July 1983, pp. 637-648.

[2] K. E. Batcher, "The Flip Network in STARAN," *1976 Int'l Conf. Parallel Processing*, Aug. 1976, pp. 65-71.

[3] T-Y Feng, "Data Manipulating Functions in Parallel Processors and their Implementations," *IEEE Trans. Computers*, Vol. C-23, No. 3, Mar. 1974, pp. 309-318.

[4] L. R. Goke and G. J. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems," *1st Ann. Symp. Computer Architecture*, Dec. 1973, pp. 21-28.

[5] M. Jeng and H. J. Siegel, "A Fault-tolerant Multistage Interconnection Network for Multiprocessor Systems Using Dynamic Redundancy," *6th Int'l Conf. Distributed Computing Systems*, May 1986, pp. 70-77.

[6] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. Computers*, Vol. C-24, No. 12, Dec. 1975, pp. 1145-1155.

[7] D. Lee and K. Y. Lee, "Control Algorithms for the Augmented Data Manipulator Network," *1986 Int'l Conf. Parallel Processing*, Aug. 1986, pp. 123-130.

[8] M. Malek and W. W. Myre, "A Description Method of Interconnection Networks," *IEEE Tech. Committee Distrib. Process.*, Quart. Vol. 1, Feb. 1981, pp. 1-6.

[9] R. J. McMillen and H. J. Siegel, "Routing Schemes for the Augmented Data Manipulator Network in an MIMD System," *IEEE Trans. Computers*, Vol. C-31, No. 12, Dec. 1982, pp. 1202-1214.

[10] R. J. McMillen and H. J. Siegel, "Performance and Fault Tolerance Improvements in the Inverse Augmented Data Manipulator Network," *9th Ann. Symp. Computer Architecture*, Apr. 1982, pp. 63-72.

[11] R. J. McMillen and H. J. Siegel, "Evaluation of Cube and Data Manipulator Networks," *J. Parallel and Distributed Computing*, Vol. 2, No. 1, Feb. 1985, pp. 79-107.

[12] K. Padmanabhan and D. H. Lawrie, "A Class of Redundant Path Multistage Interconnection Networks," *IEEE Trans. Computers*, Vol. C-32, No. 12, Dec. 1983, pp. 1099-1108.

[13] D. S. Parker and C. S. Raghavendra, "The Gamma Network: A Multiprocessor Interconnection Network with Redundant Paths," *9th Ann. Symp. Computer Architecture*, Apr. 1982, pp. 73-80.

[14] D. S. Parker and C. S. Raghavendra, "The Gamma Network," *IEEE Trans. Computers*, Vol. C-33, No. 4, Apr. 1984, pp. 367-373.

[15] M. C. Pease, III, "The Indirect Binary n-Cube Microprocessor Array," *IEEE Trans. Computers*, Vol. C-26, No. 5, May 1977, pp. 458-473.

[16] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, Lexington Books, D. C. Heath and Company, Lexington, MA, 1985.

[17] H. J. Siegel and S. D. Smith, "Study of Multistage SIMD Interconnection Networks," *5th Ann. Symp. Computer Architecture*, Apr. 1978, pp. 223-229.

[18] H. J. Siegel and R. J. McMillen, "The Multistage Cube : A Versatile Interconnection Network," *IEEE Computer*, Vol. 14, Dec. 1981, pp. 65-76.

[19] A. Varma and C. S. Raghavendra, "On Permutations Passable by the Gamma Network," *J. Parallel and Distributed Computing*, Vol. 3, No. 1, pp. 72-91, Mar. 1986.

[20] C-L. Wu and T-Y. Feng, "On a Class of Multistage Interconnection Networks," *IEEE Trans. Computers*, Vol. C-29, No. 8, Aug. 1980, pp. 694-702.

[21] C-L. Wu and T-Y. Feng, "The Reverse-Exchange Interconnection Network," *IEEE Trans. Computers*, Vol. C-29, No. 9, Sept. 1980, pp. 801-811

Appendix A1

Proof of Theorem 3.3

The "only if" part follows immediately from Theorem 3.2. To prove the "if" part, let $j \in S_i$ be the switch whose straight output link is the blocked link on the routing path and $i-k$ be the largest stage number for $i \geq k > 0$ such that a switch at stage $i-k$ has a nonstraight output link on the routing path. (i) Assume that the nonstraight link at stage $i-k$ found in backtracking is link -2^{i-k}. Clearly, as illustrated in Figure 5, the path $((j+2^{i-k}) \in S_{i-k}, (j+2^{i-k+1}) \in S_{i-k+1}, \ldots, (j+2^{i}) \in S_i, j \in S_{i+1})$ is a rerouting path for path $((j+2^{i-k}) \in S_{i-k}, j \in S_{i-k+1}, j \in S_{i-k+2}, \ldots, j \in S_i, j \in S_{i+1})$. (ii) Assume that the nonstraight link at stage $i-k$ found in backtracking is link $+2^{i-k}$; similarly path $((j-2^{i-k}) \in S_{i-k}, (j-2^{i-k+1}) \in S_{i-k+1}, \ldots, (j-2^{i}) \in S_i, j \in S_{i+1})$ is a rerouting path for path $((j-2^{i-k}) \in S_{i-k}, j \in S_{i-k+1}, j \in S_{i-k+2}, \ldots, j \in S_i, j \in S_{i+1})$.

Proof of Theorem 3.4

The "only if" part again follows immediately from Theorem 3.2. To prove the "if" part, let notations i, $i-k$ and $j \in S_i$ be the same as those in the proof of Theorem 3.3. The proof is illustrated in Figure 6. From Theorem 3.2, $(j-2^{i}) \in S_{i+1}$ and $(j+2^{i}) \in S_{i+1}$ can reach the same subset of destinations so that it does not matter which is on the rerouting path. (i) Assume that the nonstraight link at stage $i-k$ found in backtracking is link -2^{i-k}. It is self-explanatory that path $((j+2^{i-k}) \in S_{i-k}, (j+2^{i-k+1}) \in S_{i-k+1}, \ldots, (j+2^{i}) \in S_i, (j+2^{i}) \in S_{i+1})$ is a rerouting path for both paths $((j+2^{i-k}) \in S_{i-k}, j \in S_{i-k+1}, \ldots, j \in S_i, (j+2^{i}) \in S_{i+1})$ and $((j+2^{i-k}) \in S_{i-k}, j \in S_{i-k+1}, \ldots, j \in S_i, (j-2^{i}) \in S_{i+1})$. (ii) Assume that the nonstraight link at stage $i-k$ found in backtracking is link $+2^{i-k}$; similarly

path $((j-2^{t-k})\in S_{t-k}, (j-2^{t-k+1})\in S_{t-k+1}, \ldots, (j-2^{t})\in S_{t}), (j-2^{t})\in S_{t+1})$ is a rerouting path. Note that the participating input link of $j\in S_t$ may be a non-straight link; however, this is just a special case for $k = 1$. □

Proof of Corollary 4.2

First two lemmas are presented, which are to be used to prove Corollary 4.2.

Lemma A1.1 In the TSDT scheme, the links $+2^l$ and -2^l connected to a switch $j\in S_l$ are specified by tag bits $b_l b_{n+l} = \bar{j}_l j_l$ and $b_l b_{n+l} = \bar{j}_l \bar{j}_l$ respectively, and the straight link is specified by $b_l b_{n+l} = j_l j_l$ or $b_l b_{n+l} = j_l \bar{j}_l$.

Proof: Follow immediately from the definition for the TSDT scheme. □

Lemma A1.2 (i) Let $j\in S_l$ and $(j+2^l)\in S_{l+1}$ be two switches joined by a positive nonstraight link $+2^l$ and they are on a path to the destination $d_{0/n-1}$. In the TSDT scheme, the routing tag can be set to $b_l b_{n+l} = d_l \bar{d}_l$ to control routing to send the message from $j\in S_l$ to $(j+2^l)\in S_{l+1}$. (ii) Let $j\in S_l$ and $(j-2^l)\in S_{l+1}$ be two switches joined by a negative nonstraight link -2^l and they are on a path to the destination $d_{0/n-1}$. In the TSDT scheme, the routing tag can be set to $b_l b_{n+l} = d_l d_l$ to control routing to send the message from $j\in S_l$ to $(j-2^l)\in S_{l+1}$.

Proof: Only proof for (i) is given and proof for (ii) is similar. From Lemma 2.1 and the proof for Theorem 3.1, the switch $j'(= j+2^l)\in S_{l+1}$ has the label $j'_{0/n-1} = d_{0/l-1} d_l w_{l+1/n-1}$, where $w_{l+1/n-1}$ depends on network state. So $j'_l = d_l$. Additionally, $j'_l = \bar{j}_l$ because $j' = j+2^l$. Hence $j_l = \bar{d}_l$. By Lemma A1.1, $b_l b_{n+l} = d_l \bar{d}_l$. □

Proof of Corollary 4.2:

Only proofs of (i) for (a) a straight link blockage and for (b) a double non-straight link blockage are given; proofs of (ii) for cases (a) and (b) are similar. Since the destination bits always remain unchanged, only state bits need to be

considered. (a) This proof first derives the state bits controlling the rerouting path $Q^+ = ((j+2^{i-k}) \in S_{i-k}, (j+2^{i-k+1}) \in S_{i-k+1}, \ldots, (j+2^i) \in S_i)$ in Figure 5 (which illustrates the proof of Theorem 3.3). Since the links on path Q^+ are all positive nonstraight links, by Lemma A1.2, $b'_{n+(i-k)/n-1} = \bar{d}_{i-k/n-1}$ represents the state bits for path Q^+. In addition, by Theorem 3.2, the link of stage i on the rerouting path can be either link -2^i $((j+2^i) \in S_i, j \in S_{i+1})$ or link $+2^i$ $((j+2^i) \in S_i, (j+2^{i+1}) \in S_{i+1})$. Thus b'_{n+i} can be 0 or 1. (b) Notice that the rerouting paths from stage $i-k$ to stage i found in Theorem 3.3 and Theorem 3.4 are the same except the the link of stage i on the rerouting path is a nonstraight link in Theorem 3.3 (Figure 5) and it is a straight link in Theorem 3.4 (Figure 6). By Lemma A1.1, the state bit b'_{n+i}, which specifies the straight link at stage i in Theorem 3.4, can be 0 or 1. So the state bits specifying the rerouting path from stage $i-k$ to stage i are the same as those in (a). $b'_{n+(i+1)/2n-1}$ can be *arbitrary* because, *regardless of the values of* $b'_{n+(i+1)/2n-1}$, as long as the destination bits are $b_{0/n-1} = d_{0/n-1}$, the path can reach the destination $d_{0/n-1}$. □

Proof of Theorem 6.1

Consider two cube subgraphs generated by adding x and y, respectively, to the original labels of all switches of the IADM network. It is shown that $x \bmod \frac{N}{2} \neq y \bmod \frac{N}{2}$ is a sufficient condition for these subgraphs to be distinct in the sense that they differ in at least one link of the first $n-2$ stages (it is also possible to show the necessity of this condition). To prove that the subgraphs are distinct, it is shown that, given the condition above, there exists some physical switch $j' \in S_{n-2}$ such that $(j'+x)$ and $(j'+y)$ differ in their $(n-2)$-*th* bit, i.e. the switch with logical label $(j'+x)$ is an $even_i$ switch and the switch with logical label $(j'+y)$ is an odd_i switch, or vice versa. This implies

that a different nonstraight link is used and therefore the subgraphs are distinct. Let the h-th bit of $x_{0/n-2}$ and $y_{0/n-2}$ be the highest order bit such that $x_h \neq y_h$, i.e. $x_{h+1/n-2} = y_{h+1/n-2}$. Here $h \leq n-2$ since only the topology of the IADM network from stage 0 to stage $n-2$ is considered. Without loss of generality, assume that $x_h = 0$ and $y_h = 1$ and let $j'_{0/n-2} = 0_{0/h-1}1\bar{x}_{h+1/n-2}$ (where $0_{0/h-1}$ is a string of h 0's). Then

$$(j'+x)_{0/n-2} = x_{0/h-1}(0+1)1_{h+1/n-2} = x_{0/h-1}1_{h/n-3}1 \qquad \text{and}$$

$$(j'+y)_{0/n-2} = y_{0/h-1}(1+1)1_{h+1/n-2} = y_{0/h-1}0_{h/n-3}0$$

differ in the value of their $(n-2)$-th bit. Therefore there exist $\frac{N}{2}$ distinct cube subgraphs when considering only the topology of the IADM network from stage 0 to stage $n-2$. For each of these $\frac{N}{2}$ cube subgraphs, there exist 2^N subgraphs of the IADM network which differ from it only in the choice of the nonstraight links at stage $n-1$. Thus, the IADM network contains at least $\frac{N}{2} \cdot 2^N$ distinct cube subgraphs. □

Appendix A2: Proof of Algorithm BACKTRACK

Terminology and two lemmas are introduced first in order to lay the ground for the verification of algorithm BACKTRACK. Given a source and a destination, a switch on some routing path for the source/destination pair is called a *pivot*. Conversely, by the definition of a pivot, a path in the IADM network can reach the destination if and only if it passes through a pivot at each stage. The set of pivots at each stage varies with different source/destination pair and is characterized by the following lemma.

Lemma A2.1 Let $\hat{k}$ be the smallest stage number for which there exists a non-straight link on at least one routing path from a given source $s_{0/n-1}$ to a given destination $d_{0/n-1}$ in the IADM network. For this source/destination pair, there is exactly one pivot at stage k', $0 \leq k' \leq \hat{k}$, and there exists exactly two pivots at stage k'', $\hat{k}+1 \leq k'' \leq n-1$. The pivot at stage k' is $d_{0/k'-1}s_{k'/n-1}$. The pivots of stage k'' are $d_{0/k''-1}s_{k''/n-1}$ and either $(d_{0/k''-1}s_{k''/n-1}+2^{k''})$ or $(d_{0/k''-1}s_{k''/n-1}-2^{k''})$.

Proof: By definition of $\hat{k}$, the routing paths from stage 0 to $\hat{k}-1$ consist of only straight links. From Theorem 3.2, there exists a unique path from stage 0 to stage $\hat{k}$ and, therefore, the set of pivots at stage k', $0 \leq k' \leq \hat{k}$, consists of exactly one pivot. Existence of exactly two pivots at stage k'', $\hat{k}+1 \leq k'' \leq n-1$, and that their distance is $2^{k''}$ follow immediately from the single theorem in [13]. Since the IADM network functions like an ICube network when every switch in the IADM network is set to state C, $d_{0/k-1}s_{k/n-1} \in S_k$, $0 \leq k \leq n-1$, is on a routing path [6][15]; the lemma follows. □

Lemma A2.1 captures a simple characteristic of routing in the IADM network and, for each source/destination pair, it allows the discussion to focus only on the behavior of the pivots at each stage. A pivot is *unreachable* if all its participating input links (defined in Section 3) are blocked, and it is *closed* if all its participating output links are blocked. A pivot of a lower-order stage can be closed due to the closure of pivots at higher-order stages. Likewise, a pivot of higher-order stage can be unreachable due to unreachability of pivots at lower-order stages. From the definition of a pivot, an important lemma which identifies the causes for the absence of blockage-free paths between a source/destination pair is stated as follows.

Lemma A2.2 In the IADM network, for a given source/destination pair, if all pivots of some stage are closed or unreachable, there exist no blockage-free paths for the source/destination pair. □

Lemmas A2.1 and A2.2 describe the behavior of the switches and the links in the set of routing paths for each source/destination pair. These lemmas make it possible to ignore switches other than pivots and links other than participating links at each stage for a source/destination pair. These results greatly simplify the complexity of rerouting in the IADM network.

The correctness proof for algorithm BACKTRACK consists of two parts. First is that the path found by the algorithm is a valid path leading to the destination and capable of avoiding blockages in the network. Second is that algorithm BACKTRACK always finds a rerouting path if there exists any, which is equivalent to that algorithm BACKTRACK returns FAIL only if there exist no blockage-free paths. To prove these two parts, it requires examination of the conditions that terminate algorithm BACKTRACK.

The rerouting path found by the algorithm can route the message to the destination because the destination bits of the rerouting tags equal to the binary representation of the destination address. The rerouting path's ability to evade blockages is a natural consequence of Corollary 4.2, on which steps 3 and 10, the only steps in the algorithm that generate rerouting tags, are based. Notice that step 6 returns the rerouting tag if the rerouting path found from step 3 or 10 is blockage-free.

The steps that return FAIL are steps 1, 4a, 4b, 5, 8 and 9. Steps 1 and 8 return FAIL because no alternate routing paths exist. Steps 4a, 4b, 5 and 9 return FAIL because the communication between the source and the destination is broken due to the blockages in the network. So it is impossible for a

blockage-free path to exist without algorithm BACKTRACK finding it and not returning FAIL. Validity of steps 1 and 8 was discussed. Therefore, the proof for the second part is complete if steps 4a, 4b, 5 and 9 are verified.

Proof of steps 4a and 4b

In the following discussion for steps 4a and 4b, only the case where *linkfound* = 1 is explored; the cases where *linkfound* = 0 can be treated analogously. In Figure 5 (*linkfound* = 1 and $q = i$), the blockage at stage q on path P is a straight link blockage and the link at stage q on the rerouting path is chosen to be $((j+2^q)\in S_q , (j+2^{q+1})\in S_{q+1})$ by setting $b'_{n+q} = \bar{d}_q$ (Lemma A1.2). A blockage in $((j+2^q)\in S_q , (j+2^{q+1})\in S_{q+1})$ can be overcome by rerouting the message through the other nonstraight link $((j+2^q)\in S_q , j\in S_{q+1})$. This is done by complementing b'_{n+q}. If $((j+2^q)\in S_q , j\in S_{q+1})$ is also blocked, links $(j\in S_q , j\in S_{q+1})$, $((j+2^q)\in S_q , j\in S_{q+1})$ and $((j+2^q)\in S_q , (j+2^{q+1})\in S_{q+1})$ are all blocked, thus both pivots at stage q, $j\in S_q$ and $(j+2^q)\in S_q$, are closed. Hence no blockage-free paths exist. The above explains step 4a. In Figure 6 (*linkfound* = 1 and $q = i$) both nonstraight links of $j\in S_q$ on path P, $(j\in S_q , (j-2^q)\in S_{q+1})$ and $(j\in S_q , (j+2^q)\in S_{q+1})$, are blocked and thus pivot $j\in S_q$ is closed. If $((j+2^q)\in S_q , (j+2^q)\in S_{q+1})$ is also blocked, pivot $(j+2^q)\in S_q$ is also closed. Because both pivots of stage q, $j\in S_q$ and $(j+2^q)\in S_q$, are closed, there exist no blockage-free paths. This explains step 4b. □

The scope of the correctness proof for steps 6 and 10 is limited to the case where the first nonstraight link found in backtracking is -2^r (*linkfound* = 1) and assumes that the blockage at stage i is a double nonstraight link blockage. Discussions for the cases where link $+2^r$ is the first nonstraight link found in backtracking and where the blockage at stage i is a straight link blockage can be treated analogously.

An interesting property regarding the behavior of the pivots at each iteration of backtracking is discussed here. This is to be used in the correctness proof for steps 5 and 9. The discussions are associated with Figures 5 and 6 for $q = i$ and $r = i-k$. Since the links on path P from stage $r+1$ to $q-1$ are all straight links, by Theorem 3.2, there exist no alternate routing paths from $j \in S_{r+1}$ to $j \in S_q$. So the closure of $j \in S_q$ would effectively close every pivot $j \in S_l$, $r+1 \leq l \leq q-1$. Hence if $j \in S_q$ is closed, every $j \in S_l$, $r+1 \leq l \leq q$, is closed. Due to the closure of $j \in S_{r+1}$, $((j+2^r) \in S_r$, $j \in S_{r+1})$ is blocked. If $((j+2^r) \in S_r$, $(j+2^{r+1}) \in S_{r+1})$ is also blocked (step 6), both participating output links of $(j+2^r) \in S_r$ are blocked and thus $(j+2^r) \in S_r$ is closed. After j and q are updated in step 7 (i.e. $j \leftarrow j+2^r$, $q \leftarrow r$ so that $(j+2^r) \in S_r$ becomes $j \in S_q$), the same type of blockage recurs (i.e. both nonstraight output links of $j \in S_q$ are blocked and thus $j \in S_q$ is closed) as that which took place when the algorithm was first entered (i.e. $q = i$) and thus a new iteration of backtracking begins. For convenience of reference, the property described in this paragraph is formally restated as a lemma.

Lemma A2.3 In each iteration of backtracking in algorithm BACKTRACK, on path P every pivot $j \in S_l$, $r+1 \leq l \leq q$, is closed; if $((j+2^r) \in S_r$, $(j+2^{r+1}) \in S_{r+1})$ is also blocked, $(j+2^r) \in S_r$ is also closed. □

Beginning from the second iteration of backtracking, the link of stage q on the rerouting path is always a straight link, since the blockage at the onset of each iteration of backtracking is always that both nonstraight output links of $j \in S_q$ are blocked (Figure 6). Hence only step 4b is concerned in checking the blockages of stage q on the rerouting path. As a result, in Figures 5 and 6 ($linkfound = 1$), the links on path P from stage r to stage i consist of only straight links and negative nonstraight links; correspondingly, the links on the

rerouting path from stage r to stage i consist of only straight links and positive nonstraight links. Similarly, for *linkfound* = 0, the links on path P from stage r to stage $i-1$ consist of only straight links and positive nonstraight links; correspondingly, the links on the rerouting path from stage r to stage $i-1$ consist of only straight links and negative nonstraight links.

Proof of step 5

Proof of step 5 is illustrated in Figure 6 for $q = i$ and $r = i-k$. Because of Lemma A2.2, it suffices to show that, in each iteration of backtracking, a double nonstraight link blockage at stage q and an additional link blockage in $((j+2^l)\in S_l\ ,\ (j+2^{l+1})\in S_{l+1})$, for some l, $r+1 \le l \le q-1$, effectively close pivot $j\in S_{l+1}$ and make $(j+2^l)\in S_{l+1}$ unreachable. From Lemma A2.3, on path P every pivot $j\in S_{l+1}$, $r+1 \le l \le q-1$, is closed. On path $\hat{Q}$, if a link blockage also occurs in $((j+2^l)\in S_l\ ,\ (j+2^{l+1})\in S_{l+1})$, pivot $(j+2^{l+1})\in S_{l+1}$ becomes unreachable unless $j\in S_l$, the other pivot at stage l, is also connected to $(j+2^{l+1})\in S_{l+1}$. This would occur only if link $+2^{l+1}$ is a legitimate link at stage l, i.e. $2^{l+1} = 2^n \equiv 0 \ mod\ 2^n$ (a straight link). But $l \le q-1 \le (n-1)-1$ so that $l+1 \ne n$. $q \le n-1$ since q is the stage number at which a output link is blocked and stage $n-1$ the last stage that has output links. Because pivot $j\in S_{l+1}$ is closed and $(j+2^{l+1})\in S_{l+1}$ is unreachable, there exist no blockage-free paths. □

Proof of step 9

From Lemma A2.3, at the end of each iteration of backtracking, $(j+2^r)\in S_r$ and $j\in S_{r+1}$ are closed. After a new iteration of backtracking starts and step 7 is executed, $(j+2^r)\in S_r$ is relabeled as $j\in S_q$ and $j\in S_{r+1}$ is relabeled as $(j-2^q)\in S_{q+1}$. So the condition that $j\in S_q$ and $(j-2^q)\in S_{q+1}$ are both closed is a priori in the beginning of the new iteration. Since $(j-2^q)\in S_{q+1}$, one of the

pivots at stage $q+1$, is closed, any rerouting path must pass through $(j+2^q)\in S_{q+1}$, the other pivot at stage $q+1$. It is shown below that such a rerouting path does not exist if the nonstraight link at stage r found in backtracking is $+2^r$. The proof is illustrated in Figure 9. The current routing path is $((j-2^r)\in S_r\ ,\ j\in S_{r+1}\ ,\ \cdots\ ,\ j\in S_q\ ,\ (j-2^q)\in S_{q+1})$ and there exists a rerouting path $((j-2^r)\in S_r\ ,\ \cdots\ ,\ (j-2^{q-1})\in S_{q-1}\ ,\ (j-2^q)\in S_q\ ,\ (j-2^q)\in S_{q+1})$. Thus $(j-2^q)\in S_q$ and $j\in S_q$ are the two pivots at stage q. Since pivot $j\in S_q$ is closed, any rerouting path must pass through pivot $(j-2^q)\in S_q$. But $(j-2^q)\in S_q$ is not connected to $(j+2^q)\in S_{q+1}$ since link $+2^{q+1}$ is not a legitimate link at stage q, $0\le q<n-1$. Therefore, no paths that pass through $(j-2^q)\in S_q$ and $(j+2^q)\in S_{q+1}$ exist. Note that although further backtracking to a still lower-order stage is possible, as long as the nonstraight link at stage r found in backtracking is $+2^r$, the two pivots at stage q never change. That is, further backtracking will not result in a path that passes through $(j-2^q)\in S_q$ and $(j+2^q)\in S_{q+1}$. □

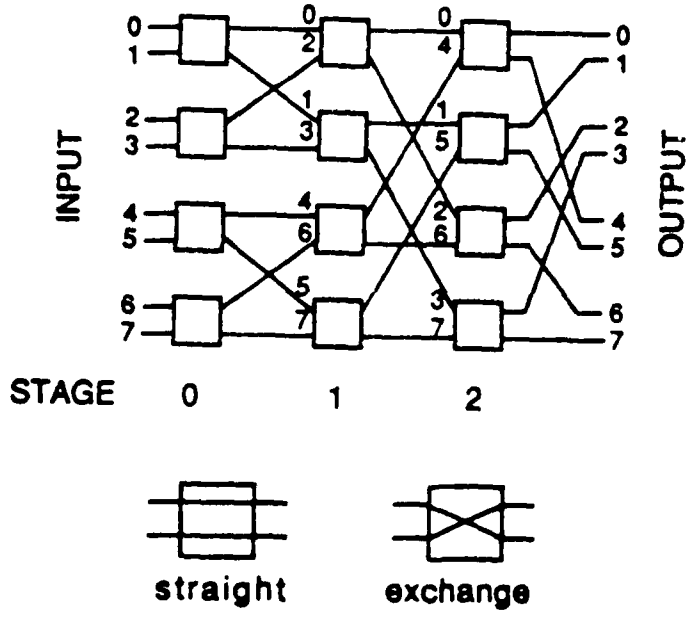


Figure 1. The Indirect Binary N-Cube (ICube) network for N=8 (according to the first graph model); two possible states for each box are shown (i.e. straight and exchange).

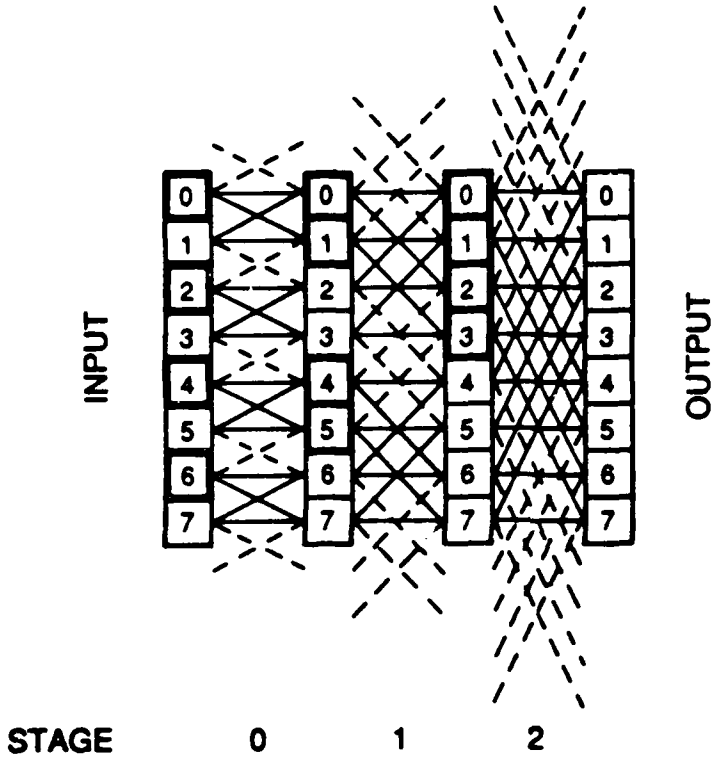


Figure 2. The IADM network for N=8 (according to the first graph model); $even_i$ and odd_i switches, $0 \leq i \leq 2$, are enclosed with bold and regular edges respectively. The solid edges (links) show the ICube subgraph.

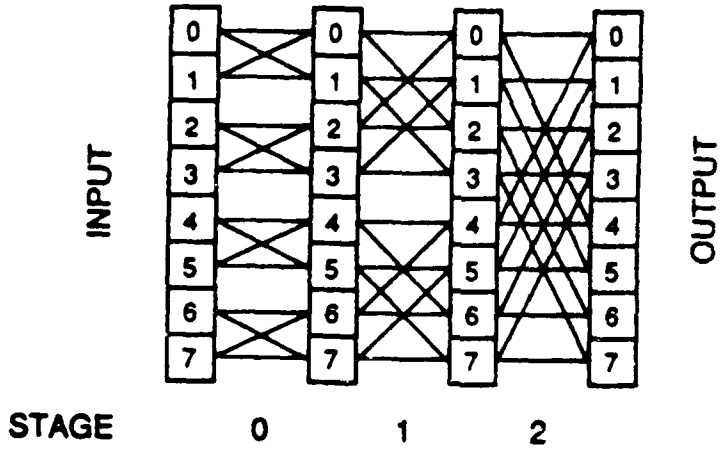


Figure 3. The Indirect Binary N-Cube (ICube) network for N=8 (according to the second graph model).

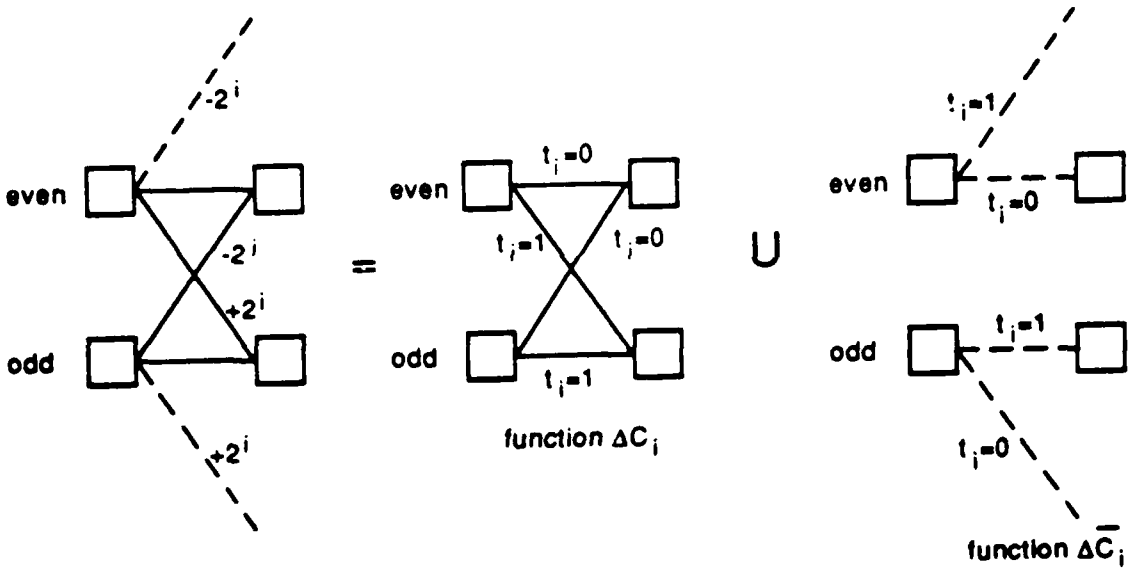


Figure 4. The connection links of stage i of the ICube network can be described by the function ΔC_i. The connection links of stage i of the IADM network can be described by the union of the functions ΔC_i and $\overline{\Delta C_i}$.

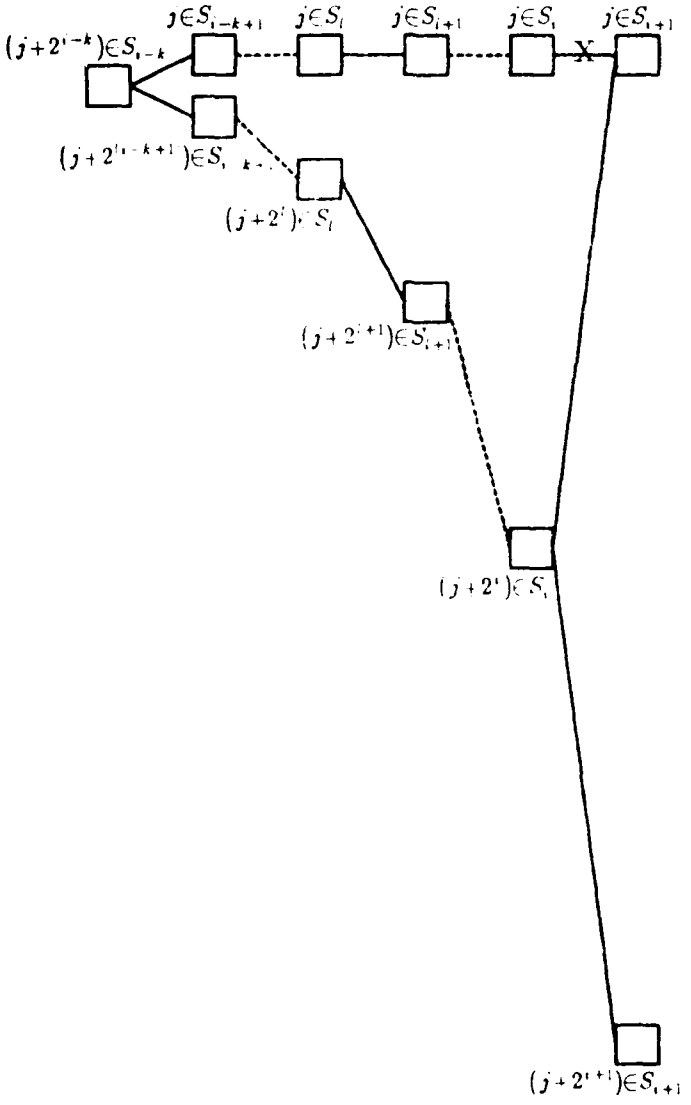


Figure 5. Rerouting for a straight link blockage in $(j \in S_i, j \in S_{i+1})$. Path $((j+2^{i-k}) \in S_{i-k}, j \in S_{i-k+1}, \cdots, j \in S_{i+1})$ is a segment of the original path; $((j+2^{i-k}) \in S_{i-k}, (j+2^{i-k+1}) \in S_{i-k+1}, \cdots, (j+2^i) \in S_i, j \in S_{i+1})$ and $((j+2^{i-k}) \in S_{i-k}, (j+2^{i-k+1}) \in S_{i-k+1}, \cdots, (j+2^i) \in S_i, (j+2^{i+1}) \in S_{i+1})$ are the rerouting paths for it.

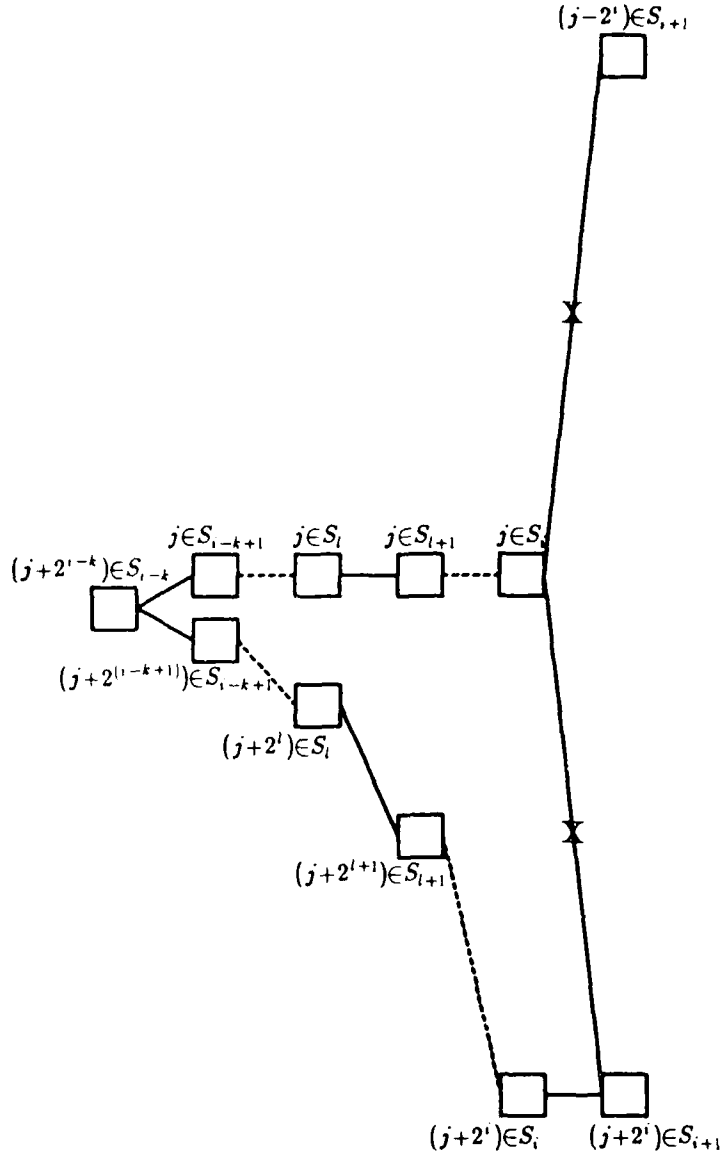


Figure 6. Rerouting for a double nonstraight links blockage in $(j\in S_i\ ,\ (j-2^i)\in S_{i+1})$ and $(j\in S_i\ ,\ (j+2^i)\in S_{i+1})$. Path $((j+2^{i-k})\in S_{i-k}\ ,\ (j+2^{i-k+1})\in S_{i-k+1}\ ,\ \cdots\ ,\ (j+2^i)\in S_i\ ,\ (j+2^i)\in S_{i+1})$ is a rerouting path for both paths $((j+2^{i-k})\in S_{i-k}\ ,\ j\in S_{i-k+1}\ ,\ \cdots\ ,\ j\in S_{i+1}\ ,\ (j-2^i)\in S_{i+1})$ and $((j+2^{i-k})\in S_{i-k}\ ,\ j\in S_{i-k+1}\ ,\ \cdots\ ,\ j\in S_{i+1}\ ,\ (j+2^i)\in S_{i+1})$.

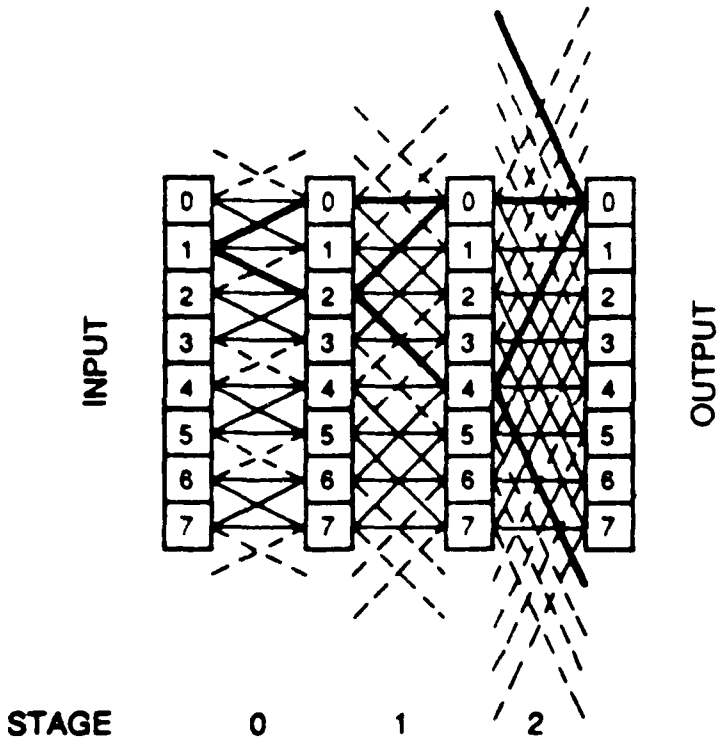


Figure 7. All routing paths from $1 \in S_0$ to $0 \in S_3$ in an IADM network of size N=8.

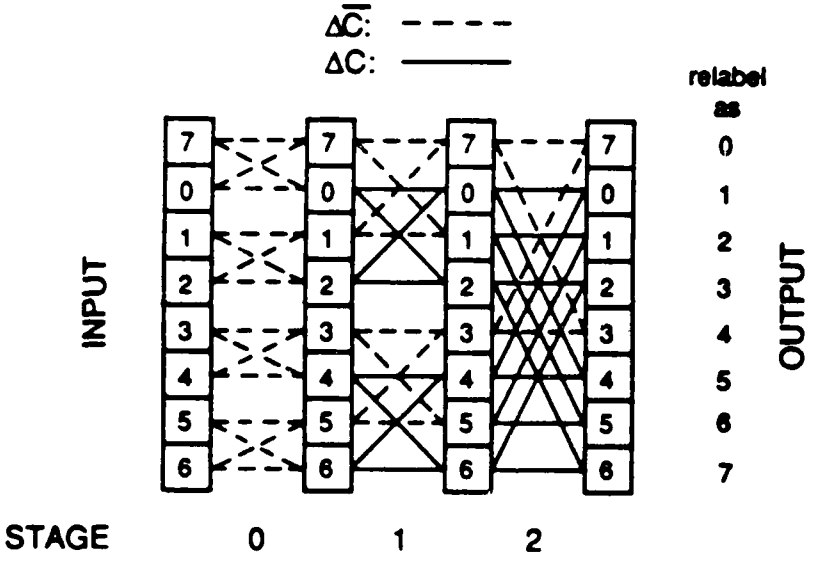


Figure 8. A cube subgraph generated by relabeling each switch j to $(j+1)$ *mod* 8 for an IADM network of size N=8.

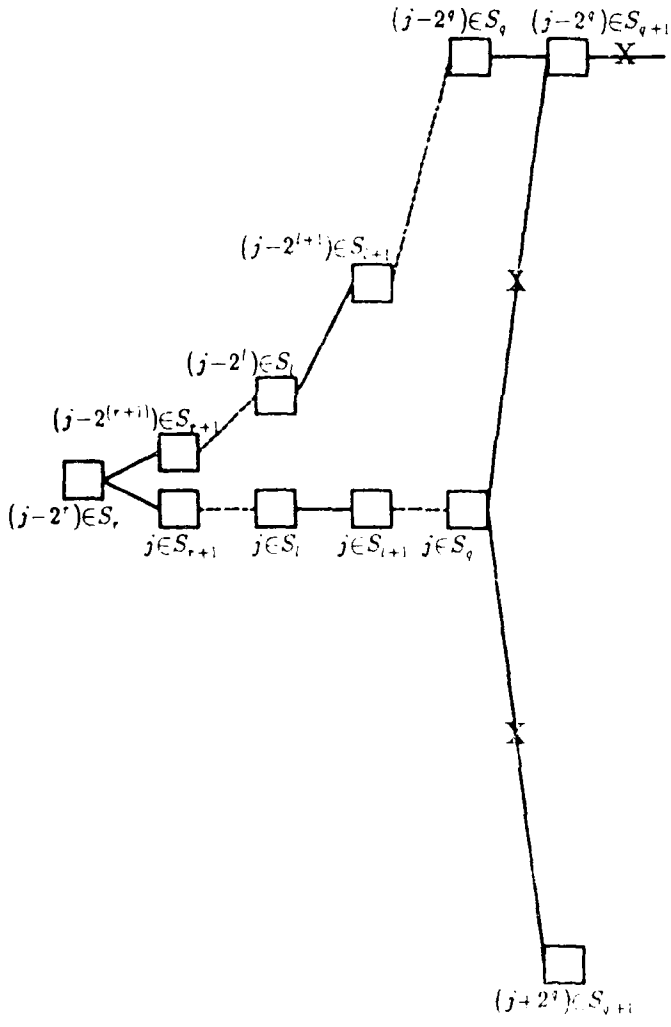


Figure 9. Rerouting for nonstraight output link blockages of switch $j\in S_q$. The original path is $((j-2^r)\in S_r, j\in S_{r+1}, \cdots, j\in S_q, (j-2^q)\in S_{q+1})$ and the nonstraight link found in backtracking is a positive link $((j-2^r)\in S_r, j\in S_{r+1})$.

REFERENCE NO. 9

Rau, D. And Fortes, J. A. B., "Partially Augmented Data Manipulator Networks: Minimal Designs and Fault Tolerance," Princeton Workshop on Algorithm, Architecture and Technology Issues for Models of Concurrent Computation, October 1987.

Partially Augmented Data Manipulator Networks: Minimal Designs and Fault Tolerance[1]

Darwen Rau and Jose A. B. Fortes

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

ABSTRACT

Augmented data manipulator networks are multistage interconnection networks which implement at each stage interconnection functions present in the single stage network known as PM2I network or barrel shifter. These multistage networks include the ADM (Augmented Data Manipulator) and IADM (Inverse Augmented Data Manipulator) networks, which have been extensively studied and proposed for use in multiprocessor systems. This paper derives new partially augmented networks based on the solution to the shortest path problem in the PM2I network. The new networks include: the HADM (Half Augmented Data Manipulator) and HIADM (Half Inverse Augmented Data Manipulator) networks which have half the number of stages of the ADM and IADM networks, the MADM (Minimum Augmented Data Manipulator) and the MIADM (Minimum Inverse Augmented Data Manipulator) networks which have the minimum link complexity required for one-to-one connections in a network of size N with $\log_t N$ stages of uniform switches, and the Extra Stage MADM and MIADM networks which are fault-tolerant versions of the MADM and MIADM networks that can tolerate at least three switch failures. The derivations of these networks are presented and their properties and advantages over other designs are analyzed.

[1]This research was supported in part by the National Science Foundation under Grant DCI-8419745 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contract No. 00014-85-k-0588

1. Introduction

Multistage interconnection networks are often designed by implementing at each stage interconnection functions characteristic of a single-stage network. This paper proposes new multistage networks which offer advantages over previously known designs based on the PM2I network [Sie77]. The new networks are derived from the solution to the shortest path problem in the PM2I network. Further analysis leads to the derivation of designs with minimal link complexity and fault-tolerance.

The plus-minus 2^i *(PM2I)* network [Sie77] is a single-stage network defined by the PM2I interconnection functions:

$$PM2I_{+i}(S) = (S + 2^i) \ mod \ N \quad 0 \leq i \leq n-1$$

$$PM2I_{-i}(S) = (S - 2^i) \ mod \ N \quad 0 \leq i \leq n-1$$

where $N = 2^n$ corresponds to the number of network nodes and S, $0 \leq S \leq N-1$, denotes a node address. Thus, in the PM2I network there exist links from a node S to nodes $PM2I_{+i}(S)$, $0 \leq i \leq n-1$, as well as links to nodes $PM2I_{-i}(S)$, $0 \leq i \leq n-1$. These links are referred to as the $+2^i$ *links* and -2^i *links*, respectively. A PM2I network of $N = 8$ nodes is illustrated in Figure 1.

The class of data manipulator networks, introduced in [Fen74], are constructed based on the PM2I functions. It includes, among others, the Augmented Data Manipulator (ADM) network [SiS78], the IADM network [McS82] and the Gamma network [PaR82][PaR84]. *The IADM network and the ADM network* differ only in that the input side of one of them corresponds to the output side of the other and vice versa. The Gamma and the IADM networks are topologically equivalent; however, they use switches of different types. Each 3×3 crossbar switch used in the Gamma network can connect simultaneously all three inputs to all three outputs whereas each switch used in the IADM network can connect only one of its three inputs to one or more of its three outputs.

The *ADM* network is composed of $n = \log N$ stages labeled from 0 to $n-1$ from the output side to the input side. Each stage consists of $3N$ connection links and N switches. The switches are labeled from 0 to $N-1$ from the top to the bottom. An extra column of switches is appended at the end of the last stage and is referred to as stage n. Each switch j of stage $i+1$ has three output links to switches $(j-2^i)$ *mod* N, j and $(j+2^i)$ *mod* N of stage i. The link joining j of stage $i+1$ and j of stage i is called a *straight link*, the link joining $(j-2^i)$ *mod* N of stage $i+1$ and j of stage i is a *plus* $(+2^i)$ *link* [McS82], and the link joining $(j+2^i)$ *mod* N of stage $i+1$ and j of stage i is a *minus* (-2^i) *link*. Each switch selects one of its input links and connects it to one or more output links. Figure 2 illustrates an ADM network of size N=8.

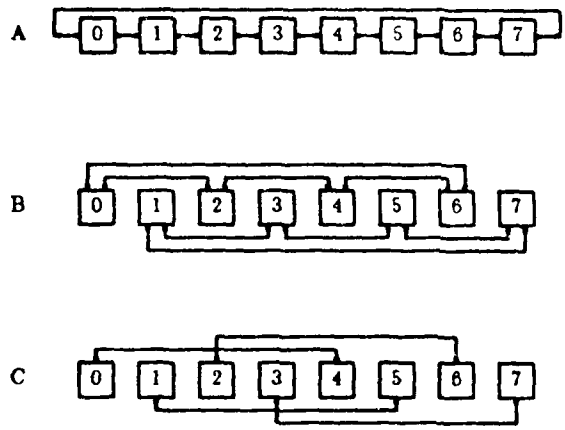


Figure 1. The PM2I network for $N = 8$. A:$PM2I_0$ B:$PM2I_1$ C:$PM2I_2$

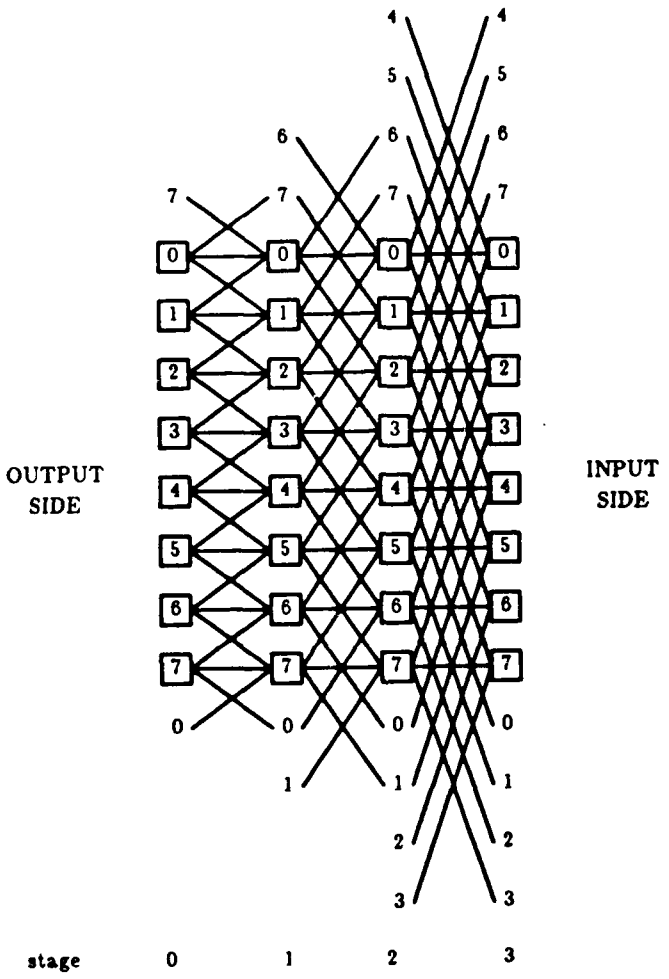


Figure 2. The ADM network for $N = 8$.

Because the only difference between the ADM and IADM networks is that their input and output sides are reversed, the stages of the IADM network are labeled from 0 to $n-1$ from the input side to the output side. Each switch j of stage i in the IADM network is connected to switches $(j-2^i)$ *mod* N, j and $(j+2^i)$ *mod* N of stage $i+1$. A plus link in the IADM network from switch j of stage i is connected to switch $j+2^i$ of stage $i+1$ is the same link as the minus link in the ADM network from switch $j+2^i$ of stage $i+1$ to switch j of stage i. Similar relationship applies to a minus link in the IADM network and a plus link in the ADM network. Due to the reversal of the input and output sides of the ADM and IADM network, stage i of the ADM network corresponds to the switches of stage i of the IADM network and the links of stage $i-1$ of the IADM network.

The results of this paper are based on the study of shortest path problem in the PM2I network. The solution to the shortest path problem for the PM2I network is derived from an algorithm [PaR82] that generates routing tags for the Gamma network. Because the IADM and Gamma network are topologically equivalent and the ADM and IADM networks differ only in their input and output sides, the results in this paper apply to all of these networks. However, the main interest of this paper is the study of the ADM network and the discussions are centered on the properties of the ADM network.

Given a string of n digits, $t = t_0 t_1 \cdots t_{n-1}$, the notation $t_{p/q}$ denotes the digits of t starting at t_p and ending at t_q. Throughout this paper, j and $j+a$ (where a is some constant) represent labels of switches. Also modulo N arithmetic is assumed, e.g. $j+a$ implies $(j+a)$ *mod* N. The notation j^i is used to indicate that a switch j belongs to stage i and (j^{i+1}, j^i) is used to represent a link joining j^{i+1} and j^i. A sequence of switches of contiguous stages $(j^{i+k}, j^{i+k-1}, \cdots, j^i)$ is used to represent a path from j^{i+k} to j^i.

Section 2 of the paper considers the formulation and solution of the shortest path problem for the PM2I network. In Section 3 these results are used to derive new networks that require less hardware complexity and transmission delay than other known augmented data manipulator networks. These new networks are called ***partially augmented data manipulator networks***. Details of routing schemes for these networks are also discussed in Section 3. Fault-tolerant topologies are proposed in Section 4 by adding an extra stage to these networks, with the result that four disjoint paths exist between any source and any destination in the networks. Section 5 concludes the paper.

2. Shortest Path Problem in the PM2I Network

Given a source node S and a destination node D in the PM2I network, the shortest path problem is to find a path from S to D which contains a minimal number of links. When circuit switching is used for communication between nodes, delays are identical for any link and transmission delay is directly proportional to the number of links on a path. Thus, the shortest path is also the one for which transmission delay is minimum.

Given a source node S and destination node D in the PM2I network, define *distance* Δ to be $(D-S)$ *mod* N; thus the range of Δ is $0 \leq \Delta \leq (N-1)$. Routing from a source S to a destination D in the PM2I network can be characterized by the *combination tag* $t_{0/2n-1} = t_0 t_1 \cdots t_{2n-1}$ such that

$$\Delta = \left(\sum_{i=0}^{n-1} t_i 2^i + \sum_{i=n}^{2n-1} t_i(-2^{i-n})\right) \ mod\ 2^n \qquad (1)$$

where Δ is the distance from the source S to the destination D and t_i's are non-negative integers. A positive value of t_i indicates that link $+2^i$, for $0 \leq i \leq n-1$, or link -2^{i-n}, for $n \leq i \leq 2n-1$, is used in the routing path whereas $t_i = 0$ indicates that the link is not used. A combination tag, as suggested by its name, specifies a combination of PM2I links that can be used to cover the distance between the source and the destination. However, the combination tag $t_{0/n-1}$ does not specify the sequence in which the links are used. Several distinct paths can be derived from a combination tag and all these paths contains the same number of links. Since the combination tag depends only on the distance Δ, it is often identified as *a combination tag of distance* Δ. A shortest path is specified by a combination tag for which the number of links $\sum_{i=0}^{2n-1} t_i$ is minimum and the problem of finding such a tag - called *minimum weight combination tag* - can be stated as follows:

Problem (P) Find $t^* = t^*_{0/n-1}$ such that

$$H_\Delta = \min \sum_{i=0}^{2n-1} t_i = \sum_{i=0}^{2n-1} t_i^*$$

$$\text{subject to} \quad \Delta = \left(\sum_{i=0}^{n-1} t_i 2^i + \sum_{i=n}^{2n-1} t_i(-2^{i-n})\right) \ mod\ 2^n$$

$$0 \leq t_i \quad for\ 0 \leq i \leq 2n-1$$

$$0 \leq \Delta \leq 2^n - 1$$

A feasible solution to this problem corresponds to a combination tag, and an optimal solution to it corresponds to a minimum weight combination tag. For convenience of discussion, the terms (i) a feasible solution and a combination tag, and (ii) an optimal solution and a minimum weight combination tag are

used interchangeably.

The next two lemmas reduce the size of the set of feasible solutions.

Lemma 2.1 If t^* is the optimal solution to (P), then $t_i \in \{0,1\}$, $0 \le i \le 2n-1$.
Proof: The proof is by contradiction. Assume that the optimal solution t^* contains $t_k^* \ge 2$, for some k, $0 \le k \le n-2$ or $n \le k \le 2n-2$. Then there exist alternate paths that, compared with paths defined by t^*, reduce traversal through link $+2^k$ (or -2^{k-n}) twice and increase its traversal through link $+2^{k+1}$ (or -2^{k+1-n}) once; i.e.

$$t_k^* 2^k + t_{k+1}^* 2^{k+1} = (t_k^* - 2)2^k + (t_{k+1}^* + 1)2^{k+1}$$

Comparing with the total delay of the path defined by t^*, the total delay of the alternate paths is reduced by one, which is contradictory to the hypothesis that t^* minimizes the routing delay. If $k = n-1$ (or $2n-1$) such that $t_{n-1}^* \ge 2$ (or $t_{2n-1}^* \ge 2$), a carry is generated in the highest order digit and t_{n-1}^* is discounted by two, denoted $t_{n-1}' = t_{n-1}^* - 2$. The carry vanishes due to ($mod\ 2^n$) operation and the total delay of the alternate paths is reduced by two; again a contradiction results. □

Lemma 2.2 If t^* is the optimal solution to (P), then $t_i^* \cdot t_{n+i}^* = 0$, $0 \le i \le n-1$; i.e. the shortest path between any source and any destination in the PM2I network cannot contain both link $+2^i$ and link -2^i for any i, $0 \le i \le n-1$.
Proof: The proof is by contradiction. Suppose the opposite is true. From Lemma 2.1, a digit of the tag representing a shortest path, can only be 0 or 1; by assumption of having both $+2^i$ and -2^i links on the routing path, $t_i^* = t_{n+i}^* = 1$. The effects of $+2^i$ and -2^i cancel each other. Thus the values for t_i^* and t_{n+i}^* can be substituted by 0 and still satisfy the equality constraint in (P) (also equation (1)). The routing delay is thus reduced by two. A contradiction results. □

From Lemma 2.2, either t_i^* or t_{n+i}^* is zero, $0 \le i \le n-1$, so that the two sums $\sum_{i=0}^{n-1} t_i 2^i$ and $\sum_{i=n}^{2n-1} t_i(-2^{i-n})$ in equation (1) can be combined to form $\sum_{i=0}^{n-1} t_i 2^i$, with the extension of the values for t_i to include negative integers. The result in Lemma 2.1 confines the values for each t_i of a tag representing a shortest path to be 0 and 1. Together with the necessary extension to include negative integers, the possible values for t_i of an optimal solution are -1, 0 or 1. Thus, the problem of finding a minimum weight combination tag can be reformulated as follows:

Problem (P) Find $t^* = t^*_{0/n-1}$ such that

$$H_\Delta = \min \sum_{i=0}^{n-1} |t_i| = \sum_{i=0}^{n-1} |t_i^*|$$

$$\text{subject to} \quad \Delta = \left(\sum_{i=0}^{n-1} t_i 2^i\right) \ mod\ 2^n$$

$$t_i \in \{-1,0,1\} \qquad \text{for } 0 \le i \le n-1$$

$$0 \le \Delta \le 2^n - 1$$

A branch-and-bound approach is used to find the optimal solution for (P), which is also a minimum weight combination tag. This approach is based on an algorithm proposed in [PaR82] that can find all signed-digit representations for the distance between any source and any destination in the Gamma network. Each signed-digit representation corresponds to a routing tag for the source/destination pair. Moreover, since the IADM network and the Gamma network are topologically equivalent, the routing tags generated by the algorithm are also valid routing tags for the IADM network. The Gamma network is constructed based on the PM2I functions and a routing tag uniquely specifies a path in it. In particular, each stage is composed of 2^n switches, and at each stage i, $0 \le i \le n-1$, each switch is connected to three output links $+2^i$, -2^i and straight link, and only one of them is on the routing path; in addition, the path in the Gamma network traverses a distance of $((D-S) \ mod\ 2^n)$ from S to D. These corresponds to the constraints in (P): $\Delta = \left(\sum_{i=0}^{n-1} t_i 2^i\right) \ mod\ 2^n$, $t_i \in \{-1, 0, 1\}$ and $-(2^n-1) \le \Delta \le 2^n-1$. Thus a routing tag that specifies a path from S to D in the Gamma network is also a feasible solution to (P). Note that $t_i = 0$ indicates that a straight link is used at stage i for routing in the Gamma network.

A routing tag for the Gamma network can be converted to a combination tag for the PM2I network: if the i-th bit of the routing tag is 1, $t_i = 1$, if it is $\bar{1}$, $t_{i+n} = 1$ (hereafter the signed-digit representation $\bar{1}$ [Avi61] is used to represent -1), and if it is 0, $t_i = t_{i+n} = 0$. A combination tag satisfying conditions (a) $t_i \in \{-1,0,1\}$ and (b) $t_i \cdot t_{i+1} = 0$ can also be converted to a routing tag for the Gamma network: if $t_i = 1$, the i-th bit of the routing tag is 1, if $t_{i+n} = 1$, the i-th bit of the routing tag is $\bar{1}$, and if $t_i = t_{i+n} = 0$, the i-th bit of the routing tag is 0. The optimal solution to (P) certainly satisfies conditions (a) and (b) and is also a minimum weight tag. Because we are only interested in the shortest path (which can be characterised by a minimum weight combination tag) in the PM2I network, given the one-to-one correspondence between a minimum weight routing tag and a minimum weight combination tag, they are used interchangeably. The algorithm in [PaR82] is stated as follows.

Algorithm ALL - TAGS ($\Delta, t_{n-1} \ldots t_0$)
$\Delta_0 = \Delta$
for $i = 0$ to $n - 1$ do

if Δ_i is even then $t_i = 0$, $\Delta_{i+1} = \dfrac{\Delta_i}{2}$

else
$t_i = 1$, $\Delta_{i+1} = \dfrac{\Delta_i - 1}{2}$
$t_i = \bar{1}$, $\Delta_{i+1} = \dfrac{\Delta_i + 1}{2}$

endif
enddo

In the algorithm, t_i is uniquely determined ($=0$) if Δ_i is even whereas freedom exists in choosing the value for t_i (1 or $\bar{1}$) if Δ_i is odd. An example is shown that generates all tags for routing from $S^1 = 1$ to $D^4 = 4$ in the IADM network of size $N = 8$. In this case, $\Delta = 3 = -5 \ mod\ 8$.

$[\Delta_0]$	t_0	$[\Delta_1]$	t_1	$[\Delta_2]$	t_2	$[\Delta]$
[3]	1	[1]	1	[0]	0	$(=3)$
[3]	1	[1]	$\bar{1}$	[1]	1	$(=3)$
[3]	1	[1]	$\bar{1}$	[1]	$\bar{1}$	$(=-5\ mod\ 8)$
[3]	$\bar{1}$	[2]	0	[1]	1	$(=3)$
[3]	$\bar{1}$	[2]	0	[1]	$\bar{1}$	$(=-5\ mod\ 8)$

As mentioned previously the focus of this paper is the ADM network, the tags generated by algorithm ALL-TAGS can also be used in the ADM network because the ADM and IADM network differ only in that the input side of one of them corresponds to the output side of the other and vice versa. In the IADM network routing is from a switch of the lowest order stage to the highest order stage while routing in the ADM network is just the opposite. Therefore, the lowest order digit of the tag is first examined by a switch for routing in the IADM network and the highest order digit is first examined for routing in the ADM network. At stage i, $0 \leq i \leq n-1$, of both networks, if t_i is 0, straight link is used for routing; if it is 1, link $+2^i$ is used; if it is $\bar{1}$, link -2^i is used. In particular, routing from S^1 to D^n in the IADM network is equivalent to routing from D^n to S^1 in the ADM network. Let $t_{0/n-1}$ be the tag for the routing from S^0 to D^n in the IADM network, it can be readily verified that tag $t'_{0/n-1}$, where $t'_i = -t_i$, $0 \leq i \leq n-1$ can be used for routing from D^n to S^0 in the ADM network, The two tags $t_{0/n-1}$ and $t'_{0/n-1}$ represent the same path, with different interpretations in the ADM and IADM networks. For example, the tags 110, $1\bar{1}1$, $1\bar{1}\bar{1}$, $\bar{1}01$ and $\bar{1}0\bar{1}$ for the IADM network in the above table can be

converted to $\bar{1}\bar{1}0$, $\bar{1}1\bar{1}$, $\bar{1}11$, $10\bar{1}$ and 101 for the ADM network, respectively. Figure 3 illustrates the routing from $D = 4$ to $S = 1$ in the ADM network using these tags.

The possibility of having two values, 1 and $\bar{1}$, for t_i, if Δ_i is odd can be used to find the optimal solution to (P). It is shown below how to choose the value for t_i so that t_{i+1} can be pre-determined as desired.

Lemma 2.3 In the process of generating tags in algorithm ALL-TAGS, if Δ_i is odd, it is always possible to make $t_{i+1} = 0$ by properly choosing the value for t_i.

Proof: Since $\frac{\Delta_i+1}{2}$ and $\frac{\Delta_i-1}{2}$ differ exactly by one, one of them is even and the other is odd. Suppose that, without loss of generality, $\frac{\Delta_i+1}{2}$ is even. Then t_i can be chosen to be -1 so that $\Delta_{i+1} = \frac{\Delta_i+1}{2}$, which makes $t_{i+1} = 0$. □

For example, one of the paths illustrated in Figure 3 is represented by a tag $t_{0/2} = 10\bar{1}$ of distance $\Delta = \Delta_0 = 3$; in this case t_0 is chosen to be 1 so that $t_1 = 0$.

Theorem 2.4 There exists an optimal solution t^* to (P) which has no adjacent nonzero digits; i.e., $t^*_{i+1} \cdot t^*_i = 0$ for $0 \le i \le n-2$. If $t^*_{n-1} = 0$ then t^* is the unique optimal solution with no adjacent nonzero digits; otherwise, there exists another optimal solution t' with no adjacent nonzero digits, where $t'_i = t^*_i$, $0 \le i \le n-2$, and $t'_{n-1} = -t^*_{n-1}$.

Proof: The proof consists of three parts. Part (i) finds a minimum weight tag, part (ii) proves the uniqueness of the minimum weight tag when $t^*_{n-1} = 0$, and part (iii) finds another minimum weight tag if $t^*_{n-1} \neq 0$.

(i) An algorithm which results from modifying algorithm ALL-TAGS is first given to construct a minimum weight tag; it is followed by a proof of its optimality.

Algorithm SHORTEST-PATH $(\Delta, t^*_{0/n-1})$
$\Delta_0 = \Delta$
for $i = 0$ to $n-1$ do
  if Δ_i is even then $t^*_i = 0$, $\Delta_{i+1} = \frac{\Delta_i}{2}$
  else if $\frac{\Delta_i-1}{2}$ is even then $t^*_i = 1$, $\Delta_{i+1} = \frac{\Delta_i-1}{2}$
    else $t^*_i = \bar{1}$, $\Delta_{i+1} = \frac{\Delta_i+1}{2}$
    endif
  endif
enddo

Since the set of tags generated by algorithm SHORTEST-PATH is a subset of those generated by algorithm ALL-TAGS, algorithm SHORTEST-PATH

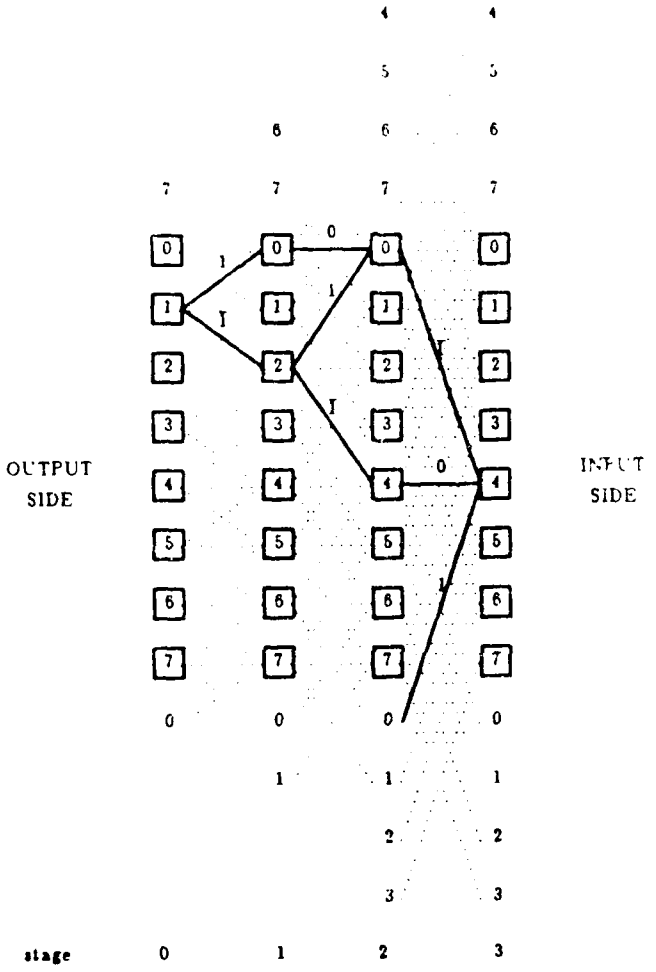


Figure 3. Routing from 4 to 1 in the ADM network for $N = 8$. The solid lines are the links on the routing paths and the dotted lines are other links of the ADM network. Labels on the links are digits of the routing tags.

correctly generates a tag of distance Δ_i. It remains to show that the tag has minimum weight. The strategy used in the algorithm SHORTEST-PATH is to generate a zero digit whenever possible (for Δ_i is odd, t_i is chosen to be such that Δ_{i+1} is even, which makes $t_{i+1} = 0$). To see why this is a good strategy, let i be the smallest index such that Δ_i is odd, and let t' and t be the solutions found by applying this strategy and and by not complying with this strategy, respectively. Assume that, without loss of generality, $\frac{\Delta_i - 1}{2}$ is even. It is shown that there are four possible cases and the terminating conditions for each case can be continued by applying the discussion for one of the four cases recursively.

Case 1

The table below illustrates the discussion for case 1 based on this assumption.

case 1	$[\Delta_i]$	t_i	$[\Delta_{i+1}]$	t_{i+1}	$[\Delta_{i+2}]$
t'	$[\Delta_i]$	1	$[\frac{\Delta_i - 1}{2}]$	0	$[\frac{\Delta_i - 1}{4}]$
t	$[\Delta_i]$	$\bar{1}$	$[\frac{\Delta_i + 1}{2}]$	1	$[\frac{\Delta_i - 1}{4}]$

Since $\frac{\Delta_i - 1}{2}$ ($= \Delta_{i+1}$ for t', denoted $\Delta_{i+1}(t')$) is assumed to be even, $t'_{i+1} = 0$. Because $\Delta_{i+1}(t) = \frac{\Delta_i + 1}{2}$ is odd, there are two possible values, 1 and $\bar{1}$, for t_{i+1}. If $t_{i+1}=1$, $\Delta_{i+2}(t) = \Delta_{i+2}(t')$. The discussion for case 1 terminates here.

Case 2

The alternative is that $t_{i+1}=\bar{1}$, which is illustrated in cases 2, 3 and 4 in the tables below. In case 2, $\frac{\Delta_i - 1}{4}$ is assumed to be even.

case 2	$[\Delta_i]$	t_i	$[\Delta_{i+1}]$	t_{i+1}	$[\Delta_{i+2}]$
t'	$[\Delta_i]$	1	$[\frac{\Delta_i - 1}{2}]$	0	$[\frac{\Delta_i - 1}{4}]$
t	$[\Delta_i]$	$\bar{1}$	$[\frac{\Delta_i + 1}{2}]$	$\bar{1}$	$[\frac{\Delta_i + 3}{4}]$

Case 2 terminates here with $\Delta_{i+2}(t')$ being even and $\Delta_{i+2}(t)$ being odd.

Case 3

In cases 3 and 4, $\frac{\Delta_i - 1}{4}$ is assumed to be odd. Case 3 is illustrated in the following table with the assumption that $\frac{\Delta_i + 3}{8}$ is even.

case 3	Δ_i	t_i	Δ_{i+1}	t_{i+1}	Δ_{i+2}	t_{i+2}	Δ_{i+3}
t'	Δ	1	$\frac{\Delta-1}{2}$	0	$\frac{\Delta-1}{4}$	$\bar{1}$	$\frac{\Delta+3}{8}$
t	Δ	$\bar{1}$	$\frac{\Delta+1}{2}$	$\bar{1}$	$\frac{\Delta+3}{4}$	0	$\frac{\Delta+3}{8}$

Since $\Delta_{i+2}(t') = \frac{\Delta-1}{4}$ is odd, $t'_{i+2} = 1$ or $\bar{1}$, and $t_{i+2} = 0$. Since $\frac{\Delta+3}{8}$ is even, the algorithm chooses $t'_{i+2} = 1$, and $\Delta_{i+3}(t) = \Delta_{i+3}(t')$. The discussion for case 3 terminates here.

Case 4

Case 4 for which $\frac{\Delta-5}{8}$ is even is illustrated in the table below.

case 4	Δ_i	t_i	Δ_{i+1}	t_{i+1}	Δ_{i+2}	t_{i+2}	Δ_{i+3}
t'	Δ	1	$\frac{\Delta-1}{2}$	0	$\frac{\Delta-1}{4}$	1	$\frac{\Delta-5}{8}$
t	Δ	$\bar{1}$	$\frac{\Delta+1}{2}$	$\bar{1}$	$\frac{\Delta+3}{4}$	0	$\frac{\Delta+3}{8}$

If $\frac{\Delta-5}{8}$ is even, the algorithm chooses $t'_{i+2}=1$. In this case, $\Delta_{i+3}(t')=\frac{\Delta-5}{8}$ and $\Delta_{i+3}(t)=\frac{\Delta+3}{8}$. The discussion for case 4 terminates here.

To conclude, cases 1 and 3 have the terminating conditions that $\Delta_{i+2}(t') = \Delta_{i+2}(t)$ and $\Delta_{i+3}(t') = \Delta_{i+3}(t)$, respectively. The discussion for $\Delta_i(t') = \Delta_i(t)$, which is the condition where the discussion for all cases begin, can be applied again to these terminating conditions. In cases 2 and 4, the terminating conditions are that $\Delta_{i+2}(t')$ is even and $\Delta_{i+2}(t)$ is odd, and $\Delta_{i+3}(t')$ is even and $\Delta_{i+3}(t)$ is odd, respectively. The discussions done for each case for iteration $i+1$ when $\Delta_{i+1}(t')$ is even and $\Delta_{i+1}(t)$ is odd can be applied again to them. Let $|t_{i/}|$ denote the number of nonzero bits of $t_{i/}$. In case 1, $|t'_{i/i+1}| = 1$ and $|t_{i/i+1}| = 2$; in case 2, $|t'_{i/i+2}| = 2$ and $|t_{i/i+2}| = 2$; in case 3, $|t'_{i/i+1}| = 1$ and $|t_{i/i+1}| = 2$; in case 4, $|t'_{i/i+2}| = 2$ and $|t_{i/i+2}| = 2$. Thus all possible cases are exhausted and no t yields a tag of smaller weight than t'.

(ii) Next the proof of uniqueness for the tag generated by algorithm SHORTEST-PATH is shown; the proof is by contradiction. Suppose there exists another tag $t_{i/i-1}$ that also has no adjacent nonzero digits. Let i be the lowest index such that $t_i \neq t'_i$; thus $t_{i/i-1} = t'_{i/i-1}$ so that $\Delta_i(t) = \Delta_i(t')$. There are three possible cases, (a), (b) and (c), for $t_i \neq t'_i$. (a) $t_i = \bar{1}$ and $t'_i = 1$ (or vice versa); then $t_{i+1} = 0 = t'_{i+1}$, since $t_i \cdot t_{i+1} = 0$ and $t'_i \cdot t'_{i+1} = 0$. But this is impossible because $\Delta_i(t) = \Delta_i(t')$ is odd so that only either $t_{i+1} = 0$ or $t'_{i+1} = 0$ (Lemma 2.3). A contradiction results. (b) $t_i = 1$ and $t'_i = 0$ (or vice versa).

Then $\Delta_i(t)$ is odd and $\Delta_i(t^*)$ is even. But this is impossible because $\Delta_i(t) = \Delta_i(t^*)$. A contradiction results. (c) $t_i = \bar{1}$ and $t_i^* = 0$ (or vice versa). The discussion is exactly the same as case (b).

(iii) Existence of the other optimal solution is shown for $t_{n-1}^* = 1$. The case that $t_{n-1}^* = \bar{1}$ can be treated analogously. If $t_{n-1}^* = 1$, then $\Delta = (\sum_{i=0}^{n-2} t_i^* 2^i + 2^{n-1}) \bmod 2^n = (\sum_{i=0}^{n-2} t_i^* 2^i + 2^{n-1} - 2^n) \bmod 2^n = (\sum_{i=0}^{n-2} t_i^* 2^i - 2^{n-1}) \bmod 2^n$; so t_{n-1}^* can also be $\bar{1}$ and the rest of digits remain unchanged. □

Actually the proof of Theorem 2.4 has a much stronger implication regarding optimality of a tag than just verifying existence of a minimum weight tag that has no adjacent nonzero digits. It is stated as Corollary 2.5.

Corollary 2.5 A feasible solution to (P) is optimal if it has no adjacent nonzero digits.

Proof: From the process of generating each digit in algorithm SHORTEST-PATH, the feasible solutions with no adjacent nonzero digits are either unique or different only at t_{n-1} (1 or $\bar{1}$). There exists an optimal solution to (P) that has no adjacent nonzero digits. So the feasible solution with no adjacent nonzero digits must be also an optimal solution. □

Corollary 2.5 only guarantees optimality of a tag that has no adjacent nonzero digits; a tag with adjacent nonzero digits may as well be a minimum weight tag. For instance, for $n = 4$, $\Delta = -6$, the tag of distance Δ can be $t_{0/6} = 0\bar{1}\bar{1}0$ or $t_{0/6} = 010\bar{1}$; both tags have a minimum weight of two.

Corollary 2.6[2] The maximum number of links on the shortest path in the PM2I network from any source to any destination is $\lceil n/2 \rceil$, i.e.

$$\max_{0 \le \Delta \le (N-1)} H_\Delta = \lceil n/2 \rceil$$

Proof: From Theorem 2.4, there exists a minimum weight tag with no adjacent nonzero digits for every distance Δ. The maximum number of nonzero digits of such a minimum weight tag is $\lceil n/2 \rceil$; i.e. the tag consists of alternating 1 and 0 digits. □

Algorithm SHORTEST-PATH is capable of finding a minimum weight routing tag for the ADM network, which can be converted to a combination tag for the PM2I network, and also deduces that the number of hops is bounded above by $\lceil n/2 \rceil$. This knowledge can be further used to investigate properties

[2] An equivalent result is reported in [HwB84]. We were unable to identify the original reference which first reported this result.

of the ADM network.

3. Construction of Half Augmented Data Manipulator Networks

Corollary 2.6 indicates that the shortest path between any two nodes in PM2I network uses at most $\lceil n/2 \rceil$ links, which implies that $\lceil n/2 \rceil$ is the least number of stages needed in a multistage network based on PM2I functions, where any source can be connected to any destination in one pass. Furthermore, from Theorem 2.4 it is possible to infer how such a network can be constructed. For convenience of discussion, assume n to be even hereafter. The path in the ADM network defined by the routing tag that has no adjacent nonzero digits includes only one of the links -2^{2k+1}, -2^{2k}, $+2^{2k}$, $+2^{2k+1}$ and straight link, for every stage k, $0 \le k \le (n/2)-1$. This implies that the links of two adjacent stages $2k$ and $2k+1$ in the ADM network can be coalesced into one stage and thus the total number of stages is reduced to $n/2$. The network is called *Half ADM (HADM) network*. The HADM network consists of $n/2$ stages ordered from 0 to $(n/2)-1$ from the output side to the input side. An extra column of switches is appended in the input side and is referred to as stage $n/2$. A source is a switch at stage $n/2$ and a destination is a switch at stage 0. Switch j of stage $k+1$ has five output links to switches of stage k: $(j+2^{2k+1})$, $(j+2^{2k})$, j, $(j-2^{2k})$ and $(j-2^{2k+1})$. An HADM network of size $N = 16$ is shown in Figure 4.

The tag generated by algorithm SHORTEST-PATH can be used as a routing tag in the HADM network. Close examination of the topology of the HADM network reveals that there exists latitude in using tags other than the ones with no adjacent nonzero digits to control routing in the HADM network; i.e. two adjacent digits of a routing tag can be both nonzero. Since, for a given source/destination pair, only one of the links -2^{2k+1}, -2^{2k}, $+2^{2k}$, $+2^{2k+1}$ and straight link is used for routing in the HADM network, as long as the tag satisfies the constraint that $t_{2k} \cdot t_{2k+1} = 0$ for $0 \le k \le (n/2)-1$, it is a valid routing tag in the HADM network. There are five possible combinations for such a pair of digits $t_{2k}t_{2k+1}$: $\bar{1}0$, 10, 00, 01 and $0\bar{1}$. If $t_{2k}t_{2k+1} = \bar{1}0$, link -2^{2k} is used; if $t_{2k}t_{2k+1} = 10$, link $+2^{2k}$ is used; if $t_{2k}t_{2k+1} = 00$, straight link is used; if $t_{2k}t_{2k+1} = 01$, link $+2^{2k+1}$ is used; if $t_{2k}t_{2k+1} = 0\bar{1}$, link -2^{2k+1} is used. The routing tags representing the same distance Δ in the HADM network are called the *equivalent routing tags*. The multitude of equivalent routing tags suggests that there may exist multiple paths for some source/destination pairs. If a routing tag has no equivalent routing tags, it is unique, and only one routing path exists for the source/destination pair.

Recall that algorithm SHORTEST-PATH always generates a zero digit whenever possible. If Δ_i is even, t_i is uniquely confined to be 0; if Δ_i is odd (for which t_i can be 1 or $\bar{1}$), then it chooses the value for t_i such that $t_{i+1} = 0$. This

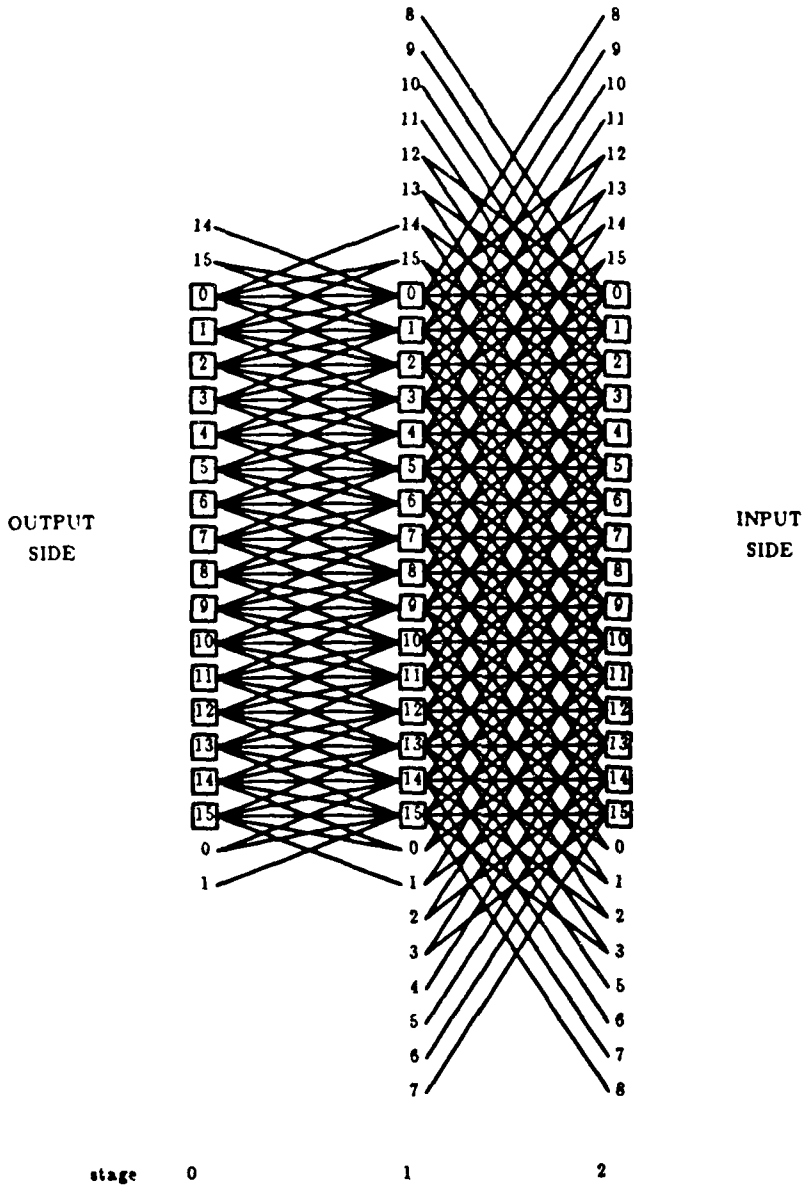


Figure 4. The HADM network for $N = 16$.

constraint can be relaxed for generating equivalent routing tags for the HADM network. For $t_{2k} = 0$ and Δ_{2k+1} odd, two subsets of equivalent tags can be generated by choosing 1 for t_{2k+1} for one of them and by choosing $\bar{1}$ for t_{2k+1} for the other. That is, if $t_{2k}t_{2k+1} = 0\bar{1}$ or 01, both 1 or $\bar{1}$ can be considered for t_{2k+1} to form equivalent routing tags, since it's always possible to make t_{2k+2} zero by properly choosing a value for t_{2k+1} *(Lemma 2.3) and satisfy the constraint* $t_{2k}\cdot t_{2k+1} = 0$ and $t_{2k+2}\cdot t_{2k+3} = 0$. For example, there are two paths from $S = 3$ to $D = 13$ in an HADM network of size $N = 16$, which are specified by the tags $t_{0/3} = 0\bar{1}\bar{1}0$ ($\Delta = -6$) and $t_{0/3} = 0101$ ($\Delta = 10$), respectively. In this example, t_0t_1 can be $0\bar{1}$ or 01; particularly the tag $t_{0/3} = 0\bar{1}\bar{1}0$ is obtained by choosing $t_2 = \bar{1}$ so that $t_3 = 0$.

Similar to the relationship between the ADM and IADM networks, the *Half IADM (HIADM) network* has the same topology as the HADM network with the input and output sides exchanged. A tag $t_{0/n-1}$ for routing from S^0 to D in the HADM network can be conveniently converted to $t'_{0/n-1}$ where $t'_i = -t_i$, $0 \le i \le n-1$, for routing from D^0 to $S^{n/-}$ in the Half IADM network. Note that tag $t'_{0/n-1}$ also satisfies the constraint $t_{2k}\cdot t_{2k+1} = 0$, $0 \le k \le (n/2)-1$.

It was shown that for some source and some destination in the ADM network, there exists only a path between them; so does in the HADM network. For example, routing for a distance $\Delta = 0$ in a HADM network of size $N = 16$ has a unique tag $t_{0/3} = 0000$, which represents a path consisting of all straight links. Thus the HADM network is not fault-tolerant. It is interesting to attempt further reduction of the network complexity while maintaining the connection between any source and any destination. It is shown in Theorem 3.1 that actually only four output links for each switch would suffice to provide connection for any source/destination pair in the HADM network.

Consider a quad-tree that consists of $\log_4 N$ levels and N leaves. Clearly the out-degree of four for each node in the quad-tree is the smallest out-degree such that the root can reach any leaf; if any node except a leaf has an out-degree less than four, some leaves can not be reached by the root. Similarly, for a network of size N that consists of $\log_4 N$ stages of uniform switches, at least four output links for each switch are needed so that any source can communicate with any destination. Such a network has the minimum number of output links for each switch required for one-to-one connections and is called a *Minimum ADM (MADM) Network*. It consists of $n/2$ stages of 4×4 switches. Each switch of stage $k+1$, $0 \le k \le (n/2)-1$, is connected to four output links: straight link, $+2^{2k}$, -2^{2k} and $+2^{2k+1}$. Figure 5 illustrates a MADM network of size $N = 16$.

The MADM and HADM networks differ only in that each switch of stage k in the MADM network is connected to only one of the $+2^{2k+1}$ and -2^{2k+1} links

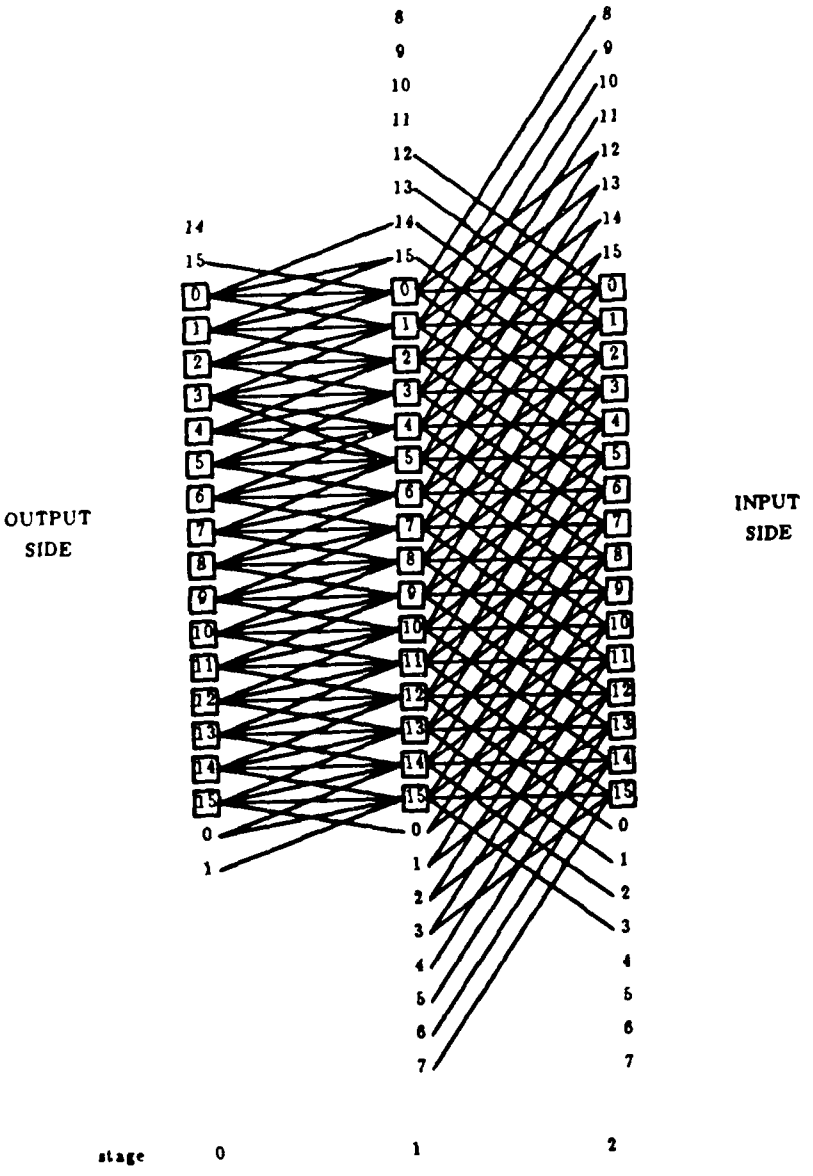


Figure 5. The MADM network for $N = 16$.

while each switch of stage k in the HADM network is connected to both links. So only a subset of routing tags for the HADM network are valid routing tags for the MADM network. In addition to the constraint that $t_{2k} \cdot t_{2k+1} = 0$ for $0 \leq k \leq (n/2)-1$, which a routing tag for the HADM network must satisfy, a valid tag for the MADM network must also satisfy the second constraint that, for Δ_{2k+1} odd, t_{2k+1} must be 1 if link $+2^{2k+1}$ is used and t_{2k+1} must be $\bar{1}$ if link -2^{2k+1} is used. The second constraint does not specify which of links $+2^{2k+1}$ and -2^{2k+1} is used at stage k; each stage can choose freely a plus or minus link. As a result, there are as many as $2^{n/2}$ types of MADM network; they differ in their choice of link $+2^{2k+1}$ or -2^{2k+1} at some stage k. The algorithm MADM-TAGS below demonstrates an example of generating routing tags for a particular type of MADM network that contains $+2^{2k+1}$ link at every stage k, $0 \leq k \leq (n/2)-1$. For convenience of discussion, this network is referred to as the MADM network.

Algorithm MADM-TAGS $(\Delta, t_{(n-1)})$
$\Delta_0 = \Delta$
for $i = 0$ to $n-1$ do
  if Δ_i is even then $t_i = 0$, $\Delta_{i+1} = \frac{\Delta_i}{2}$
  else if i is even then if $\frac{\Delta_i - 1}{2}$ is even then $t_i = 1$, $\Delta_{i+1} = \frac{\Delta_i - 1}{2}$
    else $t_i = \bar{1}$, $\Delta_{i+1} = \frac{\Delta_i + 1}{2}$
    endif
  endif
enddo

The difference between the processes of generating tags for the HADM network and for the MADM network is that, for Δ_{2k+1} odd, t_{2k+1} can be 1 or $\bar{1}$ for the HADM network while t_{2k+1} can only be 1 for generating routing tag for the MADM network. So each digit is uniquely determined in algorithm MADM-TAGS. This indicates that there exists a unique tag for each distinct Δ, which corresponds to a unique path for each source/destination pair in the MADM network.

Since there are only four output links for each switch in the MADM network, two bits per stage suffice to represent the choice of one of the four output links of a switch to send data. A total of n bits are needed to implement the signed-digit representations for routing tags. Let $r_{(n-1)}$ be such a routing tag, in which a digit can be represented by a bit. Each switch at stage k in the MADM network examines bits $r_{2k} r_{2k+1}$ to determine the output link via which

data are routed. One possible implementation is shown below.

$$r_{2k}r_{2k+1} = \begin{cases} 11 \rightarrow +2^{2k+1} \\ 00 \rightarrow \text{straight} \\ 01 \rightarrow +2^{2k} \\ 10 \rightarrow -2^{2k} \end{cases} \qquad \text{for } 0 \le k \le (n/2)-1$$

where $\rightarrow$ means "en route".

However, for the generation of tags in algorithm MADM-TAGS, two bits may be needed to represent a digit of the routing tag and thus a total of $2n$ bits are needed. Once the computation is done, the tag can be converted to $r_{0/n-1}$ for actual routing, which requires only n bits per tag.

Theorem 3.1 There exists a unique path between any source and any destination in the MADM network.

Proof: It is shown that a routing tag $t_{0/n-1}$ for the HADM network that contains $t_{2k}t_{2k+1} = 0\bar{1}$ can be recoded to become $t'_{0/n-1}$ such that $t'_{2k}t'_{2k+1} = 01$ and $t'_{2j}\cdot t'_{2j+1} = 0$, for $0 \le j \le (n/2)-1$. Case (i) If $t_{2k+2} = 0$ such that $t_{2k/2k+2} = 0\bar{1}0$, then $t'_{2k/2k+3} = 01\bar{1}t'_{2k+3}$ or $011t'_{2k+3}$. Since Δ_{2k+2} is odd ($t'_{2k+2} = 1$ or $\bar{1}$), from Lemma 2.3, either $011t'_{2k+3}$ or $01\bar{1}t'_{2k+3}$ has $t'_{2k+3} = 0$. Case (ii) If $t_{2k+2} \neq 0$ then t_{2k+3} must be equal to 0, because a tag for HADM network must satisfy the constraint $t_{2k+2}\cdot t_{2k+3} = 0$. If $t_{2k/2k+3} = 0\bar{1}10$, $t'_{2k/2k+3} = 0100$, and if $t_{2k/2k+3} = 0\bar{1}\bar{1}0$ then $t'_{2k/2k+3} = 010\bar{1}$. The discussion for recoding $t'_{2k+2}t'_{2k+3} = 0\bar{1}$ is analogous to that for recoding $t_{2k}t_{2k+1} = 0\bar{1}$. Next uniqueness of the routing path is shown. Since the out-degree of every switch in the MADM network is four and there are $n/2 = (\log N)/2 = \log_4 N$ stages, each source switch and all switches connected to it form a quad-tree. The source switch is the root and the switches connected to it are the nodes in the quad-tree, with the switches of stage 0 as the leaves. There exists a unique path from a root to a leaf in the quad-tree and thus also a unique path from a source to a destination in the MADM network. □

The topology of the *Minimum Inverse ADM* (*MIADM*) network is the same as the MADM network, with the input and output sides reversed, much like the relationship between the ADM and IADM networks and between the HADM and HIADM networks. Especially the routing tag conversion technique used for the HADM and HIADM networks can be readily applied and the proposed routing scheme for the HADM network can also be used in the HIADM network.

4. The Extra Stage MADM Network

Complexity of the MADM network is minimum in the sense that, given the constraint of network size N and $\log_2 N$ stages of uniform switches, it can provide communication for any source/destination pair in the network by using minimum number of interstage links per stage. However, this kind of topology has a drawback that it does not provide fault-tolerance; a switch failure would prevent some source/destination pairs from communicating each other. The lack of fault-tolerance suggests the use of augmentation techniques [AdS81] to improve fault-tolerance for the MADM network. First an important observations for routing in the MADM network is made.

Theorem 4.1 In the MADM network, the paths from a source S, to destinations D, $(D+(N/4))$, $(D-(N/2))$ and $(D-(N/4))$ are all disjoint.

Proof: The proof shows only that the two paths from S to D and from S to $(D-(N/4))$ are disjoint; the other cases can be treated similarly. The proof consists of two parts: (A) given the tag $t_{0/n-1}$ for routing from S to D, a tag $t'_{0/n-1}$ for routing from S to $(D-(N/4))$ can be derived from it and they differ only in digits $n-2$ and $n-1$, and (B) proof of disjointness of the two paths based on the results in (A).

(A) Since $t_{0/n-1}$ is the routing tag from S to D, $(D-S) = (\sum_{i=0}^{n-2} t_i 2^i + t_{n-1}2^{n-1}) \bmod 2^n$. So $(D-S-(N/4)) = (\sum_{i=0}^{n-3} t_i 2^i + (t_{n-2}-1)2^{n-2} + t_{n-1}2^{n-1}) \bmod 2^n$. There are three possible values, 1, $\bar{1}$ and 0, for t_{n-2}, which are discussed in cases (i), (ii) and (iii), respectively, as follows. (i) If $t_{n-2} = 1$, $(D-S-(N/4)) = (\sum_{i=0}^{n-3} t_i 2^i + 0\cdot 2^{n-2} + t_{n-1}2^{n-1}) \bmod 2^n$. That is, $t'_{0/n-1} = t_{0/n-3}0t_{n-1}$. (ii) If $t_{n-2} = \bar{1}$, t_{n-1} must be 0 because $t_{n-2}t_{n-1} = 0$ and $(D-S-(N/4)) = (\sum_{i=0}^{n-3} t_i 2^i + 0\cdot 2^{n-2} + (t_{n-1}-1)2^{n-1}) \bmod 2^n$. Then $t'_{0/n-1} = (\sum_{i=0}^{n-3} t_i 2^i + 0\cdot 2^{n-2} - 2^{n-1}) \bmod 2^n = (\sum_{i=0}^{n-3} t_i 2^i + 0\cdot 2^{n-2} - 2^{n-1} + 2^n) \bmod 2^n = (\sum_{i=0}^{n-3} t_i 2^i + 0\cdot 2^{n-2} + 2^{n-1}) \bmod 2^n$. So $t'_{0/n-1} = t_{0/n-3}01$. (iii) If $t_{n-2} = 0$, $(D-S-(N/4)) = (\sum_{i=0}^{n-3} t_i 2^i - 2^{n-2} + t_{n-1}2^{n-1}) \bmod 2^n$. There are two possible values, 0 and 1, for t_{n-1}, which are discussed in cases (a) and (b), respectively. t_{n-1} can not be $\bar{1}$ because it is assumed that link $+2^{n-1}$ is used at stage $n/2$ in the MADM network. (a) If $t_{n-1} = 0$, $t'_{0/n-1} = t_{0/n-3}\bar{1}0$. (b) If $t_{n-1} = 1$, $(D-S-(N/4)) = (\sum_{i=0}^{n-2} t_i 2^i - 2^{n-2} + 2^{n-1}) \bmod 2^n = (\sum_{i=0}^{n-3} t_i 2^i + 2^{n-2} + 0\cdot 2^{n-1}) \bmod 2^n$ so that $t'_{0/n-1} = t_{0/n-3}10$.

(B) From (A), it is seen that the two routing tags for the two paths from S to D

(B) From (A), it is seen that the two routing tags for the two paths from S to D and from S to $(D-(N/4))$ differ only in digits $n-2$ and $n-1$; i.e. $t_i = t'_i$ for $0 \leq i \leq n-3$. The two tags are the unique tags for routing from S to D and from S to $(D-(N/4))$, respectively. Let F and F' be the two switches at stage $(n/2)-1$ on the paths from S to D and from S to $(D-(N/4))$, respectively. Since $t_i = t'_i$, for $0 \leq i \leq n-3$, $\sum_{i=0}^{n-3} t_i 2^i = \sum_{i=0}^{n-3} t'_i 2^i$ (i.e. the distances that the two paths traverse from stage $(n/2)-1$ to 0 are the same), and the distance between the two destinations D and $(D-(N/4))$ is $((N/4) \bmod N)$; hence the distance between F and F' must be also $((N/4) \bmod N)$, denoted $|F-F'| = (N/4) \bmod N$. The intermediary switches at stage k, $0 \leq k \leq (n/2)-2$, on the two paths are $F+\delta_k$ and $F'+\delta_k$, respectively, where $\delta_k = \sum_{i=k}^{(n/2)-2} (t_{2i} 2^{2i} + t_{2i+1} 2^{2i+1})$. But $(F+\delta_k) \neq (F'+\delta_k) \bmod 2^n$ because $|F-F'| = (N/4) \bmod N$. That is, the two paths never share a common intermediary switch and thus are disjoint. □

The identification of disjoint paths from a source to different destinations in Theorem 4.1 can be used to improve fault-tolerance for the MADM network. The technique is to add an extra stage to the MADM network. The extra stage can be placed in the output side of the MADM network such that each switch D at the extra stage is connected to four switches at the first stage of the MADM network: D, $(D+(N/4))$, $(D-(N/2))$ and $(D-(N/4))$. Data can be sent from source S to any of the four switches and then to the destination via the extra stage. Thus there exist four disjoint paths from any source to any destination in the extra stage network. Such a network with an extra stage in the output side of the MADM network is called an *extra stage MADM network*. An extra stage MADM network consists of $(n/2)+1$ stages labeled from 0 to $n/2$ from the output side to the input side, with an additional column of switches in the input side referred to as stage $(n/2)+1$. The extra stage in the extra stage MADM network consists of the switches of stage 0 and the input links of the switches. The topology of the extra stage MADM network from stage 1 to $(n/2)+1$ is the same as that of the MADM network from stage 0 to $n/2$. The extra stage MADM network is three-fault-tolerant because of the existence of four disjoint paths for every source/destination pair and thus can withstand at least three switch failures (except the input and output switches). Since each destination in the MADM network has at four input links, which are connected to four switches in the preceding stage, at most three-fault-tolerance is possible. By appending an extra stage to the MADM network, the optimal fault tolerance is achieved.

Since the four output links of a switch at the extra stage are straight link and links -2^{n-2} $(= N/4)$, $+2^{n-2}$ $(= N/4)$ and -2^{n-1} $(= -N/2)$, the extra stage

$(n/2)+1$ to stage 1 in the extra stage MADM network. Using the tags of distances $\Delta = (D-S)$, $\Delta = (D-S-(N/4))$, $\Delta = (D-S-(N/2))$, and $\Delta = (D-S+(N/4))$, respectively, a source S (a switch at stage $(n/2)+1$) in the extra stage MADM network can send data to any of the four switches D, $(D+(N/4))$, $(D-(N/4))$ and $(D-(N/2))$ at stage 1, and then reaches the destination D at stage 0. The routing from D^1 to D^0 is controlled by tag bits 00, from $(D+(N/4))^1$ to D^0, by $\bar{1}0$, from $(D-(N/2))^1$ to D^0, by 01, and from $(D-(N/4))^1$ to D^0, by 10. So in the extra stage *M4DM* network, $n+2$ bits are needed to represent a routing tag. Note that since the four tags of distances $\Delta = (D-S)$, $\Delta = (D-S-(N/4))$, $\Delta = (D-S-(N/2))$, and $\Delta = (D-S+(N/4))$ differ only in digits $n-2$ and $n-1$, once one of them is computed, the other can be readily computed by recoding the last two digits. The proof of Theorem 4.1 demonstrates the example of recoding the tag of distance $\Delta = (D-S)$ to a tag of distance $\Delta = (D-S-(N/4))$. The table below summarizes the recoding of digits $t_{n-2}t_{n-1}$ of a tag into the other three tags that are of distance $+N/4$, $-N/4$ and $-N/2$ from it.

	$+N/4$	$-N/4$	$-N/2$
00	10	$\bar{1}0$	01
01	$\bar{1}0$	10	00
10	01	00	10
$\bar{1}0$	00	01	10

Figure 6 illustrates an extra stage MADM network of size $N = 16$. It is also shown the four disjoint paths from $S = 3$ to $D = 12$. They are represented by the tags of distances $\Delta = 9$, $\Delta = 5$, $\Delta = 1$ and $\Delta = -3$, which are $(t_{n-2}t_{n-1} =)1001$, 1010, 1000 and $10\bar{1}0$, respectively. The routing paths are $(3^1,11^2,12^1,12^0)$, $(3^1,7^2,8^1,12^0)$, $(3^1,3^2,4^1,12^0)$ and $(3^1,15^2,0^1,12^0)$, respectively. Routing from 12^2 to 12^1 is controlled by tag bits 00, from 8^1 to 12^0, by 10, from 4^1 to 12^0, by 01, and from 0^1 to 12^0, by $\bar{1}0$.

It can be similarly shown that an extra stage can also be appended in the input side of the MADM network such that a switch S at the extra stage is connected to four switches at stage $n/2$ of the MADM network: S, $(S+1)$, $(S-1)$ and $(S+2)$. Four disjoint paths result from addition of such an extra stage to the MADM network. In this type of extra stage network, the extra stage consists of the switches of stage $(n/2)+1$ and the output links of the switches, and stage $n/2$ to stage 0 has the same topology as the MADM network. The extra stage appended in the input side has the same connection patterns as stage 1 of the MADM network. A source S at the extra stage can send data to any of the four switches at stage $n/2$: S, $(S+1)$, $(S-1)$ and $(S-2)$ that are directly connected to it and uses tags of distances $\Delta = (D-S)$, $\Delta = (D+1-S)$, $\Delta = (D-1-S)$ and $\Delta = (D-2-S)$, respectively, to send data to the destinations

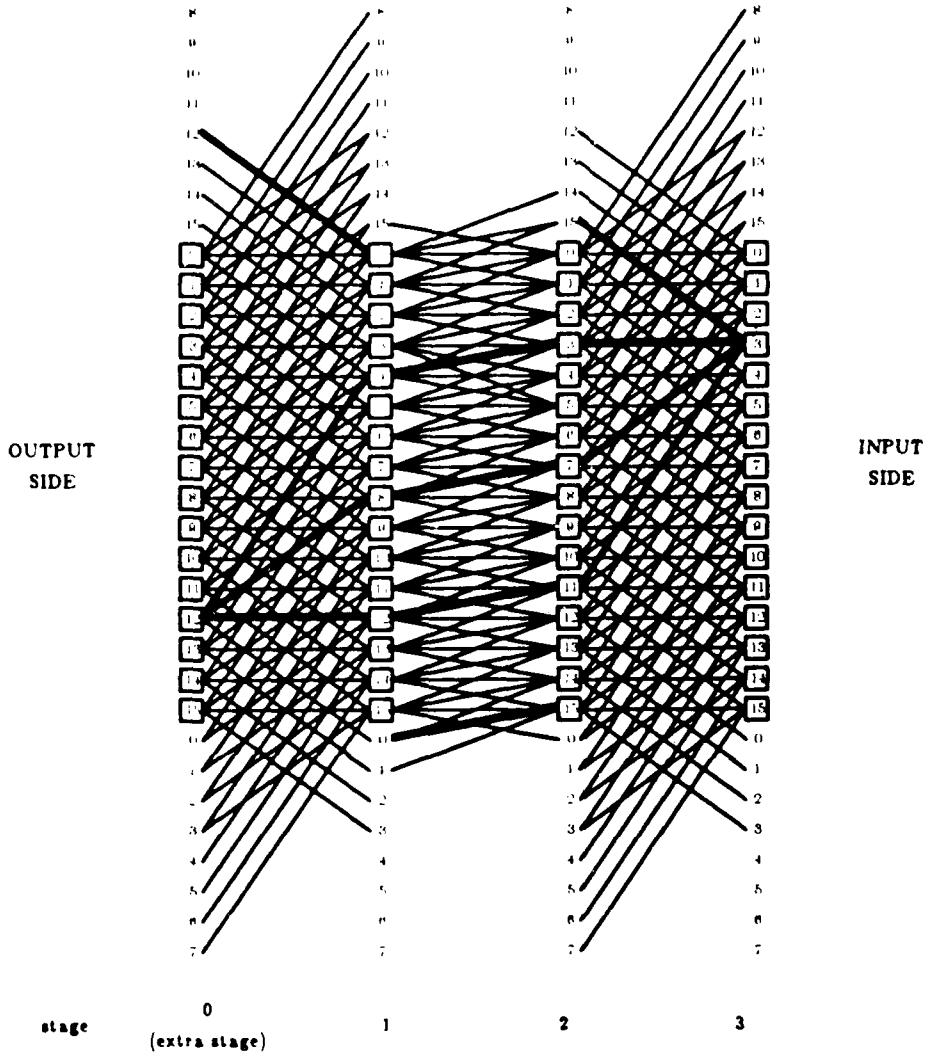


Figure 6. Routing from 3 to 12 in the extra stage MADM network for $N = 16$. The bold lines are links on the routing paths and the solid lines are links of the extra stage MADM networks.

D at stage 0. The routing from $S^{(n/2)+1}$ to $S^{n/2}$ is controlled by tag bits 00, from $S^{(n/2)+1}$ to $(S+1)^{n/2}$, by 10, from $S^{(n/2)+1}$ to $(S-1)^{n/2}$, by $\bar{1}0$, and from $S^{(n/2)+1}$ to $(S+2)^{n/2}$, by 01.

Apparently adding an extra stage to the input side of an MADM network is equivalent to adding the extra stage to the output side of the MIADM network and vice versa. Thus all discussions associated with the relationship between the MADM and MIADM networks can be applied for the extra stage networks as well.

5. Conclusion

This paper addresses the problem of designing multistage networks which are based on the implementation of PM2I functions at each stage. This type of multistage networks is referred to as augmented data manipulator networks and includes the well known ADM and IADM networks. Since the designs proposed in this paper use fewer stages and links than the ADM and IADM networks, they are referred to as partially augmented data manipulator networks. The HADM and HIADM networks derived in this paper have the least number of stages required in multistage networks (based on PM2I functions) where any source can be connected to any destination in one pass. The MADM and MIADM networks also have the least number of stages and, in addition, have the minimum number of links per switch required for one-to-one connections. The extra stage MADM and MIADM networks contain one more stage than the MADM and MIADM networks, respectively, and are fault-tolerant versions of the MADM network capable of tolerating at least three switch faults.

References

[AdS81] G. B. Adams III and H. J. Siegel, "The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems," *IEEE Trans. Computers*, Vol. C-30, No. 5, pp. 443-454, May 1981.

[Avi61] A. Avizienis, "Signed-Digit Number Representations for Fast Parallel Arithmetic," *IRE Trans. Electronic Computers*, pp. 389-400, Sept. 1961.

[Fen74] T-Y Feng, "Data Manipulating Functions in Parallel Processors and Their Implementations," *IEEE Trans. Computers*, Vol. C-23, No. 3, pp. 309-318, Mar. 1974.

[HwB84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, NY, pp. 345, 1984.

[McS82] R. J. McMillen and H. J. Siegel, "Routing Schemes for the Augmented Data Manipulator Network in an MIMD System," *IEEE Trans. Computers*, Vol. C-31, No. 12, pp. 1202-1214, Dec. 1982.

[PaR82] D. S. Parker and C. S. Raghavendra, "The Gamma Network: A Multiprocessor Interconnection Network with Redundant Paths," *9th Annu. Symp. on Computer Architecture*, pp. 73-80, Apr. 1982.

[PaR84] D. S. Parker and C. S. Raghavendra, "The Gamma Network," *IEEE Trans. Computers*, Vol. C-33, No. 4, pp. 367-373, Apr. 1984.

[Sie77] H. J. Siegel, "Analysis Techniques for SIMD Machine Interconnection Networks and the Effects of Processor Address Masks," *IEEE Trans. Computer*, Vol. C-26, No. 2, pp. 153-161, Feb. 1977.

[SiS78] H. J. Siegel and S. D. Smith, "Study of Multistage SIMD Interconnection Networks," *5th Annu. Symp. on Computer Architecture*, pp. 223-229, Apr. 1978.

REFERENCE NO. 10

Rau, D. and Fortes, J. A. B., "Destination Tag-Controlled Fault Tolerant Interconnection Networks," Symposium on Innovative Science and Technology, January 1988

Destination tag controlled fault-tolerant networks

Darwen Rau and Jose A. B. Fortes*

School of Electrical Engineering, Purdue University
West Lafayette, IN 47907

ABSTRACT

A systematic approach is proposed to construct networks that can be controlled by destination tags. The networks can be designed so that they have a unique path between any of its inputs and any of its outputs and can also be designed so that multiple paths are provided. When multiple paths exist, the network is fault-tolerant in the sense that a fault in the path does not prevent communication through a redundant path. This approach is based on the fact that for a unique-path network of size N with $\log_2 N$ stages and 2×2 switches, a binary tree structure can be associated with each input switch. The tree is such that the root, nodes, edges and levels of the tree correspond to the input switch, switches connected to the input switch, links between these switches and stages of the network, respectively. Trees of these structures can be devised and used as the underlying "skeletons" of fault-tolerant networks that are controlled by destination tags.

There are several ways to organize binary trees and map them into networks in order to implement destination tag routing. This paper concentrates on a special case of the approach and experiments are conducted on networks whose switches are 2×2, 3×3 or 4×4 switches although non-uniform switches of various sizes are also possible. Merits and advantages of these designs are discussed with respect to routing and rerouting schemes and fault-tolerance. Several existing seemingly varied and independent designs of unique-path and fault-tolerant networks are found to be governed by this approach. In particular, Augmented Data Manipulator (ADM) networks and related designs are investigated and improved with respect to fault-tolerance and routing and rerouting schemes, and new designs with better fault-tolerance capabilities are suggested.

I. INTRODUCTION

This paper proposes a systematic approach to the design of multistage *interconnection* networks that can be controlled by destination tag routing. Both unique-path and fault-tolerant networks are considered. The approach is based on the observation that such networks can be derived by coalescing multiple trees. Several seemingly varied and independent designs are found to be governed by this approach; this finding leads to more efficient routing and rerouting schemes and better fault-tolerant topologies for them.

During the past decade, a plethora of interconnection networks have been proposed. Perhaps the most popular class of multistage networks is that of multistage *cube-type* networks such as the Indirect Binary n-Cube (ICube)[1], Omega[2], Baseline[17], Modified ADM[17], STARAN flip[4] and a special case of SW-Banyan[5] networks. Among the main advantages of these networks are their very efficient destination tag routing schemes, partitionability, $O(N\log_2 N)$ cost and ability to pass useful permutations.

In a cube-type network, the binary representation of a destination address can be used directly as a routing tag. The interchange box at stage i needs only to examine the i-*th* bit of the destination routing tag for an incoming message. If the i-*th* bit is 0, the upper output of the box is taken and if the i-*th* bit is 1, the lower output of the box is taken. These schemes are known as *destination tag routing schemes*[19] and are extremely efficient and simple to implement.

Due to the requirement for efficient routing of data in interconnection networks used in real-time applications, a destination tag routing scheme is a desirable feature in interconnection networks. It is interesting to see whether or not there exist network topologies that are not necessarily equivalent to the cube-type networks and can still be controlled by destination tags. The motivation of this paper is to find a systematic approach to construct networks of this type, hereafter referred to as *destination-tag-controlled networks (DN's)*.

Let any input switch in the first stage of a network be called a *source* and an output switch in the last stage of a network be called a *destination*. Input links of a source and output links of a destination are called *terminal links*. Cube-type networks have the property that a unique routing path exists between any source and any destination; as a consequence, a single switch failure eliminates the possibility of communication between some source/destination pair. To overcome this problem, a popular approach is to provide extra interstage links to an existing unique-path topology in order to create additional paths and thus provide an ability to tolerate faults[3,12]. In fact, many networks with redundant paths can be explained as unique-path networks to which extra links have been added. For example, it has been shown that the Gamma network[13] can be regarded as a fault-tolerant version of the ICube network where one extra link is added to each switch of the ICube network[2,15,16]. An ICube network is shown in Figure 1 and a Gamma network and its relationship to the ICube network is shown in Figure 2. The Gamma network has three output links for each switch and can tolerate some source/destination pairs. However, the Gamma network may not be able to survive some

instances of a single switch failure; i.e., the communication for some source/destination pairs may be eliminated due to a switch failure. To achieve one-fault-tolerance for the Gamma network, yet another extra link is added to every switch of the Gamma network, which results in the Kappa network[6]. This is equivalent to adding two extra links to every switch of the ICube network and the Kappa network can tolerate at least one switch failure. A Kappa network and its relationship with the Gamma network is shown in Figure 3.

This paper is concerned with the construction of DN's that have multiple paths for each source/destination pair, called *fault-tolerant DN's (FDN's)*, as well as *unique-path DN's (UDN's)*. In particular, it unifies the principles that underlie the construction of the ICube, Gamma and Kappa networks and shows that a plethora of other DN topologies can also result from adding extra links for each switch of the ICube and Gamma networks, respectively.

Notation used in this paper is defined next. Any integer t has a binary representation $t_{n-1/0} = t_{n-1}t_{n-2}\cdots t_0$, where t_{n-1} is the most significant bit and n denotes the number of bits. The notation $t_{q/p}$ denotes the bits of t starting at t_p and ending at t_q. To indicate the 1's complement of bit u_i, the notation $\bar{u}_i$ is used. Throughout this paper, u and $u+a$ (where a is some constant) represent labels of switches. Also modulo N arithmetic is assumed, e.g. $u+a$ implies $(u+a)\ mod\ N$. The notation u^i is used to indicate that a switch u belongs to stage i and (u^i, v^{i+1}) is used to represent a link at stage i joining u^i and v^{i+1}. The *out-degree* of a switch is the number of output links of the switch and the *in-degree*, the number of input links.

Section 2 of the paper examines the structure of the cube-type networks and proposes binary tree structures for constructing DN's. D-constructs are presented in Section 3 as building blocks of binary tree structures and UDN's are constructed by using the D-constructs. Because UDN's are not fault-tolerant, improved topologies have been proposed in the past which, by adding one extra link per switch, provide some redundant paths. These and other similar networks are referred to as *enhanced DN's (EDN's)* and are considered in Section 4. However, EDN's may still fail to tolerate a single fault in a switch or link. In Section 5 fault-tolerant D-constructs are proposed to construct FDN's that are capable of tolerating at least one switch failure. Merits in routing and rerouting schemes and fault-tolerance advantages are also discussed for UDN's, EDN's and FDN's, respectively. Section 6 concludes the paper.

2. THE BINARY AND FAULT-TOLERANT TREE STRUCTURES

This section examines the structure of the cube-type networks, which are known UDN's, in order to derive and provide insights into a systematic approach to construct DN's.

The structures of all cube-type networks consist of $n = \log N$ stages of $N/2$ 2x2 switches. Because there are n stages of switches and each switch is connected to two switches at the next stage, for each source there exists a binary tree that contains the source as the root of the tree and the switches reachable from the root as the nodes of the tree[17]. A switch is said to be *reachable* from the other if there exists a path between them. The $N = 2^n$ output switches at the last stage are the leaves shared by all binary trees. For example, Figure 1 shows a binary tree in an ICube network. It includes the source 00 as the root, switches 00 and 01 of stage 1, and all switches of the last stage as the nodes of the binary tree. The ICube network can be regarded the coalescing or partial overlapping of N binary trees, each of which has a distinct root (source) and may or may not share with other trees switches of stage i, $0 \le i \le n-1$; for $i = n-1$, the N destinations are the leaves shared by all N binary trees.

The binary tree structure was used to assign labels to terminal links.[17] The label can be obtained by assigning a label 0 for the upper output link and 1 for the lower output link and concatenating the labels along the path from a terminal input link to a terminal output link. This is also illustrated in Figure 1.

Notice that in the ICube network a source has two terminal input links and a destination has two terminal output links. In the ICube network, any arbitrary terminal input link can be connected to any arbitrary terminal output link because there exists a path that connects the switch (source) of the terminal input link and the switch (destination) of the terminal output link. As far as one-to-one routing is concerned, only the interstage connection patterns affect the connection between a source and a destination and the number of terminal links of the source and the destination is unimportant. This allows this work to concentrate on the study of the interstage connection patterns and simplifies the discussions for the construction of DN topologies.

The above observations suggest the following sufficient conditions for a network of size $N \times N$ to be a DN: (a) there exist at least N embedded binary trees and at least one such tree is rooted at each input switch of the network and has the N output switches as its leaves, and (b) each of the two output links of a node of the binary tree is assigned a label 0 or 1 and the unique address of any destination of the network is formed by concatenating the labels along the path from the root to the destination. The DN may have different number of switches at each stage, switches of different stages do not necessarily have the same size, and even the switches of the same stage can be of varied sizes. A binary tree has $\log N$ levels and thus the DN has $\log N$ stages (the basic idea is also applicable to networks with arbitrary number of stages but this case is out of the scope of this paper.)

The notion of block structure[9] was used to describe the topologies of the cube-type networks and can be regarded as a subset of the binary tree structures. The concept of the block structure can be best explained by the illustration of the

structure of a Baseline network in Figure 4. The entire network is regarded as an $N \times N$ block and then it is divided into a stage and two subblocks. The stage is the first stage of the block and the remaining stages are divided into two subblocks. The process is repeated until the last stage is reached where a block consists of a single switch. Because a switch of the first stage of a block is connected to two switches of the first stage of the two subblocks, an input switch and all switches reachable from it constitute a binary tree. In other words, the block structure can be formed by properly coalescing binary trees. However, many possible structures which are not necessarily a block structure can result from coalescing multiple binary trees. That is, the block structure employs more restricted connections for constructing networks than those resulting from coalescing multiple binary trees.

One of the important criteria in choosing a switch size is network modularity. A module is a building block of a network that can be formed by cascading these building blocks. In order to reduce design and manufacturing costs, it is desirable that a network can be modularly constructed and the number and size of modules should be minimized. A uniform switch size for the network (and thus the same number of switches for every stage) can facilitate modular design. For this reason, only DN's of size $N \times N$ that consist of uniform switches are considered in this paper.

If the number of output links of every switch of a DN is two, then there exists a unique path in the DN for every source/destination pair because there exists a unique binary tree of which the source is the root and there exists a unique path from a root to any leaf in the tree. Such a DN is a UDN. The inability of the UDN to tolerate faults for the UDN can be improved by adding redundancy to it. This paper considers the special types of *fault-tolerant trees* resulting from adding one and two extra links to every node of the embedded binary trees of UDN's and the DN's constructed by coalescing N such fault-tolerant trees. Because modular designs are sought, only DN's with uniform 3×3 or 4×4 switches are considered. Merits and advantages of these designs with respect to fault-tolerance and routing and rerouting schemes are also discussed in later sections.

3. THE D-CONSTRUCT AND UDN

The interstage connection patterns of a DN are represented by labeling of intermediary switches of the DN. At two extremes of a DN are the source and destination which have addresses $s_{n-1/0}$ and $d_{n-1/0}$, respectively. As the data is relayed via intermediary switches, labeling of these switches reflects routing as a progressive conversion of a string of all source address bits to a string of all destination address bits. With this in mind, this section discusses the construct of a class of UDN's based on binary tree structures.

Let the sequence $(k_0, k_1, \ldots, k_i, \ldots, k_{n-1})$ be a permutation of the sequence $(0,1,\ldots,i,\ldots,n-1)$, $i = 0,1,\ldots,n-1$. Define a connection function D^i mapping a switch u^i to a switch $D^i(u^i,t_{k_i})^{i+1}$ such that

$$D^i(u^i,t_{k_i}) = \begin{cases} p & \text{for } t_{k_i} = 0 \\ q & \text{for } t_{k_i} = 1 \end{cases}$$

where $p_{k_j} = q_{k_j} = u_{k_j}$, $0 \le j \le i-1$, and $p_{k_i} = 0$, $q_{k_i} = 1$,

and the values for p_{k_j} and q_{k_j}, $i+1 \le j \le n-1$, are determined according to u^i and t_{k_i} and some design criteria. For example, if $k_i = i$, $0 \le i \le n-1$, and $p_{k_j} = q_{k_j} = u_{k_j}$, $i+1 \le j \le n-1$, the mapping function D^i is:

$$D^i(u^i,t_i) = \begin{cases} u_{n-1/i+1}0u_{i-1/0} & \text{for } t_i = 0 \\ u_{n-1/i+1}1u_{i-1/0} & \text{for } t_i = 1 \end{cases}$$

As a more complicated example, if $k_i = n-i-1$, $0 \le i \le n-1$, and $p_{k_j} = q_{k_j} = u_{k_j}$, $i+1 \le j \le n-1$, then the mapping function D^i is:

$$D^i(u^i,t_{n-i-1}) = \begin{cases} u_{n-1/n-i}0u_{n-i-2/0} & \text{for } t_{n-i-1} = 0 \\ u_{n-1/n-i}1u_{n-i-2/0} & \text{for } t_{n-i-1} = 1 \end{cases}$$

These two examples correspond to the mapping functions for the IGcube and the modified ADM networks, respectively, as explained later in this section.

For a pair of switches joined by a link, the switch at the lower-order stage is called a *predecessor* and the switch at the higher-order stage is called a *successor*. The definition of D^i indicates that, for each switch u^i, there exist two successors at stage $i+1$, p^{i+1} and q^{i+1}, and labels of u^i, p^{i+1} and q^{i+1} agree in bits $k_0, k_1, \ldots,$ and k_{i-1}. The two output links (u^i,p^{i+1}) and (u^i,q^{i+1}) are called a *0-link* and a *1-link*, respectively, and switches p^{i+1} and q^{i+1} are called a *0-successor* and a *1-successor*, respectively. This naming reflects the fact that the k_i-*th* bit of the label of the switch at stage $i+1$ connected to u^i is 0 (or 1) if messages are sent via a 0-link (or 1-link). Note that a mapping function D^i defines only the the connections between a switch of stage i and two switches of stage $i+1$ and switches of the same stage may have different mapping functions.

The set of functions $\{D^i,\ 0 \le i \le n-1\}$, henceforth called the *D-construct*, specifies connection patterns between

adjacent stages i and $i+1$ and therefore can be used as building blocks to construct networks. The definition of the D-construct specifies the number of successors for every switch to be two and the constraint of network modularity determines a uniform switch size for the network. Therefore, a switch of the network has exactly two input and two output links and all stages have the same number of switches. Routing in the network is such that a switch at stage i needs to examine the k_i-*th* bit of the routing tag to determine through which output link messages are sent. If the bit is 0, the message is sent via the 0-link and if it is 1, the message is sent via the 1-link. An interesting relationship between any routing tag and the labels of the switches on the routing path specified by the routing tag is stated next.

Property 3.1 In a network constructed by using the D-construct, let $t_{n-1/0}$ be a routing tag and v^{i+1} be a switch on the routing path specified by $t_{n-1/0}$, then $v_{k_j} = t_{k_j}$, for $0 \le j \le i$.

Proof: Let $s_{n-1/0}$ be the source of the path specified by $t_{n-1/0}$. By definition of the D-construct,

$$(D^0(s^0,t_{k_0}))_{k_0} = v^1{}_{k_0} = t_{k_0} .$$

$$(D^1(D^0(s^0,t_{k_0}),t_{k_1}))_{k_1} = v^2{}_{k_1} = t_{k_1} , (D^1(D^0(s^0,t_{k_0}),t_{k_1}))_{k_0} = v^2{}_{k_0} = t_{k_0} .$$

$$\cdots$$

$$(D^i(\cdots D^1(D^0(s^0,t_{k_0}),t_{k_1}),\cdots,t_{k_i}))_{k_j} = (v^{i+1})_{k_j} = t_{k_j} , \text{ for } 0 \le j \le i .$$

□

Property 3.1 indicates that bits k_0, k_1, ..., k_i of the labels of the switches at stage $i+1$ of the routing path specified by a routing tag are equal to the corresponding bits of the routing tag. By specifying the routing tag $t_{n-1/0} = d_{n-1/0}$, the label of a switch $(u_{n-1/0})^i$, $0 \le i \le n-1$, on the routing path has $u_{k_j} = d_{k_j}$, $0 \le j \le i-1$, and at the last stage $(i+1 = n)$, the destination of the path has the address equal to $d_{n-1/0}$. This is true regardless of the address of the source sending the message. Thus a network constructed by using the D-construct is a DN. In addition, because the out-degree of every switch of the DN is two, there exists a unique path from any source to any destination; so the network is also a UDN.

It is implicit in the construction of the UDN using the D-construct that for each source of the UDN, a binary tree composed of the source and all switches reachable from the source is embedded in the UDN. To show that a binary tree exists for each source, it needs to be shown that at stage i, $0 \le i \le n$, all the 2^i switches reachable from the same source are distinct switches. From the definition of the D-construct, the two successors (at stage 1) of the source have their k_0-*th* bit equal to 0 and 1, respectively; so these two switches must be distinct switches. By Property 3.1, the four switches at stage 2 reachable from the two successors must have the bits of their label in positions k_0 and k_1 equal to 00, 01, 10, and 11, respectively; so they are distinct switches. By simple inductive reasoning, at stage i, the labels of all switches reachable from the successors at stage $i-1$ must differ in at least one bit in bits k_0, k_1, ..., or k_{i-1}. Hence all switches at stage i reachable from the source are distinct switches.

The following discussions demonstrate the construction of a UDN based on the D-construct. It is shown how to construct interstage connection patterns between two adjacent stages; by repeating the same procedure for every stage, a UDN results. Let A^i be a subset of switches of stage i such that bits k_0, k_1, ..., k_{i-1} of the label of every switch in the same subset have the same value and $B_0(u^i)$ and $B_1(u^i)$ be subsets of switches of stage $i+1$ defined as follows:

$$B_0(u^i) = \{v^{i+1} | v_{k_j} = u_{k_j},\ 0 \le j \le i-1,\ \text{and}\ v_{k_i} = 0\ \text{and}\ u^i \in A^i\}$$

$$B_1(u^i) = \{w^{i+1} | w_{k_j} = u_{k_j},\ 0 \le j \le i-1,\ \text{and}\ w_{k_i} = 1\ \text{and}\ u^i \in A^i\}$$

By definitions of $B_0(u^i)$ and $B_1(u^i)$, they are the subsets of possible successors for u^i and can be connected to u^i through 0-links and 1-links, respectively. Since we always refer to $\{A^i, B_0(u^i)\}$ or $\{A^i, B_1(u^i)\}$ for which u is an element of A^i, notation A, B_0 and B_1 are used henceforth to represent A^i, $B_0(u^i)$ and $B_1(u^i)$, respectively. For a given value of i, $0 \le i \le n-1$, there are 2^i subsets of A, each subset consisting of 2^{n-i} switches; and 2^i subsets of B_0 and 2^i subsets of B_1, each subset consisting of 2^{n-i-1} switches. A connection pattern between A and B_0 (or between A and B_1) is a set of 0-links (or 1-links), each of the 0-links (or 1-links) joining a switch of A and a switch of B_0 (or a switch of A and a switch of B_1) such that the in-degree of every switch of B_0 is two and one link leaves from every switch of A. Since A is connected to both B_0 and B_1, the out-degree of every switch of A is two in the resulting connection pattern. Note that connection patterns of all 2^i pairs of $\{A, B_0\}$ and $\{A, B_1\}$ can be found independently. The following algorithm is used to find connection patterns for $\{A, B_0\}$. Connection patterns for $\{A, B_1\}$ can be found similarly.

Algorithm CONNECTION-1

step 0: in-degree $(v) = 0$ for all $v \in B_0$.
step 1: If $A = \phi$, done.
step 2: Connect $u \in A$ to $v \in B_0$ and in-degree(v) = in-degree$(v)+1$.
step 3: $A = A - \{u\}$ and if in-degree$(v) = 2$, $B_0 = B_0 - \{v\}$; go to step 1.

Step 0 is an initialization step. Since in the resulting connection pattern for $\{A,B_0\}$ every switch of A is connected to only one switch of B_0, a switch is removed from sets A as soon as it is determined in step 2. However, every switch of B_0 is connected to two switches of A so that a switch is not removed from B_0 until two connections are made between the switch and two switches of A. The algorithm repeats steps 1 to 3 iteratively for every switch of A.

In step 2, for a given $u \in A$, v can be any of the switches of B_0. Because each switch in B_0 can be selected twice (i.e. a switch of B_0 is connected to two switches of A), the size of B_0 can be regarded as $2 \cdot 2^{n-i-1}$. So there are $(2^{n-i-1})!$ possible connection patterns for $\{A,B_0\}$. In addition, because there are 2^i pairs of $\{A,B_0\}$ and 2^i pairs of $\{A,B_i\}$ in stage i, there are a total of $((2^{n-i-1})!)^{2^{i+1}}$ possible connection patterns between stages i and $i+1$. Thus, there are a total of $\prod_{i=0}^{n-1}((2^{n-i-1})!)^{2^{i+1}}$ distinct topologies for the UDN's.

The ICube network[14] and the modified augmented data manipulator (ADM) network[17] fall into this class of UDN's. The topology describing rules of the two networks can be readily transformed into D-construct expressions. In both networks, any switch at stage i has $p_{k_j} = q_{k_j}$, $0 \le j \le n-1$, except for $j = i$. So the D-construct of both networks can be expressed as:

$D^i(u^i,t_{k_i}) = u_{n-1/i+1}t_{k_i}u_{i-1/0}$, for the ICube network

$D^i(u^i,t_{k_i}) = u_{n-1/n-i}t_{k_i}u_{n-i-2/0}$, for the modified ADM network

and the mapping of k_i is i and $n-i-1$, $0 \le i \le n-1$, for the ICube network and the modified ADM network, respectively.

4. FAULT-TOLERANT IMPROVEMENT FOR DN'S

This section considers the simplest type of fault-tolerant tree in which only one extra link is added to every node of the binary tree and the class of networks, called *enhanced DN's (EDN's)*, that are constructed by coalescing N such fault-tolerant trees. As a result, an EDN has 3×3 switches and corresponds to the network that results from adding one extra output link to every switch of a UDN derived in Section 3.

The extra link can be either a 0-link or a 1-link. That is, a switch of the EDN has three output links: one 0-link and two 1-links, or one 1-link and two 0-links. A 0-link (or 1-link) is said to be a *conjugate 0-link* (or *1-link*) if there exists an additional 0-link (or 1-link) for the switch and they are called a *conjugate link* of each other.

The *enhanced D-construct* used to construct the EDN's is as follows.

$$D^i(u^i,t_{n-1}\cdots t_0) = \begin{cases} p & \text{for } t_{n+k}t_k = 00 \\ q & \text{for } t_{n+k}t_k = 01 \\ \hat{r} & \text{for } t_{n+k}t_k = 10 \text{ or } 11 \end{cases}$$

where $p_k = q_k = \hat{r}_k = u_k$, $0 \le j \le i-1$; and $p_k = 0$, $q_k = 1$ and $\hat{r}_k = 0$ if the conjugate links of the switch are 0-links and $\hat{r}_k = 1$ if the conjugate links are 1-links. The values for p_k, q_k and $\hat{r}_k$, $i+1 \le j \le n-1$, are determined according to u^i, t_k and t_{n+k}.

Routing in EDN's can also be controlled by the destination routing tag. If $t_{k_i} = 0$, messages can be routed via any of the conjugate 0-links and reaches the same destination; if $t_{k_i} = 1$, messages can be routed via any of the conjugate 1-links. The bit t_{n+k_i} determines which of the conjugate links is used for routing. If at some stage a conjugate 0-link (or 1-link) is blocked, rerouting can be done by sending messages via its conjugate link. A 0-link (or 1-link) being *nonconjugate* means that the 0-link (or 1-link) is the only 0-link (or 1-link) of the switch. If a nonconjugate link on a path is blocked, rerouting from the switch to which the blocked link is an output link is impossible and communication between the source and the destination of the path is eliminated.

If a nonconjugate link is blocked, the only resort for rerouting is to backtrack along the routing path (that contains the blocked link) to find a conjugate link at a lower-order stage, from which there may exist an alternate routing path that can avoid the blocked link. Backtracking can also implemented as a look-ahead scheme[12]. However, backtracking (or look-ahead) techniques require each switch to be able to detect inaccessibility of any output port (connected to a switch at the next stage) and signal presence of the blockage back to the switches of previous stages[12,15]. Even if backtracking is available, an alternate routing path may not exist if no conjugate links exist at a lower-order stage of the routing path.

The Gamma network[13,15] falls into the category of EDN's. A switch u^i of the Gamma network has three successors at stage $i+1$: $(u+2^i)^{i+1}$, u^{i+1} and $(u-2^i)^{i+1}$, $0 \le i \le n-1$. The D-construct expressions for the Gamma network are as follows.

(i) For $u_i = 0$, $p_{n-1/0} = u_{n-1/i+1}0u_{i-1/0}$, $q_{n-1/0} = u_{n-1/i+1}1u_{i-1/0}$, $\hat{q}_{n-1/i+1} = u_{n-1/i+1}-2^{i+1}$ and $\hat{q}_{i/0} = 1u_{i-1/0}$.
(ii) For $u_i = 1$, $q_{n-1/0} = u_{n-1/i+1}1u_{i-1/0}$, $p_{n-1/0} = u_{n-1/i+1}0u_{i-1/0}$, $\hat{p}_{n-1/i+1} = u_{n-1/i+1}+2^{i+1}$ and $\hat{p}_{i/0} = 0u_{i-1/0}$.
From the D-construct expressions for the Gamma network, it is clear that links (u^i, p^{i+1}) and $(u^i, \hat{p}^{i+1})$ are the conjugate 0-links of u^i (for $u_i = 1$), and links (u^i, q^{i+1}) and $(u^i, \hat{q}^{i+1})$ are the conjugate 1-links of u^i (for $u_i = 0$). Comparing with the D-construct expressions for the ICube network, it is readily seen that the D-construct expressions for the ICube network is a subset of those for the Gamma network. Links $(u^i, \hat{q}^{i+1})$ for $u_i = 0$ and $(u^i, \hat{p}^{i+1})$ for $u_i = 1$ of the Gamma network are the extra links added to the ICube network. Past routing schemes for the Gamma network are complicated distance tag routing schemes[11,13] that require the computation of the distance between the source and the destination in order to generate routing tags; by decomposing the Gamma network into fault-tolerant tree structures, it is readily realized that destination tag routing can be used for the Gamma network[15].

The Gamma network is topologically equivalent to the LADM network[5,11]; however, they use switches of different types. Since this paper is concerned with only one-to-one routing, the results in this paper equally apply to both of them. It was observed[8,12] that the Gamma and LADM networks did not have one-fault-tolerance. By transforming the topology of the Gamma network into D-construct expressions, it is easily seen that each switch of the Gamma network has only conjugate 0-links or conjugate 1-links but never both and thus the Gamma network does not have one-fault-tolerance. Past techniques rely on distance tag schemes[12] and topology comparison[8] to show this property and derive only one fault-tolerant topology for the Gamma network[8,12]. It will be shown in Section 5 that a great number of fault-tolerant topologies for the Gamma network can be systematically derived based on binary and fault-tolerant tree structures.

5. FAULT-TOLERANT D-CONSTRUCT AND FDN

The lack of adequate fault-tolerance for EDN's determines the addition of a second extra link to every node of the binary tree in order to form a fault-tolerant tree, which can be used to construct fault-tolerant networks. Also, due to network modularity requirement, the network so constructed corresponds to the one resulting from adding two extra links, one 0-link and one 1-link, to every switch of a UDN derived in Section 3. Thus every switch of the network has one pair of 0-links and one pair of 1-links. In order to maximize the fault-tolerance improvement, it is desirable to have each of the 0-links (and 1-links) be connected to a different switch at the next stage; otherwise a faulty successor may block both 0-links (or 1-links). Thus each switch is connected to four switches at the next stage and each stage of the network consists of 4N links. Such a network is an FDN that can tolerate at least one switch failure (except the failures in input and output switches, which are sources and destinations) and is capable of performing dynamic rerouting. This is true because rerouting can always be done by sending messages through the conjugate link of a blocked link.

The *fault-tolerant D-construct* used to construct the FDN's is as follows.

$$D^i(u^i, t_{n+k_i}t_{k_i}) = \begin{cases} p & \text{for } t_{n+k_i}t_{k_i} = 00 \\ \hat{p} & \text{for } t_{n+k_i}t_{k_i} = 10 \\ q & \text{for } t_{n+k_i}t_{k_i} = 01 \\ \hat{q} & \text{for } t_{n+k_i}t_{k_i} = 11 \end{cases}$$

where $p_{k_j} = \hat{p}_{k_j} = q_{k_j} = \hat{q}_{k_j} = u_{k_j}$, $0 \le j \le i-1$, and $p_{k_i} = \hat{p}_{k_i} = 0$, $q_{k_i} = \hat{q}_{k_i} = 1$. In addition, to make each of 0-links (or 1-links) be connected to a different switch at the next stage, the following conditions must be satisfied: $p_{k_j} \ne \hat{p}_{k_j}$ and $q_{k_j} \ne \hat{q}_{k_j}$, for some j, $i+1 \le j \le n-1$. So $p_{n-1/0} \ne \hat{p}_{n-1/0} \ne q_{n-1/0} \ne \hat{q}_{n-1/0}$; i.e. u^i is connected to four distinct switches at the next stage. The pair of switches p^{i+1} and $\hat{p}^{i+1}$ and the pair of switches q^{i+1} and $\hat{q}^{i+1}$ are called *conjugate switches*. (u^i, p^{i+1}) and $(u^i, \hat{p}^{i+1})$ are conjugate 0-links and (u^i, q^{i+1}) and $(u^i, \hat{q}^{i+1})$ are conjugate 1-links for u^i. A pair of conjugate switches are on two distinct routing paths to the same destination; thus they can reach the same subset of destinations.

Algorithm CONNECTION-1 can be adapted to find conjugate links for and form FDN's from a UDN. Let $U(u)$ be a successor of a switch u in the UDN, $F(u)$ be a successor of the switch u selected for the FDN, and the definitions for A and B_0 be the same as those for algorithm CONNECTION-1. Also define A' to be a set of switches that contains the switches of A whose connections with switches of B_0 were made and which were removed from A for the construction of an FDN. The following algorithm CONNECTION-2 finds a conjugate 0-link for every switch of A of a UDN.

Algorithm CONNECTION-2
step 0: in-degree(v) = 0 for all $v \in B_0$ and $A' = \phi$.
step 1: If $A = \phi$, done.
step 2: If $|B_0| = 1$, let $B_0 = \{v\}$, and if $v = U(u)$, $u \in A$, connect $u' \in A'$ to $v \in B_0$ such that $v \ne U(u')$, disconnect $u' \in A'$ and $F(u')$, and connect $u \in A$ to $F(u')$; in-degree(v) = in-degree(v) +1; go to step 4.
step 3: Connect $u \in A$ to $v \in B_0$ such that $v \ne U(u)$ and in-degree(v) = in-degree(v) +1.
step 4: $A = A - \{u\}$, $A' = A' + \{u\}$ and if in-degree(v) = 2, $B_0 = B_0 - \{v\}$; go to step 1.

Algorithm CONNECTION-2 has a more stringent constraint for selecting 0-links than that for algorithm CONNECTION-1. It is prohibited from choosing the same successor of a switch in the UDN as the conjugate 0-successor of the switch so that each switch is connected to two distinct 0-successors in the FDN. Notice that when there is only one switch v left in B_0, there may have one or two switches left in A and v may also be the successor of the switches left in A in the UDN; thus the only possible connection is to join the switches left in A and the same successors of theirs in the UDN (i.e. v). Step 2 adopts a remedial approach to exchange the connection with the one that was established in a previous iteration and whose switch of B_0 is not a successor of the switches left in A in the UDN. The exchange is always possible from stage 0 to $n-2$ because (a) if there are two switches left in A and they are the predecessors of v in the UDN, then any of the $(2^{n-i}-2)$ switches in A can be chosen for the exchange; and (b) if there is only one switch left in A (so there are $2^{n-i}-1$ switches in A) and it is also the predecessor of v in the UDN, then there must exist another switch in A that is also a predecessor of v in the UDN (for the out-degree of every switch of B_0 is two in a connection pattern of the UDN), and any of the $(2^{n-i}-1)-1$ switches in A can be chosen for the exchange. At the last stage ($i = n-1$), $|A| = 2$ and $|B_0| = 1$, so both conjugate 0-links and 1-links of a switch in A are connected to the same output switch (destination).

Assume that at the last iteration, the last switch v left in B_0 is always the successor of the switches left in A in the UDN, then at each iteration a switch in A can choose only one of the switches in $B_0 - \{v\}$ as its successor. Also assume that at each iteration of the algorithm for which, for each switch u in A, B_0 always contains $U(u)$, then a switch in A can choose only one of the switches in $B_0 - \{v\} - \{U(u)\}$ as its successor. Using the arguments similar to those used for computing the total number of possible UDN topologies, it can be shown that there exist at least $\prod_{i=0}^{n-2}((2^{n-i}-2)2^{2^{n-i}-1})$ possible FDN topologies for a given UDN.

Examples of FDN's are the F network[1] and the Kappa network[*]. The fault-tolerant D-construct expressions for these two networks are as follows.

(i) For the F network, $p_{n-1,0} = u_{n-1,i+1}0u_{i-1,0}$, $\bar{p}_{n-1,0} = \bar{u}_{n-1,i+1}0u_{i-1,0}$, $q_{n-1,0} = u_{n-1,i+1}1u_{i-1,0}$, and $\bar{q}_{n-1,0} = \bar{u}_{n-1,i+1}1u_{i-1,0}$.

(ii) For the Kappa network, $p_{n-1,0} = u_{n-1,i+1}0u_{i-1,0}$, $q_{n-1,0} = u_{n-1,i+1}1u_{i-1,0}$, and if $u_i = 0$, $\bar{p}_{n-1,i+1} = u_{n-1,i+1}+2^{i+1}$, $\bar{p}_{i,0} = 0u_{i-1,0}$, $\bar{q}_{n-1,i+1} = u_{n-1,i+1}-2^{i+1}$, and $\bar{q}_{i,0} = 1u_{i-1,0}$, and if $u_i = 1$, $\bar{p}_{n-1,i+1} = u_{n-1,i+1}-2^{i+1}$, $\bar{p}_{i,0} = 0u_{i-1,0}$, $\bar{q}_{n-1,i+1} = u_{n-1,i+1}+2^{i+1}$, and $\bar{q}_{i,0} = 1u_{i-1,0}$.

The Kappa network was derived as a result of the comparison between the topologies of the Gamma[2] and the modified Baseline[8] networks. By embedding the modified Baseline network in the Gamma network, the redundancy in the Gamma network was observed and the addition of an extra link to every switch of the Gamma network was proposed to achieve symmetric redundancy and one-fault-tolerance. By decomposing the fault-tolerant tree structures and transforming its topology describing rule into D-construct expressions, the necessary fault-tolerance in the Gamma network becomes evident (i.e. some switches have nonconjugate links), and at least $\prod_{i=0}^{n-1}((2^{n-i}-1)!)^{2^{n-1}}$ fault-tolerant topologies (by making all switches have conjugate 0-links and 1-links) can be found. The Kappa network is merely one.**

There are two possible implementations for routing schemes for FDN's and they are discussed in the next two paragraphs. The first scheme requires no computation for rerouting tags and can dynamically bypass faults in the network. The second scheme requires the computation for rerouting tags by the source of messages but has the advantage of being capable of handling more complicated multiple faults in the network.

In the first scheme the destination address can be used directly as the routing tag; i.e. $t_{n-1,0} = d_{n-1,0}$. This scheme assumes that each switch u^i uses (u^i, p^{i+1}) and (u^i, q^{i+1}) as the default 0-link and 1-link, respectively, for routing messages. If $t_{k_i} = 0$, link (u^i, p^{i+1}) is used for routing and if $t_{k_i} = 1$, link (u^i, q^{i+1}) is used for routing. Links $(u^i, \bar{p}^{i+1})$ and $(u^i, \bar{q}^{i+1})$ are regarded as spare links. If a fault in the default link is detected or the link is blocked due to failure in the switch to which the link is an input link, the switch automatically reroutes messages via the conjugate link of the blocked link. Thus rerouting is transparent to the source of the messages and no rerouting tags need to be computed. Each switch requires negligible extra hardware for the detection of blocked links.

The second scheme requires the source of messages to compute a routing tag. The source is assumed to know the locations of faulty links and switches in the network so that they can be compared against the links and switches on the routing path to decide whether faults occur on the routing path and rerouting is necessary. A routing tag consists of n digits, the k_i-*th* digit of the routing tag being represented by two bits t_{n+k_i} and t_{k_i}, called the *state bit* and the *destination bit*, respectively. The destination bit specifies the use of a 0-link or 1-link for routing and $t_{k_i} = d_{k_i}$, $0 \le k_i \le n-1$ and $0 \le i \le n-1$. If $t_{k_i} = 0$, a 0-link is used and if $t_{k_i} = 1$, a 1-link is used. The state bit is used to specify which one of the

---

*Although other fault-tolerant topologies for the Gamma network are also possible based on the comparison of topologies[4,9], the number of them is less than that derived by using the approach proposed in this paper. This is because the block structure[9] is only a subset of the binary tree structures, as explained in Section 2.

conjugate 0-links (or 1-links) is used for routing. If $t_{n+k_i} = 0$, messages are routed to p^{i+1} or q^{i+1} and if $t_{n+k_i} = 1$, messages are routed to $\hat{p}^{i+1}$ or $\hat{q}^{i+1}$. In general, if $t_{n+k_i}t_{k_i} = 00$, link (u^i, p^{i+1}) is used for routing; if $t_{n+k_i}t_{k_i} = 10$, link $(u^i, \hat{p}^{i+1})$ is used for routing; if $t_{n+k_i}t_{k_i} = 01$, link (u^i, q^{i+1}) is used for routing; if $t_{n+k_i}t_{k_i} = 11$, link $(u^i, \hat{q}^{i+1})$ is used for routing. Because the destination bits decide the message destination, it always remains unchanged. Only the state bits need to be changed for rerouting. Control of the rerouting can be done by complementing the corresponding state bit, which effectively sends messages via the conjugate link of the blocked link.

Because the F network belongs to the class of FDN, its routing can be controlled by the first and second routing schemes proposed in the paper. These schemes are more efficient than the previous algorithm[4], which requires complicated recursive procedure to compute routing tags.

Since any link on a given path results from the appropriate choice of the corresponding state bit, each path in the FDN can be specified by a routing tag computed by using the second scheme. Therefore, the second scheme can always find a fault-free path in the FDN, if such a path exists. That is, the second scheme can deal with more complicated faults that cannot be avoided by using the first scheme. For example, if the two 0-successors of a switch of stage i on some routing path are both faulty, it is impossible to reroute messages in the first scheme. However, if the conjugate switch of the switch that is the predecessor of the two 0-successors is not connected to the same successors, rerouting can be done by sending messages through the conjugate switch of stage i, which is on a path that avoids the two faulty 0-successors.

6. CONCLUDING REMARKS

The main contribution of this paper is the derivation of a systematic approach to construct network topologies that can be controlled by destination tag routing. It can be used for the construction of fault-tolerant network topologies as well as unique-path ones. The essence of the approach lies in the decomposition of the structures of UDN's into binary tree structures and in the fact that new DN topologies can be derived by coalescing binary trees. While past fault-tolerant techniques are applied to a complete network topology, this approach allows for the enhancement of underlying structures of networks and uses the enhanced structures to construct fault-tolerant topologies. In this sense, the approach presented in this paper has a finer granularity than previous ones.

This philosophy is reflected in the study of ICube and Gamma networks. While it was previously shown that the Gamma network can be considered a superset of the ICube network and the Kappa network was derived as a fault-tolerant version of the Gamma network, by decomposing the ICube and Gamma networks into binary and fault-tolerant tree structures, it is easily realized that the ICube network is a subset of the Gamma network and at least $\prod_{i=0}^{n-2}((2^{n-i}-2)!)^{2^i}$ fault-tolerant topologies for the Gamma network can be derived. In addition, more efficient routing schemes are derived for the Gamma and the F networks.

This paper considers only the addition of one and two extra links to the underlying binary trees of UDN's in order to enhance the UDN's fault-tolerance capabilities. It is certainly possible to add more extra links to achieve higher degree of fault-tolerance. As for the cases of EDN's and FDN's, it is desirable to have pairs of conjugate 0-links and 1-links for each switch. For every two extra links (one 0-link and one 1-link) added to every switch of a DN, one more switch fault-tolerance can be achieved.

The building blocks for constructing DN's in this paper are the binary trees and their fault-tolerant versions and the number of input/output switches of the DN's are assumed to be a power of 2 ($N = 2^n$). Similar ideas can be applied to generalized b-ary tree structures. A generalized network can be constructed by coalescing $N = b^n$ b-ary trees, its number of input/output switches is a power of b, labeling of switches in the network can be represented in radix b, and its routing can then be controlled by the destination tags represented in radix b. This possible extension is now under investigation.

This paper experiments on the addition of extra links to the underlying binary trees of a UDN to improve its fault-tolerance capability, which, due to the modularity requirement, correspond to the fault-tolerant technique of adding extra links to the UDN. There are other fault-tolerant techniques such as adding extra input/output switches[7], extra stage[1], etc. The structures of these fault-tolerant topologies may also be explained under the scope of binary and fault-tolerant tree structures. In addition, the results presented in this paper can be explained and extended in terms of a generalization of a network state model[15]. This extension is presented in a forthcoming paper.

7. ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation under Grant DCI-8419745 and by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contract No. 00014-85-k-0588.

8. REFERENCES

1. G. B. Adams III and H. J. Siegel, "The extra stage cube: a fault-tolerant interconnection network for supersystems," *IEEE Trans. Computers*, Vol. C-30, No. 5, May 1981, pp. 443-454.
2. D. Agrawal, "Graph theoretical analysis and design of multistage interconnection networks," *IEEE Trans. Computers*, Vol. C-32, No. 7, July 1983, pp. 637-648.
3. K. E. Batcher, "The flip network in STARAN," *1976 Int'l Conf. Parallel Processing*, Aug. 1976, pp. 65-71.
4. L. Ciminiera and A. Serra, "A connecting network with fault tolerance capabilities," *IEEE Trans. Computers*, Vol. C-35, No. 6, June 1986, pp. 578-580.
5. T-Y Feng, "Data manipulating functions in parallel processors and their implementations," *IEEE Trans. Computer* Vol. C-23, No. 3, Mar. 1974, pp. 309-318.
6. L. R. Goke and G. J. Lipovski, "Banyan networks for partitioning multiprocessor systems," *1st Ann. Symp. Computer Architecture*, Dec. 1973, pp. 21-28.
7. M. Jeng and H. J. Siegel, "A fault-tolerant multistage interconnection network for multiprocessor systems using dynamic redundancy," *6th Int'l Conf. on Distributed Computing Systems*, May 1986, pp. 70-77.
8. S. C. Kothari, G. M. Prabhu and Robert Roberts, "The kappa network with fault-tolerant destination tag algorithm," Technical Report #85-20, Department of Computer Science, Iowa State University, Iowa, July 1985.
9. S. C. Kothari, G. M. Prabhu and R. Roberts, "Fault-tolerant strategies for networks with block structure," Technical Report #85-30, Department of Computer Science, Iowa State University, Iowa, Nov. 1985.
10. D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Computers*, Vol. C-24, No. 12, Dec. 1975, pp. 1145-1155.
11. R. J. McMillen and H. J. Siegel, "Routing schemes for the augmented data manipulator network in an MIMD system," *IEEE Trans. Computers*, Vol. C-31, No. 12, Dec. 1982, pp. 1202-1214.
12. R. J. McMillen and H. J. Siegel, "Performance and fault tolerance improvements in the inverse augmented data manipulator network," *9th Ann. Symp. Computer Architecture*, Apr. 1982, pp. 63-72.
13. D. S. Parker and C. S. Raghavendra, "The Gamma network: a multiprocessor interconnection network with redundant paths," *9th Ann. Symp. Computer Architecture*, Apr. 1982, pp. 73-80.
14. M. C. Pease, III, "The indirect binary n-Cube microprocessor array," *IEEE Trans. Computers*, Vol. C-26, No. 5, May 1977, pp. 458-473.
15. D. Rau, J. Fortes and H. J. Siegel, "Destination tag routing techniques based on a state model for the IADM network," Technical Report TR-EE 87-39, School of Electrical Engineering, Purdue University, Oct. 1987.
16. H. J. Siegel and S. D. Smith, "Study of multistage SIMD interconnection networks," *5th Ann. Symp. Computer Architecture*, Apr. 1978, pp. 223-229.
17. C-L. Wu and T-Y. Feng, "On a class of multistage interconnection networks," *IEEE Trans. Computers*, Vol. C-29, No. 8, Aug. 1980, pp. 694-702.

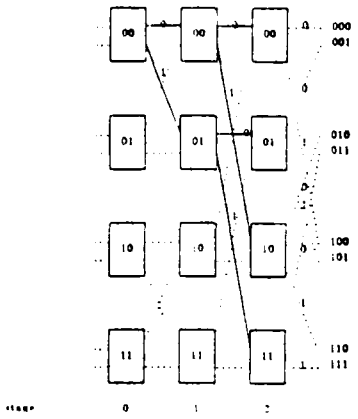


Figure 1 A binary tree exists in an 8×8 ICube network that includes the source $(00)^0$ as the root of the tree and $(00)^1$, $(01)^1$, $(00)^2$, $(01)^2$, $(10)^2$ and $(11)^2$ as the nodes of the tree. The bold links manifest their interconnection.

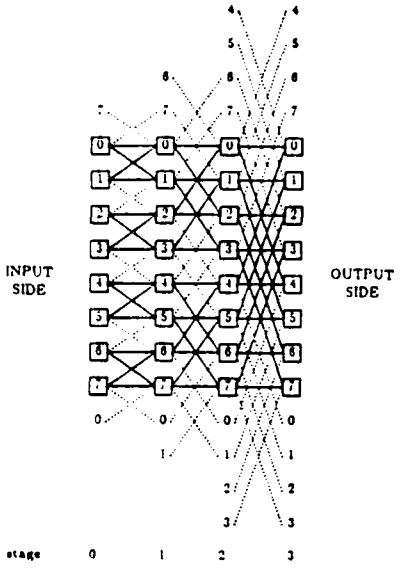


Figure 2. The Gamma network for $N = 8$. The dotted links are the links of the ICube network and the solid links are the extra links. The bold links manifest their interconnection.

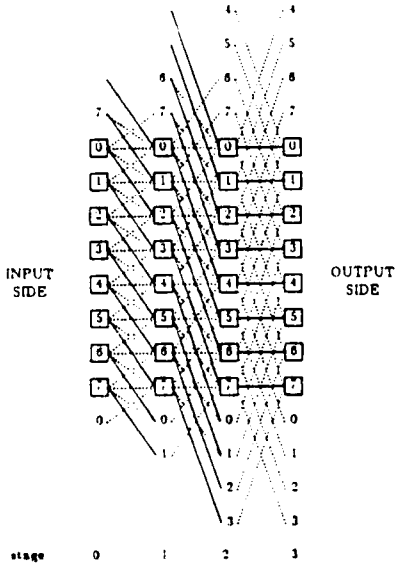


Figure 3. The Kappa network for $N = 8$. The dotted links are the links of the Gamma network and the solid links are the extra links. The solid links of stage 2 are overlapped with dotted links.

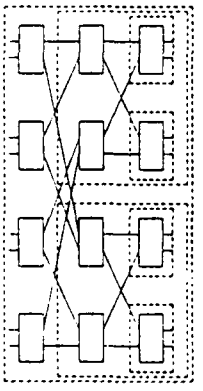


Figure 4 An 8×8 baseline network with block structure. A dashed box encloses a sub-block.

REFERENCE NO. 11

Wang, Y-X. and Fortes, J. A. B., "Hierarchical Approaches to Fault-Tolerance in Processor Arrays," Symposium on Innovative Science and Technology, January 1988.

Hierarchical Approaches to Fault Tolerance in Processor Arrays

Y.A. Wang and Jose A. B. Fortes
School of Electrical Engineering, Purdue University
West Lafayette, IN 47907

ABSTRACT

Because processor arrays have only limited connections between neighboring processors, fault-tolerance schemes may require additional interconnect, switching and control hardware in order to allow for reconfiguration when faults occur. In general, the larger the reconfiguration capability, the greater is the probability that a processor array can survive a given distribution of faults. In other words, the coverage of the reconfiguration procedure increases directly with the amount of extra hardware required to support it. However, this is true only if the added hardware does not fail itself. For this reason and depending on, among other factors, the size of the processor array and the size of each processor, distinct reconfiguration schemes may be best suited for different arrays. Also, in general, previously proposed schemes may still result in unacceptably low reliabilities in very large processor arrays.

This paper proposes a class of reconfiguration schemes which have a hierarchical nature. According to this approach, a processor array is logically partitioned into smaller subarrays and, once faults occur, reconfiguration takes place within each of the subarrays (where faults are present) if possible and, otherwise, the full subarray is replaced by a spare subarray. Arrays of this type are referred to as bi-level fault-tolerant processor arrays and, by allowing several levels of reconfiguration, multi-level arrays can be defined similarly. While several levels of reconfiguration are possible, the case of two levels is emphasized in this paper. Also, the reconfiguration schemes used in each level are not necessarily identical. This class of hierarchical reconfiguration schemes provide much higher reliability than previously proposed ones, particularly in the case of very large arrays.

To design a hierarchical reconfiguration scheme for a given processor array it is necessary to choose the size of the subarrays for every level in the hierarchy as well as the reconfiguration scheme at that level. A design methodology is provided which mathematically solves these problems, i.e. it enables the choice of the subarrays size and the reconfiguration scheme to be used at each level so to obtain a processor array with optimal reliability.

1. INTRODUCTION

In a fault-tolerant processor array (FTPA) redundancy may or may not be provided for every type of component of the array. As a typical possible case, an FTPA may have spare processors but no redundancy provided for links, switches, and control logic to implement the fault-tolerance scheme. This paper discusses the impact of *non redundant hardware* on the reliability of an FTPA and shows that FTPA's with superior reliability result if the fault-tolerance scheme used has a *hierarchical structure*. *These FTPA's are called multi level FTPA's and a methodical procedure* for the design of optimally reliable MLFTPA's is also described in this paper.

The relevance of processor arrays (PA's) stems not only from their ability to meet the computational demands of many real-time applications but also from their suitability for efficient implementations using Very Large Scale Integration/Wafer Scale Integration (VLSI/WSI) technology[1]. Large PA's can be implemented by replicating many times a small basic hardware module which might include not only the processors but also some hardware for switching, control and interconnections. The design of the basic module can be effectively optimized and, therefore, it can be used to implement area- and time-efficient arrays of arbitrary size with a lower cost. In this paper, it is assumed that a basic module *corresponds to a single processor*. However, since not all interconnect, switching and control logic can be implemented in modules, the results of this paper apply also when other types of modules are used. Assume the reliability of each module is r_m, the reliability of PA with N modules is r_m^N which can be rather low when N is very big. This is equivalent to the statement that, regardless of the reliability per unit of area of the PA, the reliability of a PA with large area is very small, unless fault-tolerance is provided. From this perspective, it becomes clear that the first statement is true independently of the type of module used since what really matters is the overall amount of area taken by the array and, assuming no fault-tolerance, the reliability of a large array with small modules is the same as that of an array with the same area but larger modules. Thus, reliability prevents the exploitation of two of the main advantages of processor arrays, i.e., modularity and extensibility.

Extensive work has been done towards devising FTPA designs and a non-exhaustive list of references is [2-11] and the references therein. Reconfiguration approaches that *resemble* those described in some of these references are used to illustrate the basic ideas and results of this paper. These methods include Column Redundancy ([3-6] and others), Diogenes [5], Complex Fault Stealing [7] and Triplicated Modular Redundancy [8]. Succinct descriptions of the approaches that resemble these methods are made in Section 3 of this paper.

Typically, previously proposed FTPA designs assume the non-redundant hardware, i.e., the links, switches and control logic never fail during array operation [10-11]. It is assumed that these components are small and simple and can therefore be designed carefully and conservatively so to minimize the probability of operational failure. Historically, this approach finds its origins in the nature of earlier electronic systems where, due to a less reliable technology and the large size of logic devices, bulky active logic was much more likely to fail than smaller and easier-to-implement passive connections and simple switches. Another justification for the validity of the "fault-free interconnect" assumption is the belief that the reliability of interconnect can always be improved by replicating "inexpensive" wires whereas more elaborate

schemes with limited redundancy are required for the "expensive" logic. With some current technologies the same type of assumptions may still be valid in some cases, e.g., fabrication faults in the wafer-scale integrated systems described in [18] occur with probability of 97% in wires versus 50% in processing elements (PE's) and reconfiguration mechanisms are 99.99% reliable. However, it is well known that these assumptions must be re-evaluated in light of available Very Large Scale Integration/Wafer Scale Integration (VLSI/WSI) technology, particularly when operational faults (instead of manufacturing defects) are to be considered [19]. In fact, in VLSI/WSI systems, interconnect can take up to 50% of the total area and, in simple terms, it is made by using processes and materials similar to those used for active logic. Thus, it is more appropriate to characterize the lifetime reliability of different types of components of a system in terms of the area that they require and the reliability per unit of area of the medium used for their implementation.

The understanding of the impact of the operational reliability of non-redundant hardware in the overall operational reliability of an FTPA is one of the goals of this paper. Similar studies have been done for other type of systems (for example, in[20, 21]) but, to our knowledge, they have not been reported for FTPA's. The idea of considering multi-level FTPA's is a consequence of the understanding just mentioned, in the sense that the hierarchical structure of these FTPA's reduces the amount of non-redundant hardware in the overall array. It is appropriate to mention here that related or similar concepts have been independently proposed before in [10-13]; however, the motivations for putting forward those ideas, while valid, seem to be different from the reasons presented in this paper.

Section 2 of this paper presents the basic ideas and models that underly the approach and studies reported in this paper. In Section 3, the reliability characteristics of single-level FTPA's are discussed. Four different type of FTPA's are described and area and reliability estimates for these FTPA's are presented and discussed. The approach used to compute these estimates is also explained. Section 4 introduces the concept of multi-level FTPA's and provides reasons why their reliability is potentially higher than for other single-level FTPA's. The issue of how to design optimal bi-level FTPA's is the topic of Section 5. It is shown that it is not feasible to attempt to derive optimal design parameters directly from the analytical expressions that describe the reliability of a bi-level FTPA. To solve this problem, accurate functional approximations of those expressions are proposed and used for optimization purposes. A case study is briefly described that illustrates the very high reliability improvement achievable by bi-level FTPA's in contrast with the very poor reliability possible with a single-level FTPA. Section 6 is dedicated to conclusions.

2. BASIC IDEAS AND MODELS

A very general model is used to represent characteristics of FTPA's that are relevant to the purpose of this paper. A processor array is described by a 4-tuple (P, L, S, C) where P corresponds to the part of the array used for processing, L denotes all the link components in the array, S is the set of switching components, and C denotes the set of control logic components responsible for the control of processing, linking and switching elements. Clearly, in order for an array to be fully operational, each of these component parts of an array must operate correctly. In general, a component can belong to only one of the sets P, L, S and C. However, in order to introduce fault-tolerance in the component P of a PA=(P,L,S,C), a new array (P',L',S',C') is obtained where not only P' but also L', S' and C' result from adding elements to P, L, S and C, respectively. Consequently, the reliability of L', S' and C' may also differ from that of L, S and C. The difference in reliability depends on both the added elements and the logical and physical organization of the components of the array.

Without loss of generality, assume that an FTPA = (P,L,S,C) contains $(n\times n)+k$ PE's, where k corresponds to the number of spares. Let $A_p(n,k)$, $A_l(n,k)$, $A_s(n,k)$ and $A_c(n,k)$ denote the areas used to implement P, L, S and C, respectively, and let $A(n,k) = A_p(n,k) + A_l(n,k) + A_s(n,k) + A_c(n,k)$ denote the total area of the array. Let r_p, r_l, r_s, and r_c denote the reliability of the unit of area of an element in P, L, S and C, respectively. Similarly, let $R_p(n,k)$, $R_l(n,k)$, $R_s(n,k)$ and $R_c(n,k)$ refer to the reliability of P, L, S and C. Then the reliability of a processor array is given by

$$R(n,k) = R_p(n,k) \cdot R_l(n,k) \cdot R_s(n,k) \cdot R_c(n,k) \qquad (2.1)$$

and the overall reliability of the array is at most as good as the lowest of the reliabilities of P, L, S and C.

In general, proposed reconfiguration schemes adopt as the basic replacement unit a module and assume that the hardware outside the basic module (i.e., the non-redundant hardware) is fault-intolerant. This suggests that the model just proposed can be generalized further. In essence, the area of a FTPA consists of some fault-tolerant area $A_{ft}(n,k)$ and the remaining area $A_{fi}(n,k) = A(n,k) - A_{ft}(n,k)$ which is fault-intolerant. Denoting the reliability of these areas by $R_{ft}(n,k)$ and $R_{fi}(n,k)$, respectively, the reliability of the FTPA is

$$R(n,k) = R_{ft}(n,k) \cdot R_{fi}(n,k) \ . \qquad (2.2)$$

In other words, given an FTPA with $(n\times n)+k$ modules (where k is the number of spares and the unit of area is defined as the area of a module) and letting c_i denote the probability that the FTPA can recover from the failure of i modules, $i \le k$, equation (2.2) becomes

$$R(n,k) = r_{fi}^{A_{fi}(n,k)} \cdot \sum_{i=0}^{k} c_i \binom{n^2+k}{i} r_{ft}^{(n^2+k-i)}(1-r_{ft})^i \qquad (2.3)$$

Expressions (2.2) and (2.3) are valid for any type of FTPA. Without loss of generality and in order to facilitate the presentation and discussion of the ideas and results, several assumptions apply to the remainder of this paper and are described next. Given any FTPA (P,L,S,C), it is assumed that, (1) P is fault tolerant and L,S and C are fault-intolerant, i.e., $A_{fi}(n,k) = A_l(n,k) + A_s(n,k) + A_c(n,k)$, $R_{fi} = R_l(n,k)\cdot R_s(n,k)\cdot R_c(n,k)$, $A_{ft}(n,k) = A_p(n,k)$, $r_{ft} = r_p$, and $R_{ft}(n,k) = R_p(n,k)$; (2) at the exception of the reconfiguration scheme used, all other procedures required for fault tolerance are perfect (e.g., fault detection and location) and, therefore, c_i in equation (2.3) corresponds to the probability of

successful reconfiguration given that i faults occurred.

From equation (2.3) it is possible to infer important characteristics that apply to all FTPA's. In particular, it is of interest to answer the questions of how $R(n,k)$ varies when more spare PE's are added to the FTPA so that (a) the ratio between the number of processors n^2 and the number of spares k stays constant, in other words, the *redundancy ratio* $\eta=(n^2+k)/k$ stays constant or (b) the number of processors n^2 remains constant. To answer these questions, the variations of $R_R(n,k)$ and $R_D(n,k)$ with increases in k are first analyzed. Because $R_D(n,k) = r_0^{M(n,k)}$, it is clear that this term decreases in both cases. With respect to $R_R(n,k)$, the summation term in (2.3) it is useful to consider the special case when c_i equals a constant c_0 for all $1 \le i \le k$ and use the DeMoivre-Laplace limit theorem[12] which states that, for sufficiently large n^2+k, $\frac{1}{c_0} R_R(n,k)$ can be approximated by the Normal Gaussian distribution function $\Phi(x(n,k))$ where

$$x(n,k) = \frac{k - (n^2+k)(1-r_p)}{\sqrt{(n^2+k)r_p(1-r_p)}}$$

For large k, (which implies that n^2+k is also very large) $\Phi(x(n,k))$ can be approximated as

$$\Phi(x(n,k)) \approx 1 - \frac{1}{x(n,k)\sqrt{2\pi}\, e^{\frac{x(n,k)^2}{2}}} \qquad (2.4)$$

It is clear that $R_R(n,k) > R_R(n,k-1)$. However, in first question, (e.g., $(n^2+k)/k = ((2n)^2+4k)/4k$) for sufficiently large n and k ($(n^2+k) >> (1-r_p)^{-1}$), $R_R(2n,4k)$ is negligibly larger than $R_R(n,k)$ since, using (2.4), the following ratio approaches 1 for large n and k

$$\frac{R_R(2n,4k)}{R_R(n,k)} \approx \frac{\Phi(x(2n,4k))}{\Phi(x(n,k))} \approx 1 \qquad (2.5)$$

This means the overall reliability $R(2n,4k) = R_R(2n,4k)R_D(2n,4k)$ will go down for n and k sufficiently large because $R_D(2n,4k) < R_D(n,k)$. To answer the second question above, consider an increment on the k with n remaining constant, using approximation (2.4) again, when n and k are sufficiently large,

$$\frac{R_R(n,k+1)}{R_R(n,k)} \approx \frac{\Phi(x(n,k+1))}{\Phi(x(n,k))} \approx 1 \qquad (2.6)$$

This indicates that increasing the number of spares above a certain number does not necessarily result in reliability improvement, because the resulting reduction in $R_D(n,k)$ may be more significant that the improvement in $R_R(n,k)$. It can be concluded now, a very large FTPA may result a unacceptable reliability even using the same redundancy ratio as a small FTPA, and an extended version of an FTPA where the redundancy ratio is preserved may have less reliability than the original FTPA since a reduction in $R_D(n,k)$ may be more significant than a possible improvement in $R_R(n,k)$.

The next section discusses single-level FTPA's and considers several examples which confirm and make more concrete the conclusions reached above.

3. RELIABILITY OF SINGLE-LEVEL FTPA'S

This section considers four single-level FTPA designs, discusses how their reliabilities are estimated and analyzes how these estimates vary with the *size of the array and* the number of spares. Four different FTPA design approaches, denoted CR, DI, CFS and TMR are considered[5-11]. It is not important that the schemes described in this paper, do or do *not correspond* exactly to the descriptions of similar approaches in the references just mentioned. The basic ideas and results of this paper are generally valid regardless of possible variations on how the schemes proposed in those references are interpreted and implemented.

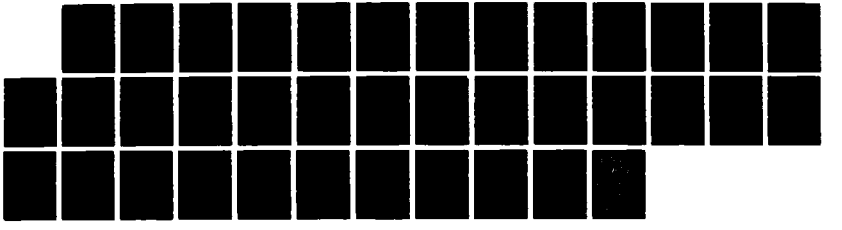
In an FTPA using the CR approach, one or more columns of spare processors are provided and upon occurrence of a fault in a PE in a given column of the array, the complete column is replaced by a spare column. The reconfiguration succeeds if and only if faults occur in a number of columns less than or equal to the number of spare columns. Switches and additional links are required for each processor so that a full column can be bypassed. A single control signal is needed to set each column of switches; this signal is local with respect to each column but global to all processors in the same column. Additional switches are needed to "align" the input and output ports with the processors. These switches require individual control signals which are global (i.e., they are not local's).

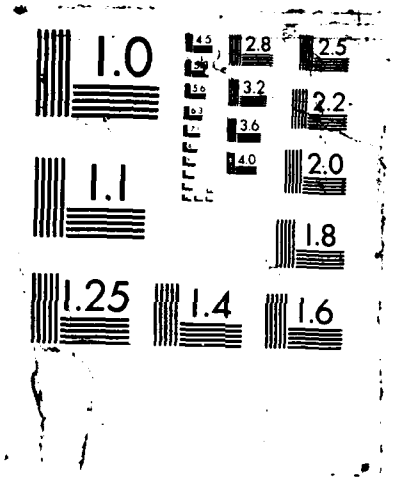
An FTPA using the DI approach is physically laid out as a linear array of processors above the full array. The desired configuration (in this paper, an orthogonal array) is obtained through switches that appropriately connect processors to wires. After the occurrence of faults a new setting is determined for the switches and the array is reconfigured. Until redundancy is exhausted, every processor is connected to four wires to implement an orthogonal array. However, all switches associated with a processor. However, all switches associated with a processor are set identically. In the *Diogenes approach*[5] there are also individual control signals to indicate whether a processor is faulty or not (since all processors are identical). (This is not necessary in the DI approach.)

An FTPA using the CFS approach consists of processors. In addition,

AD-A190 918 DESIGN AND EVALUATION OF FAULT-TOLERANT VLSI/WSI PROCESSOR ARRAYS(U) PURDUE UNIV LAFAYETTE IN J A FORTES 31 DEC 87 N00014-85-K-0580 3/3

UNCLASSIFIED F/G 12/6 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5

column and each row. Two extra busses and corresponding switches are added to allow for the "alignment" of input and output ports with operational I/O processors. All switches are individually controlled according to a distributed algorithm. Switch control signals are generated by neighboring logic whose inputs are originated from similar control logic associated with four next- neighbor processors. The reconfiguration algorithm is successful as long as the number of faults is not very close to the number of spares and, otherwise, the success rate decays as the number of faults increase.

In an FTPA using the TMR approach, each processor is triplicated and its outputs are voted. This corresponds to a static form of redundancy and there is no reconfiguration. Therefore, the only extra hardware added includes the voting logic and the minimal wiring required to link it to three processors. This "naive" TMR approach is not to be confused with the more elaborate approach proposed in[9].

A precise estimate of the areas required by FTPA's is possible only if their designs are carried to completion. However, the conclusions of this section do not depend on extremely accurate area estimates because relative rather than absolute information about area is sought. In particular, it is the goal of this section to illustrate (a) how the reliability of each FTPA design varies with the array size and the number of spares used and (b) how the reliability of different FTPA designs compare with each other for different array sizes. Thus, it suffices to guarantee that the rules and assumptions used to estimate area are consistent across different FTPA designs and capture the variation of reliability with array size and number of spares. These rules and assumptions are described next.

The unit of the area is defined as the area of a PE, i.e., $A_p(n,k) = n^2+k$ = total number of PE's in the FTPA. The total area taken by links is $A_l(n,k) = a_l \sum_{i=1}^{n_l} l_i$, where n_l is the number of links, l_i is the length of the ith link and a_l is the area of a link with unit length (a_l can also be thought of as the wire width.) The length of the links depends on the geometry of the FTPA layout. Assuming that PE's correspond to points in the Cartesian plane, the length of a wire is assumed to be the Manhattan distance between the source and destination of the wire. The total area taken by switches is $A_s = a_s \sum_{i=1}^{n_s} in_i \times out_i$ where n_s is the number of switches, in_i and out_i correspond to the fan-in and fan-out of the ith switch, respectively, and a_s is the area of a switch for which $in_i = out_i = 1$.

The area taken by control circuitry consists of the area taken by the logic that generates the control signals and the area taken by the links that propagate those signals. In the case of locally generated control signals the area taken by links is assumed to be zero. Also, it is assumed that the area of the logic necessary to control a switch is proportional to the size of the switch. Then, $A_c = A_{clocal} + A_{cglobal}$, where $A_{clocal} = a_c \sum_{i=1}^{n_{sc}} in_i \times out_i$, n_{sc} is the number of the switches with local control and a_c is the area of the smallest logic element (e.g., a 1×1 switch in which case $a_c = a_s$), and $A_{cglobal} = \sum_{i=1}^{n_{gs}} (a_c \times in_i \times out_i + a_l \times l_{ci})$, where n_{gs} is the number of global signals, in_i and out_i correspond to the fan-in and fan-out of the switches controlled by the ith global control signal and l_{ci} is the length of the wire used to propagate that signal.

Several area estimates were derived for FTPA's using CR, DI, CFS and TMR approaches, assuming different values for a_l, a_s and a_c and using different assumptions about the geometry of the layout. These area estimates and equation (2.3) were used to obtain reliability estimates assuming different values for $r_{ft} = r_p$, r_{fi}, r_c, r_s, and r_l. These estimates were then plotted as functions of the number of spares k and as functions of the number n^2 of processors in the logical array. Figures 1, 2 show some of these curves and additional ones are reported in[24]. Two general conclusions resulted from these studies:

(1) as predicted by the analysis done in section 2, the overall reliability of an FTPA increases as the number of spares increases up to a certain number and then starts to decrease due to the reliability of non-redundant hardware; this is clearly illustrated in Figure 1 where the reliabilities of the fault-tolerant and fault-intolerant areas (R_{ft} and R_{fi}, respectively) for an FTPA using the CR approach are plotted as function of the number of spare columns; in this particular example, the FTPA reliability (i.e., the product of R_{ft} and R_{fi}) shows no improvement as the number of spare columns is increased beyond 28 because R_{fi} decreases faster than R_{ft} increases after this value;

(2) The choice of an FTPA design with the best reliability depends on the size of the array, on the number of spares and other technological parameters (r_{fi}, r_p, a_c, a_s, etc.); in other words, FTPA's of different size may require different design approaches in order to achieve maximal reliability with a given technology, number of spares and layout geometry; this is illustrated in Figure 2 where CFS and DI approaches may or may not provide better reliability depending on the array size.

4. MULTI-LEVEL FTPA'S

The basic idea behind the design of a multi-level FTPA is best explained if the particular case of a bi-level FTPA is considered first. *A bi-level FTPA* consists of a fault tolerant array of FTPA's. In other words, the full array is partitioned into subarrays and can be thought of as an array of subarrays. Both the subarrays and the array of subarrays use some fault-tolerance scheme. The subarrays are hereon denoted as *1st-level FTPA*'s and the array of subarrays is referred to as the *2nd-level FTPA*. A 2nd-level FTPA can be thought of as an FTPA where the basic modules are themselves FTPA's and, physically, it is the same as the bi-level FTPA. The extension to multi-level FTPA's easily made by realizing that an n-level FTPA consists of an FTPA whose basic modules are (n-1)-level FTPA's which are FTPA's composed of (n-2)-level

FTPA's, etc. For convenience of presentation, bi-level arrays are assumed hereon and, unless stated otherwise, the basic ideas and results apply to multi-level arrays as well.

When faults occur in a bi-level array, reconfiguration is first attempted at the 1st-level FTPA's, and, when reconfiguration fails at the 1st-level, then reconfiguration at the 2nd-level FTPA is attempted. Intuitively, bi-level FTPA's can be expected to have better reliability than single-level FTPA's for two reasons: (1) the area of non-redundant (fault-intolerant) hardware in bi-level FTPA's is smaller than that in single level FTPA's and (2) the size of arrays and the reconfiguration approach used at each level can be chosen so that optimal reliability results, thus avoiding the inevitable reliability degradation that occurs when the size of single-level arrays grows too large.

Reason (1) can be restated in different terms in order to provide additional insight. One can think of the structure of "extra hardware" (non-redundant hardware, e.g., switches and control logic) in single-level FTPA's as a series "success" diagram[23] where individual extra hardware elements correspond to the modules of the diagram (not to be confused with the FTPA modules.) In this type of serial structure, a failure in a single module implies failure of the complete system. On the other hand, the structure of the extra hardware in a multi-level FTPA is such that the corresponding success diagram is a series-parallel diagram with as many stages as there are levels in the multi-level FTPA. Clearly, the failure of any module now results in the failure of the associated reliability paths [23] but not necessarily a total failure. In relation to reason (2) above, the advantage of being able to choose the size and fault-tolerance method used at each level is that it becomes possible to use the best fault-tolerance method for a given size of array or, given the method, find the array size which is the best, or both. In other words, the dependency on array size, discussed in section 3 as a disadvantage of single-level FTPA's, can be advantageously exploited in each level of multi-level FTPA's.

Having realized the potential benefits of multi-level FTPA's, from an engineering point of view, it is essential to have a systematic and formal approach to the design these systems so to optimize the reliability of the overall FTPA. This approach is described in the remainder of this paper.

5. OPTIMAL BI-LEVEL FTPA'S

For the purposes of this section, it is convenient to refer to the reliability of an FTPA as given by (2.3) as an explicit function not only of n and k but also of $r_p = r_{fp}$. Let the size of the 1st-level and 2nd-level FTPA's be $(n_1 \times n_1) + k_1$ and $(n_2 \times n_2) + k_2$, respectively. Here, k_1 and k_2 correspond to the number of spare processors in the 1st-level FTPA and the number of spare 1st-level FTPA's in the 2nd-level FTPA, and can be expressed as functions of n_1 and n_2, respectively. Also, the number of processors in the bi-level array is $(n \times n) + k$ where $n = n_1 \times n_2$ and $k = k_1 n_2^2 + k_2 (n_1^2 + k_1)$. The reliability of the bi-level FTPA is essentially the reliability of the 2nd-level FTPA, i.e., $R_2 = R(n_2, k_2, R_1)$ where R_1 is the reliability of one 1st-level FTPA, i.e., $R_1 = R(n_1, k_1, r_p)$. In order to find the values of n_1 and n_2 for which R_2 is optimal it is necessary to solve the equation

$$\frac{dR_2}{dn_2} = \frac{\partial R_2}{\partial n_2} + \frac{\partial R_2}{\partial R_1} \cdot \frac{\partial R_1}{\partial n_2} + \frac{\partial R_2}{\partial k_2} \cdot \frac{\partial k_2}{\partial n_2} = 0 \tag{5.1}$$

Once the solution n_2^* is obtained, n_1^* can be found because n is assumed to be known. Unfortunately, for even the simplest bi-level FTPA design, equation (5.1) is very hard to solve. An example illustrating the "impossibility" of this approach for a simple case is discussed in [24]. In order to deal with this problem, a new approach is proposed here which uses very good functional approximations of R_1 and R_2 that are explained next.

The type of function chosen to approximate R_1 as a function of n_1 is the well known Weibull reliability function, i.e.

$$R_1 = e^{-\alpha' n_1^{\beta}} \tag{5.2}$$

where α' and β are real positive constants whose value depends on, among other characteristics of the FTPA, the reconfiguration method used. With this type of approximation it has been possible to approximate with negligible errors all the reliability estimates discussed in section 3 that have been attempted so far. While it is not easy to show analytically that R_1 (given by equation 2.3) can always be approximated by a Weibull function, the general characteristics of the curves obtained for R_1 and the experience gained so far indicate that it is highly likely that this is possible in general[23].

The approximation used for R_2 as a function of n_2 is more involved than that used for R_1 as a function of n_1. This is because R_1 is approximated as a function of n_1 alone (the remaining variables are functions of n_1 or fixed) while R_2 must be approximated as a function of n_2 and R_1 (because, for the 2nd-level FTPA, $r_{fp}=R_1$ is a function of $n_2 = n/n_1$). From (2.2) and (2.3) it results that R_2 can be expressed as

$$R_2 = R(n_2, k_2, R_1) = R_g(n_2, k_2) R_{fl}(n_2, k_2, R_1) \tag{5.3}$$

where

$$R_g(n_2, k_2) = r_g^{A(n_2, k_2)} \tag{5.4}$$

$$R_{fl}(n_2, k_2, R_1) = \sum_{i=0}^{k_2} c_i \binom{n_2^2 + k_2}{i} R_1^{(n_2^2 + k_2 - i)} (1 - R_1)^i \tag{5.5}$$

The form of the approximation function for (5.4) has also been chosen to be that of a Weibull reliability function, i.e.,

$$R_{fi}(n_2,k_2) = r_{fi}^{a' n_2^b} = e^{(\ln r_{fi}) a' n_2^b} = e^{-a n_2^b} \tag{5.6}$$

where a' and b are some positive real constants, and $a = (\ln r_{fi}) \cdot a'$. From (5.4) and (5.6), it is clear that this amounts to approximating $A_{fi}(n_2,k_2)$ by $a \cdot n^b$, i.e., the area of fault-intolerant hardware is assumed to grow proportionally with some power of the number of processors in the array. Such an approximation may not suffice when $A_{fi}(n_2,k_2)$ varies in a more complicated way. However, in the worst case, it can always be approximated by a polynomial expression on n, in which case $R_{fi}(n_2,k_2)$ would be approximated by a product of Weibull functions. However, experience shows that (5.6) provides rather good approximations and is assumed hereon.

The approximation used for $R_{ft}(n_2,k_2,R_1)$ as a function of n_2 and R_1 (the k_2 used here is a function of n_2) is also in the form of a Weibull function, i.e.,

$$R_{ft}(n_2,k_2,R_1) = R_{ft}(n_2,R_1) \simeq e^{-(-\ln R_1)^u \cdot v \cdot n_2^w} \tag{5.7}$$

where u, v and w are positive real constants. In order to justify (5.7), consider the expression of $R_{ft}(n_2,k_2,R_1)$ as a function of time for fixed n_2, k_2 and R_1, i.e., $R_{ft}(t) \simeq e^{-\lambda t^{\delta}}$ for some positive real constants λ, δ. This is a commonly adopted Weibull approximation to the reliability of a system as a function of time. Similarly, consider the expression of R_1 as a function of time as $R_1(t) = e^{-\gamma t^{\phi}}$ for some positive real constants γ and ϕ. The variable t can be expressed as a function of R_1 and replaced in the expression of $R_{ft}(t)$ to obtain $R_{ft}(n_2,k_2,R_1)$ as a function of R_1 as

$$R_{ft}(R_1) \simeq e^{-\lambda \left(\frac{-\ln R_1}{\gamma}\right)^{\delta/\phi}} \tag{5.8}$$

and, since $R_{ft}(n_2,k_2,R_1)$ depends on n_2 according to another Weibull function (just as R_1 depends on n_1), it results that

$$R_{ft}(n_2,R_1) \simeq (R_{ft}(R_1))^{x \cdot n_2^y} \tag{5.9}$$

for some positive real constants x and y. Substituting the expression (5.8) for $R_{ft}(R_1)$ in (5.9) and letting $u = \delta/\phi$, $v = \lambda \cdot x \cdot \gamma^{-\delta/\phi}$ and $w = y$ yields (5.7).

In summary, the approximation used for (5.3) is

$$R_2 \simeq e^{-a n_2^b} \cdot e^{-(-\ln R_1)^u \cdot v \cdot n_2^w} \tag{5.10}$$

Since $n_1 = n/n_2$, and letting $\alpha = \alpha' \cdot n^{\beta}$, (5.2) can be rewritten as

$$R_1 \simeq e^{-\alpha n_2^{\beta}} \tag{5.11}$$

Replacing R_1 in (5.10) according to (5.11) yields

$$\begin{aligned} R_2 &= e^{-(a n_2^b + \alpha^u \cdot n_2^{\beta u} \cdot v \cdot n_2^w)} \\ &= e^{-(a n_2^b + \alpha^u \cdot v \cdot n_2^{w+\beta u})} \end{aligned} \tag{5.12}$$

Equation (5.1) can now be solved, i.e.,

$$\frac{dR_2}{dn_2} = \left[a \cdot b \cdot n_2^{b-1} + \alpha^u \cdot v (w - \beta u) \cdot n_2^{(w-\beta u-1)}\right] e^{-(a n_2^b + \alpha^u \cdot v \cdot n_2^{w+\beta u})} = 0 \tag{5.13}$$

and because $n_2 \neq 0$ and $R_2 \neq 0$, $\frac{dR_2}{dn_2} = 0$ if

$$a \cdot b \cdot n_2^b + \alpha^u \cdot v (w - \beta u) \cdot n_2^{(w-\beta u)} = 0 \tag{5.14}$$

A sufficient and necessary condition for the existence of a real positive solution to (5.14) is that

$$w - \beta u < 0 \tag{5.15}$$

If there exists no real positive solution to (5.14) then there is no bi-level FTPA (using some fixed fault-tolerant schemes in each level) with better reliability than a single-level scheme using one of the fault-tolerant schemes used in the 1st and 2nd-level arrays. Equation (5.14) can be rewritten as

$$a \cdot b \cdot n_2^{b-(w-\beta u)} + \alpha^u \cdot v (w - \beta u) = 0 \tag{5.16}$$

and letting $\theta = w - \beta u$ and $\phi = -\alpha^u \cdot v \cdot \theta/(a \cdot b)$, the solution is

$$n_2^* = \phi^{1/(b-\theta)} \tag{5.17}$$

Substituting (5.17) in (5.12) yields the expression for the maximum reliability attainable with a bi-level FTPA using two given fault-tolerance schemes as

$$R_2^* = e^{-[a \phi^{b/(b-\theta)} + \alpha^u \cdot v \cdot \phi^{\theta/(b-\theta)}]} \tag{5.18}$$

In order to verify the potential gains in reliability of bi-level FTPA's several cases studies were undertaken. As an example, one of these studies looked at the problem of designing a logical (36×36) processor array (i.e., the number (36×36) does not include spares). It was decided that the simple CR approach would be used for 1st-level FTPA's while the DI scheme would be used for the 2nd-level FTPA in order to maximize utilization of spare 1st-level FTPA's. For both CR and DI schemes a column of spare processors is used. The area and reliability estimates were computed for both the CR and DI methods and the reliabilities for each of the levels were approximated as

$$R_1 \simeq e^{-14.88 \cdot 2^{-n_2}} \tag{5.19}$$

$$R_2 \simeq e^{-[0.027 n_1! \cdot 2^{-n_1} + 0.64 (\ln \frac{1}{R_1})^{n_1} n_2^{(n_2)}]} \tag{5.20}$$

The following parameter values were used : $r_p = r_{fi} = 0.99$, $a_s = a_c = 1/800$ and $a_l = 1/400$. The values of the variables in (5.17) are $\theta \simeq -11.24$, $b \simeq 1.26$, $\phi \simeq 1.144 \times 10^9$ and the optimal value of n_2 is $n_2^* = 5.3 \sim 5$. This indicates that $n_1^* = 36/5 = 7.2 \simeq 7$. Since the targeted array has a size of (36×36), i.e. $n = 36$, and since $n^* = n_1^* \cdot n_2^* = 35$, one can decide to build a slightly smaller array using the extra processors as spares or change $n_2^* = 6$ and $n_1^* = 6$. When $n_1^* = 7$ and $n_2^* = 5$ the actual physical array (i.e., including spares) contains 1680 processors, i.e., a redundancy factor of 1.37. When $n_1^* = 6$ and $n_2^* = 6$ there are 1764 processors, i.e., a redundancy factor of 1.36. The reliability for the last case is 0.75, where for a single-level array using the DI approach or the CR approach with the same redundancy ratio the reliabilities are 0.08 and 0.31 respectively.

6. CONCLUSIONS

Several important related conclusions can be made from the work reported in this paper. First, it has been shown that non-redundant hardware and extra logic added for fault-tolerant purposes do limit the usefulness of single-level FTPA's above a certain size. A second conclusion is that, based on reliability estimates for different types of FTPA's for different array sizes and different area and technology parameters, there is not a single type of FTPA which is universally optimal. In other words, FTPA's based on different fault-tolerance methods are optimal for different array sizes for a given technology. A third conclusion is that multi-level FTPA's do not suffer from the disadvantage pointed out in the first conclusion for single-level FTPA's and can take advantage of the fact pointed out in the second conclusion - the net result being a highly reliable FTPA.

To achieve the third conclusion mentioned above, the problem of designing optimal bi-level FTPA's was addressed and a methodology for its solution has been described. The key to this methodology is the realization that, by using accurate functional approximations of the reliability of FTPA's at different levels, the complexity of the exact analytical expressions is avoided. These approximations are based on Weibull reliability functions.

The work reported in this paper represents significant progress towards a theory and solutions for FTPA design. However, it also opens a very large number of questions which relate to how better area and reliability estimates can be obtained, variations of the optimization criteria (e.g. compound measures of performance, area and reliability) subject to other constraints (e.g. fixed redundancy ratio), extension to multi-level FTPA's, implementation issues, development of tools, etc. It is our belief that the framework presented in this paper provides the basis and some ingredients for a sound theory that might lead to the solution of these new problems. Work in this direction is now in progress.

7. ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation under Grant DCI-8419745 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contract No. 00014-85-k-0588.

8. REFERENCES

[1] C. Mead and L. Conway, "Introduction to VLSI Systems," Addison-Wesley, Reading, Mass., 1980.

[2] J. A. Abraham, P. Banerjee, C-Y. Chen, W. K. Fuchs, S-Y. Kuo and A. L. N. Reddy, "Fault Tolerance Techniques for Systolic Arrays," Computer, July 1987, pp. 65-74.

[3] I. Koren and M. A. Breuer, "On Area and Yield Considerations for Fault-Tolerant Processor Arrays," IEEE Trans. on Computers, Jan. 1984, pp. 21-27.

[4] S. Y. Kung, "VLSI Array Processors," Prentice-Hall, Englewood Cliffs, N.J. 1987.

[5] A. L. Rosenberg, "The Diogenes Approach to Testable Fault Tolerant Arrays of Processors," IEEE Trans. on Computers, Oct. 1983, pp. 902-910.

[6] J. A. B. Fortes, C. S. Raghavendra, "Gracefully Degradable Processor Arrays," IEEE Trans. on Computers, Nov. 1985, pp., 1033-1044.

[7] M. Sami and R. Steffanelli, "Fault-Tolerance and Functional Reconfiguration in VLSI Arrays," International Conference on Circuits and Systems, 1986.

[8] H. T. Kung and M. S. Lam, "Fault-Tolerance and Two-Level Pipelining in VLSI Systolic Arrays," Proc. Conf. Advanced Research in VLSI, Jan. 1984, pp. 71-83.

[9] J-H. Kim and S.M. Reddy, "A Fault-Tolerant Systolic Array Design Using TMR Method," Proc. IEEE Int'l Conf Comp. Design: VLSI in Computer, Oct. 1985, pp. 769-773.

[10] K.S. Hedlund and L.Snyder, "Systolic Architectures-A Wafer Scale Approach," IEEE 1984 Int'l. Conf. on Comp. Design: VLSI in Computers, 1984, pp. 601-610.

[11] S.Y. Kung, C.W. Chang and C.W. Jen, "On Fault-Tolerance in Array Processors," IEEE 1985 Int'l. Conf. on Comp. Design, pp.764-768 1985.

[12] C. Jesshope and L. Bentley, "Techniques for Implementing two-dimensional wafer-scale processor arrays," IEE Proceedings, Vol. 134, Pt.E, No.2, March 1987, pp. 87-92.

[13] J.H. Hwang and C.S.Raghavendra, "VLSI Implementation of Fault-Tolerant Systolic Arrays," Proc. IEEE Int'l. Conf. on Comp. Design: VLSI in Computers, 1986, pp. 110-113.

[14] R.A. Evans, "A Self-Organizing Fault-Tolerant, 2-Dimensional Array," VLSI 85, E.Horbst (editor) Elsevier Science Publishers B. V. (North-Holland), IFIP, 1986, pp. 239-248.

[15] J.von Neumann, "Probabilistic Logics and the Synthesis of Studies, C.E. Shannon and J. McVarthy (Ed.), Priceton University Press, Princeton, New Jersey, 1956.

[16] I. Koren, "Comments on "The Diogenes Approach to Testable Fault-Tolerant Arrays of Processors,"" IEEE Trans. on Comp. Vol. C-35, No.1, Jan.1986, pp. 93.

[17] A.L. Rosenberg, "Author's Reply," IEEE Trans. on Comp. Vol. C-35, No. 1, Jan. 1986, pp. 94.

[18] F.M. Rhodes, "Performance Characterization of the RVLSI Technology," Proc. IFIP Workshop on Wafer-Scale Integration, Grenoble, 1986.

[19] "Components Quality/Reliability Handbook," Intel,1985

[20] A.D.Ingle and D.P. Siewiorek, "A Reliability Model for Various Switch Designs in Hybrid Redundancy," IEEE Trans. on Computers, Vol. C-25, Feb. 1976, pp. 115-133.

[21] S.A. Elkind and D.P. Siewiorek, "Reliability and Performance of Error-Correcting Memory and Register Arrays," IEEE Trans. on Computers, Vol. C-29, Oct. 1980, pp.920-927.

[22] A. Papoulis, "Probability, Random Variables, and Stochastic Processes," McGraw Hill, New York, N.Y, 1984 (Second Edition).

[23] D.P. Siewiorek and R.S. Swarz, "The Theory and Practice of Reliability System Design," Digital Press, Educational Services, DEC, Bedford, MASS., 1982.

[24] Y.Wang and J.A.B. Fortes, "Multi-Level Fault-Tolerant Processor Arrays," Technical Report, in preparation.

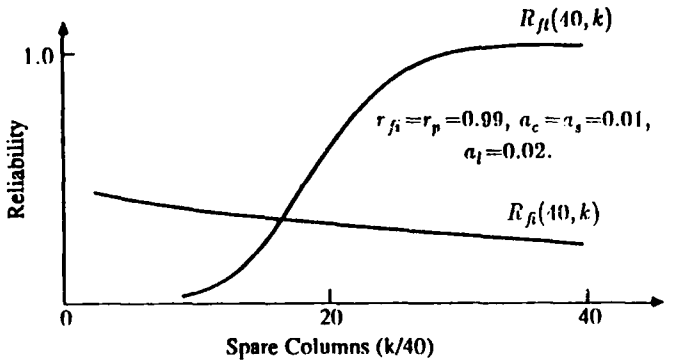


Figure 1. The reliability $R(40,k)$ of an array using the CR-method is the product of $R_{fi}(40.k)$ and $R_{ft}(40,k)$; it decreases with k for $k/40 \geq 28$.

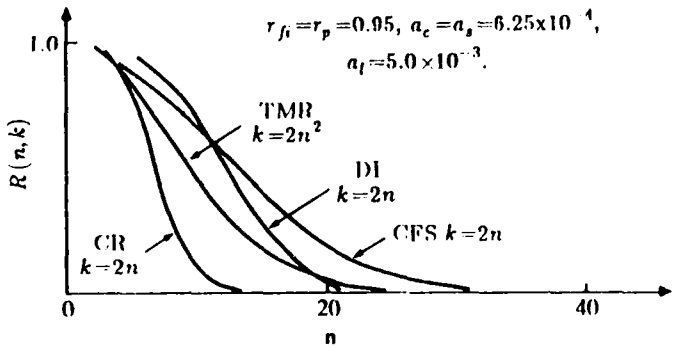


Figure 2. The reliability of different single-level FTPA's schemes with $(n\times n)+k$ processors; different approaches are optimal for different values of n and low reliability results for large arrays.

REFERENCE NO. 12

Fortes, J. A. B., Fu, K. S. and Wah, B. W., "Systematic Design Approaches for Algorithmically Specified Systolic Arrays," in Computer Architecture: Concepts and Systems," Milutinovic, V. M., eds., Elsevier Science Publishing Co., Inc., New York, pp. 448-488, 1988.

11

Systematic Design Approaches for Algorithmically Specified Systolic Arrays

Jose A. B. Fortes
*King-Sun Fu**
Benjamin W. Wah

11.1. INTRODUCTION

The evolution in Very Large Scale Integration (VLSI) technology has had a great impact on computer architecture (114). Many existing algorithms in pattern recognition and image and signal processing can be implemented on a VLSI chip using multiple, regularly connected processing elements (PEs) to exploit the great potential of pipelining and multiprocessing in applications in command, control, and communications systems (4). This type of array processor has been referred to as a *systolic array*, and the concept was introduced in the pioneering paper of H. T. Kung and C. E. Leiserson (67, 69). A good survey on the state of the art is provided by Ullman (116). A list of the sample problems for which systolic solutions exist is shown in Table 11.1 (46).

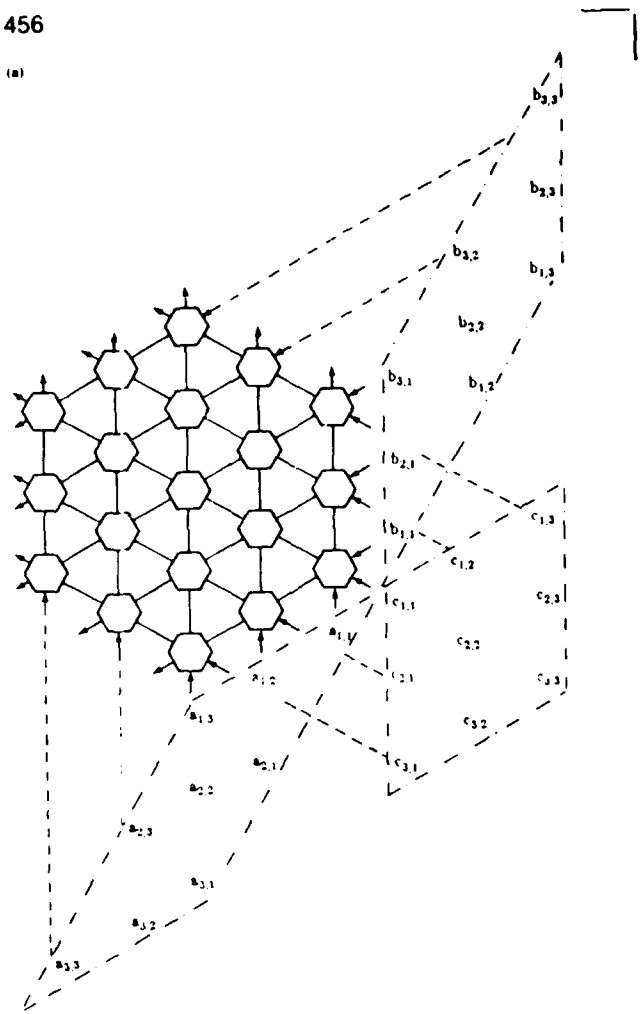
* Deceased.

Research supported by National Science Foundation Grants ECS 80-07580, DMC 85-19649, and DCR 84-19745, and was also supported by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contract No. N00014-85-K-0588.

Table 11.1 A Sample of the Applications for Which Systolic Arrays are Available

Signal and image processing and pattern recognition
FIR, IIR filtering and 1-D convolution
2-D convolution and correlation
discrete Fourier transform
interpolation
1-D and 2-D median filtering
geometric warping
feature extraction
order statistics
counters
minimum distance classifier
covariance matrix computation
template and pattern matching
seismic signal classification
cluster analysis
syntactic pattern recognition
radar signal processing
curve detection
dynamic scene analysis
image resampling
scene matching
Matrix arithmetic
Toeplitz matrix–vector multiplication
matrix–matrix multiplication
matrix triangularization
QR and LU decompositions
sparse matrix operations
solution of triangular linear systems
polynomial multiplication division
Nonnumeric applications
data structures—stacks and queues, searching, priority queues
graph algorithms—transitive closure, minimum spanning trees, connected components
language recognition
dynamic programming
arithmetic arrays
relational database operations
sorting

An example of a systolic array for multiplying matrices A and B to form C is shown in Figure 11.1. The dataflows of the three rhomboidal data blocks are in three directions. A moves toward the north, B moves toward 120° north, and C moves toward 60° north. During a clock cycle, each PE receives three data items from three different pipes and executes a multiply-add operation. These data items advance into neighboring PEs along their own pipes synchronously in the next clock cycle.



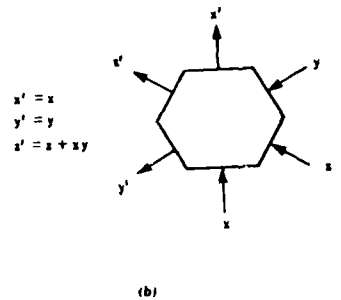


Figure 11.1 The systolic processor for two-dimensional matrix multiplication. (a) Systolic processor. (b) Structure of PE.

One of the many advantages of the systolic approach is that each input-data item can be used a number of times once it is accessed, and thus a high computational throughput can be achieved with only a modest bandwidth. Other advantages include modular expandability, simple and regular data and control flows, and simplicity and uniformity of PEs.

Systolic arrays have been classified into semisystolic arrays with global data communications and pure systolic arrays without global data communications (67). In a *semisystolic array*, a data item accessed from memory is broadcast to and used by a number of possibly nonidentical PEs concurrently. Although this approach is potentially faster than systolic arrays without data broadcast, providing (or collecting) a data item to (or from) all the PEs in each cycle requires the use of a global bus that may eventually slow down the processing speed as the number of PEs increases. On the other hand, a *pure systolic array* eliminates the use of broadcast buses and implements the algorithm in pipelines extending in different directions. Several data items flowing along different pipes with the same or different rates may meet and interact. The PEs operate synchronously with one or more clocks, and all the necessary operands to be processed by a PE in each computational step must arrive at this PE simultaneously. This mode of pipelining is referred to as *systolic processing*.

One of the important design problems in systolic processing is the development of a systematic methodology for transforming an algorithm represented in some high-level constructs into a systolic architecture specified by the timing of data movements and the interconnection of processing elements such that the design requirements are satisfied. In this chapter, we survey 19 methodologies proposed in the literature. The applicability, capabilities, and results derived for each methodology are identified.

11.2. Systematic Methodologies for Designing Systolic Arrays

The common characteristic of most previously proposed methodologies is the use of a transformational approach—i.e., systolic architectures are derived by transforming the original algorithm descriptions that are unsuitable for direct VLSI implementation. Distinct transformational systems for systolic architecture design (hereafter referred to as transformational systems) can be characterized by how algorithms are described, what formal models are used, how systolic architectures are specified, and what types of transformations are used on and between these representations. In other words, we can visualize each transformational system as a three dimensional space, where dimensions (or axes) are associated with the algorithm representation, algorithm model, and architecture specification. To the axis of algorithm representation, we associate different forms or levels to present an algorithm to the transformational system. The axis of algorithm model shows different levels of abstraction used to represent relevant features of the algorithm. The axis of architecture specification is associated with the hardware model or level of design in which the systolic array is described.

This three dimensional space can be graphically depicted as a Y chart (Figure 11.2), where directed arcs can be drawn to illustrate transformations that map a given representation into another representation in the same axis and level (a self loop), in the same axis and different level, or between distinct dimensions.† Arcs drawn in full lines represent systematic transformations, whereas those drawn in broken lines represent ad hoc transformations. The Y charts allow us to classify and describe the large number of approaches taken to design systolic arrays. Before we do this, we will use the Y chart in Figure 11.3 to explain Kuhn's approach [63].

Kuhn's methodology starts with a naive high-level language cyclic-loop program—i.e., an algorithm written without regard to how it is implemented in VLSI. In an ad hoc manner, additional subscripts for variable referencing are introduced such that the possibility of broadcasts of variables does not exist. The algorithm model assumed in Kuhn's method is a set of computation nodes (which correspond to the loop-body assignment statements) indexed by the vector value of the indices of the iteration when they are computed (Figure 11.3). The structural information is modeled by the dimensionality of the iteration space and the dependency vectors (which are the vector difference of the indices of dependent computation nodes). The geometry of the algorithm is represented by the iteration space and how different variables are associated to points in that space. This model is derived from the program in a systematic manner by using analysis techniques

† We borrowed from Cuckov's paper [4] the idea of using Y charts to improve the clarity of our presentation; however, our Y charts are numerically related to those used in that reference.

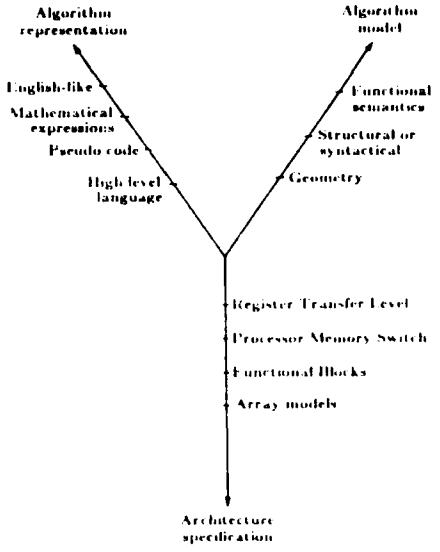


Figure 11.2 Y-chart for transformational systems.

similar to those used in optimizing compilers. A re-indexing transformation is then sought in an ad hoc fashion until a favorable set of dependencies is obtained. Once this transformation is known, one can systematically generate not only the new dependency vectors but also the range of the new indices of the loops and the subscript functions used to reference variables. By projecting the new iteration set into all but one of its dimensions, and by identifying the iterations in which input variables are used, the size, dimension, and input/output ports of the architecture can be systematically generated.

Each point in the projected space corresponds to a PE in the array whose function is totally described by the statements in the loop body. The interconnections and the direction, speed, and timing of data movements are systematically derived from the new set of dependencies that resulted from applying the re-indexing transformation. This completes our example of the use of Y-charts to explain a methodology. We defer the analysis of the capabilities of Kuhn's method until after we introduce a classification of the different approaches in terms of the Y-charts.

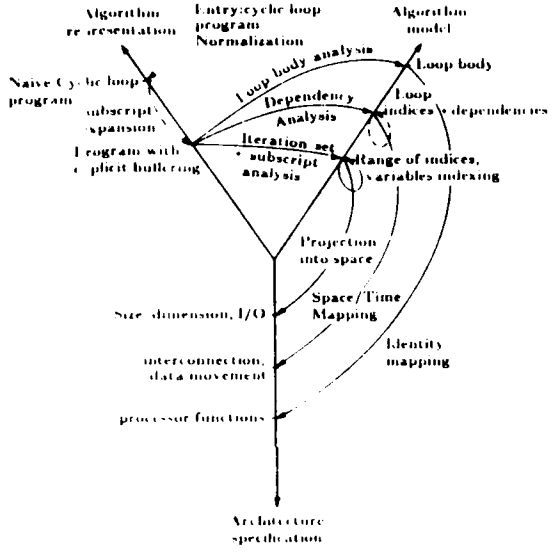


Figure 11.3 Kuhn's method

The various methodologies can be grouped into the following classes:*

1. Those that allow transformations to be performed at the algorithm-representation level and that advocate a direct mapping from this level to the architecture specification. These include:
   Cohen, Johnsson, Weiser, and Davis' method
   Lam and Mostow's method
2. Methods that prescribe transformations at the algorithm-model level and that require procedures for deriving the model from the algorithm representation (analysis) and for mapping the model into hardware (synthesis). These include:
   Gannon's method
   H. T. Kung and Lin's method
   Kuhn's method
   Moldovan and Fortes' method
   Miranker and Winkler's method
   Cappello and Steiglitz's method
   S. Y. Kung's method
   Quinton's method
   Ramakrishnan, Fussell, and Silberschatz's method
   Li and Wah's method
   Cheng and Fu's method
   Jover and Kailath's method
   Schwartz and Barnwell's method
   Ibarra, Kim, and Palis' method
3. Methods that transform a previously designed architecture into a new architecture. We will consider only one work in this class:
   Leiserson, Rose, and Saxe's method
4. Methods that abstract the function implemented by a given systolic architecture and that use symbolic manipulations and transformations to prove the correctness of the design. Two studies in this class are considered:
   Chen and Mead's method
   Kuo, Levy, and Musicus' method

In the following sections, we will describe these methods in an arbitrary order, show their applicability, discuss their capabilities, and summarize the major results. The discussion in some of these studies may be vague, and we have tried to infer their results from our understanding of the published work.

* The order in which the methodologies are described is chosen at random.

11.2.1. Cohen, Johnsson, Weiser and Davis' Method (33, 57–59, 119)

Description

Starting from a mathematical expression involving subscripted variables, which conceptually represent data sequenced in time or space, this method begins by deriving a new expression where a well-defined operator Z is used to model displacements in time (e.g., the storage of data) or shifts in space (e.g., the allocation of a data stream to PEs). Symbolic manipulation is used to transform the derived mathematical expression into equivalent ones by using the properties of the Z operator and the functional operators in the expression. From a particular expression, the execution order of the operations can be derived from known precedence rules. The number, placement, and interconnection of operator PEs can also be derived. Timing and storage requirements are inferred from the placement of delay PEs (which correspond to the Z operators) (Figure 11.4).

Applicability

This method seems to be best applicable to algorithms that can be described by relatively simple and concise mathematical expressions.

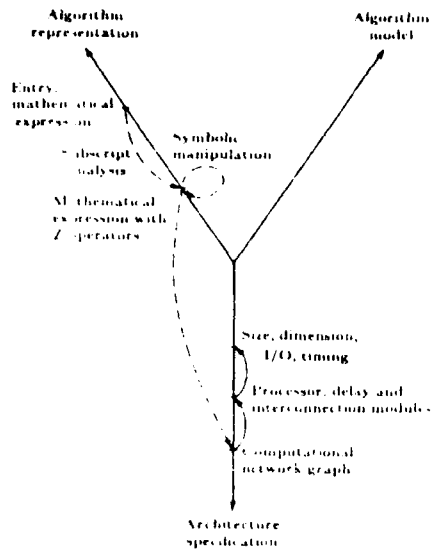


Figure 11.4 Cohen, Johnsson, Weiser, and Davis' method

Capabilities

Computational rate, performance, delay, modularity, and size can be easily derived from the equations; interconnection and communication characteristics can also be derived when the architecture is regular. This method may yield implementations with both parallel and sequential features, requiring hardware of a size smaller than the problem size. The method treats control signals in the same way as data signals. The optimal design is searched in an ad hoc manner.

Results

Formal derivations have been reported for architectures intended for the following problems: finite-impulse-response (FIR) filters, discrete Fourier transform (DFT), matrix-vector product, string matching, solution of triangular linear equations, product of band matrices, synthetic aperture radar (SAR), and multiplication and division of polynomials. A set of data set operators defined in terms of the Z operator was also proposed for treating

sets of data as wavefront entities in expressions and their graphical representation.

11.2.2. Lam and Mostow's Method (SYS) (78, 101)

Description

SYS accepts as input an algorithm suitable for systolic implementation—i.e., an algorithm obtained by software transformations from a high-level specification that results in segments of code executed repeatedly with a regular pattern of data accesses. The algorithm is mapped into a systolic design described by a structure and a driver. The structure describes the hardware PEs (which are functionally equivalent to code segments), interconnections, and input–output ports. The driver defines data streams in terms of the original variables in the algorithm. *The mapping of iterative* algorithms uses three basic allocation schemes named sequential, parallel, and compositional. SYS has a special language for representing a given design. *Initially, SYS generates a simple-minded implementation of the given* algorithm. Systematic and user determined transformations are then used to optimize and to obtain new designs (Figure 11.5).

Applicability

SYS can process algorithms with simple FOR loops and BEGIN-END blocks, simple unnested function calls, and scalar and array variables. As reported in the references, SYS cannot deal with conditional execution, computed iteration bounds and array indices, and other high-level software constructs.

Capabilities

SYS can derive the structure and driver of a systolic design. Specification of the structure includes the number and dimensionality of ports of PEs, *hierarchical definition of PEs*, arrays and compounds of PEs, and interconnections among them, including broadcasts and directional links. The driver describes data streams and timing schemes that include delay, skew of streams, and ready time (time allowed between two consecutive inputs to the structure).

Results

Reported designs obtained by SYS include two systolic arrays for polynomial evaluation and a circuit for computing the greatest common divisor of two polynomials. Other nonsystolic designs using a transformational system related to SYS include a chip for color shading and hidden-surface elimination and a multichip switching network for marker passing semantic networks. All designs were previously known.

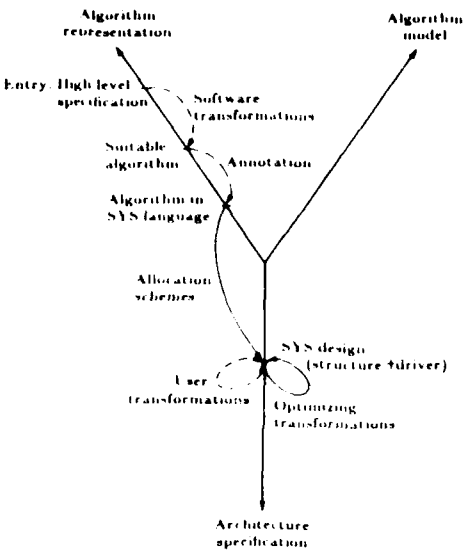


Figure 11.5 Lam and Mostow's method

11.2.3. Gannon's Method (48)

Description

From a given algorithm, a functional specification is derived by using vector operators that explicitly represent parallelism. These vector operators are defined in terms of basic functions that correspond to small units of sequential computation and that map directly into the functional specification of the PEs of the systolic architecture. The vector operators include a product operator, which represents the concurrent operation of basic functions, permutation and data-movement operators; a chain operator, which represents the iterative composition of basic functions; and the systolic-iteration operator, which describes basic functions that are "reused." The global functional specification of the algorithm is viewed as a dataflow graph which, depending on the properties of the functions and operators used, can be mapped into a systolic architecture. Different architectures result from expressing the same algorithm with different operators (Figure 11.6).

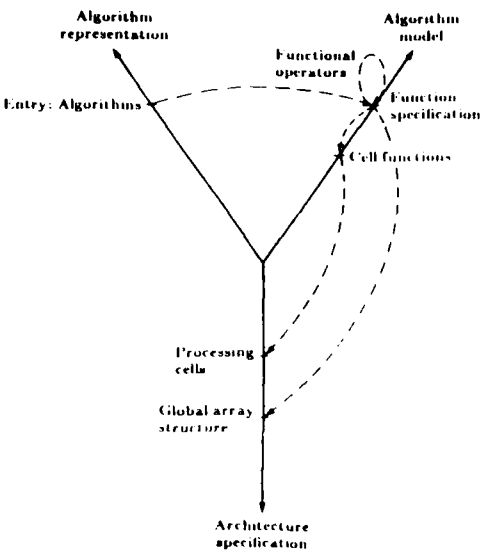


Figure 11.6 Gannon's method

Applicability

This method seems to be suitable to those algorithms that can be reexpressed by vector operators. For these algorithms, the methodology seems hard to apply without human assistance.

Capabilities

The functional description of PEs and the interconnection topology can be easily derived. Additional information such as data movement and timing is *present in implicit form.*

Results

A previously known design of a recurrence solver was rederived. The formalism used proved the theoretical result that systolic versions of computation graphs perform asymptotically as fast as fully concurrent execution of the original dataflow graph.

11.2.4. H. T. Kung and Lin's Method (70)

Description

This method starts by deriving a straightforward and obviously correct algebraic representation from the mathematical representation of the algorithm. The canonical algebraic representation consists of two matrical expressions of the form (a) $v \leftarrow Av + bx$, and (b) $y = c^T v$, where x represents the input, y represents the output, and v represents variables generated by implicit functions. The $(n \times n)$ matrix A and the column vectors b and c represent the delay cycles between the availability and the use of variables, and each entry is either 0 or Z^{-k}, where k corresponds to the number of delays. For example, the ith component of v in Expression (a) is

$$v_i \leftarrow Z^{-a_{i1}} v_1 + \cdots + Z^{-a_{in}} v_n + Z^{-b_i} x,$$

which means that

$$v_i(t) = f_i[v_1(t - a_{i1}), \ldots, v_n(t - a_{in}), x(t - b_i)]$$

for some implicit function f_i associated with node i. To this canonical representation, algebraic transformations are then applied. There are two major types of transformations: retiming and "k-slowing," which can also be described algebraically. These transformations determine the distribution of delays and the input/output periods of the systolic architecture. There exists a direct correspondence between the algebraic representations and a hardware-related representation denoted as the Z graph. The Z-graph has an edge for each variable and a node for each computation. Each edge is labeled as Z^{-k} if k delay cycles (i.e., registers) exist between the availability of the variable and its use as an operand or output (Figure 11.7).

Applicability

The method is suitable for algorithms for which a canonical algebraic representation can be found.

Capabilities

Functional description of PEs, interconnections, and timing for input/output and data communication can be derived systematically; designs and transformations can be expressed algebraically; theoretical results on retiming and k-slowed designs can be proved easily by algebraic manipulation.

Results

Designs for FIR and IIR filters and matrix–matrix multiplication were derived. New results include the derivation of two-level pipelined systolic arrays and systolic architectures for LU decomposition.

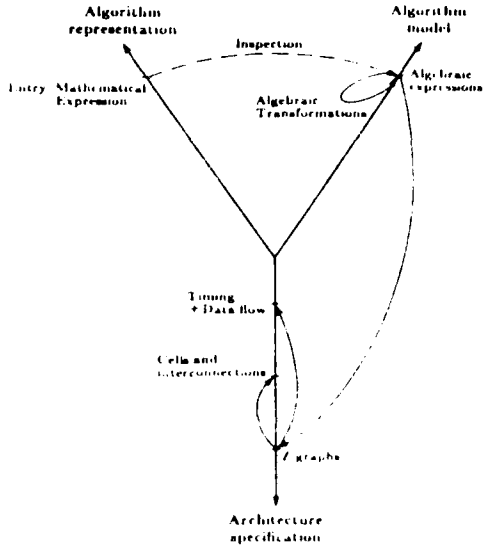


Figure 11.7 H. T. Kung and Lin's method.

11.2.5. Kuhn's Method (62, 63)

Description

As described early in this chapter.

Applicability

This method is best suited for algorithms described as cyclic loop programs with constant execution time and dependencies in loop bodies.

Capabilities

Size, dimension, topology, input–output ports, execution time, data movement, timing, and functional descriptions of PEs of an architecture can be systematically derived. However, nothing can be said about the optimality of the design, and the choice of transformations is done in an ad hoc manner.

Results
Designs for the following problems were derived using this method: matrix–matrix multiplication, matrix–vector multiplication, recurrence evaluation, solution of triangular linear systems, constant-time priority queue, on-line sort, transitive closure, and LU decomposition.

11.2.6. Moldovan and Fortes' Method (37–41, 96–99, 100, 104)

Description
From a program or a set of recurrence equations, an algebraic model of the algorithm is derived by using systematic techniques similar to those used in software compilers. This model consists of a structured set of indexed computations that operate on a set of inputs to obtain a set of outputs. Typically, programs include loops, and indexing of computations is related to the loop indices. However, unlike Kuhn's approach, which associates the body of loops with the corresponding loop indices, each computation has an index. The algebraic representation of the algorithm is then transformed by local and global transformations. Local transformations are used to rewrite computations that are mapped into the functional and structural specifications of the PE of the systolic architecture. Global transformations composed of time and space transformations are used to restructure the algorithm. They are chosen in such a way that the new algorithm has a set of dependencies that favors VLSI implementation. Time transformations determine the execution time of the algorithm and the timing for data communication. Space transformations determine the interconnections and the direction of data movement. The *projection* of the index set of the algorithm into space determines the size, dimension, I/O ports, and geometry of the architecture (Figure 11.8).

Applicability
This method is best suited to algorithms described by either programs with loops or recurrence equations.

Capabilities
Because this method is an extension of Kuhn's approach, it has the same capabilities as that method. Additionally, it allows or eliminates broadcasting, designs fixed-size architectures for arbitrarily large algorithms, implements fault-tolerance schemes, and optimizes execution time.

Results
Systematically obtained designs have been reported for matrix–matrix multiplication, LU Gaussian elimination, dynamic programming, partitioned matrix–vector multiplication, convolution, partitioned QR eigenvalue de-

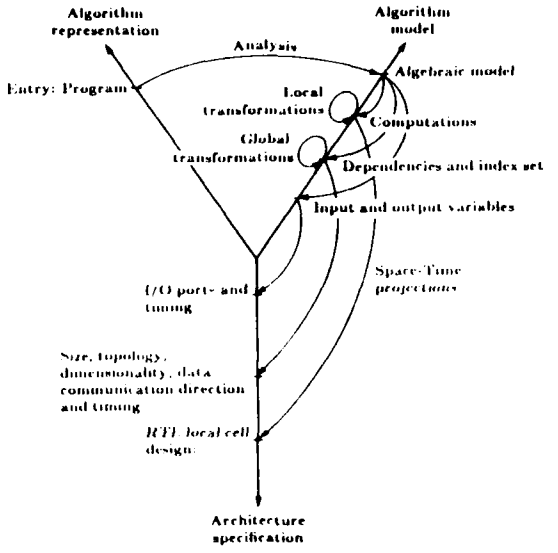


Figure 11.8 Moldovan and Fortes' method

composition, and partial differential equations. Theoretical results include the necessary and sufficient conditions for the existence of global transformations and broadcasts, sufficient conditions for the partitionability of algorithms, and a method for finding optimal linear schedules for systolic algorithms.

11.2.7. Miranker and Winkler's Method (95)

Description
This method is an extension of Kuhn's method and is similar to Moldovan's method. An algorithm is represented as either a mathematical expression or a cyclic-loop program. One extension is to allow the rewriting of mathematical expressions by using the properties of the operators in an ad hoc fashion. The other extension is to use the graph embeddings based on the knowledge of the longest path of the computation graph when this graph is too irregular and simple matrix transformations are not useful (Figure 11.9).

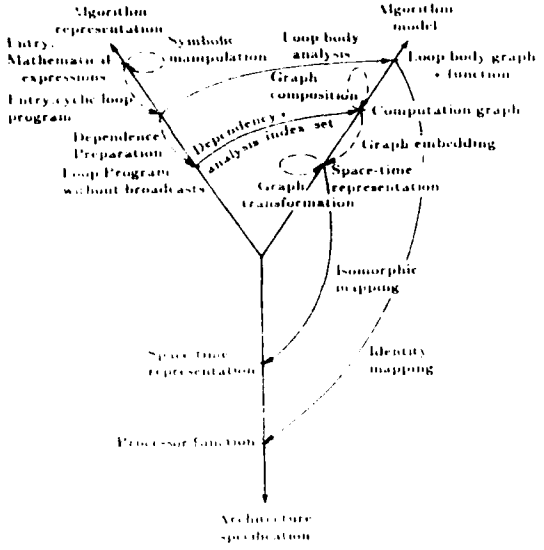


Figure 11.9 Miranker and Winkler's method

Applicability
Theoretically, it can be applied to any algorithm, although systematic design seems possible only for those algorithms described by programs with loops

Capabilities
Size, dimension, topology, data movement, timing, functional description of PEs, and execution time can be systematically derived

Results
Designs of architectures for the computation of discrete Fourier transform and the solution of a triangular linear system of equations were systematically derived

11.2.8. appello and Steiglitz's Method (15–17)

Description
Starting from a set of recurrence equations describing the algorithm, a canonical representation is obtained by adjoining an index representing time

to the definition of the recurrence. Each index is associated with a dimension of a geometric space, where each point corresponds to a tuple of indices on which the recurrence is defined. To each such point, a primitive computation is associated, and its implementation is left unspecified. Primitive computations are mapped directly into functional specifications of PEs in the systolic architecture. From the geometric representation and an ordering rule, the topology and size of the architecture and the timing and direction of dataflows are derived systematically. By selecting different geometric transformations, distinct geometric representations and their corresponding architectures can be derived (Figure 11.10).

Applicability
This method is best suited to algorithms described by recurrence equations

Capabilities
Geometric representations help the designer's understanding of a systolic architecture, and geometric transformations are easily and succinctly rep-

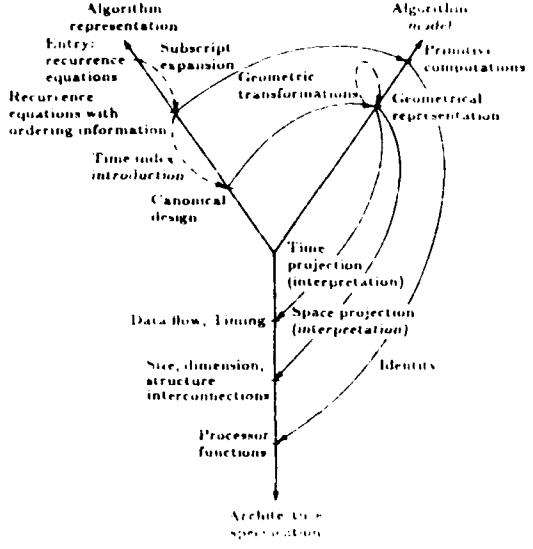


Figure 11.10 Cappello and Steiglitz's method

resented as matrix transformations. The presence of broadcasts, data pipelining, topology, area, and timing of the architecture are easily perceived from the geometric representation.

Results
Designs of architectures for matrix–vector multiplication, convolution, matrix–matrix product, and matrix transposition were formally related and rederived. For some, it was shown that they are asymptomatically optimal, and for others, alternative designs were provided.

11.2.9. S.Y. Kung's Method (74, 75)

Description
Given a Signal Flow Graph (SFG) representing an algorithm, this method starts by choosing basic operational modules that correspond to the functional description of PEs of the architecture. Localization rules are then applied to derive a regular and temporally localized SFG. The localization procedure consists of selecting cut-sets of the SFG and reallocating scaled delays to edges "leaving" and "entering" the cut-set in such a way that at least one unit of time is allowed for communicating a signal between two nodes. Delays are combined with operational modules to obtain a full description of the operation of a basic systolic module. The resulting SFG maps straightforwardly into the systolic array by mapping basic modules into PEs and edges into interconnections. Timing and data movements can be derived from the basic modules due to the localized spatial and temporal characteristics of the SFG (Figure 11.11).

Applicability
This method is applicable to all algorithms described by computable SFGs with some regularity.

Capabilities
Size, dimension, functional, and structural description of PEs, timing, direction of dataflow, and interconnections of the architecture can be derived from the SFG of the algorithm. Design verification can be done by applying Z transform techniques to the SFG.

Results
Systolic arrays have been derived from SFGs for autoregressive filter, matrix multiplication, banded-matrix–full-matrix product, banded-matrix multiplication, and LU-matrix decomposition. Theoretical results include the proof that all computable SFGs are temporally localizable and the equivalence between SFGs and dataflow graphs.

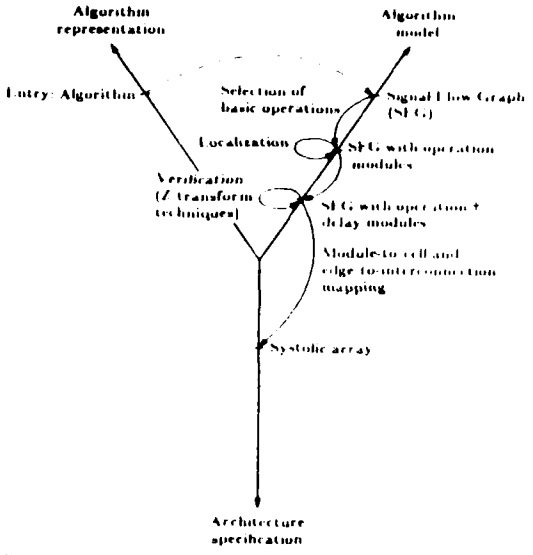


Figure 11.11 S. Y. Kung's method

11.2.10. Quinton's Method (106, 107)

Description
Given a system of n uniform recurrence equations defined over some convex subset D in Z^M and with some characteristic dependency vectors, (which, together define a dependency graph), this method starts by finding a timing function that maps points of D into time. This requires the identification of a convex space of feasible timing functions from which one can be chosen heuristically. Such space can be found systematically from the knowledge of the dependency vectors and $D(D)$ can be thought of as the index set of the recurrence). Next, an allocation function is chosen, which projects D into space along a preselected direction such that two points in D with the same image under the timing function do not map into the same point in space. Once the timing and allocation functions (which are quasi-affine functions) are known, the systolic array can be systematically generated from D. Each point of D is mapped into a PE that computes the recurrence function, and receives and sends data from and to PEs that are the image of

points dependent and depending on the point under consideration, with delays given by the timing function (Figure 11.12).

Applicability
The method is specifically intended for algorithms described by uniform recurrence equations.

Capabilities
The functional description of PEs, the size, dimension, and topology of the array, and the execution time, direction, and timing of data communication can all be derived systematically.

Results
Derived architectures include arrays for convolution (including a block convolver and a ring convolver) and matrix product. Extensions of the method allow the derivation of arrays for LU decomposition and dynamic programming.

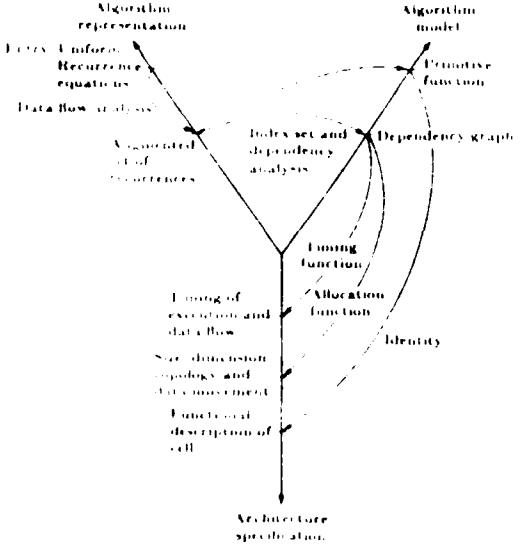


Figure 11.12 [illegible] method.

11.2.11. Ramakrishnan, Fussell, and Silberschatz's Method (108, 118)

Description
This method starts with a dataflow description of the algorithm (an acyclic program graph), which is partitioned into sets of vertices that are mapped into the same PE (this partitioning is called diagonalization). A syntactically correct mapping is then used to map computation vertices onto PEs and time steps, and the labels and edges to map communication delays and interconnections (Figure 11.13).

Applicability
The method applies only to homogeneous graphs with connected subgraphs that satisfy certain properties. Moreover, the method can only be used to generate linear arrays.

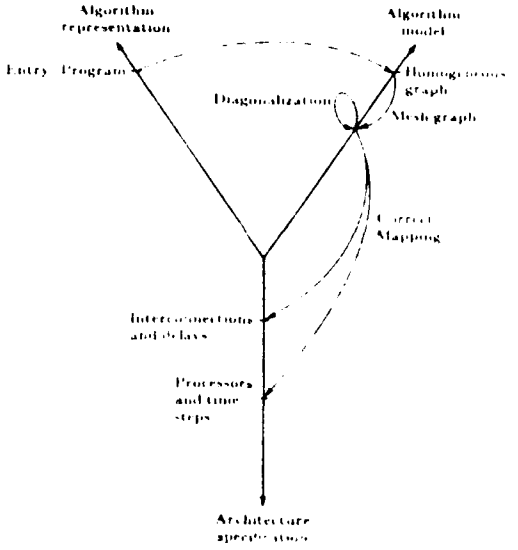


Figure 11.13 Ramakrishnan, Fussell, and Silberschatz's method.

Capabilities
The method yields the number of PEs, their functional description, and the number of I/O ports. Additionally, it gives the direction of data communication, time used, and timing.

Results
Linear-array designs were synthesized for band matrix–vector multiplication, convolution, dynamic programming, and transitive closure.

11.2.12. Li and Wah's Method (85–88)

Description
Starting from an algorithm described as a set of linear recurrence equations, this method derives three classes of parameters: velocities of dataflows, spatial distributions of data, and periods of computations. The relationships among these parameters are represented as constraint vector equations that must be satisfied in a correct design. The performance of a design can also be expressed in terms of the defined parameters. Performance can be defined as execution time or the product of the square of execution time and the number of PEs. *Optimal* designs are then searched in the space of solutions that satisfy the constraint equations. This search is done by ordered enumeration over a limited search space in time polynomial to the problem size. The functional description of the PEs is derived from the definition of the recurrence equations. The interconnections among PEs are found from the defined parameters (Figure 11.14).

Applicability
The method is best suited to algorithms that can be described by sets of linear recurrences.

Capabilities
Functional description of PEs, timing and spatial distribution of dataflows, execution time, number of PEs, and interconnections can be systematically derived. Optimal designs can be found.

Results
Systematically derived architectures include systolic arrays for finite-impulse response (FIR) filtering, matrix multiplication, discrete Fourier transform, polynomial multiplication, deconvolution, triangular matrix inversion, and tuple comparison.

11.2.13. Cheng and Fu's Method (26–30)

Description
Starting from a recursive formula with several indices or a program-loop with a simple expression, this method starts by designing the basic PE to

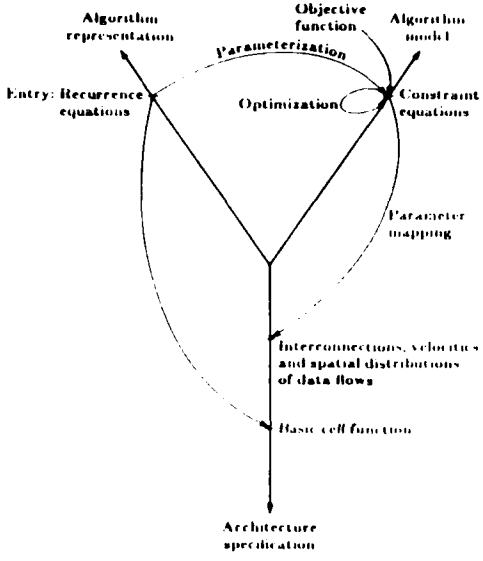


Figure 11.14 Li and Wah's method

compute the simple expression. This basic PE is then expanded in time and space in such a way that indices of the loop (or recursion) become associated with time and space. This expansion is done according to rules that maintain the consistency of time and space. Time-space expansion can be applied in any degree varying from full-time expansion (i.e., purely sequential single-processor architecture) to full-space expansion (i.e., fully parallel single-time execution). Time and space expansions implicitly determine dataflow timing and direction as well as PE interconnections (Figure 11.15). The method can be implemented at the gate level, register level, processing-unit level, and system level. Because there is no restriction on the dimensionality of the processing array, the method can be applied to design high-dimensional VLSI architectures. A computational model and partition rules can be derived that partition any problem suitable for the method and implement the problem on a fixed-size VLSI architecture.

Applicability
The method is suitable for algorithms described by recurrence expressions or programs with loops.

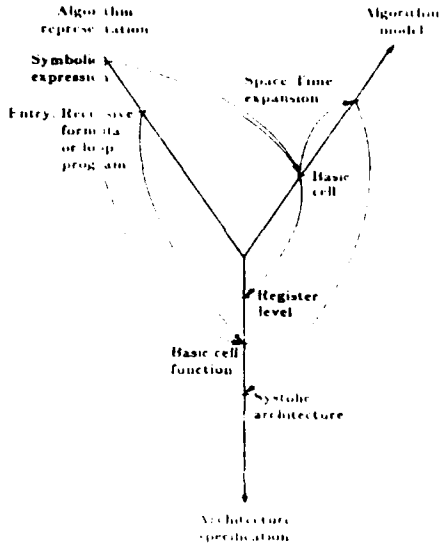


Figure II.15 Cheng and Fu's method

Capabilities
Functional description of PEs, timing and directions of dataflows, interconnections, execution time, and number of PEs can be systematically generated.

Results
This method has been applied to construct computational structures for computing vector inner product, matrix multiplication, convolution, comparison operations in relational databases, fast Fourier transform, hierarchical scene matching, transitive closure, string matching, pattern matching, recognition of hand-written signals, and recognition of context-free languages.

II.2.14. Jover and Kailath's Method (60)

Description
This method is based on the use of Lines Of Computation (LOCs) to determine whether a given topology is suitable for VLSI implementation. LOCs

are directional straight lines with several equally spaced nodes, and can be interpreted either as the history of how a given value was computed or as a stream of values in different stages of a computation. Because of the properties of LOCs, one can easily check if the LOCs chosen for a given algorithm define a systolic array or a systolic-type array (not necessarily planar and fully regular) (Figure II.16).

Applicability
Suitable for algorithms from which one can easily identify LOCs.

Capabilities
The topology of the systolic array can be easily derived from LOCs. Additionally, throughput, efficiency, data interval, initial conditions, interval and external delays, and pipeline ability can also be found from LOCs and the knowledge of execution times of basic operations.

Results
Three designs for matrix multiplication were derived.

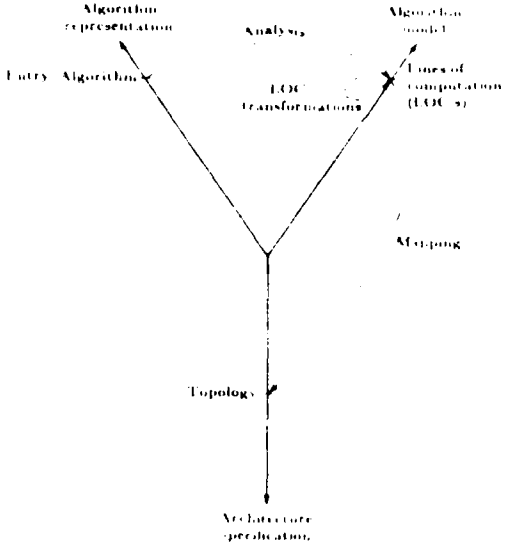


Figure II.16 Jover and Kailath's method

11.2.15. Sc wartz and Barnwell's Method (5, 110)

Description

This metho starts with an algorithm described as a fully specified flow graph—i e directed graph in which nodes represent operations and edges represent s nal paths. Node operations are fundamental operations performed by t e PEs of the architecture. For a given flow graph, it is possible to derive a ound in the sampling period and a bound in the static-pipeline sampling pe iod (i e , the minimum sampling period achievable if the graph is implemen ed as a static pipeline). Different systolic solutions are generated by distribut g delay nodes throughout the flow graph such that correctness is preserved and data transfers can be simultaneous. The transformations of flow graphs consist of data interleaving and the cut-set of delay transformation that are shown to preserve equivalence. The transformed flow graph is mapped into a systolic array by mapping nodes into PEs and delays and edges it to interconnections (Figure 11.17).

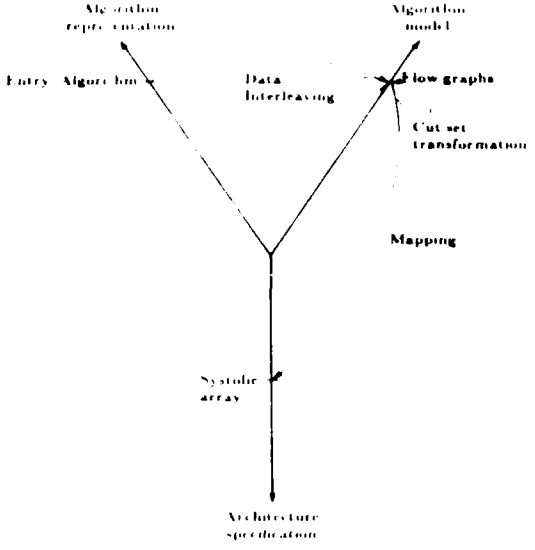


Figure 11.17 Schwartz and Barnwell's method.

Applicability

This method can be used with algorithms representable as shift-invariant flow graphs.

Capabilities

The following information about the systolic architecture can be systematically derived: number and functional description of PEs, interconnections, data movement, and time. The optimality of the resulting designs can be analyzed.

Results

Previously known and new architectures have been derived for FIR and IIR filters and two multiplier Markel–Gray lattice filters.

11.2.16. Ibarra, Palis, and Kim's Method (55, 56)

Description

Sequential-Machine (SM) models are used to simulate linear and orthogonal Systolic Arrays (SAs). Given an algorithm, this methodology starts by generating an SM characterization that consists of a serial program for a simple sequential machine with a single PE and an infinite array of registers (called the "worktape(s)"). SM characterizations that are obtained heuristically are easier to program and analyze than their SA counterparts. The conversions from SM to SA characterizations and vice versa are done systematically (Figure 11.18).

Applicability

This method seems to be best applicable to algorithms that can be easily programmed in an SM model. These are likely to be relatively simple and regular algorithms.

Capabilities

Because one starts with a given type of systolic array, certain features of the architecture (e.g., number, placement, direction of interconnections, and inputs) are known beforehand. In this sense, the methodology synthesizes or maps a given algorithm into a type of systolic array. From the SM program, the functional description of the PEs and the direction and timing of data movement can be derived automatically. The search and selection of the best type of systolic array and the best SM algorithm is done in an ad hoc manner.

Results

Systolic architectures and algorithms have been reported for priority queues, real-time bitwise multiplication, and language recognition. For linear systolic

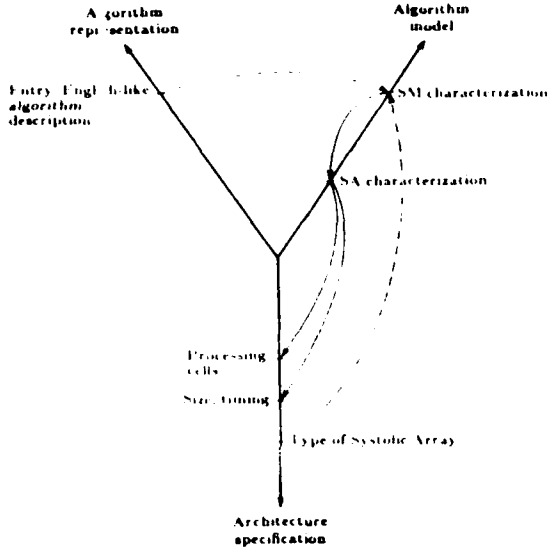


Figure 11.18 Ibarra, Kim, and Palis' method

arrays, several general results on speedup, computational power, the effects of adding global control, and the use of one- and two-way communications were reported.

11.2.17. Leiserson, Rose, and Saxe's Method (81–84)

Description

This method starts with the design of a synchronous circuit (not necessarily systolic) whose correctness is either obvious or easily verifiable. This design is modeled as a finite, rooted, vertex-weighted, edge-weighted, directed multigraph, where nodes represent functional PEs and edges represent interconnections. Weights represent delays of nodes and register delays of interconnections. Transformations are then applied to the original design to obtain a systolic design without global broadcasts (Figure 11.19). The transformations applied include retiming, k-slowdown, broadcast and census elimination, coalescing, interlacing, code motion, resetting, register elimination, and parallel-serial compromises.

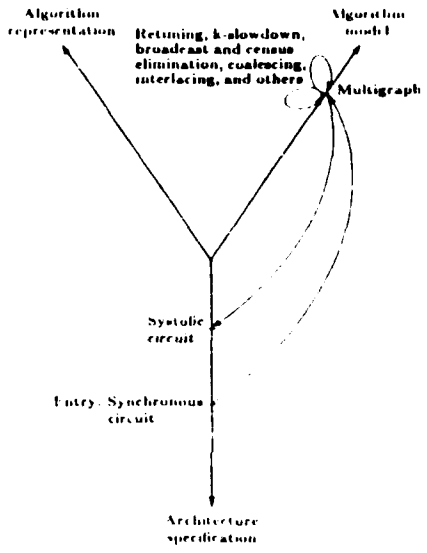


Figure 11.19 Leiserson, Rose, and Saxe's method

Applicability

It applies to any synchronous system.

Capabilities

The function of PEs, layout, and number of pins are preserved by the transformations. Optimal retiming transformations can be selected either by reducing this problem to an efficiently solvable mixed-integer linear-programming problem such that the transformed circuit has the smallest clock period, or by solving a linear-programming dual of a minimum-cost flow problem such that the total number of registers is minimum. Extensions of the optimization procedures used can also take into account fanout, interconnection-bus width, multiple hosts, host timing constraints, and geometric constraints like the number of registers per interconnection.

Results

Systolic designs were derived from synchronous versions of a digital correlator and palindrome recognizer. Other derived circuits include priority

queues, search trees, priority multiqueue, counters, matrix–vector multiplication, matrix–matrix multiplication, and LU decomposition.

11.2.18. Chen and Mead's Method (19, 25)

Description

The goal is to verify that a given systolic design computes the function for which it was intended instead of the generation of a systolic architecture to compute a given function. However, one can see this method as the verification component of a design methodology in which systolic architectures are designed heuristically. Given a systolic architecture, the method generates a CRYSTAL program that describes the algorithm executed by the architecture in a set of space-time equations (19). This representation consists of several equations describing processes executed by local PEs, equations describing connections between PEs, functions representing data streams, and functions describing the relation between the structure of input and output data and the systolic-array structure. From fixed-point theory, the minimum solution of the system of recursive equations is the function computed by the systolic architecture (Figure 11.20).

Applicability

The generality and power of the formalism used makes the methodology widely applicable; however, for the same reason, it is not clear how practical and feasible it is to automate the steps and reasoning involved in this method.

Capabilities

Any systolic array with homogeneous or heterogeneous PEs and interconnections, and synchronous and self-timed systems can be verified.

Results

The method has been demonstrated in verifying the correctness of published designs for synchronous and self-timed systolic architectures for matrix–matrix multiplication.

11.2.19. Kuo, Levy, and Musicus' Method (20)

Description

This method starts from the knowledge of the action and position of each PE in the systolic array, the data "waves" present, their movements, and the way their components are indexed. A "wave" is simply a collection of related data that moves as a block during execution such that the relative positions are preserved (e.g., a matrix). By inspection, Space Time-Data (STD) equations can be derived for each data wave. These equations relate

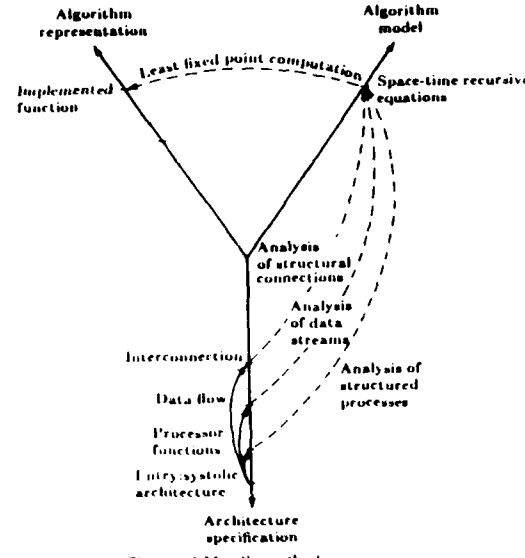


Figure 11.20 Chen and Mead's method

the PE coordinates, time, and indices of data for each wave. Two functions can be derived from the STD equations: the position function and the memory function. The position function gives the coordinates of the PE at which some indexed data arrive at a given instant of time. The memory function is the inverse of the position function and gives the index of the data that arrives at a given PE at a given time. The direction and speed of data movement are described by velocity vectors that correspond to the difference between the coordinates of a PE receiving the data and those of the PE sending the same data. Verification is done by simulating the systolic network to either (a) track the activity of each PE over time by using the memory functions to identify the data being used at any given time, or (b) track each wave of data through the array by using the position functions to identify the PE being visited by a piece of data and the memory functions to identify other data present in that PE. If this simulation does exactly the same operations on the same data as the original algorithm, systolic algorithm is correct (Figure 11.21).

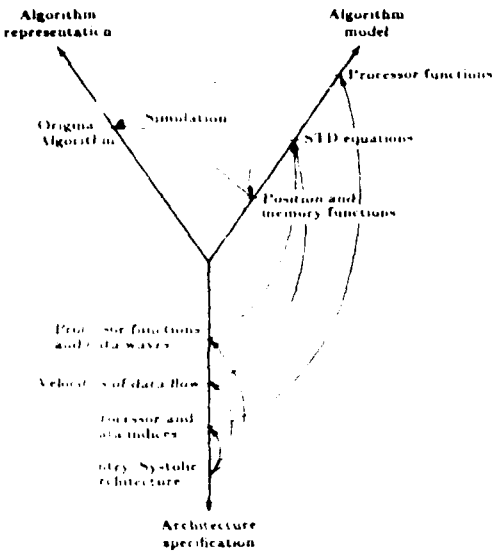


Figure 11.21 Kuo, Levy, and Musicus' method

Applicability
This methodology is best suited for verifying spatially invariant systolic arrays for which data flows are independent of the values computed by the PEs. In practice, this means that the systolic algorithms must have regular dataflows through an array with regular geometry.

Capabilities
Computational wavefronts can be rigorously described by position and memory functions that are linear functions of time, data indices, and PE coordinates.

Results
The verification of matrix–matrix multiplication on a hexagonal array has been reported.

11.3. Final Remarks

In this chapter, we have surveyed 19 systematic methods for synthesizing algorithmically specified VLSI computational arrays. From a global point of view, it is clearly indicated that the two greatest limitations in the state of the art of existing transformational systems are the nonexistence of powerful systematic semantic transformations and the inability to systematically achieve optimality in the resulting designs. This will be the directions of future research in designing better methodologies.

Problems

1. Starting with an algorithmic description of matrix–matrix multiplication, derive the systolic array design of Figure 11.1 using the following methods: (a) Moldovan and Fortes' method, (b) Li and Wah's method, and (c) S. Y. Kung's method. (Hint—review references (75, 88, and 96).)
2. In "A mathematical model for the verification of systolic networks" by R. G. Mehlem and W. C. Rheinboldt (SIAM J. Comp., Vol. 13, No. 3, August 1984), a methodology is proposed for the verification of the correctness of systolic designs. Derive the Y-chart for this method and compare it with the verification methods mentioned in this chapter.
3. In reference (67), six systolic designs are described for the convolution problem. For each of the methodologies mentioned in this chapter, rederive those designs. (Review the relevant references for each methodology because most of them consider convolution as an example.)
4. (Project.) For each of the applications mentioned in Table 11.1, use the methods described in this chapter to verify, design, or redesign the systolic architectures described in the corresponding references. Compare the power, versatility, and effectiveness of the methods with respect to each application. List the main limitations and advantages of each method.

References

1. Agrawal DP, Pathak GC. Design of VLSI based multicomputer architecture for dynamic scene analysis. In Fu KS (ed): *VLSI for Pattern Recognition and Image Processing*, Springer-Verlag, New York, 1984.
2. Apostolico A, Negro A: Systolic algorithms for string manipulations. *Trans. on Computers*, Vol. C-33, No. 4, April 1984, pp. 361–364.
3. Banâtre JP, Frison P, Quinton P. A network for the detection of words in continuous speech. *Acta Informatica*, Vol. 18, 1983, pp. 431–448.

4 Barbe DF: VHSIC systems and technology. *Computer*, Vol. 14, No. 2, Feb. 1981, pp. 13–22.

5 Barnwell TP III, Schwartz DA: Optimal implementations of flow graphs on synchronous multiprocessors. *Proc. 1983 Asilomar Conference on Circuits and Systems*, Pacific Grove, CA, Nov. 1983.

6 Baudet GM, Preparata FP, Vuillemin JE: Area-time optimal VLSI circuits for convolution. *Trans. on Computers*, Vol. C-32, No. 7, July 1983, pp. 684–688.

7 Bentley JL: A parallel algorithm for constructing minimum spanning trees. *Journal of Algorithms*, Jan. 1980, pp. 51–59.

8 Bentley JL, Kung HT: A tree machine for searching problems. *Proc. International Conference on Parallel Processing*, Aug. 1979, pp. 257–266.

9 Blackmon J, Kuekes P, Frank G: A 200 MOPS systolic processor. *Proc. SPIE: Real-Time Signal Processing IV*, Vol. 298, SPIE, 1981.

10 Bojanczyk A, Brent RP, Kung HT: Numerically stable solution of dense systems of linear equations using mesh-connected processors. *Journal on Scientific and Statistical Computing*, Vol. 5, No. 1, March 1984, pp. 95–104.

11 Bongiovanni G: A VLSI network for variable size FFTs. *Trans. on Computers*, Vol. C-2, No. 8, Aug. 1983, pp. 756–760.

12 Bongiovanni G: Two VLSI structures for the discrete Fourier transform. *Trans. on Computers*, Vol. C-32, No. 8, Aug. 1983, pp. 750–754.

13 Brent RP, Kung HT: Systolic VLSI arrays for linear-time GCD computation. In Anceau F, Aas EJ (eds): *VLSI '83*, North-Holland, Aug. 1983.

14 Brent RP, Luk FT, Loan CV: Computation of the singular value decomposition using mesh-connected processors. Technical Report 82-528, Cornell University, Ithaca, NY, March 1983.

15 Capello PR: VLSI Architecture for Digital Signal Processing. Ph.D. Thesis, Princeton University, Princeton, NJ, 1982.

16 Cappello PR, Steiglitz K: Unifying VLSI array designs with geometric transformations. *Proc. International Conference on Parallel Processing*, 1983, pp. 448–45.

17 Cappello PR, Steiglitz K: Unifying VLSI array design with linear transformations of space-time. In Preparata F (ed): *Advances in Computing Research*, JAI Press, Inc., 1984, pp. 23–65.

18 Cappello PR, Steiglitz K: Digital signal processing applications of systolic algorithms. In Kung HT, Sproull RR, Steele G Jr. (eds): *VLSI Systems and Applications*, Computer Science Press, Rockville, MD, 1981.

19 Chen N: Space-Time Algorithms: Semantics and Methodology. Technical Report 5090-TR-83, California Institute of Technology, May 1983.

20 Chen NC: A Methodology for Hierarchical simulation of VLSI Systems. Research Report, YALEU/DCS/RR-325, Yale University, New Haven, CT, Aug. 1984.

21 Chen MC: A Synthesis Method for Systolic Design. Research Report YALEU/DCS/RR-334, Yale University, New Haven, CT, Jan. 1985.

22 Chen MC: Synthesizing Systolic Designs. Research Report YALEU/DCS/RR-374, Yale University, New Haven, CT, March 1985.

23 Chen MC: The Generation of a Class of Multipliers: A Synthesis Approach to the Design of Highly Parallel Algorithms in VLSI. Research Report YALEU/DCS/RR-406, Yale University, New Haven, CT, July 1985.

24 Chen MC: A Parallel Language and its Compilation to Multiprocessor Machines or VLSI. Research Report YALEU/DCS/RR-412, Yale University, New Haven, CT, Aug. 1985.

25 Chen MC, Mead CA: Concurrent algorithms as space-time recursion equations. *Proc. USC Workshop on VLSI and Modern Signal Processing*, University of Southern California, Los Angeles, CA, Nov. 1982, pp. 31–52.

26 Cheng HD: Space-time domain expansion approach to VLSI and its application to pattern recognition and image processing. Ph.D. Thesis, Purdue University, West Lafayette, IN, 1985.

27 Cheng HD, and Fu KS: Algorithm partition for a fixed-size VLSI architecture using space-time domain expansion. *Proc. Seventh Symposium on Computer Arithmetic*, IEEE, June 1985.

28 Cheng HD, Fu KS: VLSI architectures for pattern matching using space-time domain expansion approach. *Proc. International Conference on Computer Design: VLSI in Computers*, IEEE 1985.

29 Cheng HD, Lin WC, Fu KS: *Proc. Seventh International Conference on Pattern Recognition*, IEEE, July 1984.

30 Cheng HD, Lin WC, Fu KS: Space-time domain expansion approach to VLSI and its application to hierarchical scene matching. *Trans. on Pattern Analysis and Machine Intelligence*, IEEE, May 1985, pp. 306–319.

31 Chiang YP, Fu KS: Parallel parsing algorithms and VLSI implementations for syntactic pattern recognition. *Trans. on Pattern Analysis and Machine Intelligence*, IEEE, Vol. PAMI-6, No. 5, May 1984, pp. 578–580.

32 Clarke MJ, Dyer CR: Systolic array for a dynamic programming application. *Proc. 12th Workshop on Applied Imagery Pattern Recognition*, IEEE, 1983.

33 Cohen D: Mathematical approach to iterative computation networks. *Proc. Fourth Symposium on Computer Arithmetic*, IEEE, 1978, pp. 226–238.

34 Cohen D, Tyree V: VLSI system for synthetic aperture radar (SAR) processing. In *Digital Processing of Aerial Images*, Vol. 186, SPIE, May 1979, pp. 166–177.

35 Dyer CR, Clarke MJ: Optimal curve detection in VLSI. *Proc. Conference on Computer Vision and Pattern Recognition*, IEEE, 1983, pp. 161–162.

36 Fisher AL: Systolic algorithms for running order statistics. In Kung HT, et al. (eds): *Signal and Image Processing VLSI Systems and Computations*, Computer Science Press, Rockville, MD, Oct. 1981, pp. 265–271.

37 Fortes JAB: Algorithm transformations for parallel processing and VLSI architecture design. Ph.D. Thesis, University of Southern California, Los Angeles, Dec. 1983.

38 Fortes JAB, Parisi-Presicce F: Optimal linear schedules for the parallel execution of algorithms. *Proc. International Conference on Parallel Processing*, IEEE, Aug. 1984.

39 Fortes JAB, Raghavendra CS: Dynamically reconfigurable fault-tolerant array processors. *Proc. 14th International Conference on Fault-Tolerant Computing*, IEEE, 1984.

40 Fortes JAB, Moldovan DI: Parallelism detection and transformation techniques useful for VLSI algorithms. *Journal of Parallel and Distributed Computing*, Vol. 2, Academic Press, 1985.

41 Fortes JAB, Moldovan DI: Data broadcasting in linearly scheduled array pro-

cessors. *Proc. 11th Annual Symposium on Computer Architecture*, ACM/IEEE, 1984, pp. 224–231.

42. Foster MJ, Kung HT. The design of special-purpose VLSI chips. *Computer*, Vol. 13, No. 1, Jan. 1980, pp. 26–40.
43. Foster MJ, Kung HT. Recognize regular languages with programmable building blocks. *In VLSI 81*, Academic Press, Aug. 1981, pp. 75–84.
44. Fouse SD, Nudd GR, Cumming AD. A VLSI architecture for pattern recognition using residue arithmetic. *Proc. Sixth International Conference on Pattern Recognition*, IEEE, 1982.
45. Frison P, Quinton P. A VLSI parallel machine for speech recognition. *Proc. ICASSP*, 1984, pp. 25B.3.1–25B.3.4.
46. Fu KS. VLSI for Pattern Recognition and Image Processing. Springer-Verlag, New York, 1984.
47. Gajski DL, Kuhn RH. Guest editor's introduction: New VLSI tools. *Computer*, Vol. 16, No. 12, Dec. 1983, pp. 11–14.
48. Gannon L. Pipelining array computations for MIMD parallelism: A functional specification. *Proc. International Conference on Parallel Processing*, 1982, pp. 284–286.
49. Gentleman WM, Kung HT. Matrix triangularization by systolic arrays. *Proc. SPIE, Real-Time Signal Processing IV*, Vol. 298, 1981.
50. Guibas LJ, Kung HT, Thompson CD. Direct VLSI implementation of combinatorial algorithms. *Proc. Conference Very Large Scale Integration*, California Institute of Technology, Jan. 1979, pp. 509–525.
51. Guibas L, Liang FM. Systolic stacks, queues, and counters. *Proc. Conference on Advanced Research in VLSI*, Massachusetts Institute of Technology, Jan. 1982.
52. Horowitz E. VLSI architecture for matrix computations. *Proc. International Conference on Parallel Processing*, Aug. 1979, pp. 124–127.
53. Hwang J, Cheng YH. VLSI computing structure for solving large-scale linear systems of equations. *Proc. International Conference on Parallel Processing*, Aug. 19.., pp. 217–227.
54. Hwang ., Su SP. VLSI architectures for feature extraction and pattern classificatio.. *International Journal on Computer Vision, Graphics, and Image Processing*, Vol. 24, Academic Press, Nov. 1983, pp. 215–228.
55. Ibarra O, Kim S, Palis M. Designing systolic algorithms using sequential machines. *Proc. 25th Annual Symposium on Foundations of Computer Science*, ACM, Oct. 1984, pp. 46–55.
56. Ibarra O, Kim S, Palis M. Some results concerning linear iterative (systolic) arrays. *Journal of Parallel and Distributed Computing*, Vol. 2, 1985, pp. 182–218.
57. Johnsson L, Cohen D. Mathematical approach to modeling the flow of data and control in computational Networks. In Kung HT, et al (eds): *VLSI Systems and Computations*, Computer Science Press, Rockville, MD, 1981, pp. 213–225.
58. Johnsson L, Cohen D. Computational arrays for the discrete ..urier transform. *Proc. COMPCON*, 1981, pp. 236–244.
59. Johnsson L, Weiser U, Cohen D, Davis A. Towards a formal treatment of VLSI arrays. *Proc. Second Caltech Conference on VLSI*, California Institute of Technology, Jan. 1981.

60. Javer JM, Kailath T. Design framework for systolic type arrays. *Proc. ICASSP*, 1984, pp. 8.5.1–8.5.4.
61. Jutand F, Demassieux N, Viard D, Chollet G. VLSI architectures for dynamic time warping using systolic arrays. *Proc. ICASSP*, pp. 34A.5.1–34A.5.4, IEEE, 1984.
62. Kuhn RH. Transforming algorithms for single-stage and VLSI architectures. *Proc. Workshop on Interconnection Networks for Parallel and Distributed Processing*, April 1980, pp. 11–19.
63. Kuhn RH. Optimization and interconnection complexity for parallel processors, single stage networks and decision trees, Ph.D. Thesis, Technical Report 80-1009, University of Illinois, Urbana-Champaign, IL, 1980.
64. Kung HT. Special-purpose devices for signal and image processing: An opportunity in VLSI. *Proc. SPIE, Real-Time Signal Processing III*, Vol. 241, July 1980, pp. 76–84.
65. Kung HT. Highly concurrent systems introduction to VLSI system. In Mead CA, Conway LA (eds): *Introduction to VLSI Systems*, Addison-Wesley, 1980.
66. Kung HT. Use of VLSI in algebraic computation: Some suggestions. *Proc. Symposium on Symbolic and Algebraic Computation*, ACM SIGSAM, Aug. 1981, pp. 218–222.
67. Kung HT. Why systolic architecture. *Computer*, Vol. 15, No. 1, Jan. 1982, pp. 37–46.
68. Kung HT, Lehman PL. Systolic (VLSI) arrays for relational database operations. *Proc. International Conference Management of Data*, ACM SIGMOD, May 1980, pp. 105–116.
69. Kung HT, Leiserson CE. Systolic arrays (for VLSI). In Sparse Matrix Proc. 1978, pp. 256–282.
70. Kung HT, Lin WT. An algebra for VLSI algorithm design. *Proc. Conference on Elliptic Problem Solvers*, Monterey, CA, 1983.
71. Kung HT, Picard RL. Hardware pipelines for multi-dimensional convolution and resampling. *Proc. Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, Nov. 1981, pp. 273–278.
72. Kung HT, Ruanc LM, Yen DWL. A two-level pipelined systolic array for convolutions. In Kung HT, et al (eds): *VLSI Systems and Computations*, Computer Science Press, Rockville, MD, Oct. 1981, pp. 255–264.
73. Kung HT, Song SW. A systolic 2-D convolution chip. Technical Report CMU-CS-81-110, Carnegie-Mellon University, Pittsburgh, PA, March 1981.
74. Kung SY. From transversal filter to VLSI wavefront array. *Proc. International Conference on VLSI*, IFIP, 1983.
75. Kung SY. On supercomputing with systolic/wavefront array processors. *Proc. IEEE*, Vol. 72, No. 7, 1984.
76. Kung SY, Hu YH. A highly concurrent algorithm and pipelined architecture for solving toeplitz systems. *Trans. on Acoustics, Speech, and Signal Processing*, Vol. ASSP-31, No. 1, Feb. 1983, pp. 66–76.
77. Kuo CJ, Levy BC, Musicus BR. The specification and verification of systolic wave algorithms. In *VLSI Signal Processing*, IEEE Press, 1984.
78. Lam M, Mostow J. A transformational model of VLSI systolic design. *IFIP Sixth International Symposium on Computer Hardware Descriptive Languages and their Applications*, Carnegie-Mellon University, Pittsburgh, PA, May 1983.

79. Lehman PL: A systolic (VLSI) array for processing simple relational queries. In Kung HT, Sproull RF, Steele GI (eds): *VLSI Systems and Computations*, Computer Science Press, Rockville, MD, 1981.
80. Leiserson CE: Systolic priority queues. *Proc. Conference on Very Large Scale Integration*, California Institute of Technology, Jan. 1979, pp. 199–214.
81. Leiserson CE: Systolic and semisystolic design (extended abstract). *Proc. International Conference on Computer Design/VLSI in Computers*, IEEE 1983.
82. Leiserson CE: Area-efficient VLSI Computations. Massachusetts Institute of Technology Press, Cambridge, MA, 1983.
83. Leiserson CE, Rose FM, Saxe JB: Optimizing synchronous circuitry by retiming. In Bryant R (ed): *Proc. Third Caltech Conference on Very Large Scale Integration*, Computer Science Press, Rockville, MD, 1983.
84. Leiserson CE, Saxe JB: Optimizing synchronous systems. *Twenty-second Annual Symposium on Foundations of Computer Science*, ACM, Oct. 1981, pp. 23–36.
85. Li GJ: Array pipelining algorithms and pipelined array processors. M Sc. Thesis, Institute of Computer Technology, Chinese Academy of Science, Beijing, China, 1981.
86. Li G, Wah BW: Optimal design of systolic arrays for image processing. *Proc. Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, IEEE, Oct. 1983, pp. 134–141.
87. Li G, Wah BW: The design of optimal systolic algorithms. *Proc. Computer Software and Applications Conference*, IEEE, 1983, pp. 310–319.
88. Li G, Wah BW: The design of optimal systolic arrays. *Trans. on Computers*, Vol. C-34, No. 10, Jan. 1985, pp. 66–77.
89. Liu LH, Hu KS: VLSI algorithm for minimum distance classification. *Proc. International Conference on Computer Design*, IEEE, 1983.
90. Liu LH, Fu KS: A VLSI systolic processor for fast seismic signal classification. *Proc. International Symposium on VLSI Technology, Systems and Applications*, Taipei, Taiwan, March 30–April 1, 1983.
91. Liu LS, Young TY: VLSI array architecture for picture processing. In Fu KS, Kunii TL (eds): *Picture Engineering*, Springer Verlag, New York, Vol. 6, 1982.
92. McCanny JV, McWhirter JG: Implementation of signal processing functions using 1-bit systolic arrays. *Electronic Letters*, Vol. 18, No. 6, March 1982, pp. 241–43.
93. McCanny JV, McWhirter JG: Completely iterative, pipelined multiplier array suitable for VLSI. *Proc. IEEE*, Vol. 129, No. 2, April 1982, pp. 40–46.
94. McWhirter JG: Systolic array for recursive least square minimization. *Electronic Letters*, Vol. 19, No. 18, Sept. 1983, pp. 729–730.
95. Miranker WL, Winkler A: Space-time representations of computational structures. *Computing*, Vol. 32, 1984, pp. 93–114.
96. Moldovan DI: On the analysis and synthesis of VLSI algorithms. *Trans. on Computers*, Vol. C-31, No. 11, Nov. 1982, pp. 1121–1126.
97. Moldovan DI: On the design of algorithms for VLSI systolic arrays. *Proc. IEEE*, Vol. 1, No. 1, Jan. 1983, pp. 113–120.
98. Moldovan DI, Fortes JAB: Partitioning of algorithms for fixed size VLSI architectures. Technical Report PPP-83-5, Department of Electrical Engineering Systems, University of Southern California, Los Angeles, CA, 1983.

99. Moldovan DI, Varma A: Design of algorithmically specialized VLSI devices. *Proc. International Conference on Computer Design: VLSI in Computers*, 1983, pp. 88–91.
100. Moldovan DI, Wu CI, Fortes JAB: Mapping an arbitrarily large QR algorithm into a fixed size VLSI array. *Proc. International Conference on Parallel Processing*, Aug. 1984.
101. Mostow J, Lam M: Transformational VLSI design: A progress report. Unpublished manuscript.
102. Narayan SS, Nash JG, Nudd GR: VLSI processor array for adaptive radar applications. *Proc. of SPIE 27th International Technical Symposium*, 1983.
103. Ni LM, Jain AK: Design of a pattern cluster using two-level pipelined systolic array. In Fu KS (ed): *VLSI for Pattern Recognition and Image Processing*, Springer-Verlag, New York, 1984.
104. Nishida S: Application of mapping algorithm to VLSI architectures. Technical Report, Department of Engineering Systems, University of Southern California, Los Angeles, 1984.
105. Ottmann T, Rosenberg AL, Stockmeyer LJ: A dictionary machine (for VLSI). Technical Report RC9060, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1981.
106. Quinton P: The Systematic design of systolic arrays. Technical Report 193, IRISA, April 1983.
107. Quinton P: Automatic synthesis of systolic arrays from uniform recurrent equations. *Proc. 11th Annual Symposium on Computer Architecture*, ACM/IEEE, 1984, pp. 208–214.
108. Ramakrishnan IV, Fussell DS, Silberschatz A: On mapping homogeneous graphs on a linear array-processor model. *Proc. International Conference on Parallel Processing*, IEEE, 1983, pp. 440–447.
109. Savage, C: A systolic data structure chip for connectivity problems. In Kung HT, Sproull RF, Steele GL Jr. (eds): *VLSI Systems and Computations*, Computer Science Press, Rockville, MD, 1981.
110. Schwartz DA, Barnwell TP III: A graph theoretic technique for the generation of systolic implementations for shift-invariant flow graphs. *Proc. ICASSP, IEEE*, 1984, pp. 8.3.1–8.3.4.
111. Song SW: On a high-performance VLSI solution to database problems. Ph D Dissertation Carnegie-Mellon University, Pittsburgh, PA, July 1981.
112. Stroll Z, Kang SC: VLSI-based image resampling for electronic publishing. In Fu KS (ed): *VLSI for Pattern Recognition and Image Processing*, Springer-Verlag, New York, 1984.
113. Travassos RT: Real-time implementation of systolic Kalman filters. Technical Report, Systolic Systems Inc., 1983.
114. Treleaven PC: VLSI processor architectures. *Computer*, Vol. 15, No. 6, June 1982, pp. 33–45.
115. Tur M, Goodman JW, Moslehi B, et al: Fiber-optic signal processor with applications to matrix-vector multiplication and lattice filtering. *Optics Letters*, Vol. 7, No. 9, Sept. 1982, pp. 463–465.
116. Ullman JD: Computational Aspects of VLSI. Computer Science Press, Rockville, MD, 1984.
117. Varman PJ, Fussell DS, Ramakrishnan IV, Silberschatz A: Robust systolic

algorithms for relational database operations. *Proc. Symposium on Real Time Systems*, Dec. 1983.

118. Varman PJ, Ramakrishnan IV. Dynamic programming and transitive closure on linear pipelines. *Proc. International Conference on Parallel Processing*, 1984, pp. 359–364.

119. Weiser V, Davis A. A wavefront notion tool for VLSI array design. In Kung HT, et al (eds). *VLSI Systems and Computations*. Computer Science Press, Rockville, MD, 1981.

120. Weste V, Burr DJ, Ackland BD. Dynamic time warp pattern matching using an integrated multiprocessing array. *Trans. On Computers*, Vol. C-32, No. 8, Aug. 1983, pp. 731–744.

121. Wing O. A content-addressable systolic array for sparse matrix computation. Technical Report, Columbia University, New York, 1983.

122. Yeh CS, Reed IS, Truong TK. Systolic multipliers for finite fields. *Trans. on Computers*, Vol. C-33, No. 4, April 1984, pp. 357–360.

123. Yen DW, Kulkarni AV. Systolic processing and an implementation for signal and image processing. *Trans. on Computers*, Vol. C-31, No. 10, Oct. 1982, pp. 1000–1008.

124. Zhang C-N, Yun DYY. An area-time optimal systolic network for discrete Fourier transform. *Proc. 11th Annual International Symposium on Computer Architecture*, ACM/IEEE, June 1984.

REFERENCE NO. 13

Fortes, J. A. B. and O'Keefe, M. T., "Current Architectures and a Tool for the Design and Programming of Bit Level Processor Arrays," 1988 IEEE Int. Symposium on Circuits and Systems, June 1988.

Current Architectures and a Tool for the Design and Programming of Bit Level Processor Arrays

Jose A. B. Fortes and Matthew T. O'Keefe [+]

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Extended Abstract

This paper surveys current bit level processor array architectures and describes a tool for designing and programming these arrays. The survey emphasizes arrays that have been implemented rather than proposed architectures. The essential features shared by these arrays, and those that differentiate them are characterized and used to develop a taxonomy for bit level processor arrays. The second part of the paper discusses programming tools, with an emphasis on RAB, a large program used to map a class of algorithms written in 'C' onto bit level processor arrays. The basic components and extensions to RAB are discussed, along with examples which include the mapping of arithmetic, numeric and neural network algorithms onto processor arrays. Directions for future research and design of bit-level processor array architectures and their programming environments are also discussed.

[+] This work was supported in part by the National Science Foundation under Grant DMC-8419745 and in part by the Office of Naval Research under contract no. 00014-85-k-0588.

I. Introduction

The rapid pace of innovation in VLSI has provided an implementation medium for highly parallel computer architectures. To fully realize the tremendous computing power of VLSI technology requires that its characteristics, often subtle and complex, be understood. First, the design complexity introduced by the increasing chip size and density calls for architectures composed of repetitive, modular structures. Second, interconnections between devices consume more power and require more area than the devices (transistors) themselves. Third, the computing environment offered by VLSI is I/O-bound, not compute-bound, due to the limited number of I/O pins available in comparison to logic gates. These considerations have led to the development of highly parallel bit level processor arrays, which are distinguished by their communication strategy - digital signals are transmitted bit sequentially on single wires as opposed to simultaneous transmission on parallel busses - and their large number of simple 1-bit word processing elements (PEs). This leads to efficient communication both internally and between chips and provides a high degree of parallelism. The regular, repeatable structures inherent in bit level arrays have the following advantages over other structures implemented using VLSI:

(a) design complexity is reduced, an important feature as VLSI densities reach millions of transistors per chip;
(b) several techniques for introducing fault-tolerance into such structures are available, and this aspect is particularly important in wafer scale integration implementations;
(c) functional verification of the design is simpler;
(d) high packing densities are possible as the chip designer can concentrate on optimising a single cell which is then repeated, as is done in memory circuits;
(e) the arrays are scalable to higher VLSI densities and can be pipelined to the bit level to provide very high throughput rates;
(f) measures of active resources, i.e., the percentage of logic gates and memory involved in a computation, favor large bit level arrays and indicate that they have the potential for very large throughput.

The advantages from an algorithmic standpoint, resulting from the flexibility at the bit level, include:

(a) using symmetries and optimizations that are possible at the bit level to reduce computation time, e.g., squaring can take one-half the time necessary for multiplication;
(b) the precision used in the array can be matched to that necessary for a particular computation;
(c) some of the more flexible processor arrays can change the level of parallelism available to match the parallelism contained in an algorithm.

II. Current Architectures

Bit level processor arrays are characterized by a set of four features:

- the basic building blocks are simple, modular, and repeatable processing elements that can be placed in large numbers on a single VLSI chip
- computation and communication are performed at the bit level
- SIMD (single instruction multiple data) control, with some variations, is used reduce the complexity of control for these very large arrays and amortize the expensive control hardware over many PEs
- the dominating feature of these arrays is potentially massive concurrency realized through VLSI architectures.

Within the context of these features, we will discuss five classes of bit level arrays and architectures representative of each class. These five classes, shown in Table 1 with representative systems, are:

1) systolic arrays;
2) image processing arrays;
3) reconfigurable, or adaptive, arrays;
4) associative processing arrays;
5) high level network arrays.

Bit level systolic arrays have been developed to perform several digital signal processing tasks, including convolution and correlation, rank-order filtering, and the Discrete Fourier Transform (DFT) [1]. Several of these chips are now sold as commercial products. These arrays are specialized to perform one particular algorithm, and each processing element is optimized for the particular algorithm being implemented. This fact, along with the systolic concepts of extensive pipelining and local communication applied down to the bit level yield extremely fast clock speeds, as the clock cycle time is reduced to that of the bit level processing element.

Image processing arrays include the [illegible] APP chip [2], Goodyear Aerospace MPP [3], GEC GRID chip [4], and the University Co[illegible] London CLIP [5]. These bit level arrays are oriented towards image processing as [illegible]r primary application, and have special image manipulation features in hardware suc[illegible] as data reformatting buffers, bit plane I/O, and processing elements optimized for certai[illegible] image transformations. PAPIA [6] is a processor array using a pyramid architecture to rea[illegible]e multiresolution pipelined image processing; each PE in PAPIA operates at the bit level, and has connections to 4 PEs (sons) on a lower plane and a singe PE (father) in a higher plane.

Reconfigurable arrays have the ability to adapt to the different degrees of concurrency available for different algorithms and within the same algorithm. Bit level processor arrays representative of this class include the ICL DAP [7], NTT's AAP chip [8], and the University of Southampton RPA [9]. In the latter two arrays, part of the microcode word is held locally within each PE, allowing some degree of independence in both computation and in communication with other PEs.

Associative processing arrays include the Brunel University SCAPE [10] and the Airborne Associative Processor (ASPRO) [3], a VLSI version of the STARAN associative processor [11]. The salient feature of these processors is a content-addressable memory that is used to perform bit level operations in parallel. These arrays are particularly adept at fast and efficient searching.

High level network arrays have the same basic architecture as other bit level processor arrays, but in addition to a nearest neighbor network these arrays will have, for example, a hypercube, cube connected cycles, or multistage cube interconnection network. These high level networks provide high bandwidth communication paths between non-neighbor processing elements, a requirement for many algorithms. Examples of this class include the Connection Machine [12] (hypercube), the DEC Massively Parallel Architecture [13] (multistage cube), and the Boolean Vector Machine [14] (cube connected cycles).

Future architectures could combine the features of reconfigurable arrays with high level networks to provide arrays with varying degrees of parallelism and flexible communication. This approach holds the promise of efficiently mapping a broader class of algorithms onto these arrays.

A more comprehensive survey and elaborate taxonomy will appear in the final paper.

Bit Level Processor Arrays				
Systolic	Image Processing	Reconfigurable	Associative Processing	High Level Network
DFT	GAPP	AAP	SCAPE	DEC MPA
Rank-Order Filter	MPP	RPA	ASPRO	Connection Machine
Convolution	Grid	DAP		Boolean Vector Machine
	CLIP			
	PAPIA			

Table 1

III. Programming and Design Tools

Despite the advantages of bit level processor arrays in both VLSI implementation and algorithm execution, they can be difficult to program (in the case of an existing general purpose architecture) or design (in the case of a special purpose architecture). This problem is accentuated by the need to implement high level computations (e.g., matrix computations, convolution) using bitwise operations. For the architectures described previously, several approaches to this problem have been used. These include:

- subroutine libraries accessed through a standard high-level language which have been optimized for a particular architecture
- parallel languages, which are often architecture or machine dependent
- microcoded routines to handle standard word level operations in a general way, without using bit level symmetries or optimizations.

These approaches lack portability among different machines and sometimes ignore optimizations possible at the bit level. It is often difficult to prove the optimality of a given mapping using these methods. In order to solve the problems associated with current approaches, it is desirable to develop methodologies and tools which enable the systematic mapping of algorithms onto processor arrays. In the past, several research efforts have been pursued in this direction and a good survey can be found in [15]. Many of these methodologies, which were intended for word level processor arrays, are applicable to bit level arrays. However, besides some of the limitations that still characterize those methodologies, systematic bit level designs present additional problems. RAB (Reconfiguration Algorithm for Bit level code), an automated design tool which maps a class of algorithms programmed in 'C' into bit level arrays, represents an attempt to understand and solve the open questions and problems involved in the systematic design of bit level processor arrays.

In practice, potential users of processor arrays are given an algorithm and must devise means for its execution using one of the following options: (1) use an existing processor array, (2) design a special purpose processor array, or (3) design an array that uses a number of existing small processor array modules as the basic components. Option (1) requires mapping the algorithm into an existing array taking into consideration size limitations, fixed interconnection schemes, and predesigned processing elements. In this option, which we refer to as full mapping, the programming decisions are subordinated to the characteristics of the array. Option (2) allows the user to design the hardware taking into consideration only the characteristics of the algorithm and perhaps some rather general VLSI design constraints (i.e., planarity, limited pinout, etc). This option is referred to as full design. It corresponds to the front end of a silicon compiler, and could provide the structural and behavioral description of an array to the placement and layout tools of the silicon compiler. Option (3) is a compromise between full mapping and full design, where the designer can decide the overall organization (i.e., shape, size, interfaces) of the array, but uses given basic blocks which are themselves fully defined "small" processor arrays. We refer to this option as partial mapping/design.

The input to RAB consists of C programs which describe word level algorithms. These algorithms correspond to nested **for** loops with static behavior. RAB first expands the computations in the input program into bit level operations as shown in Figure 1. This expansion phase replaces word level computations with a bit level implementation of the arithmetic operations using a library of macro expansions. This phase is followed by data dependence and broadcast analysis using the Dependence Arc Set Analysis technique [16]. The result of this analysis is a formal description of the internal structure of the bit level algorithm. This structural information is used to generate an algorithm transformation which yields a restructured algorithm suitable for mapping onto a bit level processor array. The mapping may be a full design of an algorithmically defined array or full (partial) mapping for a fixed (variable) size array corresponding to the fourth level of modules in Figure 1. The transformed algorithm structure is then converted into an intermediate representation which can used to generate code for several different architectures. The last two modules in Figure 1, code generation and code optimization, comprise the phase in which code is generated from the intermediate representation for a particular target architecture. This code is then optimized using a standard compaction technique. RAB has been applied to numerical, arithmetic and neural network algorithms.

IV. Summary

This paper describes bit level processor arrays and the characteristics that make them ideal for VLSI and high speed computations. It presents a taxonomy for current bit level arrays that provides different perspectives on this class of architectures. The design and programming problem is addressed, and an automated design tool, RAB, is presented as a solution to some of these problems. RAB has the twin advantages of portability among different architectures and systematic optimization techniques down to the bit level.

References

[1] J.V. McCanny and J.G. McWhirter, "Some Systolic Array Developments in the United Kingdom," *IEEE Computer*, Vol. 20, No. 7, pp. 51-64, July 1987.

[2] R. Davis and D. Thomas, "Systolic Array Chip Matches the Pace of High-Speed Processing," *Electronic Design* Oct. 31, 1984.

[3] K.E. Batcher, "Bit-Serial Parallel Processing Systems," *IEEE Trans. on Computers*, Vol. C-31, No. 5, pp. 377-384.

[4] D.K. Arvind, I.N. Robinson, I.N. Parker, "A VLSI Chip for Real-Time Image Processing," *Proc. IEEE Symp. on Circuits and Systems*, pp. 405-408, May 1983.

[5] M.J.B. Duff and T.J. Fountain, **Cellular Logic Image Processing**, Academic Press:

London, 1986.

[6] V. Cantoni, *et al,* "The PAPIA Image Analysis System," *SPIE Vol. 596, Arch. and Alg. for Dig. Image Processing (1985),* pp. 88-95.

[7] R.W. Hockney and C.R. Jesshope, **Parallel Computers: Architecture, Programming and Algorithms,** Adam Hilger Ltd.: Bristol, 1981, pp.178-192.

[8] T. Kondo, *et al,* "An LSI Adaptive Array Processor," *IEEE Journal Solid-State Circ.,* Vol. SC-18, No. 2, pp. 147-156.

[9] C.R. Jesshope, *et al,* "The Structure and Application of RPA - A Highly Parallel Adaptive Architecture," **Highly Parallel Computers,** Elsevier Science Publishers, 1987, pp.81-95.

[10] I.P. Jalowiecki, "A 256-Element Associative Parallel Processor," *1987 IEEE Int. Solid-State Circuits Conf.,* pp. 196-197.

[11] K.E. Batcher, "STARAN Series E," *Proc. 1977 Int. Conf. Parallel Processing,* Aug. 1977, pp. 140-143.

[12] W.D. Hillis, **The Connection Machine,** MIT Press: Cambridge, MA, 1985.

[13] R. Grondalski, "A VLSI Chip Set for a Massively Parallel Architecture," *1987 IEEE Int. Solid-State Circuits Conf.,* pp. 198-199.

[14] R. Wagner, "The Boolean Vector Machine (BVM)," *Proc. 10th Ann. Symp. on Computer Architecture,* pp. 59-66.

[15] J.A.B. Fortes, B.W. Wah, and K.S. Fu, "Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays," *IEEE Int. Conf. Acoustics, Speech, and Signal Processing,* 1985, pp.300-303.

[16] D.J. Kuck, *et al,* "Dependence Graphs and Compiler Optimizations," *Proc. 8th ACM Symp. on Principles of Programming Languages,* Jan. 1981, pp. 207-218.

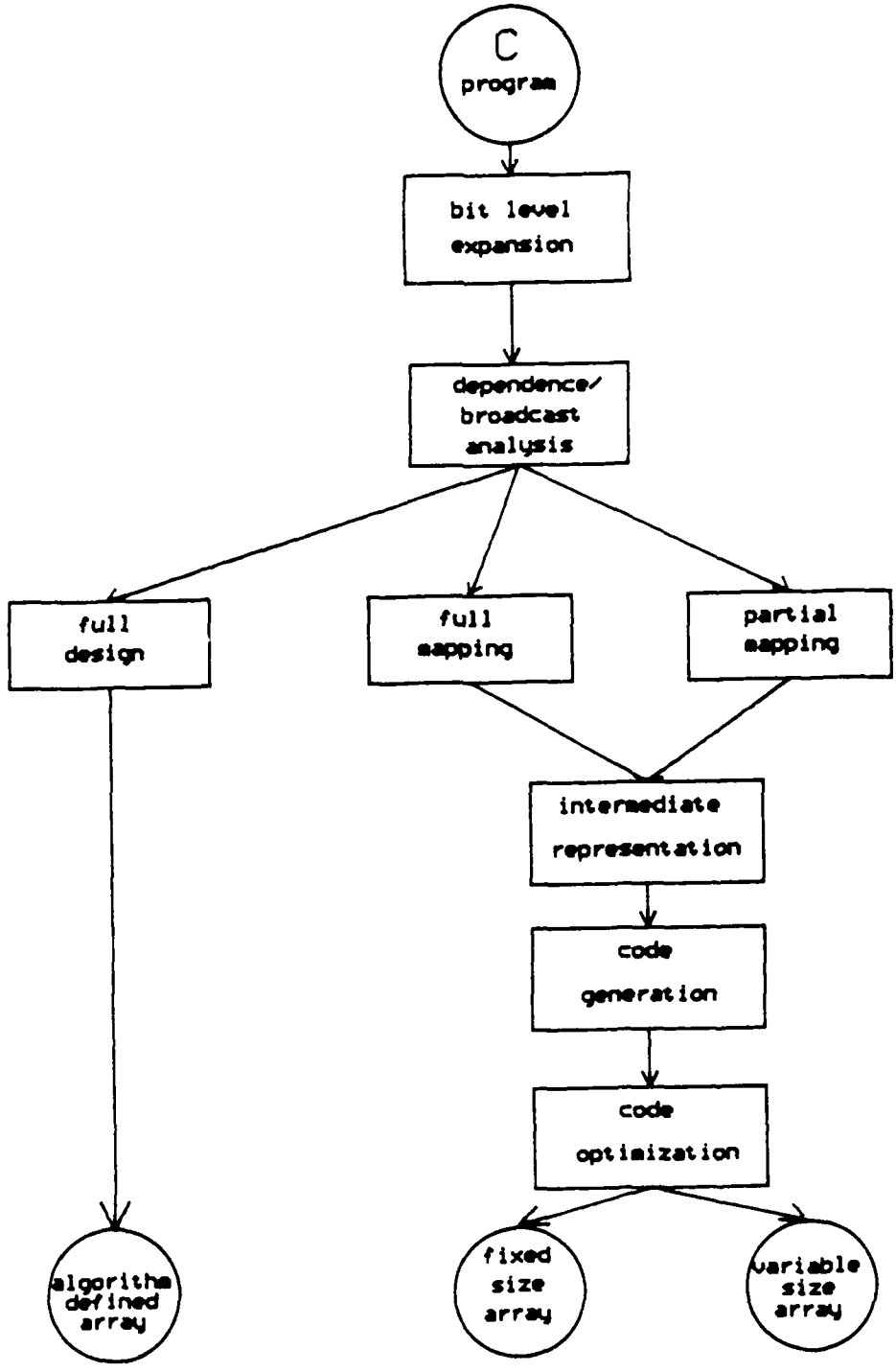


Figure 1. Flow diagram of RAB.

END

DATE

FILMED

5-88

DTIC