Reconfigurable SIMD Massively Parallel Computers

HUNGWEN LI, SENIOR MEMBER, AND QUENTIN F. STOUT, MEMBER, IEEE

Reconfigurable SIMD parallel processor is a member of SIMD architectures. Its most distinguished feature is the utilization of the reconfigurability of the interconnection network to 1) establish a network topology well mapped to the algorithm communication graph so that higher efficiency can be achieved, and to 2) remove faulty processors from the network so that the system operation can be kept uninterrupted while maintaining the same or slightly degraded efficiency. This paper describes several existing reconfigurable SIMD parallel architectures and their reconfiguration mechanism, demonstrates the effectiveness of algorithm mapping through reconfiguration, and discusses fault tolerant schemes via reconfiguration.

I. Introduction

This paper gives a brief introduction to a new class of computers, the reconfigurable massively parallel computer. Its most distinguished feature is the utilization of the reconfigurability of the interconnection network to establish a network topology well mapped to the algorithm communication graph so that higher efficiency can be achieved, and to remove faulty processors from the network so that the system operation can be kept uninterrupted while maintaining the same or slightly degraded efficiency.

Reconfigurable massively parallel computers are primarily of SIMD architecture due to their massive parallelism nature, however, as will be discussed, their architecture deviates from the SIMD paradigm to allow autonomy for each processor in the system. Reconfiguration is accomplished by one type of the autonomy the connection autonomy of the network, which allows each processor to select different local connectivity to accomplish a global desired topology. Such a reconfiguration strategy, based on the connection autonomy, facilitates the mapping of an algorithm to the network connecting all processors and the fault tolerance of the system.

We will focus our discussion on several existing reconfigurable parallel architectures and their reconfiguration mechanism, the effectiveness of the algorithm mapping through reconfiguration, and the fault tolerant schemes via reconfiguration.

Manuscript received May 17, 1990; revised September 15, 1990.

H. Li is with the IBM Research Division, Almaden Research Center, San Jose, CA 95120–6099.

Q.F. Stout is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109–2122.

IEEE Log Number 9042728.

Historically, the SIMD parallel computers are categorized as computers consisting of many processing elements, each of which behaves identically with the others under a centralized control [1]. In reality, none of the SIMD machines has been implemented in such a restricted manner. Certain deviation from the absolute SIMD paradigm has always been made, and *local autonomy* has been inserted to exploit more flexibility of the SIMD architecture. Local autonomy can be provided in different areas ranging from local activity control to local program control [2], [3]. Amid of the local autonomy, we are particularly interested in the *connection autonomy* which allows the reconfiguration of the network be individually and locally controlled in a SIMD system.

Connection autonomy [4] is a principle aiming at the efficient mapping of the algorithm graph onto the hardware network topology via dynamic network reconfiguration subject to the local condition of every processor in a SIMD system. It shares the same goals of other reconfigurable computers [5]–[7] in the aspects of efficient mapping, high reliability, and high availability. However, in the domain of SIMD massively parallel computer, the connection autonomy can be very different from other reconfigurable computers in the method of graph embedding due mainly to the massive parallelism, the granularity, the network topology, and the locality of the control mechanism. A very unique feature offered by the reconfigurable SIMD architectures is its capability of configuring one network topology at every instruction by manipulating local data at each processor. Such a feature can be viewed as an extension of the SIMD paradigm from computing to configuration.

With the connection autonomy, the reconfigurable SIMD architectures distinguish themselves as a unique computing model. This can be shown in many new algorithms, to be discussed in Section III, whose performance approach an ideal PRAM model. The major reason for such an improvement is due to the fact that the reconfigurable SIMD

0018–9219/91/0400–0429$01.00 © 1991 IEEE

architectures are able to, on a per instruction basis, configure a topology which mirrors the algorithm connectivity by using the data-dependent switch setting.

For reconfigurable SIMD architectures, fault tolerance is treated in the same way as the algorithm mapping by the use of reconfiguration. Many built-in features of the switch design for the architectures have readily supported the fault tolerance. The control of the switch generated for the algorithm mapping is no different from that for fault tolerance. This is a significant step toward a uniform handling of fault as a normal operating condition in a massively parallel system.

This paper is structured as follows. Section II presents several reconfigurable SIMD architectures and their mechanisms to support the reconfiguration. Section III discusses many algorithms that exploit the benefits of reconfiguration. Section IV discusses several fault tolerance schemes that are based on the reconfiguration principle.

II. Architectures

A. Polymorphic-Torus

Polymorphic-torus [8]–[10] is a massively parallel SIMD architecture aimed toward a more than 1 000 000-processor system. Study on the VLSI and packaging technology at the early stage of the design convinced one of the authors that a feasible approach is to choose a bounded-degree network with a low-degree of connection. A comparison of such wiring complexity for a massively parallel system at different levels of packaging hierarchy (e.g., chip, printed circuit board and chassis) for various networks has been studied [9]. Briefly, this comparison shows that a hypercube network can have a wiring complexity at three orders of magnitude higher than a mesh. This leads to the conclusion that a high-degree network may not be a good choice for a massively parallel system because the packaging consideration is a genuine and serious technology constraint for the feasibility of a massively parallel system.

Regarding the choice of network for a massively parallel system, the following explains the understanding through many algorithms studies for various networks.

1. A high-degree network may not have a substantial communication benefit over the lower-degree ones.
2. Many algorithms are performed in a high-degree network by embedding a lower-degree network.
3. The complexity required to support the communication in a high-degree network slows down the clock rate, which leads to a degraded overall communication bandwidth.

These understandings stimulate the thinking that a low-degree network can be enhanced to more suitably support the connectivity of the massively parallel computers.

The connection autonomy was identified as a key mechanism for the development of the Polymorphic-torus to enhance the performance of a low-degree network. The Polymoprhic-torus selects a two-dimensional mesh as the base (physical) network and adds "a switch" to each intersection of the mesh. These switches can be independently controlled by the status of each local processor. At every clock tick, the open–close positions of the switches determine a new network topology for the system. In a sense, this is an extension of the SIMD paradigm because, for each instruction, the data manipulating the connectivity are controlled in exactly the same way as the data for computing. Circuit switching was adopted to establish the connection, and as a result, a very long path can be established in a large system. Many processors therefore can be attached to the path and can communicate as if they are one unit distance apart. Communication bandwidth is consequently increased. Many paths can coexist in the system and in many cases, multiple paths can be configured systematically. Since the length of the path can be different, a variable clock was devised to support the operation of the connection autonomy.

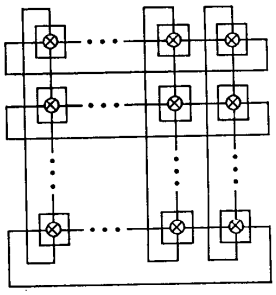

Fig. 1. A Polymorphic-torus network.

In a Polymorphic-torus architecture consisting of NxN processors (Fig. 1), a processor is located at each node of an $N \times N$ two-dimensional torus (called base network BNET) and a collection of $N \times N$ switches are distributed over every node of the BNET. Since the switches are internal to the processors, we call them the internal network (INET) for the ease of discussion. We use the coordinate as the identification (PID) of the processor/switch and simply denote it as $P(i,j)$. For connecting, each processor $P(i,j)$ is equipped with four ports $N(i,j), E(i,j), W(i,j)$, and $S(i,j)$. The wiring of these ports to BNET follows mesh/torus pattern, which is fixed and nonprogrammable.

In contrast, the INET is totally programmable. At the (i,j) node of the BNET, there resides an $INET(i,j)$ which is a complete graph of four ports $(N(i,j), E(i,j), W(i,j)$, and $S(i,j))$. By "complete", we mean that each port can be connected to every other port. For example, any two ports can be connected (e.g., S and N), two pairs of ports can be connected simultaneously (e.g., N connected to S and E connected to W), a triple of ports can be connected (e.g., N connected to both E and W), or all four ports are connected together. As will be seen later, the functionality of the INET is a superset of other reconfigurable massively parallel architectures discussed in

the following. In Section III, we will point out the benefit of having the INET capability.

Circuit switching is adopted for the INET switch. This means that the connected ports are "short-circuited"; they share the same logic level along the short-circuit path so that any datum appearing on the path can be accessed in the same machine cycle by all processors attached to the path. This is heavily used for long-distance communication and broadcasting. To establish a "circuit," each processor refers to the instruction and its local condition, then decides a switch setting. A communication path is accordingly formed as the result of the collective switch setting.

Four types of controls are used to achieve the connection autonomy: the unconditional control, the PID-dependent control, the mask-dependent control, and the data-dependent control. They are discussed next.

Unconditional INET control. This is interpreted and executed uniformly by every processor regardless of the difference of local status. For example, a "connect all ports" command will glue all processors in the same path and is useful for broadcasting. An unconditional "connect E and W" configures the system into N rows of buses allowing multiple broadcasting.

PID-dependent control. The processor identification (PID) or the coordinate of the processor in the BNET is the most frequently used condition for the Polymorphic-torus connection autonomy. N binary trees each with N leaves can be emulated by shorting E-W ports of the processors with $PID[i] = 0$ subsequently in $i = 1$ to $\log N$ steps [10]. Similarly, a pyramid can be emulated by systematically controlling the PIDs.

Mask-dependent control. A connection mask is a known "condition pattern" prepared as part of the algorithm before or at compile time. For an N-processor Polymorphic-torus, the mask consists of N "conditions", one for each processor. In the simplest case, each condition is a bit and each mask bit is used to control the connection function. The mask is usually stored in the memory and retrieved when needed. An irregular, but known, connectivity (such as sparse matrix) [11] is usually implemented on the Polymorphic-torus by the mask-dependent control. In fact, the PID-dependent control for an N-processor system can be stored as $\log N$-bit masks and an unconditional control can be performed as a conditional one with an "all 1" mask.

Data-dependent control. Connection autonomy driven by local data is the most powerful mechanism for reconfiguration of the SIMD massively parallel processors. The data-dependent connection autonomy establishes Polymorphic-torus and many architectures to be discussed as a distinguished computing model. Many algorithms (connected component [12], Boolean [13], and transitive closure [14], etc.) approach the performance of an ideal CRCW shared-memory PRAM model. We discuss this in detail in Section III. For example, one can distribute an $N \times N$ image uniformly over the Polymorphic-tours and can then perform the connected component algorithm to segment the image into several regions, each of which carries a distinct label. The label then can be used as the condition such that pixels carrying the same label can be connected (or short-circuited). Thus the connection autonomy using data-dependent condition can dynamically group the data that share the same property. This is similar to the content associative processing referred in [15]. It is also worth noting that multiple groups of content associative processing can be conducted simultaneously in the Polymorphic-torus.

In summary, the Polymorphic-torus demonstrated that the concept of the connection autonomy can be treated as an extension of the SIMD architecture. Network topologies can be configured on a per instruction basis by using local data in exactly the same way as used in arithmetic operations. It depicts that a low-degree network can be enhanced to compete with a high-degree network. The advantage, however, is that its implementation is more suitable for today's VLSI and packaging technology. An implementation of the Polymorphic-torus is referred to in [14]. Many algorithms that exploit the unique capability of the Polymorphic-torus are discussed in [13].

B. Gated-Connection Network

Gated Connection Network (GCN) [16], [17] is a communication structure of part of a larger system called Image Understanding Architecture (IUA) being developed for computer vision applications [15], [18]. The IUA is a hierarchical heterogeneous architecture consisting of three different types of processing elements: arrays of microprocessors at the highest level, the digital signal processors at the middle, and, to the interest of this paper, the bit-serial processors at the lower level. A 512×512 array is proposed for the lower level and the GCN is the connecting network for the array. We give a brief description of the GCN and its contrast to the Polymorphic-torus in the following.

The GCN consists of a mesh array of eight simple transmission gates per bit-serial processor as shown in Fig. 2. These transmission gates are the "local switches" superimposed on the nodes of the physical mesh network. The processor situated on the mesh node is connected to its "local switch" via a dedicated register X (output) and a dedicated wire (S/N input). Communication between processors is determined by the switch setting of the transmission gates. These gate settings are determined by an eight-bit register which resides in the memory space of each processor. The transmission gates and the switch control registers implement the connection autonomy. A control pattern can be stored in advance or created dynamically in the control registers, and subsequently used to reconfigure the network topology.

As in the case of Polymorphic-torus, path with variable length can be configured and signal delay may become unacceptable. GCN adopts precharged circuits to shorten the signal delay. The GCN network is precharged and processors sending "1" to the path will pull down the precharged circuit. This is equivalent to a wired-OR circuit. The wired-OR implements the Boolean and MIN/MAX algorithms in constant time and is an important improvement over the physical mesh networks. All four types of controls for the connection autonomy mentioned in previous section

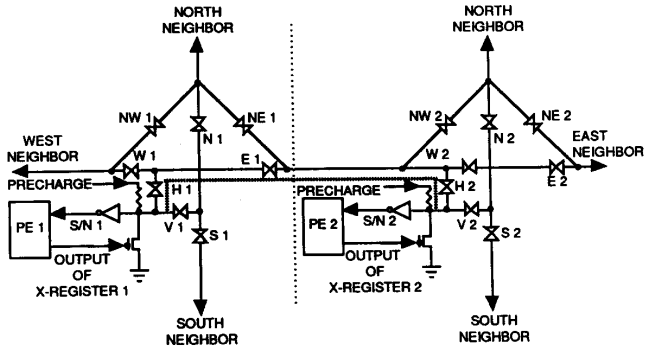

Fig. 2. A gated-connection network.

can be implemented by these transmission gates. Many important vision algorithms, such as Hough transform and connected component labeling have been implemented on GCN with significant performance improvement. Several symbolic algorithms also enjoy the benefit of GCN.

A VLSI implementation of the GCN exists. The chip contains 64 bit-serial processors with their corresponding GCN and local memory of 320 bits per processor. A difference in implementing the connection autonomy between the GCN and the Polymorphic-torus is that GCN has an internal 8-bit data path to load/store the 8-bit control register which open/close the eight transmission gates while the Polymorphic-torus encodes part of the open/close switch patterns into the instruction set and leaves only one bit of datum locally for autonomy. The tradeoff is that GCN has individual control of each switching gate, however, it may take more instruction cycles to configure a new topology. On the other hand, the encoding technique used in the Polymorphic-torus selectively supports important patterns. However, it can deliver one of these important patterns on a per instruction basis.

C. Flat Pyramid

Pyramid architectures [19]–[23] were developed to handle multiresolution processing in image processing and computer vision paradigms. The goal is based on the observation that salient information can be extracted by sensory data at different scales of representation. Coarse solution can be obtained quickly by a reduced version of image and can be refined by focusing on areas where evidences exist that further processing at higher resolution is useful. This is an analogy of the human eye called foveation.

Two approaches have been attempted to perform the pyramidal processing: the physical pyramid architecture and the use of mesh to emulate the pyramid. The former, the physical pyramid architecture, requires a high-degree network (e.g., 8 neighbors at the same pyramid level, 4 wires for children at the level below, and 1 wire for parent at the level above) whose wiring complexity usually prevents such a fixed structure from qualifying a good architecture for massively parallel computer because of the limited packaging capacity.

On the second approach, one method of using mesh to emulate pyramid is shown in [24]. Although simple and modular, the mapping incurs high overhead for data transfer among neighboring nodes. This mapping also requires a large amount of memory because processors at different levels are mapped to the same processor in the mesh.

The above-mentioned difficulties for pyramidal massively parallel processing can be largely alleviated by embedding switches in a mesh and by reconfiguring a pyramid out of the mesh. The added reconfigurability reduces the wiring complexity and increases the communication bandwidth.

PAPIA2 is an architecture which configures pyramid topology out of a flat mesh by adding connection autonomy into a mesh array. Figure 3 illustrates the basic idea by an example of a 32×32 array. A processor at the base of the pyramid is represented by "+" while "1, 2, 3, 4, and 5" represent an extra pyramidal processor at a higher level mapped on the base processors. A processor at level k is mapped to the mesh location (i, j), $i = (p \times 2^{k-1})_{\mathrm{mod}N}$ and $j = (q \times 2^{k-1})_{\mathrm{mod}N}$ where p and $q = 1, 3, 5, \ldots$. The communication at base level uses the mesh. For interprocessor communication at level 1, the base processors (marked as "+") can short their E-W switch or N-S switch to facilitate the communication. More base processors need to be shorted for interprocessor communication at higher levels, for example, 3 shorted base processors for level 2, 7 for level 3, and 15 for level 4.

For the proposed mapping, the interlevel communication occurs at the diagonal direction. Short-circuited paths along SE and NW direction can be established by setting appropriate switches. The interlevel communication is limited in its concurrency; for example, when level 1 and 2 are communicating, communication paths between level 2 and 3 are blocked. However, the proposed mapping is highly regular and extendable, and no extra wiring is required beyond

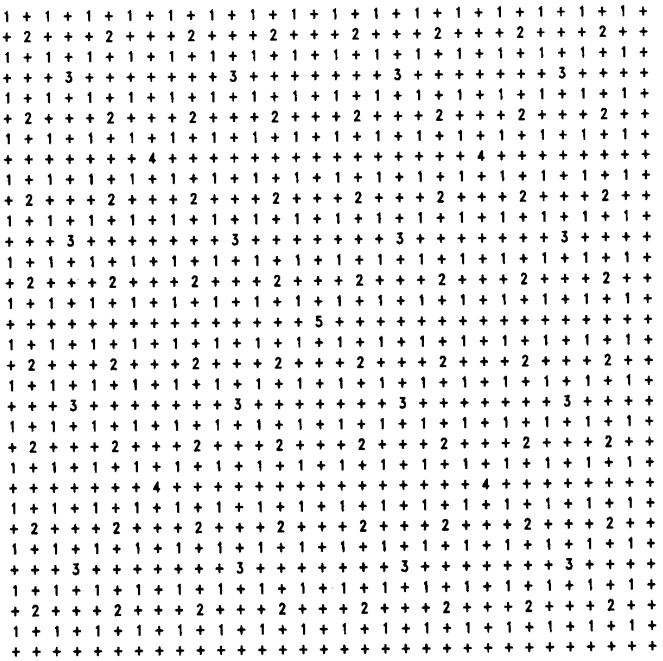

Fig. 3. A PAPIA2 Flat pyramid on a 32 × 32 mesh.

the original mesh links. By adding connection autonomy to the mesh network, one can efficiently configure a mesh into a pyramid which is suitable for a large category of hierarchical-structured algorithms.

D. CLIP 7

Researchers at University College London have gone through several generations of SIMD processor array designs designated as CLIP series [25]–[30]. The CLIP 7 represents the latest research focusing on the autonomy.

The CLIP7 chip contains one processor whose internal structure is shown in Fig. 4. Several features are implemented in CLIP7 chip to support the local autonomy and reconfigurability. A 16-bit C register serves the purpose of autonomy. When used in autonomous mode (determined by a pin in CLIP7 chip), the meanings of the bit assignment are described in the following. The use of these bits supports several categories of autonomy and configuration.

Bit 15 of the C register is equivalent to the activity (or enable/disable) bit common to many SIMD arrays. Results from arithmetic operation (e.g., carry, overflow, zero, or sign) can be stored into this bit to subsequently determine the participation of the processor in the array operations.

The network reconfiguration is supported by CLIP7 in a unique way. Bit 0–7 of the C register select the gated input from one of the eight neighbors when in binary operation. In the case of 8-bit operation, data from neighbors are collected through the N inputs (Fig. 4) serially and stored in the N registers. Bit 0–2 of the C register control the multiplexer to select one of the 8 N registers By programming the C register, the processors in the array can receive data from different neighbors under one instruction. This effectively changes the logical topology of the network dynamically. In analogy to the Polymorphic-torus and Gated-Connection Network, the C register is the internal switch that controls the flow of the data (or the reconfiguration) in the entire network. Since the CLIP7 chip contains only one processor and is not committed to any "physical network", this leaves a lot of room to conduct research on connection autonomy by using the same chip. Many networks can be emulated by choosing a physical network and by programming the C registers.

A prototyping CLIP7A system is physically connected as a linear array of 256 processors. Besides its easiness and relative low cost to construct, the choice of linear network has the benefit of lowest wiring complexity and easiness to emulate a two-dimensional array and other network. Being a linear array, the CLIP7 is left with more luxury in wiring; a CLIP7 chip contains a 16-bit processor and communicates with other processor via 8-bit ports. Since the program chose a 64-pin package, the packaging constraint forced the port communication in serial. The choice of the CLIP7A is a good indication of the seriousness of the wiring constraint put on the engineering of a massively parallel computer.

E. Reconfigurable Bus Architecture

Reconfigurable bus architectures [14], [31]–[33] are a

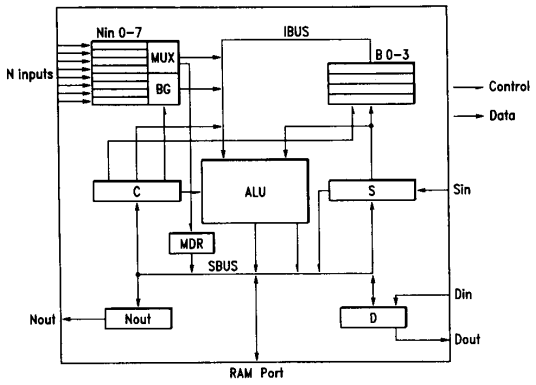

Fig. 4. A CLIP 7 processor structure.

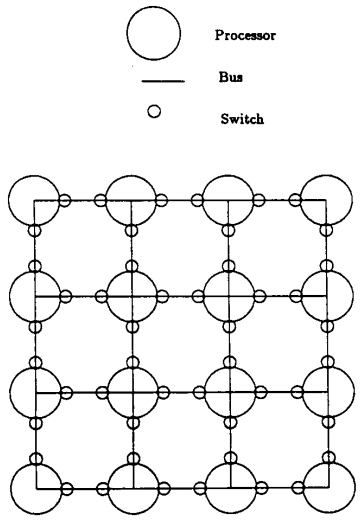

Fig. 5. Reconfigurable bus architectures.

family of computing models that have exhibited its unique strength in a wide spectrum of computations. A two-dimensional processor array with a reconfigurable bus system consisting of $N \times N$ processors connected to a mesh-shaped reconfigurable bus system (called reconfigurable mesh) is shown in Fig. 5. The configuration of the bus system can be dynamically changed by adjusting the switches within each processor. Different shape of buses such as rows, columns, diagonals, zig-zag, and staircase can be formed by properly adjusting the switches/ports. There exists a range of reconfigurable bus architecture proposals: one assumes that only one processor can broadcast value at a time in a connected path (bus); the second supports simpler switch that has only two ports, one controls the attachment of the processor to the row bus and the other to the column bus; and a third supports only N-S and E-W short-port capability. These are examples with subset capability of the model discussed in the Polymorphic-torus section. They differ in the performance of algorithm mapping. Extension of the reconfigurable bus architecture to higher dimension is conceptually straightforward however requires careful engineering considerations.

The uniqueness of the reconfigurable bus system is displayed through many algorithms tailored to its reconfigurability and is discussed in detail in Section III.

F. Engineering and Technology Constraints

As discussed above, the reconfigurable SIMD architectures are all of two-dimensional topology. This is due to the evolutionary nature that the knowledge generated by many mesh-oriented architectures [29], [34]–[39] have laid a solid theoretical and technological foundation on which the reconfigurable massively parallel architectures are based. The lack of knowledge in high-dimensional topology also forces the infancy of the reconfigurable arrays starts from two-dimension. Besides the evolutionary reasons, the engineering and technological constraints play a very important role in the architecture choice. Several constraints that are generic to massively parallel systems are discussed but are of particular pertinence to the reconfigurable massively parallel architectures.

The choice of the physical network is of fundamental impact to the design of a reconfigurable massively parallel array. The bounded degree network has been a favorable choice and a degree-4 mesh is among the most popular. This is because of its low pin requirement and two-dimensional topology, both of which are well suited for today's VLSI and packaging technology. Beyond obviousness, the mesh network is chosen for its ability to deliver good performance in algorithm mapping (Section III) and for its better known fault tolerant schemes (Section IV). As to be discussed in Section III, many algorithms on this degree-4 network performs equally well as that of a higher-degree network. The secret of achieving equal performance with less wires lies on the reconfigurability or the local connection autonomy to make full utilization of the wires for efficient communication.

The choice of circuit switching versus packet switching is another important consideration. Most systems that use the mesh network adopt circuit switching for communication. The circuit switching has a low overhead in terms of VLSI implementation as demonstrated in the Polymorphic-torus and the Gated-Connection Network. Such a low overhead is extremely important for a single-bit array. One implication of choosing the circuit switching is the potential large signal delay due to a long chain of "shorted" path. This issue can be resolved by adopting all-active and precharged circuit for local switches. For the packet switch implementation, the switch design is more complicated and the communication time can be longer than the data movement in the mesh network. Nevertheless, the power and the flexibility of the

packet switching [40]–[42] can be a promising enhancement to the reconfigurable architectures.

In a massively parallel system, clocking technique is a primary consideration. This is especially compelling for a reconfigurable system with circuit switching implementation because an indefinite length of "shorted" path can be established. If a fixed length clock is designed to accommodate the worst case "shorted" path, the clock for the massively parallel system will be degraded. In Section III, many constant time algorithms can be developed for a reconfigurable system. Although theoretically correct, the claim made by constant time algorithms does not consider the clock implementation. The readers must be reminded of the possibly degraded clock rate for a fair comparison. Many clocking schemes are possible to accommodate the worst case path while not affecting the average clock performance. One such proposal is the variable length clock [10] that adjusts the length of clock to the length of the path.

III. Algorithms

In this section, some algorithms for reconfigurable architectures will be given to illustrate the flexibility of these architectures. Some attention will also be paid to differences in the architectures that affect performance. For most purposes we will start with the Polymorphic-torus model described earlier, and then describe how algorithms for other architectures differ from this. The base mesh for our analyses will be a $\sqrt{n} \times \sqrt{n}$ mesh, with a total of n processors.

One critical factor in the analysis of reconfigurable algorithms is the time needed to propagate a signal. Some authors treat this as a unit-time operation no matter how far the signal must travel, while others note that the signal takes longer to propagate in a larger machine and hence the time must be a function of the number of processors. If a logarithmic function is used, so that the speed of propagation is proportional to the log of the distance traveled, then the time to send a signal across the entire machine is $\Theta(\log n)$. On the other hand, if a linear function is used so that the speed is proportional to the distance traveled, then the time is $\Theta(\sqrt{n})$ if the entire mesh is one circuit, or as much as $\Theta(n)$ if the mesh has been configured as a long snake-like circuit. Physically, the time must be at least linear in the distance traveled, but since the speed of propagation is much faster than the clock period a linear time analysis is misleadingly pessimistic over the relevant size range. For the remainder of this section, time analyses will assume a constant time, called the *constant-delay model*, or a logarithmic time, called the *logarithmic-delay model*.

We assume that the controller does not issue a new instruction until the previous signal has had sufficient time to propagate. Thus programmers must either be able to specify a suitable delay period, or else the controller must make the pessimistic assumption that every signal propagation is across the entire machine. The former puts a significant burden on the programmer, while the latter loses some of the potential of the machine. In practice perhaps both would be used, in that the programmer could specify a delay less than the most pessimistic one in the few cases where it was known to be appropriate. The development of some carefully tuned standard routines could result in much of the communication taking place with nearly optimal timings. For the following algorithms the pessimistic delay is the appropriate one, with the exception of the XOR algorithm for the logarithmic-delay model, in which the analysis assumes that an optimal delay is used.

A. OR

Perhaps the simplest function which can be computed is a global OR, in which each processor starts with a boolean value and the goal is to obtain the OR of all of these values. To solve this on the Gated-Connection network, all gates are closed so that all processors are connected together and the circuit is precharged. Then each processor that has a "true" (1) pulls down the circuit. The time is just the time for a signal to propagate across the machine, and so is $\Theta(1)$ in a constant-delay model, and is $\Theta(\log n)$ in a logarithmic-delay model.

If each processor opens its N and S gates before the operation, then the result will be the OR in each row, in the same time bounds as for the global OR. This is an example of the unconditional control mentioned in Section II-A. Further, this can be simultaneously used on any set of disjoint circuits, a fact that will be exploited below in Section III-C.

On the reconfigurable mesh model considered in [33] the OR function is slightly more complicated, due to the restriction that only a single processor can broadcast at any one time on any given circuit. To see how this can be accomplished, consider just a single row of processors. First, each processor with a 1 opens the gate to its right and closes the gate to the left, while each processor with a 0 closes both gates. In each subcircuit created, there is exactly one processor with a 1, and it is the rightmost processor within its subcircuit. The only exception to this is the rightmost circuit, which has no processor with a 1 unless the rightmost processor of the row has a 1. Each processor with a 1 then transmits that value. At this point all processors know the correct value, except those in the rightmost subcircuit since they have not seen a value transmitted and do not know if their subcircuit is the entire row. To remedy this the left-right directions are reversed. Now each processor with a 1 opens the gate to its left and closes the gate to its right, each processor with a 0 closes both gates, and again each processor with a 1 transmits the 1 on its subcircuit.

The total time is $\Theta(1)$ in the constant-delay model, and $\Theta(\log n)$ in the logarithmic-delay model. Further, a global OR can be computed in similar time bounds by computing OR within each row, and then within each column, or by first setting the gates to form a Hamiltonian path through the processors and treating the path in a manner similar to a single row.

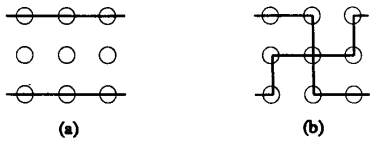

Fig. 6. A block switch setting for XOR.

B. XOR and Addition

Computing the XOR shows some of the power of the reconfigurable architectures, and illustrates some of the significant differences among the variations. Suppose each processor starts with a boolean value, and the goal is to take the XOR of all of these values.

1) Polymorhic-torus algorithms: In the Polymorphic-torus model, first the XOR will be computed in 3×3 blocks. This can be done in constant time, after which each processor in each block knows the result for its block.

Then rows of blocks (i.e., groups of 3 rows) compute the XOR as follows. In each block in which the XOR is 0, the gates will be set as in Fig. 6(a), while in each block in which the XOR is 1, the gates will be set as in Fig. 6(b). Then the gates for the first and third rows are closed between each block, and a signal is started in the leftmost processor of the first row. If it arrives at the rightmost processor of the first row then the result of the XOR is 0, while if it arrives at the leftmost processor then the result is 1. Closing all the gates in the row of blocks, the result can then be transmitted to the entire row. To compute the XOR over the entire mesh, a similar process can be used in a column of blocks. The time for the XOR is $\Theta(1)$ in the constant-time model, and $\Theta(\log n)$. in the logarithmic-time model.

The XOR circuit can also be used to form the sum of the boolean values. To simplify discussion, only the sum within a row of blocks will be discussed, with the extension to the entire array being straightforward. The XOR has computed the lowest-order bit of the sum, and note that a carry should be issued every time a block has an XOR of 1 and the outgoing signal is a 0. Therefore, at the end of the first XOR, within each block there is either a 0 or 1 for a carry, and this can be determined by the processor in the lower left corner of the block since it will know if the outgoing signal is 0 and whether the block had an XOR of 0 or 1. Further, since the original block had 9 values in it, there can be an initial carry of as much as 4 to be added to this. Again the proper circuit in each block can be set up, and a new XOR of these carry bits formed, giving the second-lowest bit. After a total of $\log_2 n$ iterations, the correct answer has been determined. The total time a $\Theta(\log n)$ in the constant-time model, and $\Theta(\log^2 n)$ in the logarithmic-time model. If each processor starts with a b-bit number then the global sum can be found in $\Theta(b+\log n)$ time in the constant-time model, *and* $\Theta(b\log n+\log^2 n)$ time in the logarithmic-time model. It is not known if these are the best possible times for these models.

2) Reconfigurable mesh algorithms: The nonplanar connection possibilities of the Polymorphic-torus play a decisive role in the above algorithm for XOR, in that the circuit for a block with a 1 needs to construct a crossing. In the reconfigurable mesh model of [33], or the Gated-connection network, such crossings are not possible and as a result it does not seem possible to compute the XOR quite as quickly. One natural technique is to use a pyramidal approach, computing the XOR bottom-up in the pyramid with each node above the base taking the XOR of the results of its children. This will take $\Theta(\log n)$ time in the constant-time model, and $\Theta(\log^2 n)$ in the logarithmic-time model.

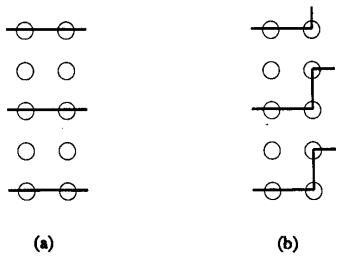

Fig. 7. A column switch setting for XOR.

Following the work in [33], a faster algorithm can be developed by noting that the XOR of $\sqrt{n}$ bits can be done quickly on the reconfigurable mesh. To see that this is so, first suppose that each even column has a single bit. Pair columns up, and transmit the bit of the even column throughout the pair. For each pair of columns, if the bit is 0 then the column switches are set as in Fig. 7(a), if the bit is 1 then they are set as in Fig. 7(b). Then the switches between pairs of columns are closed, and a signal started at the lower left processor. In the rightmost column, if the signal reaches a processor with row coordinate x then the sum of the bits in the even columns is x div 2, and the XOR is (x div 2) mod 2. If each column has a single bit, then two applications of this procedure can be used to find the sum or XOR. The time is $\Theta(1)$ in the constant-time model and $\Theta(\log n)$ in the logarithmic-time model.

To efficiently find the XOR or sum of all of n bits a more complicated approach is needed. In [33], the XOR problem on a mesh of n processors is solved recursively, finding the XOR in submeshes of $\sqrt{n}$ processors, and then moving the results to different columns and using the above procedure to find the XOR of the remaining bits. This approach gives a time of $\Theta(\log\log n)$ on the constant-delay model, and $\Theta(\log n)$ on the logarithmic-delay model. This algorithm can be extended to find the sum of n b-bit numbers in $\Theta(b\log\log(n)+\log n)$ time on the constant-delay model, and $\Theta(b\log(n)+\log^2 n)$ on the logarithmic-delay model.

It is not known if the $\Theta(\log\log n)$ time is the best possible for the XOR or the constant-delay model. If it is, then this is an example of a nontrivial problem for which the time on the logarithmic-delay model is less than log (n) times the constant-delay time. It would also be an example of a problem on which the reconfigurable mesh is more than a constant multiple slower than the Polymorphic-torus.

3) Bus automata algorithms: Finally, note that the

Polymorphic-torus solution would also be applicable in a weaker bus automaton model if it also allowed the cross-over connections of the Polymorphic-torus, as long as each automaton knew its relative position with its block. However, the reconfigurable mesh solution is not as easily applicable to the bus automaton because the calculations needed to perform the divide-and-conquer are significantly more complicated, and it is not trivial to have these performed efficiently in an automaton model. Using the clerk idea from [43], after a $\Theta(\sqrt{n})$ initialization period, a bus automaton could then form an XOR in $\Theta(\log n)$ time. However, the details are formidable and would never be used in practice. The fastest possible XOR on a bus automaton without cross-over connections is an open question.

C. Component Labeling

An interesting and practical use of the OR occurs in component labeling. Suppose each processor contains a pixel of a black/white image, and the goal is to assign a label to each black pixel so that two black pixels have the same label if and only if there is a connected all-black path from one to the other. Here we say that two pixels are *connected* if they share an edge. For this problem we assume each processor has a unique id of length $\log_2 n$, typically a concatenation of its row and column coordinates. The label in each component will be the largest id of any processor in the component.

1) Polymorphic-torus algorithms:

To label the components in the Polymorphic-torus, first each processor with a white pixel opens all of its gates, while each processor with a black pixel closes all of its gates. This has the result of making turning all of the processors in a single component into a single circuit. These switch settings stay fixed for the remainder of the algorithm. Now a sequence of steps occurs, for $i = 1, \ldots, \log_2 n$. Initially each black processor is "live", and during the ith step, if a live black processor has a 1 in the ith highest position of its ID, then it will set a local flag to 1; otherwise the flag is set to 0. Then an OR is computed within the component, just as in the global OR. If the result was 1 then all live PE's that had a true local flag remain live, and all other PE's become "dead"; otherwise the result was 0, and all live PE's remain live.

The effect of this algorithm is to compute the component label from highest order bit down to lowest bit. At the start of stage i, the only processors alive in any component are those that have an id with the same first $i-1$ bits as the largest id in the component. At the end, only one processor in each component is live, namely the one with the largest id.

This algorithm is an illustrative example data-dependent switch settings, and on broadcasting information based on locally computed data and on data received from earlier broadcasts. It also shows how the machine can be configured to have circuits which exactly mirror the components in the image, a flexibility unmatched in fixed connection networks.

2) Reconfigurable mesh algorithms:

On the reconfigurable mesh model of [33], the component labeling is significantly more complicated, again because of the restriction that only a single processor may broadcast on any circuit at any one time. Because the shape of the component is not known in advance, and because it can be quite convoluted, there does not seem to be a simple adaptation of the above technique which works for all images. However, if the components are sufficiently nice then there is a reasonable approach. Note that if the processor ids are formed by the usual process of concatenating row and column coordinates, then in each component the processor with the largest id is on the border. Suppose each figure is sufficiently nice so that the border processors form a simple cycle, e.g., there are no bad components such as a dumbbell shaped component where the bar is only a single pixel wide. This can be checked by having each processor determine the pixels in its neighbors, and then by deciding if it is a border processor and if so then by deciding whether its local piece of the border is simple.

If all the components have borders that are simple cycles, then the bit-by-bit determination of the component label can be accomplished by using a technique similar to the OR within a row. Now each border processor with a 1 opens the gate in the counterclockwise traversal of the border (this can be determined locally by thinking of a counterclockwise traversal as one in which you walk around an object by keeping your left hand on it), and closes the gate in the clockwise traversal direction, while each border processor with a 0 closes both such gates. Because the border is a cycle, only one round of messages needs to be sent rather than the two used in a row.

At the end of the entire process, all the processors with a black pixel close all gates, all white processors open all gates, and the single live processor in each component broadcasts its id to the entire component. The time for this algorithm is the same as the time for the Polymorphic-torus algorithm, to within multiplicative constants.

Unfortunately, not all components have such nice borders and it is not known how to orient arbitrary borders. In [33], a more complicated divide-and-conquer approach is used, labeling regions with subsquares of $(n/2) \times (n/2)$ processors and then combining the labeling information of the subsquares to obtain a label for the entire image. This algorithm takes $\Theta(\log^2 n)$ time on the constant-delay model and $\Theta(\log^3 n)$ time on the logarithmic-delay model.

D. Simulation and Sorting

One measure of the power of an architecture is its ability to rapidly simulate other architectures. In general, the reconfigurable mesh can easily simulate any architecture which has a planar layout, since a given layout can be mapped into appropriate switch settings. Once the switches are set, each communication step of the given architecture can be simulated with a single communication step of the reconfigurable mesh. This may not be quite as good as one would hope since it is possible for an architecture with

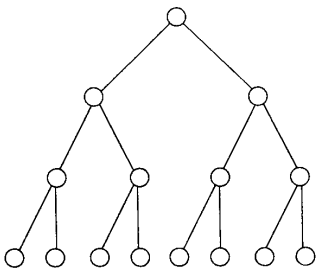

Fig. 8. Row tree processors above the base processor projected.

m processors to require as much as m^2 area for a planar layout, and hence a reconfigurable mesh of n processors may only be able to simulate an instance of the architecture with $\sqrt{n}$ processors.

1) Pyramid and mesh-of-trees: For architectures with a not-quite planar layout, the reconfigurable mesh can sometimes do quite well. Two such architectures of interest are the pyramid and mesh-of-trees. For the pyramid, the "flat pyramid" embedding noted in Section II-C is such that all East–West connections at all levels of the pyramid can be simultaneously set up on the reconfigurable mesh. Similarly, all North–South connections can be simultaneously set up, as can all parent-child connections between pyramid levels. Thus only three communication steps are needed to simulate all the communication of the pyramid. Further, a pyramid with an $m \times m$ base, and a total of $(4m^2 - 1)/3$ processors, can be so simulated on a $(2m - 1) \times (2m - 1)$ reconfigurable mesh.

The mesh-of-trees is constructed by taking an $m \times m$ base, for m a power of 2 and by adding a complete binary tree above row and a complete binary tree above column, where these trees have only their leaves (the base processors) in common. To simulate a mesh-of-trees with a $\sqrt{n} \times \sqrt{n}$ base on a $\sqrt{n} \times \sqrt{n}$ reconfigurable mesh, each processor will simulate a base processor of the mesh of trees, at most one row-tree processor, and at most one column-tree processor. This uses a projection of the row and column trees onto the base, as illustrated in Fig. 8. To simulate a communication step of the mesh-of-trees, first the base mesh communication is performed, then the row-tree communication, and then the column-tree communication. Since the row-tree and column-tree communication is similar, only the row-tree communication will be described.

The row-tree communication is simulated level by level, starting at the bottom. Notice that the projection in Fig. 8 has the property that each parent node is between its two children nodes, and that all the communication links between children at height i and their parents at height $i+1$ in the row-tree can occur simultaneously in the reconfigurable mesh. Therefore, only $\Theta(\log n)$ communication steps in the reconfigurable mesh as needed to simulate all possible row-tree communication.

The above has shown that any algorithm for a pyramid can be stepwise simulated on a reconfigurable mesh, taking only $\Theta(1)$ time per step, any algorithm for the mesh-of-trees can be simulated in only $\Theta(\log n)$ time per step. The pyramid and mesh-of-trees are normally analyzed in a constant-time model, but if they were analyzed in a logarithmic-time model then again the time on the reconfigurable mesh would only be multiplied by the indicated amount.

There are a large number of efficient algorithms for the pyramid and mesh-of-trees, typically involving images or graphs (see [44]), that therefore can be simulated to yield reasonably efficient algorithms for the reconfigurable mesh. Further, some of the mesh-of-trees algorithms have the property that only one level of the row- or column-trees are being used at any one time, and therefore, the stepwise simulation time on the reconfigurable mesh is $\Theta(1)$ instead of $\Theta(\log n)$. Such algorithms are called *normalized* in [33].

2) PRAM simulation and sorting: As a general-purpose computer, the reconfigurable mesh has the same deficiency that the plain mesh does, namely the fact that it must take $\Omega(\sqrt{n})$ time to sort n values. This is because a slice through the center cuts only $\sqrt{n}$ wires, and to sort on average half the items must cross those wires, taking at least $\Omega(\sqrt{n})$ time. For some applications this is a serious deficiency, but for others it can be gotten around by noting that the reconfigurable mesh does quite well on sorting $\sqrt{n}$ values if they are initially stored one per row or one per column.

To accomplish this, the items are first moved to the diagonal processors. Then in each row the value in the row's diagonal processor is broadcast, and then in each column the value in the column's diagonal processor is broadcast. Now each processor compares its row value to its column value, and if the row is greater then it sets a flag 1, otherwise it sets the flag to 0 (in case of ties it sets the flag equal to 1 if the row coordinate is at least as large as the column coordinate). In each row, the rank of the item in the diagonal processor is the sum of the flags in the row. These flags can be summed in a tree-like fashion, first forming the sum of pairs, then the sum of four consecutive processors, etc. Using bit-serial communication, this would take a total of $\Theta(\log^2 n)$ time on the constant-delay model and $\Theta(\log^3 n)$ time on the logarithmic-delay model. In the reconfigurable model considered in [33] the communication is assumed to be wordwise, and hence for it the time would be $\Theta(\log n)$ on the constant-delay model, and $\Theta(\log^2 n)$ on the logarithmic-delay model.

Another use of the diagonal processors comes from simulating algorithms for a Parallel Random Access Machine (PRAM) with exclusive write and either exclusive or concurrent read. If there are just $\sqrt{n}$ processors and memory cells of the PRAM to be simulated, then the diagonal processors simulate these. For each communication step, if PRAM processor i is sending information to process of memory cell j, then first simultaneous row broadcasts are used to give the destination and value being sent. Each processor checks to see if the destination is equal to its column. Then simultaneous column broadcasts are used to

send the value to the appropriate diagonal processor. As long as no two messages are simultaneously being sent to the same location, there will be no conflicts, and a step of the PRAM can be simulated in $\Theta(1)$ communication step on the reconfigurable mesh.

This PRAM simulation can be useful in divide-and-conquer image algorithms where first a problem is solved on subimages, and then the pieces are put together to solve the entire problem. The reason for this is that often the merging operation involves $\theta(n)$ words of information and hence the above techniques can be applied. The PRAM simulation is used in the component labeling algorithm in [33], and in algorithms for finding nearest neighbors and convex hulls.

IV. Fault Tolerance

Fault tolerance has been treated as a luxurious subject in the computer development and has been emphasized only in areas where a malfunction of the computer may lead to fatal results such as the space shuttle and nuclear power plant control, etc.. For massively parallel processing, the fault tolerance needs to be treated as part of the design because the probability of the occurrence of a fault within such a large amount of devices is very high. Many levels of fault tolerance are possible, e.g., circuit or memory; nevertheless this section focuses on the use of reconfiguration as a means for fault recovery. We describe several reconfiguration schemes for fault tolerance of massively parallel computers.

A. Row/Column Replacement for Mesh Network

Row/Column replacement is a popular reconfiguration scheme for fault tolerance [36], [45], [46] of a two-dimensional array. In such a scheme (Fig. 9), processors in a row (column) which contains one or more faulty processors are bypassed and treated as "connecting processors" via a built-in switch or multiplexer, and a spare row (column) is switched in to make up the loss of the bypassed row (column). The advantage of such a scheme is its simplicity: the extra hardware to support the bypassing is minimal, and the algorithm to replace the fault is efficient. The disadvantage however is the waste of the entire row (column) when there is only one or few processors at fault.

The row/column replacement scheme can be readily supported by the reconfiguration architectures discussed in Section II. For example, the bypassing of a column can be performed by connecting the East and West ports of all processors in the column in the Polymorphic-torus network. Similarly, for the Gated-Connection Network, bypassing a processor involves the activation of two gates of the switch in the processor (e.g., W and E) and activating the appropriate gates of the entire row accomplishes the column replacement. It is important to note that for a reconfigurable architecture the faults are recovered via reconfiguration in exactly the same way as an algorithm mapping by controlling the open/close position of the switches. The design of a reconfigurable architecture considers a faulty condition as a normal operating mode of the system. A unified mechanism (i.e., reconfiguration) is provided to handle both the algorithm mapping and the faults.

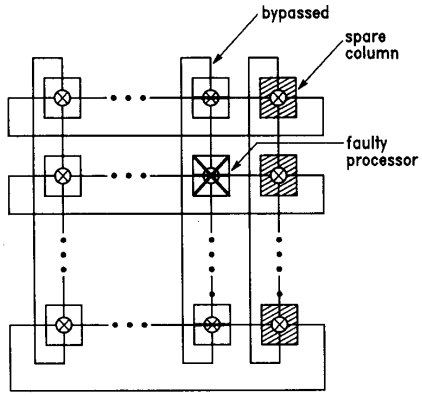

Fig. 9. A row/column replacement scheme by reconfiguration.

The row/column replacement is performed on PAPIA2 via reconfiguration for fault tolerance in a similar manner but at a higher cost. When a processor is found at fault, two adjacent columns are disabled (i.e., the column containing the faulty processor and the previous or the following column). Two spare columns are activated to make up the loss.

The waste of processors in a massively parallel system using row/column replacement scheme can be large. This can be improved by localizing the replacement. In a localized row/column replacement scheme, the massively parallel system logically arranged as a two-dimensional array of size $M \times N$ is decomposed into many smaller $m \times n$ arrays, each of which is equipped with local spare rows and columns and the replacement takes place only locally. That is, a spare in $m(i) \times n(i)$ subarray can replace a faulty processor only in the same subarray. The waste of the spare processors caused by one fault is restricted to $\max(m, n)$ which can be a large saving when $M \gg m$ or $N \gg n$. Furthermore, considering that $m \times n$ can be fitted into a chip, the signal delay due to the bypassing is small and the worst signal delay can be estimated regardless of the system size. Consequently, the system clock can be designed in a more controllable manner. The localized scheme offers advantages in engineering a large system.

The simplicity of the row/column replacement can be explained from the viewpoint of mapping. Removing a row/column entirely from the network is equivalent to maintaining the topology of the network such that the mapping of an algorithm onto the network remains unchanged. This is a tradeoff of using either hardware or software for fault recovery. The row/column replacement demonstrates that using large amount of hardware resource (i.e., bypassed nonfaulty processors), although expensive, is an effective way of simplifying the mapping.

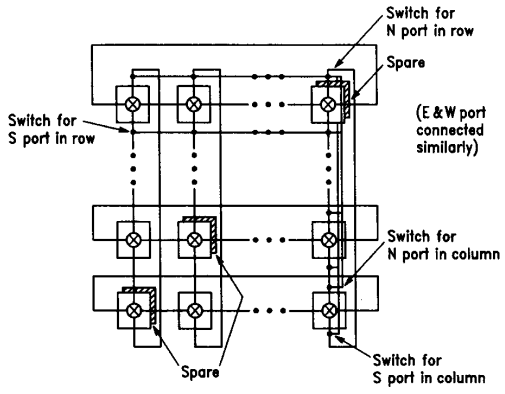

Fig. 10. A diagonal replacement scheme.

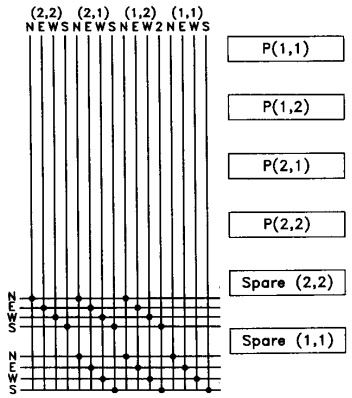

Fig. 11. A layout for a diagonal replacement scheme.

The idea of row/column replacement can be extended to topologies other than the two-dimensional mesh. The goal is to remove a small number of nonfaulty processors while maintaining the original topology. Such a goal requires to include enough redundancy in the original design. The localized row/column replacement is one such example. By recognizing that a small array is a subset of a large array and by building redundancy in each subarray, one can remove a fault at a smaller cost.

B. Localized Diagonal Replacement Scheme

The waste of the bypassed nonfaulty processors in the row/column replacement scheme can be improved by the localized diagonal replacement scheme which uses one spare processor for each processor in the diagonal location of the two-dimensional array. The spare processor at (i,i) location is connected to every processor in the ith row and i-th column, and it can replace any faulty processor in the i-th row or ith column. Since one spare processor can recover one fault, the utilization of the spare is increased in comparison with the row/column replacement scheme. However, when multiple faults occur at the same row i, the spare (i,i) can only be used to recover one fault in the row, and the rest of the faults need to be recovered by the spares located in their column indices. One can therefore create a fault pattern consisting of faulty processors at (s,t), (s,y) and (x,t) locations; such a fault condition can not be recovered by the diagonal replacement scheme. Enhancement to improve the recovery of such a fault pattern will be discussed shortly.

Extra wires and switches are needed to implement the diagonal replacement scheme. The N port of a spare processor (d,d) is connected to all N ports of the regular processors in the same row and to all N ports of the regular processors in the same column through switches (Fig. 10). The extra connections for the S, E, and W ports of the spare processor are established in a similar way. One switch is inserted in each extra connection wire and is turned on

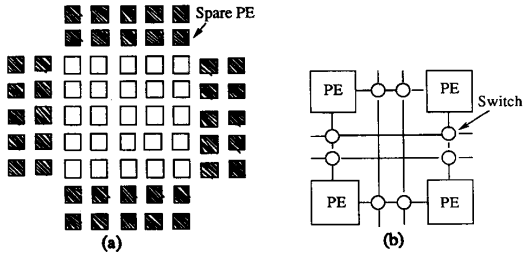

Fig. 12. A multitrack scheme.

when the spare is to be used. Considering the case that the faulty processor (d,j) is to be replaced by the spare (d,d), the switches controlling the four ports of $P(d,j)$ to $SP(d,d)$ are turned on so that the data flow into the faulty processor are guided to the spare. The fault is isolated.

At the first glance, the extra wires required for the diagonal replacement seems to destroy the regularity of the mesh connectivity. However, one layout pattern shows that it can be a systematic and efficient fault tolerant scheme. Figure 11 shows that the ports of the regular processors are routed to the side of the processor rows and are available for connection vertically through the entire chip. A spare processor is placed in row direction and its ports are connected to all appropriate ports via "opening contacts" at the vertical connection. When the scheme is localized, the suggested layout imposes only a small overhead in VLSI implementation.

The above described layout for the diagonal replacement scheme also provides an efficient way to add more spare processors to every diagonal position to increase the probability of recovery. More than one spare processor can be added to the diagonal positions so that more faults can be recovered. The connectivity of the spare to the

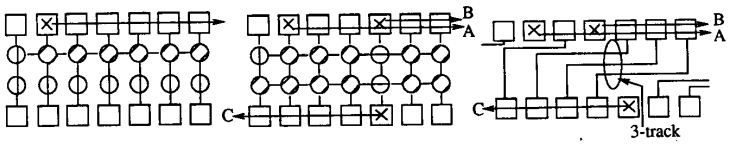

Fig. 13. A fault recover example using a multitrack scheme.

regular processors is a straightforward duplication. It is also possible to put spares in positions other than the diagonal. Since the vertical wiring for the ports of the regular processors is readily available, the difference will be "opening the contacts" at nondiagonal connections. This later option further enhances the utilization of the spare processors.

Generalizing the scheme to the extreme, the area where the vertical ports meet the ports of spare processors can be considered as a "reconfiguration area" in which a spare processor can be programmed as a spare to certain rows and columns at the chip fabrication stage, and can be controlled for replacement at the run time. What is more intriguing is that when three-dimensional silicon structure is realized [47] one layer of the silicon structure can be dedicated as "the reconfiguration area" for fault tolerance purpose only.

C. Multitrack Scheme

As shown in Fig. 12, the multitrack scheme [48]–[51] inserts several tracks of switches between two rows of processors (Fig. 12(b)) and provides spares around the edges of the two-dimensional array (Fig. 12(a)). By reconfiguring the switches, the ports of the regular and spare processors are reconstructed as an array. The purpose of providing more tracks is to recover more faults with the same number of spares. The scheme is an attempt to improve upon the row/column replacement scheme by using one spare (instead the entire row of spares) to recover one fault. It is recognized that more alternative paths are needed when multiple faults occur, as a result, multiple tracks of switches provide the alternative paths. An example of recovering multiple faults using multiple tracks is shown in Fig. 13. Research has been widely conducted to determine the number of tracks, the probability of successful recovery, the recovery algorithm, and the track structure.

The multitrack and the diagonal replacement scheme share the same goal to recover one fault by one spare. They all use the reconfiguration as the fundamental mechanism to achieve the recovery. However, they differ in the distribution of the reconfiguration and extra switches. The diagonal replacement scheme centralize the "reconfiguration area" while the switches for the reconfiguration of the multitrack scheme are dispersed among the array. A spare in the multitrack scheme replaces a faulty processor in the same row while the diagonal scheme allows the spares to replace faults in the row or column direction.

Reconfigurable SIMD architectures such as Polymorphic-torus and GCN support multitrack scheme as follows. For example, of a 2-track system, two rows of processors/switches below a processor row can be treated as the switch tracks dedicated solely for connecting purpose. Reconfiguring the switch track for fault tolerant is no different from reconfiguration for algorithm mapping. Although the processing capability of the processors in the tracks are not used, the multitrack scheme on reconfigurable architectures enjoys a unified approach to both the mapping and the fault tolerance. Furthermore, there is no need to determine in advance the number of tracks because there is no distinction between switches and processors. A processor row can be converted to a switch track as demanded by the fault pattern.

D. Summary of Fault Tolerance by Reconfiguration

We have discussed three schemes to recover faults via reconfiguration as representative examples among a large body of available schemes [52], [53]. Fault recover schemes for higher dimensional topology is less understood and is an active research field [54], [55]. Besides their pertinence to the reconfigurable massively parallel architectures, the choice of presenting these schemes indicates the simplicity of the row/column replacement scheme, and the quantum jump in complexity of the diagonal and multitrack schemes. The discussion in previous sections reveals encouraging messages that these schemes are consistently supported by reconfigurable SIMD architectures via executing different fault recovery algorithms. Such a feature is an important step toward the design of a reconfigurable SIMD architecture that encompasses sufficient built-in connection autonomy so that new and more powerful fault recovery algorithms can be applied without hardware redesign.

Two subjects, the algorithm mapping and the fault-tolerance discussed in the previous sections, were treated in the past as two separate areas. The mapping is concerned with the transformation of an algorithm graph into a processor graph while the fault tolerance is concerned with the transformation of the same algorithm graph into a degraded (and/or modified) processor graph. The contribution of many reconfigurable architectures discussed in the paper is the attempt to unify the treatment of mapping and fault tolerance via a switching network with rich local autonomy.

V. Summary

We have presented in this paper a family of reconfigurable massively parallel architectures that employ the idea

of *connection autonomy* as the mechanism to efficiently support the algorithm mapping and the fault tolerance. Different implementations of the connection autonomy for reconfiguration were illustrated. We also showed the improved algorithm efficiency accomplished via reconfiguration. Faults are treated by these reconfigurable architectures in a uniform way as the algorithm mapping. A further detailed description of this architecture can be found in [56].

REFERENCES

[1] M. J. Flynn, "Some computer organizations and their effectiveness", *IEEE Trans. Comput.*, vol. 21, pp. 948–960, Sept. 1972.

[2] T. J. Fountain, "Introducing Local Autonomy to Processor Arrays," in *Machine Vision: Algorithms, Architectures and Systems,* H. Freeman, Ed. New York: Academic, 1988.

[3] M. Maresca, M. Lavin, and H. Li, "Parallel Architectures for Vision," *Proc. IEEE*, vol. 76, pp. 970–981, Aug. 1988.

[4] M. Maresca and H. Li, "Connection autonomy in SIMD architectures: A VLSI implementation," *J. Parallel and Distributed Processing*, vol. 7, no. 2, pp. 302–320, Oct. 1989.

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

[6] S. Yalamanchili and J. K. Aggarwal, "Reconfigurable strategies for parallel architectures," *IEEE Comput.*, Dec. 1985.

[7] C. Davis, S. P. Kartashev, and S. I. Kartashev, "Reconfigurable multicomputer networks for very fast real-time applications," in *Proc. NCC*, pp. 167–184.

[8] H. Li and M. Maresca, "Polymorphic-Torus Network," in *Proc. Int. Conf. Parallel Processing*, 1987.

[9] H. Li and M. Maresca, "Polymorphic-torus network," *IEEE Trans. Comput.*, vol. 38, pp. 1345–1351, Sept. 1989.

[10] H. Li and M. Maresca, "Polymorphic-torus architecture for computer vision," *IEEE Trans. Patt. Anal. Mach. Intell.*, vol. 11, pp. 233–243, Mar. 1989.

[11] H. Li and M. Sheng, "Sparse matrix vector multiplication on polymorphic-torus," in *Proc. FRONTIERS '88, 2nd Symp. Frontier of Massively Parallel Computation*, Fairfax, VA, Oct. 1988,.

[12] H. Li and M. Sheng, "Connected component labeling algorithm on polymorphic-torus architecture," in *Proc. Int. Comput. Symp.* Dec. 15–17, 1988, Taipei, China.

[13] M. Maresca, H. Li and M. Sheng, "Parallel computer vision on polymorphic torus architecture," in *Int. J. Comput. Vision and Appl.*, Nov. 1989.

[14] B. F. Wang and G. H. Chen, "Constant time algorithms for the transitive closure problem and its applications," in Proc. Int. Conf. Parallel Processing, 1990.

[15] C. C. Weems, S. P. Levitan, A. R. Hanson, E. M. Riseman, D. B. Shu, and J. G. Nash, "The image understanding architecture," *Int. J. Comput. Vision*, vol. 2, pp. 251–282, 1989.

[16] D. B. Shu and J. G. Nash, "The gated interconnection network for dynamic programming," in *Concurrent Computations,* S. K. Tewsburg, *et al.*, Eds. New York: Plenum, 1988.

[17] D. B. Shu, L. W. Chow, J. G. Nash, and C. Weems, "A content addressable, bit-serial associate processor," in *IEEE Workshop on VLSI Signal Processing,* Montery CA, Nov. 1988.

[18] D. B. Shu, J. G. Nash, and C. Weems, "Image understanding architecture and applications," in *Advances in Machine Vision,* J. L. C. Sanz, Ed. New York: Spring-Verlag, 1989.

[19] V. Cantoni and S. Levialdi, Eds., *Pyramidal System for Computer Vision*. New York: Spring-Verlag, 1986.

[20] L. Uhr, "Pyramidal multiprocessor structures, and augmented pyramid," in *Computer Structure for Image Processing,* M. J. B. Duff, Ed. New York: Academic, 1983.

[21] S. L. Tanimoto and T. Pavlidis, "A hierarchical data structure for picture processing," *Comput. Graphics and Image Processing*, vol. 4, pp. 104–109, 1975.

[22] D. H. Schaefer, P. Ho, J. Boyd, and C. Vallejos, "The GAM Pyramid," in *Parallel Computer Vision,* L. Uhr Ed. New York: Academic, 1987.

[23] V. Cantoni, M. Ferretti, S. Levialdi, and R. Stefanelli, "PAPIA: Pyramidal architecture for parallel image analysis," in *Proc. of 7th Symp. Comput. Arithmetic*, 1985.

[24] A. P. Reeves, "Pyramidal algorithms on processor arrays," in *Pyramidal System for Computer Vision*, V. Cantoni and S. Levialdi, Ed. New York: Spring-Verlag, 1986.

[25] T. J. Fountain, *Processor Array-Architectures and Applications*. New York: Academic, 1987.

[26] M. J. B. Duff, B. M. Jones, and L. J. Townsend, "Parallel processing pattern recognition system UCPR1," *Nucl. Instr. Meth.*, vol. 52, pp. 284–288.

[27] D. M. Watson, *The application of cellular logic to image processing,* PhD. dissertation, University College London, 1974.

[28] M. J. B. Duff, D. M. Watson, T. J. Fountain, and G. K. Shaw, "A cellular logic array for image processing," *Pattern Recognition*, vol. 5, pp. 229–247, 1973.

[29] T. J. Fountain and V. Goetcherian, "CLIP4 parallel processing system," *Proc. Inst. Elec. Eng.* vol. 127E, pp. 219–224.

[30] M. J. B. Duff, "Real Applications on CLIP4," in *Integrated Technology for Parallel Image Processing,* S. Levialdi, Ed. London: Academic.

[31] R. Miller, V. K. Prasanna Kumar, D. Reisis, and Q. F. Stout, "Image computations on reconfigurable VLSI arrays," in *Proc. Conf. Comput. Vision and Pattern Recognition*, 1988.

[32] J. Rothstein, "Bus automata, brains, and mental models," *IEEE Trans. Syst. Man, Cybern.*, vol. SMC-18, 1988.

[33] R. Miller, V. K. Prasanna Kumar, D. Reisis, and Q. F. Stout, "Parallel computations on reconfigurable meshes," *IEEE Trans. Comput.* vol. 39, 1990.

[34] S. H. Unger, "A computer oriented toward spatial problems," *Proc. IRE*, vol. 46, pp. 1744–1750, 1958.

[35] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The Illiac IV Computer," *IEEE Trans. Comput.*, vol. 17, pp. 746–757, 1968.

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

[37] P. M. Flanders, D. J. Hunt, F. S. Reddaway, and D. Parkinson, "Efficient high-speed computing with the distributed-array processor," in *High Speed Computing and Algorithm Organization,* D. J. Kuck, D. H. Lowrie and A. H. Sameh Eds., 1977.

[38] D. K. Arvind, I. N. Robinson, and I. N. Parker, "A VLSI chip for realtime image processing," in *IEEE Int. Symp. Circuits Syst.*, pp. 405–408, 1983.

[39] T. Kondo, T. Nakashima, M. Aoki, and T. Sudo, "An LSI adaptive processor," *IEEE J. Solid-State Circuits*, vol. 18, 1983.

[40] W. J. Dally, "A VLSI Architecture for concurrent data structure," Ph.D. dissertation, Calif. Inst. Technol., 1986.

[41] W. J. Dally and C. L. Seitz, "Torus routing chip," in *Distributed Computing*, vol. 1, no. 4, 1986.

[42] W. J. Dally and P. Song, "Design of a self-timed VLSI multiprocessor communication controller," in *Proc. Int. Conf. Comput. Design: VLSI in Computer and Processors*, Oct. 1987.

[43] Q. F. Stout, "Using clerk in parallel processing," in *Proc. 23rd IEEE Symp. Found. Comput. Sci.*, 1982.

[44] R. Miller and Q. F. Stout, *Parallel Algorithms for Regular Architectures*.Cambridge, MA: MIT Press, 1990.

[45] I. Koren, "A reconfigurable and fault tolerant VLSI multiprocessor," in *8th Symp. Comput. Architecture*, pp. 425–442, 1981.

[46] S. Y. Kuo and W. K. Fuchs, "Efficient spare allocation for reconfigurable arrays," *IEEE Design and Test*, Feb. 1987.

[47] G. R. Nudd, R. D. Etchells, and J. Grinberg, "Three-dimensional VLSI architecture for image understanding," *J. Parallel and Distributed Computing*, vol. 2, no. 1, pp. 1–29, Feb. 1985.

[48] S. N. Jean, H. C. Fu, and S. Y. Kung, "Yield enhancement for WSI array processor using two-and-half-track switches," in *Proc. Int. Conf. Wafer Scale Integration*, Jan. 1990.

[49] M. Sami and R. Stefanelli, "Reconfigurable architectures for VLSI processing array," *Proc. IEEE*, vol. 74, pp. 712–722, May 1986.

[50] L. Jerris, F. Lombardi, and D. Sciuto, "Orthogonal mapping: A reconfigurable strategy for fault tolerant VLSI/WSI 2D arrays," in *Proc. Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, Oct. 1988.

[51] G. Saucier, J-L. Party, E-F. Kouka, T. Midwinter, P. Ivey, M. Huch, and M. Glesner, "Defect tolerance in a wafer scale array for image processing," in *Proc. Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, Oct. 1988.

[52] M. Chean and J. A. B. Fortes, "A taxonomy of reconfigurable techniques for fault-tolerant processor arrays," *IEEE Comput.*, Jan. 1990.

[53] W. R. Moore, "A review of fault tolerant techniques for the enhancement of integrated circuit yield," *Proc. IEEE*, May 1986.
[54] M. S. Chen and K. G. Shin, "On hypercube fault tolerant routing using global information," in *Proc. 4th Conf. Hypercube Concurrent Computers and Applications*, Montery, CA, Mar. 1989.
[55] J. M. Gorden and Q. F. Stout, "Hypercube message routing in the presence faults," in *Proc. 3rd Conf. Hypercube Concurrent Computers and Applications*, Jan. 1988.
[56] H. Li and Q.F. Stout, *Reconfigurable Massively Parallel Computers*. Englewood Cliffs, NJ: Prentice Hall, 1991.

Hungwen Li was born in Taiwan in 1951. He received the BSEE degree in 1973 from National Taiwan University. He received the M.S. and Ph.D. degrees in electrical engineering from the University of Pittsburgh, PA, in 1977 and 1979, respectively.

Since 1976, he has been conducting research in various branches of parallel processing including architecture, algorithm, language, simulation, performance evaluation, and application. He has applied parallel computing to power system, satellite control, digital signal processing, image processing, and computer vision. He joined the Advanced Technology Laboratory of the RCA Government System Division in 1980 and in 1982 he became Manager of the Advanced Signal Processor Architecture group working primarily on the VLSI digital signal processor, advanced parallel architectures and its applications. He joined the Industrial Machine Vision group of the IBM T.J. Watson Research Center in 1983 to initiate a research project on VLSI polymorphic architecture, parallel language, and parallel vision algorithms. In 1988, he joined the IBM Almaden Research Center. His current research involves massively parallel processing for solid and fluid mechanical design and analysis for computer disk storage systems. He was the guest editor of PROCEEDINGS OF THE IEEE Special Issue on Computer Vision in 1988. He is a coeditor of the book *Reconfigurable Massively Parallel Computers* (Prentice Hall, 1991). He holds several U.S. patents on parallel computers.

Quentin F. Stout (Member, IEEE) learned computing in the public schools of Euclid, OH. He received the B.A. degree from Centre College, Danville, KY and the Ph.D. degree from Indiana University.

Since 1984 he has been an Associate Professor in the Department of Electrical Engineering and Computer Science of the University of Michigan, Ann Arbor, where he is a member of the Advanced Computer Architecture Laboratory and the Laboratory for Scientific Computing. His primary research interests are in parallel algorithms, parallel computing, and parallel architectures. With H. Li, he recently coedited the book *Reconfigurable Massively Parallel Computers* (Prentice Hall, 1991), and with R. Miller he is completing the book *Parallel Algorithms for Regular Architectures* (MIT Press, 1991).

Dr. Stout is a member of the Association for Computing Machinery and the American Mathematical Society, and serves on the editorial boards of INFORMATION PROCESSING LETTERS and the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS.