

where h is the decision tree height, or the number of radial cuts, whose i cross edges can be chosen independently:

$$h = (f^{L-k} - 1)/(f - 1).$$

Therefore, the number of permutations having the number of cross edges $i = 1$ in a stage- k subgraph of an $(3, L)$ CC-banyan is

$$C_{3,k}^1 = 3^{(3^{L-k} - 1)/2}, \quad 0 \leq k \leq L-2. \quad (4)$$

Now we can evaluate the total number of permutations in CC-banyan with fan-out 3.

Theorem 6: The number of permutations in an (f, L) CC-banyan with fan-out $f = 3$ is

$$P = (3!)^{3^{L-1}} * \prod_{k=0}^{L-2} [2 + 3^{(3^{L-k} - 1)/2}] 3^k.$$

Proof: The expression for the total number of permutations follows immediately from (2), (3), and (4). []

The method of decision trees can also be used for enumeration of all permutations in the case of general fan-out f . We have been able to evaluate the number of permutations in CC-banyans with fan-out $f = 4$. However, the process of constructing decision trees becomes quite complex and elaborate for large values of f . This is due to the fact that the analysis of all possible permutations in a radial sector of size $(f - 1)$ becomes a complex combinatorial problem by itself, when f is large. Even so, for practical purposes, the number of permutations for larger fan-outs can be easily obtained by combining analytical expressions (2) and (3) with computer-generated exhaustive enumeration for $C_{f,k}^1$.

V. CONCLUSIONS

In this correspondence, we presented a graph-theoretic approach to the analysis of rectangular CC-banyan networks with an arbitrary fan-out f and an arbitrary number of stages L . It is shown that CC-banyans, like many other networks (e.g., SW-banyans, Benes networks) can be constructed recursively from the networks of smaller size. This recursiveness enables the modular structure of CC-banyans and can be used in the analysis of its partitioning properties.

We also presented a general approach to the analysis of permuting properties of CC-banyans. The analytical expressions for the number of permutations performable by CC-banyan with fan-outs 2 and 3 are derived. In the case of general fan-out f , we proposed a method, based on decision tree analysis, for a systematic enumeration of all permutations.

REFERENCES

- [1] G. J. Lipovski and M. Malek, *Parallel Computing*. New York: Wiley-Interscience, 1987.
- [2] C.-L. Wu and T. Y. Feng, "On a class of multistage interconnection networks," *IEEE Trans. Comput.*, vol. C-29, pp. 694-702, Aug. 1980.
- [3] D. P. Agrawal, "Graph theoretical analysis and design of multistage interconnection networks," *IEEE Trans. Comput.*, vol. C-32, pp. 637-648, July 1983.
- [4] V. Cherkassky, "Performance of non-rectangular multistage interconnection networks," in *Proc. Int. Conf. Distrib. Comput. Syst.*, May 1986, pp. 2-7.
- [5] U. V. Premkumar, R. Kapur, M. Malek, G. J. Lipovski, and P. Horne, "Design and implementation of the banyan interconnection network in TRAC," in *AFIPS Conf. Proc.*, vol. 49, NCC, Los Angeles, CA, 1980, pp. 643-653.
- [6] L. R. Goke and G. J. Lipovski, "Banyan networks for partitioning multiprocessor system," in *Proc. 1st Annu. Symp. Comput. Architecture*, Dec. 1973, pp. 21-28.
- [7] D. P. Agrawal and S.-C. Kim, "On non-equivalent multistage interconnection networks," in *Proc. 1981 Int. Conf. Parallel Processing*, Aug. 1981, pp. 234-237.
- [8] V. Cherkassky and E. Opper, "Fault diagnosis and permuting properties of CC-banyan networks," in *Proc. Real-Time Syst. Symp.*, Dec. 1984, pp. 175-183.
- [9] J. H. Patel, "Performance of processor-memory interconnections for multiprocessors," *IEEE Trans. Comput.*, vol. C-30, pp. 771-770, Oct. 1981.
- [10] R. J. McMillen and H. J. Siegel, "Routing schemes for the augmented data manipulator network in MIMD system," *IEEE Trans. Comput.*, vol. C-31, pp. 1202-1214, Dec. 1982.
- [11] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnik, and R. A. Stokes, "The ILLIAC IV computer," *IEEE Trans. Comput.*, vol. C-17, pp. 746-757, Aug. 1968.
- [12] U. V. Premkumar and J. C. Browne, "Resource allocation in rectangular SW-banyans," in *Proc. 9th Annu. Symp. Comput. Architecture*, Apr. 1982, pp. 326-333.
- [13] V. Cherkassky and M. Malek, "On permuting properties of regular rectangular SW-banyans," *IEEE Trans. Comput.*, vol. C-34, pp. 542-546, June 1985.
- [14] H. J. Siegel, "The theory underlying the partitioning of permutation networks," *IEEE Trans. Comput.*, vol. C-29, pp. 791-801, Sept. 1980.
- [15] C. Wu and T. Feng, "The reverse-exchange interconnection network," *IEEE Trans. Comput.*, vol. C-29, pp. 801-811, Sept. 1980.
- [16] D. C. Opferman and N. T. Tsao-Wu, "On a class of rearrangeable switching networks, parts I and II," *Bell Syst. Tech. J.*, pp. 1579-1618, May-June 1971.
- [17] E. Opper, M. Malek, and G. J. Lipovski, "Resource allocation in rectangular CC-banyans," in *Proc. 10th Annu. Symp. Comput. Architecture*, June 1983, pp. 178-184.

Optimal Matrix Multiplication on Fault-Tolerant VLSI Arrays

P. J. VARMAN AND I. V. RAMAKRISHNAN

Abstract—The Diogenes methodology, proposed by Rosenberg, for the design of easily testable and configurable fault-tolerant VLSI arrays, results in *collinear* layouts of processors (PE's) that are configured into the desired array structure by appropriate switch settings on buses running parallel to the PE's. While possessing attractive mechanisms for fault-tolerant implementations, Diogenes designs of two-dimensional (2-D) arrays require more area than a two-dimensional implementation and result in long wires between logically adjacent PE's.

In this paper, we present a collinear VLSI array that retains all the desirable fault-tolerance characteristics of Diogenes designs but avoids the degradation in throughput (caused by a lower system clock rate) that long inter-PE wire lengths would impose. Just as in the systolic model, all signals in our array travel a fixed physical distance in any clock cycle.

On this model, we show a lower bound of $\Omega(n\sqrt{n})$ on the time complexity required to multiply two $n \times n$ matrices by computing the n^2 scalar products. Furthermore, we present an *optimal* $O(n\sqrt{n})$ time systolic algorithm using $n\sqrt{n}$ PE's and requiring $O(n^2)$ area. Our algorithm is superior in time performance and/or area requirements to previous matrix multiplication algorithms on this model.

The use of *retiming* enables us to maintain the correctness of our

Manuscript received July 31, 1985; revised March 6, 1986 and January 7, 1988. P. J. Varman was supported by and IBM Faculty Development Award and I. V. Ramakrishnan was supported by ONR Grant N00014-84-K-0530 and NSF Grant ECS-84-04399.

P. J. Varman is with the Department of Electrical and Computer Engineering, Rice University, Houston, TX 77001.

I. V. Ramakrishnan is with the Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794.

IEEE Log Number 8820889.

algorithm despite the presence of faulty PE's. Since the clock rate in our design is independent of the number and distribution of faulty PE's, their presence contributes only an *additive* increase (equal to the number of bypassed faulty PE's) to the time complexity of our algorithm.

Index Terms—Collinear, Diogenes, fault tolerance, retiming, systolic

I. INTRODUCTION

Wafer-scale integration [13] of processor arrays requires development of effective fault-tolerance techniques to handle the *ubiquitous* presence of defective processing elements. Such techniques involve testing for faulty PE's and configuring the working processors into the desired array structure. Several approaches to the design of fault-tolerant arrays of processors, with a view towards wafer-scale integration, have been proposed in the recent past [5], [6], [8], [9], [11], [19]. The designs resulting from all of these varied techniques, however, either ignore or fall short of meeting some of the desirable features of an effective fault-tolerant design—that the design facilitate testability, be easily reconfigurable, have short inter-PE wire lengths, and make good utilization of nonfaulty PE's.

In [16], Rosenberg proposed the Diogenes methodology for the design of fault-tolerant VLSI arrays. The essence of this methodology is the *linearization* of processor networks which are mapped onto collinear layouts of processors, and configured into the desired array structure by appropriate switch settings on buses running parallel to the PE's. It provides scan-in/scan-out capability to enhance testability and configuration is achieved using a few control lines per processor. (The reader is referred to [16] for a detailed description of this methodology and for a comparison between it and the approaches in [5], [6], [8], [9], [11], and [19].)

While possessing mechanisms for convenient testing and configuration necessary for fault tolerance, Diogenes designs of two-dimensional mesh arrays occupy a significantly larger area than a two-dimensional implementation and require the wires connecting logically adjacent PE's to span a large physical distance. Note that this is the case even if all the PE's in the array are assumed to be fault free. The long wires, inherent in such an approach, necessitate the use of a slower system clock with a corresponding reduction in system throughput. Fig. 1 below illustrates a collinear implementation of a 3×3 mesh array obtained using the Diogenes methodology.

In this paper, we propose a model of a fault-tolerant VLSI matrix multiplication array that retains all the excellent fault-tolerance characteristics of Diogenes designs, but avoids the degradation in throughput (caused by a lower system clock rate) that long inter-PE wire lengths would impose.

Informally, our model consists of a collinear layout of processors that communicate by a multibus structure running parallel to the line of PE's. In order to make the clock rate independent of the inter-PE wire length, we introduce buffers in the buses at fixed physical separations (say between every pair of physically adjacent PE's) so that signals are clocked in and out of these buffers at every cycle.

In common with other models [5], [6], [8], [9], [11], [16], [19], we assume that the buses and switches are completely reliable. Fault tolerance is achieved by bypassing the faulty PE's and hooking only the working processors onto the buses. These use of retiming [9], [19] enables us to maintain the correctness of our algorithm despite the presence of faulty PE's.

The main results in this paper are the following. On our model (formalized in the next section), we establish a lower bound of $\Omega(n\sqrt{n})$ on the time required to multiply two $n \times n$ matrices by any algorithm that computes the n^3 scalar products. Furthermore, we present an *optimal* $O(n\sqrt{n})$ time *systolic algorithm* [10] that uses $n\sqrt{n}$ PE's and requires an *optimal* $O(n^2)$ area. From the viewpoint of fault tolerance, since the clock rate in our design is independent of the number and the distribution of faulty PE's, their presence contributes only an *additive* increase (equal to the number of bypassed faulty PE's) to the time complexity of our algorithm.

The algorithm described in this paper is superior in time performance to matrix multiplication algorithms on a linear array that

require $O(n^2)$ time and $O(n^2)$ area [1], [7], [14], [15]. A naive simulation of the well-known two-dimensional systolic array algorithm for matrix multiplication [10] on the collinear pipelined structure obtained by linearizing a two-dimensional mesh array (a-la Diogenes) also requires $O(n^2)$ time and uses $O(n^3)$ area. This may be seen by observing that every communication step that transferred data elements between adjacent PE's along a column in the two-dimensional mesh would now require $n - 1$ clock cycles to move the element between the same two PE's in the collinear-pipelined structure (assuming linear propagation delay). Thus, the $O(n)$ time two-dimensional systolic algorithm would require $O(n^2)$ time to execute on this collinear pipelined implementation. By observing that the collinear structure consists of n^2 PE's and $2n$ parallel buses, it follows that such an implementation requires $O(n^3)$ area. More interestingly (as discussed further in Section II), a consequence of the lower bound results of Gentleman [4] is that *any* matrix multiplication algorithm on this structure would require $O(n^2)$ time.

The rest of this paper is organized as follows. In the following section, we establish the lower bound result. In Section III, we will describe the processor array and the matrix multiplication algorithm that executes on it. For ease of exposition, our algorithm will be described for a fault-free processor array. The proof of correctness of the algorithm appears in [18]. In Section IV, we discuss an implementation similar in spirit to the approach in [16] with additional hardware to incorporate retiming necessary to ensure the correctness of our algorithm in the presence of faulty PE's. Finally, comparisons to other algorithms on this model and conclusions are presented in Section V.

II. COMPUTATION MODEL

The computation model consists of an unlimited number of processors each with its own local memory and arranged unit distances apart along a line. The processors are interconnected by a system of buses running parallel to the line along which the processors are arranged. Each processor has an index from the set of integers $\{\dots - 1, 0, 1 \dots\}$ and we will refer to the processor with index i as PE_i . The arrangement of PE's may be visualized as illustrated in Fig. 2.

We will assume that it takes $c|j - i|$ ($c \geq 0$ is some constant) time steps to transfer a data element from PE_i to PE_j . This, in k ($k \geq 0$) time steps, a data element initially at PE_i can be made available to at most $2k + 1$ PE's, namely, to those with indexes in the range $[i - k, \dots, i + k]$. Note, however, that this model does not forbid the use of pipelining within the interconnection network, so that at any time step there may be several data items within the network in transit from a source PE to a destination PE. Such a model captures in an abstract setting the two features of our model described earlier.

A closely related model was previously studied by Gentleman [4] who established a lower bound of $\Omega(n^2)$ for multiplying two $n \times n$ matrices on a 1-D array processor with constant storage per processor and unlimited inter-PE bandwidth. In contrast, we relax the first assumption and place *no restrictions* on the number of data elements that may initially be stored in each PE's memory. The requirement of constant storage per PE in Gentleman's model [4], required that there be at least n^2 processors to hold the elements of the matrices to be multiplied, and consequently (as our proof suggests) spread the elements so far apart that the total time to multiply the matrices was dominated entirely by the time to *communicate* the data to the requisite PE's. A consequence of this result is that *any* matrix multiplication algorithm on a collinear array using $\Omega(n^2)$ PE's must require $\Omega(n^2)$ time. Our results show that by storing several elements in a PE, we can reduce the time complexity for matrix multiplication to $O(n\sqrt{n})$.

The elements of matrices A and B are initially stored *without duplication* in the local memories associated with the PE's. The PE in whose local memory an element of matrix A or B is initially stored will be referred to as its *home* PE. In our model, the home PE for any $a_{ij} \in A$ (and $b_{ij} \in B$) is unique. As mentioned above, we place no restrictions on the number of data elements that may initially be

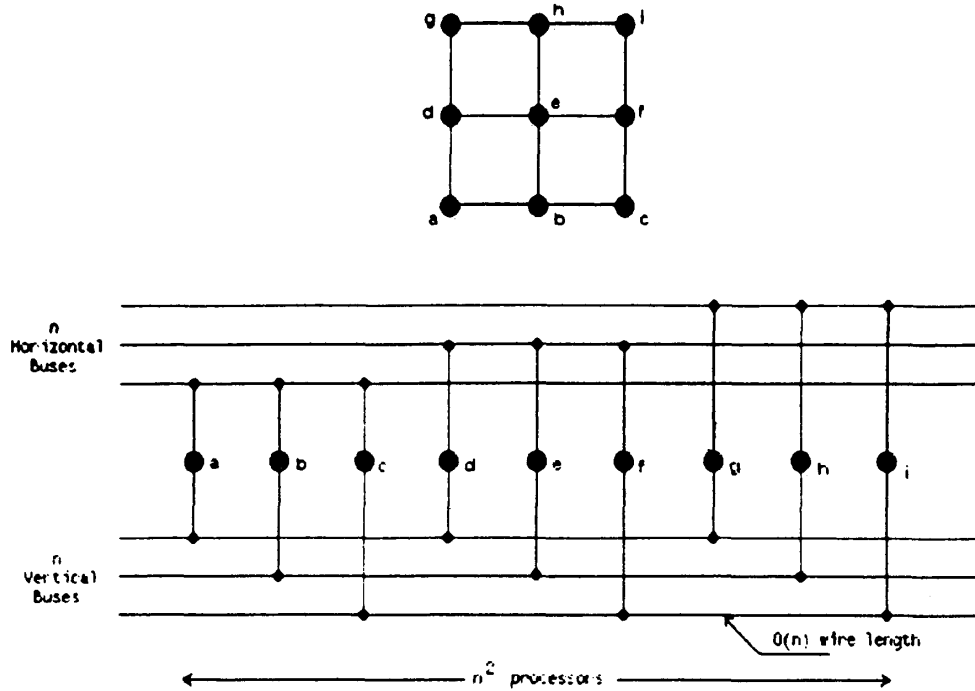


Fig. 1. Linearization of 3×3 mesh.

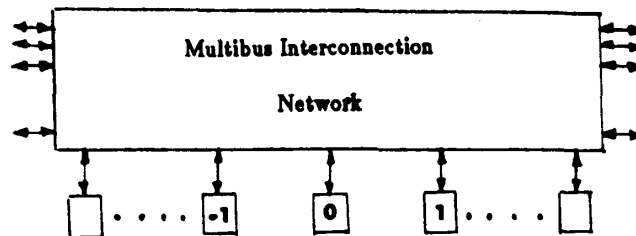


Fig. 2.

stored in each PE's memory, that is, several elements of matrix A (B) may have the same home PE.

Finally, in common with previous studies of matrix multiplication in systolic arrays [10] and VLSI models [12] we assume that elements are drawn from a finite ring and that computing a scalar product involving these elements takes unit time. In this model, we will now establish a lower bound of $\Omega(n\sqrt{n})$ on the time required to multiply two matrices by computing the n^3 scalar products.

Lemma 2.1: Let d denote the maximum number of PE's away from its home PE that any data element a_{ij} or b_{ij} participates in the computation of a scalar product. Let t denote the earliest time by which any element of matrix C gets its final update (that is, it has accumulated all its product terms). Then all the n^3 scalar products are computed in at most $1 + 8 \max(t, d)$ PE's.

Proof: Without loss of generality, let c_{ij} be the first element of C to get its final update at time t in PE_w . Now c_{ij} requires each of a_{ik} and b_{kj} ($\forall k = 1, \dots, n$) in its computation. Hence, if any of them had its home PE more than t PE's away from PE_w , then they could not have contributed to c_{ij} by time t . Thus, the home PE's of a_{ik} and b_{kj} must be in the range $[w - t, \dots, w + t]$.

Now consider an arbitrary a_{is} . All the n scalar products $a_{is}b_{sq}$ must be computed within d PE's of the home position of a_{is} , that is, in the range $[w - t - d, \dots, w + t + d]$. Since no data element could

have participated in the computation of a scalar product more than d PE's away from its home position, the home PE's for all b_{sq} must be in the range $[w - t - 2d, \dots, w + t + 2d]$. Since s ($1 \leq s \leq n$) was arbitrary, it follows that $\forall s$ and $\forall q$, b_{sq} must have its home PE in the range $[w - t - 2d, \dots, w + t + 2d]$.

Finally, it follows that all n^3 scalar products must be computed within at most d PE's of the home PE's of all b_{sq} , that is, in the range $[w - t - 3d, \dots, w + t + 3d]$. Thus, all n^3 scalar products are computed in at most $2t + 6d + 1$ PE's, which is less than $1 + 8 \max(t, d)$. \square

Theorem 2.1: Any matrix multiplication algorithm that computes n^3 scalar products on the model described above must take time $T = \Omega(n\sqrt{n})$.

Proof: By definition of d and t in Lemma 1, it follows that $T \geq t$ and $T \geq d$. Hence,

$$T \geq \max(t, d). \tag{1}$$

By Lemma 1, at most $1 + 8 \max(t, d)$ PE's participate in the computation of the n^3 scalar products. Thus,

$$T \geq \frac{n^3}{1 + 8 \max(t, d)}.$$

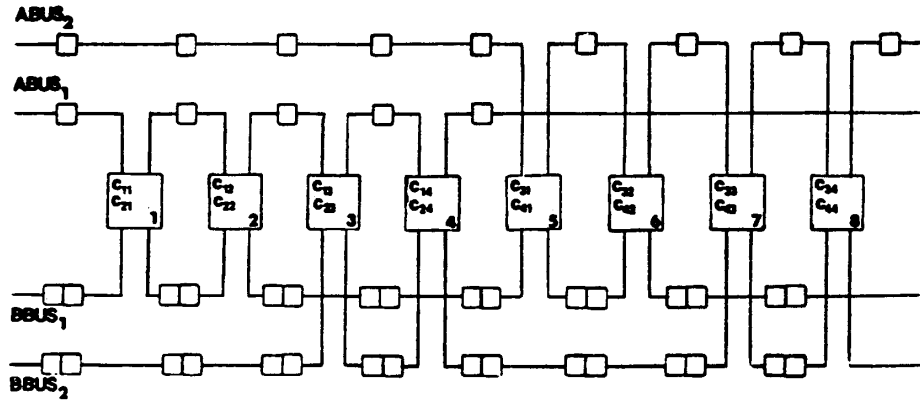


Fig. 3. 1-D arrangement of processors for $n = 4$.

Since

$$t \neq 0, T \geq \frac{n^3}{9 \max(t, d)}. \quad (2)$$

From (1) and (2), $T^2 \geq n^3/9$ and hence, $T = \Omega(n\sqrt{n})$. \square

Theorem 2.2: If two $n \times n$ matrices are multiplied by forming n^3 scalar products in time $O(n\sqrt{n})$ on the model described above, then the number of processing PE's required is $\Omega(n\sqrt{n})$.

Proof: Since n^3 scalar products have to be computed within time $cn\sqrt{n}$ (for some constant $c > 0$) at least $n^3/cn\sqrt{n}$ of these must be computed in every machine cycle. Thus, the number of PE's required is at least $(1/c)n\sqrt{n}$ which is $\Omega(n\sqrt{n})$. \square

We would like to make the following remark on the difficulty of directly using the lower bound results in [12] and [17] for VLSI matrix multiplication. These results are obtained by bounding the total information-flow requirements, from which we can only establish a weak $\Omega(1)$ lower bound on the time complexity for matrix multiplication in our model. The intuitive reason for this is that although we need to transfer $\Omega(n^2)$ bits of information between any two approximately equal-sized partitions of the array [17], we cannot rule out the possibility that it may be possible to accomplish this in $O(1)$ time by using $O(n^2)$ buses. Using the results in [2], we can similarly establish a lower bound of $\Omega(n)$. However, our lower bound of $\Omega(n\sqrt{n})$ arises by bringing in the constraints imposed by the maximum path length and the number of scalar computations.

III. MATRIX MULTIPLICATION ARRAY

We will now describe the computing structure for our matrix multiplication algorithm. We begin with a description of the processor and the array model. For ease of exposition we will assume no faulty PE's in this description. We will assume that n , which is the size of the matrices being multiplied, is a perfect square.

Let A , B , and C denote three $n \times n$ matrices, where $C = A \times B$. The row and column indexes of each matrix range from 1 to n . The $n\sqrt{n}$ PE's used for matrix multiplication are indexed 1, 2, ..., $n\sqrt{n}$ and arranged along a line as shown in Fig. 3 below for the case $n = 4$.

Each PE has \sqrt{n} words of local storage, with addresses ranging from 1 to \sqrt{n} . The n^2 elements of matrix C (all initialized to 0) are stored in the local memories of the PE's, one per word, and updated *in situ* as elements of matrices A and B are made available to the PE's.

There are three distinct types of buses—ABUS, BBUS, and CNTRLBUS. The first two buses transport data elements while the CNTRLBUS transports control signals.

There are \sqrt{n} buses of each type and each PE is hooked to *exactly one* bus of each type. Thus, each PE is receiving (inserting) *at most one element* from (into) each of the three buses in every cycle. Note,

therefore, that our PE's have *constant* fan-in and fan-out making our structure attractive for implementation purposes.

We can visualize all the $n\sqrt{n}$ PE's in the layout as being subdivided into n contiguous blocks where each block is a contiguous sequence of \sqrt{n} PE's. An ABUS is connected to all the PE's in \sqrt{n} contiguous blocks, that is, to n consecutive PE's. The first ABUS is connected to PE's 1, 2, ..., n , the second one is connected to PE's $n + 1, n + 2, \dots, 2n$, and so on. A BBUS is connected to each of the processors in \sqrt{n} different blocks, where consecutive blocks that are connected to the same BBUS are separated by a run of $\sqrt{n} - 1$ blocks that are connected to the remaining $\sqrt{n} - 1$ BBUS's. Thus, the first BBUS is connected to each of the processors in the first block comprising of PE's 1, 2, ..., \sqrt{n} and then to each of the processors in the block consisting of PE's $n + 1, n + 2, \dots, n + \sqrt{n}$, and so on. The second BBUS is connected to the block with PE's $\sqrt{n} + 1, \sqrt{n} + 2, \dots, 2\sqrt{n}$ and then to the block with PE's $n + \sqrt{n} + 1, n + \sqrt{n} + 2, \dots, n + 2\sqrt{n}$, and so on. The organization of the CNTRLBUS is similar to the ABUS. The reader may observe that the above interconnection results in connecting the PE's in each block into a linear array of \sqrt{n} processors.

Formally, let α and β be two integers such that $0 \leq \alpha, \beta < \sqrt{n}$ and let $i = \alpha\sqrt{n} + \beta + 1$. Then, element c_{ij} is stored in memory location $\beta + 1$ in $PE_{\alpha n + j}$. The buses of each type are assigned indexes 1, 2, ..., \sqrt{n} . Let $(ABUS)_i$, $(BBUS)_i$, and $(CNTRLBUS)_i$ denote the i th ABUS, BBUS, and CNTRLBUS, respectively. Each PE is connected to *exactly one* ABUS, BBUS, and CNTRLBUS as follows. Let $1 \leq p, q \leq \sqrt{n}$, and $1 \leq r \leq n$. Then $(ABUS)_i$ and $(CNTRLBUS)_i$ are connected to PE's indexed $(i - 1)n + r$ and $(BBUS)_i$ is connected to PE's indexed $(i - 1)\sqrt{n} + (p - 1)n + q$. For instance, in Fig. 3, $(ABUS)_1$ is connected to PE's 1, 2, 3, and 4 and $(ABUS)_2$ to PE's 5, 6, 7, and 8. $(BBUS)_1$ is connected to PE's 1, 2, 5, and 6 whereas $(BBUS)_2$ is connected to PE's 3, 4, 7, and 8. Finally, all the processors are assumed to operate synchronously.

We will denote two processors i and j as *logically adjacent* with respect to a bus if i and j are connected to the same bus and there is no processor with an index k which lies between i and j , that is also connected to the same bus. Thus, in Fig. 3, PE's 1 and 2 are logically adjacent with respect to $(BBUS)_1$. PE's 2 and 5 are also logically adjacent with respect to the same bus. Associated with each bus is a delay equal to the number of clock ticks required to move a data element traveling along the bus between consecutively indexed PE's i and $i + 1$. The delay associated with the ABUS and CNTRLBUS is 1, as shown by the "unit-delay" buffers on these buses (\square 's in Fig. 3). Similarly, the delay in traversing between adjacent PE's i and $i + 1$ along the BBUS is two clock ticks ($\square\square$'s in Fig. 3). Note that if PE's i and j ($i < j$) are logically adjacent with respect to a BBUS, then it would take $2(j - i)$ block ticks to transport an element from PE_i and PE_j along that BBUS. Since PE's i and j are separated by a physical distance proportional to $j - i$, no element has to traverse

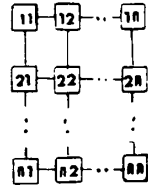


Fig. 4.

more than a fixed physical distance in one clock cycle and this distance is *independent* of the *size* of the array.

The control signals on the 2-bit wide CNTRLBUS are load (Φ_L), unload (Φ_U), transport (Φ_T) and compute (Φ_C). In every cycle, the PE decodes the control signals and either allows the signals to pass through the PE without performing any action (if Φ_T is true) or updates some c_{ij} stored in its local memory. A Φ_L signal causes the PE to latch-in the data available on its BBUS into an internal buffer BBUF and a Φ_U control signal causes the PE to unlatch the data stored in its BBUF onto the BBUS. On receiving a Φ_T signal, the PE merely transmits all the data at its input ports onto the corresponding output ports without performing any action. Every PE has a memory address register (MAR) that is reset to 1 when it receives a Φ_L control signal and is incremented by 1 every time it receives a Φ_C or Φ_U signal. The locations in the RAM are accessed in a cyclic sequence and hence in an implementation can be replaced by a more compact set of cyclically rotating registers. The $O(\lg n)$ access time associated with the RAM can therefore be eliminated.

Our algorithm is an adaptation of the following matrix multiplication algorithm on a 2-D $n \times n$ systolic array shown in Fig. 4. c_{ij} is accumulated in the processor at the intersection of the i th row and j th column. In the first iteration, the elements in the first column of matrix A interact with the elements in the first row of matrix B to form the first product terms of matrix C ($a_{i1}b_{1j}$). In the second iteration, the second column of matrix A and the second row of matrix B interact to form the next set of product terms and so on. Note that the systolic array computes the n^2 product terms in an iteration in $O(n)$ time. Each iteration is begun in the cycle following the commencement of the iteration immediately preceding it and hence all the n^3 product terms can be computed in $O(n)$ time.

We will now implement this algorithm on our model. First divide the $n \times n$ mesh into submeshes, each of size $\sqrt{n} \times \sqrt{n}$. The computations performed in a $\sqrt{n} \times \sqrt{n}$ submesh are simulated on a linear array of \sqrt{n} PE's obtained by collapsing a column of the submesh onto a single PE of the linear array. Observe, therefore, that we will now require $O(\sqrt{n})$ storage per PE in order to store the c_{ij} 's in one column of the submesh. Moreover, each b_{ij} now needs to stay in a PE for \sqrt{n} cycles. Observe, therefore, that we need control signals to latch and unlatch the appropriate b_{ij} 's in a PE. By interconnecting the linear arrays as shown in Fig. 3 and carrying out the algorithm described below we can multiply the two matrices.

We would like to mention that Fisher [3] has also independently proposed a similar simulation technique to transform 2-D systolic algorithms into linear array algorithms.

The algorithm consists of a host inserting the elements of matrices A and B and control signals at appropriate clock ticks as follows. Let n , the size of the matrices being multiplied, be a perfect square. Recall that the array consists of $n\sqrt{n}$ PE's that are indexed $1, 2, \dots, \sqrt{n}$. Let α and β be two integers such that $0 \leq \alpha, \beta < \sqrt{n}$. The algorithm consists of the following steps.

1. Let $i = \alpha\sqrt{n} + \beta + 1$. Then, insert a_{ij} into $(ABUS)_{\alpha+1}$ at tick $t_0 + \alpha(n + \sqrt{n} - 1) + \beta + (j - 1)(2\sqrt{n} - 1)$.
2. Let $j = \alpha\sqrt{n} + \beta + 1$. Then, insert b_{ij} into $(BBUS)_{\alpha+1}$ at tick $t_0 + (i - 1)(2\sqrt{n} - 1) - \alpha\sqrt{n} - \beta$.
3. Insert a Φ_L signal and a Φ_U signal along with $a_{(\alpha\sqrt{n}+1)j}$ and $a_{(\alpha\sqrt{n}+\sqrt{n})j}$, respectively, into $(CNTRLBUS)_{\alpha+1}$.

4. Insert a Φ_C control signal along with $a_{(\alpha\sqrt{n}+x)j}$ into $(CNTRLBUS)_{\alpha+1}$, for all x , $1 < x < \sqrt{n}$.
5. Insert a Φ_T signal into $(CNTRLBUS)_{\alpha+1}$ at other times.

Proof of correctness of this algorithm appears in [18].

Theorem 3.1: The time complexity of the matrix multiplication algorithm is $O(n\sqrt{n})$.

Proof: The first element to be inserted in the array is b_{1n} . It is inserted at time $t_0 - (n - 1)$. $a_{nn}b_{nn}$ is the last product term that is evaluated by the array. It is evaluated at time $t_0 + 4n\sqrt{n} - n - 3\sqrt{n}$. Unloading of the c_{ij} 's can be done in $O(n\sqrt{n})$ using \sqrt{n} buses.

IV. IMPLEMENTATION AND FAULT TOLERANCE

In this section, we will describe an implementation of the array described earlier and indicate how fault tolerance is achieved by restructuring and retiming. Our realization is similar in spirit to that proposed by Rosenberg [16] for obtaining Diogenes designs in the presence of faulty PE's. Just as in Rosenberg's scheme, we also use a few bits per PE that make the buses behave like a stack and/or a queue in order to configure our array in the presence of faulty PE's. The bits associated with the i th PE are labeled G_i and E_i and their complements are labeled \bar{G}_i and \bar{E}_i , respectively. Along with each PE for every ABUS, we also associate a buffer that is used for retiming as explained later. This retiming buffer delays the signals passing through it by one clock cycle.

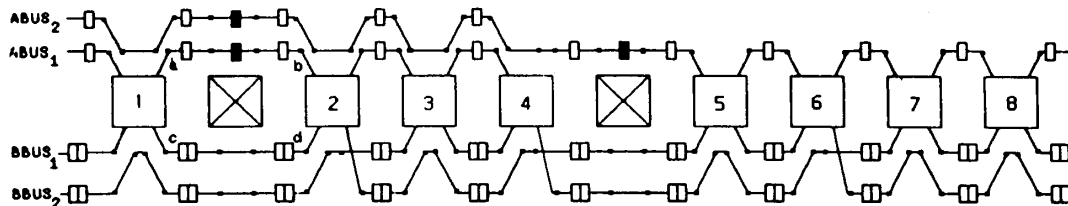
The bundle of ABUS's behave as a stack. G_i is set high if the PE is not faulty. E_i is a control bit that is set high only if the PE is the last processor connected to a particular ABUS (like PE₄ that is connected to ABUS₁ in Fig. 3). This is so because subsequent processors should no longer be hooked to this ABUS. The signals on the ABUS's pass through the retiming buffer associated with a PE only when it is faulty. The good PE's are hooked to the CNTRLBUS's in a manner similar to the ABUS's.

Recall that a single BBUS is connected to \sqrt{n} blocks of \sqrt{n} consecutive PE's where each block is separated by a run of $n - \sqrt{n}$ consecutive PE's that are connected to the remaining $\sqrt{n} - 1$ BBUS's. Such a configuration is achieved by making the bundle of BBUS's behave selectively as a stack or as a queue. The stack is used to hook \sqrt{n} consecutive PE's onto the same BBUS (similar to the hooking of the ABUS's) and the queue is used to defer the hooking of this BBUS until the next $n - \sqrt{n}$ consecutive good PE's have been hooked onto the remaining $\sqrt{n} - 1$ BBUS's. $G_i \wedge E_i$ forces the buses to behave as a stack, $G_i \wedge \bar{E}_i$ makes it behave as a queue, and $\bar{G}_i \wedge E_i$ is used to bypass the PE. In a straightforward implementation when the BBUS's behave as a queue, the signal emerging from the output of the PE needs to travel a vertical distance proportional to \sqrt{n} (corresponding to the \sqrt{n} buses) in order to join the tail end of the queue. This would seemingly negate our assumption that all signals travel a fixed physical distance independent of the size of the array. However, by splitting the bundle of buses into two halves and interleaving the buses from each half, we can ensure that the top and bottom of the queue are the two buses immediately adjacent to the PE.

Fig. 5 is an array of eight working PE's out of ten PE's. The difference between this and Fig. 3 is the presence of the two faulty PE's and the extra buffers (marked ■ in the figure) through which the signals on the ABUS travel while bypassing faulty PE's.

In the figure, the "dips" taken by the ABUS's on top of a good PE correspond to their stack-like behavior. The input line to a good PE gets popped from the stack on the left of that PE and the output line is pushed onto the stack at its right. The BBUS's behavior is similar to that of the ABUS's as long as contiguous PE's are hooked onto the same BBUS. However, at PE's 2, 4, and 6 (where the next run of consecutive PE's have to be hooked to a different BBUS) the bundle of BBUS's behave as a queue. BBUS₁ emerging from the right of PE₂ and PE₆ joins the end of the queue. Similarly, BBUS₂ emerging from the right of PE₄ joins the end of the queue.

The extra buffers achieve the retiming necessary to ensure correctness of the matrix multiplication algorithm described earlier (see Section III) for the fault-free array. Retiming as a means of

Fig. 5. Configured array in the presence of faulty PE's ($n = 4$).

achieving fault-tolerant systolic algorithms was proposed independently in [9] and [19]. If a systolic algorithm is comprised of only unidirectional data streams (that is all the data values move in one direction only as in our matrix multiplication algorithm), then in order to preserve relative alignment of data in all these streams, retiming requires that the *additional* delay encountered by data elements in each stream when bypassing a faulty PE be identical. For instance, in Fig. 5, a signal on ABUS₁ takes three cycles to reach point *b* from point *a* whereas in a fault-free array (see Fig. 3) it would require one cycle. Similarly, a signal on BBUS₁ requires four cycles to travel from *c* to *d* whereas it requires only two cycles to do so in a fault-free array. The additional delay of two cycles encountered in both these buses (and by the signals in all the other buses when bypassing this faulty PE) is the same. Thus, the single additional buffer on the ABUS's ensures the correctness of the algorithm when the faulty PE's are bypassed.

V. CONCLUDING REMARKS

In this paper, we have developed a fault-tolerant array for matrix multiplication that explicitly incorporates mechanisms for easy testability and reconfigurability. More importantly all signals in our array travel only a constant distance (independent of array size) in any clock cycle. On this model, we designed an optimal-time algorithm for multiplying matrices. The algorithm is an efficient simulation of a 2-D systolic algorithm for multiplying matrices.

Generalization of the simulation technique presented in this paper appears in [20] wherein we present a parameterized family of algorithms on this model (parameterized by the number of buses and storage per PE). The asymptotic processor and time complexities of all linear-array matrix multiplication algorithms and of the algorithm described in this paper are obtained as special cases of the parametrized algorithm.

REFERENCES

- [1] J. Bentley and T. Ottmann, "The power of one-dimensional vector of processors," Universitat Karlsruhe, Bericht 89, Apr. 1980.
- [2] B. Chazelle and L. Monier, "A model of computation for VLSI with related complexity results," *J. ACM*, vol. 32, pp. 573-588, July 1985.
- [3] A. L. Fisher, personal communication.
- [4] W. M. Gentleman, "Some complexity results for matrix computations on parallel processors," *J. ACM*, vol. 25, pp. 112-115, 1978.
- [5] J. W. Greene and A. Gamal, "Area and delay penalties in restructurable wafer-scale arrays," *J. ACM*, Oct. 1984.
- [6] K. S. Hedlund, "Wafer scale integration on parallel processors," Ph.D. dissertation, Comput. Sci. Dep., Purdue Univ., Aug. 1982.
- [7] A. V. Kulkarni and D. W. L. Yen, "Systolic processing and an implementation for signal and image processing," *IEEE Trans. Comput.*, vol. C-31, pp. 1000-1009, Oct. 1982.
- [8] I. Koren, "A reconfigurable and fault-tolerant VLSI multiprocessor array," in *Proc. Eighth Annu. Symp. Comput. Architecture*, Aug. 1982, pp. 262-264.
- [9] H. T. Kung and M. Lam, "Wafer-scale integration and two-level pipelined implementation of systolic arrays," in *Proc. MIT Conf. Adv. Res. VLSI*, Jan. 1984.
- [10] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proc. 1978*, I. S. Duff and G. W. Stewart, Eds., SIAM, 1979, pp. 256-282.
- [11] F. T. Leighton and C. E. Leiserson, "Wafer-scale integration of systolic arrays," *IEEE Trans. Comput.*, vol. C-34, pp. 448-461, May 1985.
- [12] F. P. Preparata and J. E. Vuillemin, "Area-time optimal VLSI networks for multiplying matrices," *Inform. Processing Lett.*, vol. 11, no. 2, pp. 77-80, 1980.
- [13] J. I. Raffel, "On the use of nonvolatile programming links for restructurable VLSI," in *Proc. Caltech Conf. VLSI*, Jan. 1979.
- [14] I. V. Ramakrishnan, D. S. Fussell, and A. Silberschatz, "Systolic matrix multiplication on a linear array," in *Proc. Twentieth Annu. Allerton Conf. Comput., Contr., Commun.*, Oct. 1982.
- [15] I. V. Ramakrishnan and P. J. Varman, "Modular matrix multiplication on a linear array," *IEEE Trans. Comput.*, vol. C-33, pp. 952-958, Nov. 1984.
- [16] A. Rosenberg, "The Diogenes approach to testable fault-tolerant networks of processors," *IEEE Trans. Comput.*, vol. C-32, pp. 902-910, Oct. 1983.
- [17] J. E. Savage, "Area-time tradeoffs for matrix multiplication and related problems in VLSI models," *J. Comput. Syst. Sci.*, vol. 20, no. 3, pp. 230-242.
- [18] P. J. Varman and I. V. Ramakrishnan, "Optimal matrix multiplication on fault-tolerant VLSI arrays," Tech. Rep., State Univ. New York at Stony Brook, Aug. 1985, (also appeared in *12th ICALP*, Springer Verlag LNCS, vol. 194, July 1985, pp. 487-496).
- [19] P. J. Varman, "Wafer-scale integration of linear processor arrays," Ph.D. dissertation, The Univ. Texas, Austin, Aug. 1983.
- [20] P. J. Varman and I. V. Ramakrishnan, "A fault-tolerant VLSI matrix multiplier," in *Proc. 1986 Int. Conf. Parallel Processing*, Aug. 1986, pp. 351-357.

K-Way Bitonic Sort

TOSHIO NAKATANI, SHING-TSAAN HUANG,
BRUCE W. ARDEN, AND SATISH K. TRIPATHI

Abstract—This paper presents *k-way bitonic sort*, which is a generalization of *Batcher's bitonic sort*. This algorithm is based on a *k-way decomposition* instead of a *two-way decomposition*. We prove that *Batcher's bitonic sequence decomposition theorem* still holds with this

Manuscript received July 15, 1986; revised July 23, 1987. S.-T. Huang was supported by the National Science Council of the Republic of China under Contract NSC-76-0408-E007-07.

T. Nakatani is with the Department of Computer Science, Princeton University, Princeton, NJ 08544.

S.-T. Huang is with National Tsing Hua University, Institute of Computer and Decision Sciences, Hsinchu, Taiwan 30043, Republic of China.

B. W. Arden is with the College of Engineering and Applied Science, University of Rochester, Rochester, NY 14627.

S. K. Tripathi is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

IEEE Log Number 8820891.