Virtual Channels for Fault-Tolerant Programmable Two-Dimensional Processor Arrays

H.T. Kung and O. Menzilcioglu

December, 1986
CMU-CS-87-171

ABSTRACT

A programmable two-dimensional (2D) processor array is fault-tolerant if faulty processors can be detected, and then avoided during program execution. In the literature there are many schemes on detecting faulty processors and reconfiguring data routing to avoid them. However, an efficient implementation of these schemes on a 2D array can be an extremely difficult programming task if application, fault detection and reconfiguration must all be considered at the same time. The virtual channels mechanism of this paper allows these concerns to be dealt with separately and efficiently.

An application or fault detection program may assume that every logical connection between processors is implemented by a dedicated physical connection. A physical connection is composed of a sequence of virtual channels. Since the number of virtual channels between any two processors is not bounded by the number of available physical channels, all dedicated physical connections required by the program can be implemented. The mapping of logical connections to physical connections and the scheduling of a physical channel to implement multiple virtual channels are totally transparent to a program, and can be optimized independently. Various fault tolerance schemes are now readily implementable without programming difficulty. For example, it is straightforward to have concurrent execution of application and fault detection programs on the same 2D array.

A switch architecture is presented for implementing the virtual channels mechanism. This architecture is planned to be used in building a fault-tolerant 2D Warp array.

The research was supported in part by Defense Advanced Research Projects Agency (DOD), monitored by U. S. Army Engineer Topographic Laboratories under Contract

1. Introduction

This study results from a joint project with General Electric for developing a fault-tolerant programmable 2D array of Warp cells. A Warp cell is a programmable processor with 10 MFLOPS computation power, and is currently used in the one-dimensional processor array inside a high-performance system called Warp [1, 2].

A 2D processor array, such as a systolic array, can provide very high throughput. This is because numerous processors, called cells in this paper, work together in an interdependent fashion to solve a problem, and the 2D interconnection provides high bandwidth inter-cell communication. Because a large number of cells can be present in a 2D array, the probability of a failure can be high. Therefore, fault tolerance is an important issue for 2D processor arrays.

A fault tolerance scheme for a 2D array basically consists of identifying faulty cells, called fault detection, and replacing faulty cells with redundant ones, called reconfiguration. Many fault tolerance schemes have been proposed in the literature.

However, an efficient implementation of such a scheme on a 2D array is often a non-trivial programming task. In general, programming a perfect 2D array just for an application or for fault detection alone is already a difficult task. Unless the application, fault detection, and reconfiguration can each be programmed independently from the others, programming a fault-tolerant 2D array is impractical.

To make the programs independent from each other and from the configuration of the physical array, we use a program model that allows an application or fault detection program to assume a perfect logical array dedicated to its own use. The program does not specify how the logical array is mapped to the physical array during program execution. The mapping is done by a separate mechanism transparent to the program.

With this program model, reconfiguring the array for fault tolerance means implementing a new mapping. Previous work on fault-tolerant 2D arrays has mostly concentrated on 2D arrays of simple cells with VLSI and wafer-scale implementations [3, 4, 5, 6, 7, 9, 11]. Mapping mechanisms which only utilize some of the functional cells may be acceptable for arrays with simple cells. With cells as powerful as the Warp cell (which is implemented on a large 15″×17″ board), one cannot afford inefficient mapping mechanisms which result in a significant performance degradation due to an under-utilization of functional cells.

Implementing a mapping mechanism which achieves a high utilization of functional cells is the problem. For a given fault distribution, it may be difficult, or even impossible, to find a physical connection for every one of the logical connections required by the program. This is because the number of physical connections between a cell and other cells is limited, while the number of logical connections to be implemented can be very large.

The mapping mechanism can be further complicated when run-time fault detection schemes, such as shadowing [12] and concurrent testing of spare cells, are considered. These schemes require logical connections between cells beyond those required by the application program.

The problem of implementing an efficient mapping mechanism is solved in this paper by providing as many virtual connections between cells as needed for any particular mapping. Virtual connections are implemented using switches, and channels which are connections between switches. The mechanism is called "virtual" channels because multiple channels are implemented by a single physical channel. With the virtual channels mechanism, many fault tolerance schemes considered in the literature can be implemented efficiently without programming difficulty.

This paper covers motivation, programming, architecture, and implementation issues, and is not on fault tolerance schemes per se. Many fault tolerance schemes have been proposed in the literature [8, 13], and some of those are illustrated in this paper. Following is the outline of the rest of the paper. Section 2 describes some fault tolerance schemes considered for the 2D array and presents a physical array for their implementation. Section 3 defines the program model. Section 4 discusses how to implement the program model and introduces the concept of virtual channels together with a switch for virtual channels. Section 5 covers the scheduling strategies for physical channels and switches in implementing virtual channels. A switch architecture for virtual channels is presented in Section 6, while board and semi-custom chip implementations of the architecture are discussed in Section 7. Section 8 is a summary with some concluding remarks.

2. General background for fault-tolerant 2D processor arrays

Consider an application program requiring a 3×3 processor array as shown in Figure 1, where cells are represented by squares and I/O buffers by rectangles.

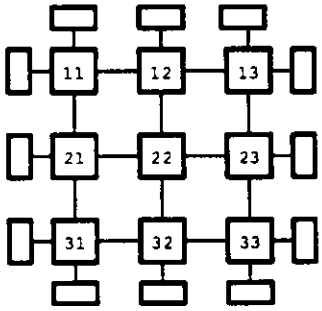

Figure 1. 3×3 array for application program

Fault tolerance for the 3×3 array can be provided by using a 4×4 array as shown in Figure 2. A cell can be either functional or faulty. A functional cell used by the application program is called a *normal cell*, while a functional cell not used by the program is called a *redundant cell*. When a normal cell fails, it is replaced by a redundant one;

therefore, for the example of Figure 2, up to 7 cell failures can be tolerated until the array is no longer functional. The figure shows a configuration with 7 faulty cells, each identified by an **F**. Replacing a faulty cell with a redundant cell and reestablishing the necessary connections between normal cells is called *reconfiguration*. The rest of the paper assumes that there exists a central controller, called the *reconfiguration master*, which monitors and reconfigures the array as needed.

Whether a cell is functional or faulty is determined by *fault detection* mechanisms. In addition to any fault detection mechanism internal to each cell, such as parity checking, fault detection mechanisms can be implemented at the array level to test one or more cells externally or by each other [10, 12, 14]. Two types of fault detection mechanisms are considered to illustrate the idea: *shadowing* and *off-line testing* [12].

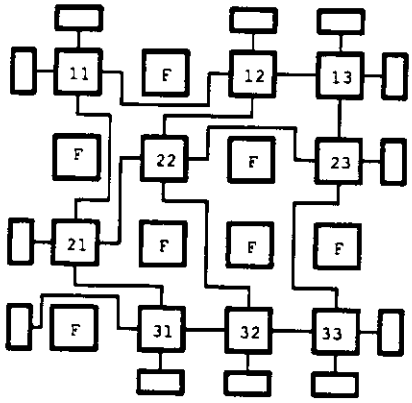

Figure 2. Reconfigured 4×4 array to implement a 3×3 array

Shadowing involves a pair of cells called the *shadowing pair*. A shadowing pair consists of a normal cell called the *shadowed* cell and a redundant cell called the *shadowing* cell. By duplicating the program of the shadowed cell on the shadowing cell and providing the same inputs to both cells one expects that both cells will generate identical outputs. If there is an inconsistency, the shadowed cell is replaced by another redundant cell, and the shadowing pair are tested further to identify whether the shadowed cell or the shadowing cell was the faulty one. Figure 3 shows the case where normal cell **23** is shadowed by a redundant cell **s23**. Assume that only cells **13** and **22** provide inputs to **22**. The same inputs are also provided to **s23** as identified by the highlighted connections. One scheme of comparing outputs of cells **23** and **s23** uses a data compression mechanism at the output of each cell. This implies that outputs of the shadowing pair need not be checked every cycle.

Off-line testing involves running a series of exhaustive tests on a cell to determine whether a cell is functional or faulty, and if faulty whether the faults are permanent or transient. Off-line testing is typically used on functional cells, normal or redundant, to ensure that they are still functional. Also, when a cell is suspected of being faulty, as indicated by internal fault detection mechanisms or shadowing, the cell can be removed from the normal array for off-line testing. In off-line testing, a cell can be either self-tested or tested externally. Multiple cells can also be

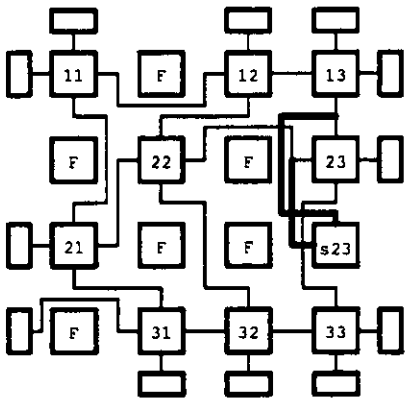

Figure 3. Reconfigured array with shadowing

tested together or independently. Figure 4 shows the case where the cell identified as T is being tested off-line and externally. The highlighted connections provide input and output paths for the off-line testing. It is important that off-line testing and application programs run concurrently on the array for the following reason. Since off-line testing involves exhaustive testing of the cell, it may not be practical or possible, for certain applications, to halt the application program frequently to test the array. On the other hand, if off-line testing does not take place frequently then the functionality of a redundant cell cannot be ensured with confidence.

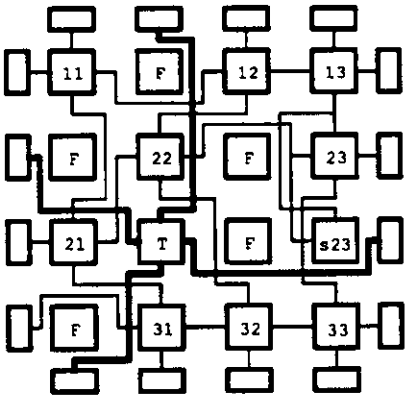

Figure 4. Reconfigured array with shadowing and off-line testing

The physical array must provide the capability of routing around faulty cells to implement connections of the application program, as well as the connections needed for shadowing and off-line testing. Figure 5 shows a 4×4 physical array proposed in this paper. The squares and rectangles represent the cells and I/O buffers as before. The circles, called *switches*, are connected to cells and I/O buffers with bidirectional lines called *physical channels*. A switch has 6 ports, each connected to a physical channel, and can establish any connection pattern between its ports. For a particular switch, the four ports which are connected to neighboring switches are identified as N, S, E, and W corresponding to the four directions in the 2D array geometry. The two switch ports connected to the cell are identified as X and Y.

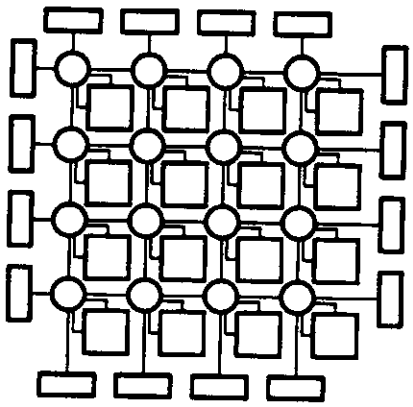

Figure 5. Physical array

All cells and logical connections in Figure 4 must be mapped to cells and physical connections in Figure 5. The rest of this paper describes a general mechanism to implement such a mapping.

3. Program model

We describe the program model we propose to use for fault-tolerant programmable arrays.

A *physical node* is a cell or I/O buffer, and a *physical path* connecting two physical nodes is a chain of physical channels connected by switches. An application or fault detection *program* is a directed graph of *logical nodes*. A directed edge between two logical nodes is called a *logical path*. Before program execution, each logical node is mapped to a dedicated physical node and each logical path is mapped to a physical path connecting the end nodes of the logical path. During program execution, the functionality of the logical node is implemented by the physical node and the data associated with the logical path is transmitted by the physical path.

The mapping and the mechanisms in the physical array to implement the mapping are totally transparent to the program. The separation of the program from the mapping mechanism and its implementation is the essence of this program model. The program model has the following advantages.

Programs are independent from the physical array configuration. Programs see only perfect logical arrays and need not be changed when the physical array is reconfigured.

Programs are independent of each other. Multiple programs can execute concurrently on the physical array by mapping the logical nodes and logical paths of *all* programs together onto the physical array. As described in the previous section, it is important for fault detection programs, such as shadowing and off-line testing of redundant cells, to execute concurrently with the application program. Furthermore, programs can be developed independently, that is, in writing a program one will not have to be aware of other programs that may run concurrently with

that program on the 2D array.

4. The concept of virtual channels

There are two ways of mapping logical paths to physical paths. One way is to map each logical path to a separate physical path. To map the logical paths required by all the programs (for application and fault detection), many physical channels may be needed between neighboring cells, as shown in Figure 4. Providing as many physical channels as can be needed for all possible mappings to the array is not a realistic alternative, especially if the array size is not very small. If there are not many physical channels, the reconfiguration requires sophisticated placement and routing processes which can take a long time. To map a given program onto a given physical array configuration, one may have to exhaust all possible mappings before determining that not all logical paths can be mapped to a dedicated physical path. This for example can deter the reconfiguration master from changing configurations frequently to rotate the cells for off-line testing.

The other way of mapping logical paths to physical paths is to multiplex the use of a physical channel among multiple logical paths. This is the only feasible mechanism to implement many logical paths with limited number of physical channels. However, a careful implementation of multiplexed channels is necessary to ensure efficient utilization of physical channels, and to avoid potential problems of starvation and deadlock.

4.1. Virtual channels

Consider a physical channel shared by one or more logical paths. The physical channel will implement a number of *virtual channels*, each dedicated to a single logical path. The data transmissions in a virtual channel will be totally independent from transmissions in all other virtual channels implemented by the same physical channel. Hereafter, we will assume that each logical path is mapped to a dedicated *virtual path* consisting of a chain of virtual channels connected by switches. For a program, a dedicated virtual path is no different from a dedicated physical path. Suppose that each physical channel in Figure 5 can implement up to four virtual channels, then we have the array shown in Figure 6.

4.2. Switch for virtual channels

A switch has six ports, each connected to a physical channel implementing a number of virtual channels. We say that these virtual channels are connected to the switch. Hereafter, a switch is viewed as a switch for the virtual channels connected to it.

Inside the switch, if data is routed from one virtual channel to another virtual channel, then we say that there is a *virtual link* from the former to the latter. The function of a switch is to route all the data according to a set of virtual links defined by the reconfiguration master. If data from one virtual channel is broadcasted to more than one virtual channel, then multiple virtual links share the same virtual channel as their source. Note that a virtual channel can be the source of multiple virtual links, but can only be the destination of a single virtual link.

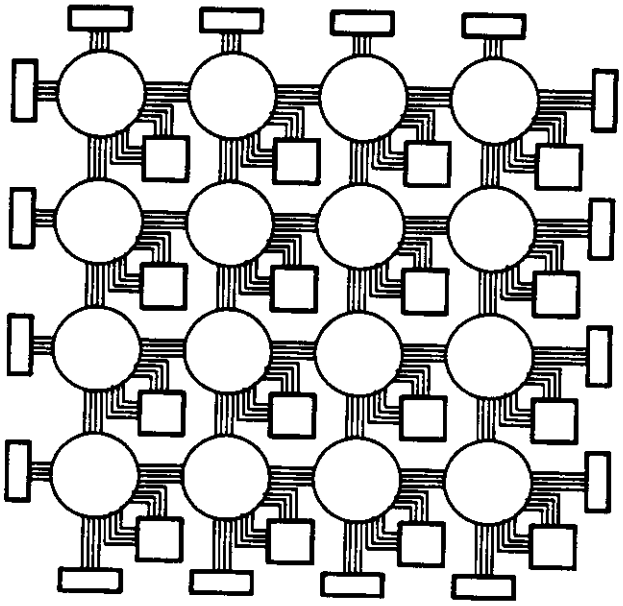

Figure 6. Virtual channels implemented by the physical array of Figure 5

To implement the virtual links we use *physical links* between switch ports. To have a dedicated physical link for each virtual link can be very costly in hardware. Therefore, physical links and physical ports may have to be multiplexed to implement all the virtual links. The total bandwidth of the physical links and physical ports determine the performance and cost of the switch.

5. Scheduling strategies for physical channels and switches

Consider a physical channel implementing several virtual channels. A *cycle* is defined as the time it takes to transmit a unit of data, called a *word*, across the physical channel. *Allocating* the physical channel to a virtual channel means transmitting a word for the virtual channel over the physical channel. In one cycle, the physical channel can be allocated only to one virtual channel. *Scheduling* the physical channel means defining the order in which the physical channel is allocated to virtual channels.

Scheduling the switch means that defining the order in which a set of virtual links will be routed. The scheduling involves scheduling the physical ports as well as the physical links. The scheduling problem can be non-trivial if one wants to minimize the number of cycles to route all the virtual links while assuming a modest switch hardware.

5.1. Static vs. dynamic scheduling

Consider scheduling a physical channel. If the schedule is *static*, every cycle the scheduler allocates the physical channel to a virtual channel according to some pre-determined order, whether or not the virtual channel has a word to transmit in that cycle. If the virtual channel doesn't have a word to transmit in that cycle, then a cycle is wasted. Therefore, static scheduling can lead to poor utilization of physical channels. If the schedule is *dynamic*, the scheduler allocates the physical channel only to a virtual channel that has a word to transmit. Since dynamic

scheduling requires knowing every cycle which virtual channels have a word present, it can be costlier to implement.

Similarly, the scheduling of a switch can be static or dynamic.

5.2. Global vs. local scheduling

Global scheduling means that scheduling the ports of the switch is not decoupled from scheduling the physical channels to which the ports are connected. In this case, the schedules of the physical channels and the switches must be carefully coordinated. In particular, all the switches that are connected by virtual paths must be scheduled together. Figure 7 shows an example where three switches and several virtual paths are involved. It is impossible to schedule any switch or physical channel independent from other switches and channels. The figure shows one global schedule where the order of allocating virtual channels and routing virtual links are indicated by the numbers next to each one. It is assumed that there is only one physical channel between any two neighboring switches and a switch has two physical links, so that two virtual links can be routed in one cycle. Not decoupling the scheduling of physical channels from that of switch ports can lead to a simple switch hardware. However, global scheduling is extremely difficult, especially for 2D arrays, because many switches and physical channels may have to be scheduled together. It also requires a static scheduling, and a synchronous array where the data movements between nodes in any cycle are predetermined.

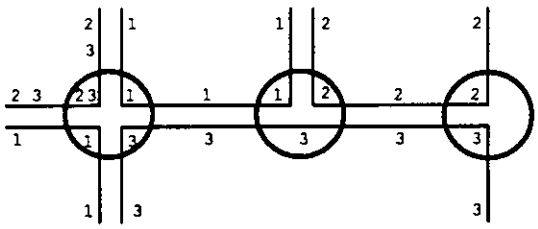

Figure 7. Example of global scheduling

Local scheduling means that each switch can be scheduled independently from other switches. This can be achieved by using queues to decouple the switch ports from the physical channels as shown in Figure 8. Across a physical channel, data are transferred between a pair of queues connected by a virtual channel, and within the switch data are transferred between a pair of queues linked by a virtual link. Since queues provide the necessary buffering, the schedule of a physical channel or switch is independent from other physical channels and switches. In Figure 8, the schedule for each physical channel and switch is assigned arbitrarily. Note that, the switch is still assumed to have four physical ports and two physical links. With local scheduling, the switches and the physical channels can have their own independent schedules, which can be dynamic or static, and the array can be synchronous or asynchronous.

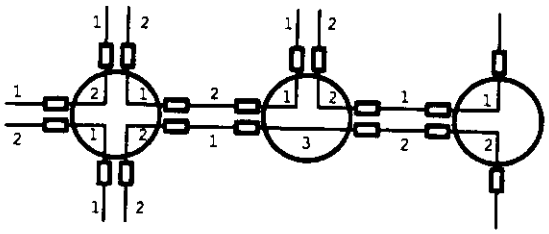

Figure 8. Example of local scheduling

5.3. Choosing the scheduling strategy

Local scheduling is preferred over global scheduling because global scheduling is very difficult, especially for large 2D arrays.

While static scheduling is relatively easy to implement, it can lead to significantly poorer performance than dynamic scheduling. In the worst case, static scheduling for either the switch or the physical channel can require L times the cycles required by dynamic scheduling to transfer the same amount of data, where L is the maximum number of virtual channels implementable by a physical channel. However, in the following cases static scheduling in the switch or the physical channel may be acceptable.

Both switch and physical channels may use static scheduling without loss of performance if the overall inter-node communication is low compared to computation. In this case, lower utilization of physical channels would not have an impact in the overall performance since communication is not a bottleneck. If the traffic in virtual channels in general is balanced and the queues at switch ports are large, static and dynamic scheduling should have similar performances. Finally, if the switch or the physical channel is significantly faster than the other, then static scheduling can be acceptable for the faster component.

Therefore, although in the next section we present a switch architecture using dynamic scheduling for both the switch and the physical channels, static scheduling may still be preferred sometimes depending on the performance requirements and implementation constraints.

6. A switch architecture for virtual channels

In this section, a switch architecture, similar to that shown in Figure 8, is proposed which supports dynamic scheduling for both the switch and the physical channels.

6.1. Switch interface

Figure 9 shows the interface between two switches. The interface consists of data, tag, transfer, ownership, request, and destination status lines. Hereafter, the collection of these lines will be referred to as the *switch interface*. To support dynamic scheduling for physical channels, each word is accompanied by a tag indicating to which virtual channel the word belongs. Transfer of a valid word to and from the switch is accomplished by transfer signals. In case of a synchronous array, the transfer signal is a single valid bit. In case of an asynchronous array, it consists of

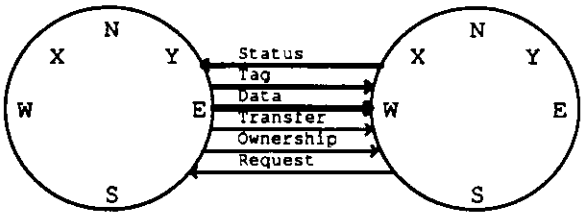

Figure 9. Switch interface

handshaking signals indicating the start and completion of a valid data transfer. Ownership indicates the current owner and direction of the channel. In Figure 9, E is the owner. Ownership remains with the current owner, until a request for ownership is received from the destination. One cycle after the request, the ownership is switched which is indicated by switching the state of the ownership line by the current owner. Destination status lines use the tag of a virtual channel to indicate that 8 more words have been removed from the destination queue of that virtual channel.

6.2. Queues

Figure 10 shows the queues in one port of the switch. There is a dedicated queue for each virtual channel which is the source of a virtual link. These queues are called *input queues*, or simply queues when there is no possibility of confusion. In Figure 10, only four input queues are shown assuming that the maximum number of virtual channels implementable by a physical channel is four. It is very important to note the difference between having a single queue for all virtual links originating at a port, and having a dedicated queue for each virtual link originating at a port. In the case of a single queue, blocking of one virtual path could stop the queue, therefore blocking the traffic in all other virtual paths sharing the queue. This would violate the principle of dedicated virtual paths.

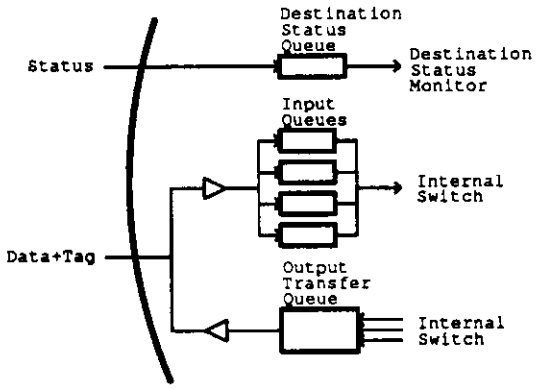

Figure 10. Queues in one port

An input queue is partitioned into segments of 8 words. As each segment is crossed by the read pointer of the queue, a segment crossing signal is generated which sends the tag of the input queue to the opposite port of the neighboring switch.

An input queue is said to be *active* if it has data, and there is space available in the destination queue to which the data is to be transferred. The status of the destination queue is determined by keeping a count of the words already transferred and by monitoring the segment crossing signals sent by the destination queue. If an input queue is broadcasting, then it is considered active when all of its destinations have space available.

The *output transfer queue* holds the words to be transferred out of the port. The output transfer queue has an input bandwidth of three and an output bandwidth of one word per cycle, and it is considered to be full if there are less than 3 available slots in the queue. The *destination status queue* brings in the segment crossing signals of the input queues at the opposite port of the neighboring switch.

6.3. Internal switch

So far, the term *switch* has been used to refer to the overall node to which the physical channels connect, which includes the switch interface and various queues. The actual routing of data for a given set of virtual links is performed by an *internal switch* which transfers data from input queues to output transfer queues as shown in Figure 11. The input queues and output transfer queues serve to isolate the schedule of the internal switch from the schedule of the physical channels. Note that the terms *input port* and *output port* refer to ports of the internal switch which are directional, while the term *port* refers to a bidirectional port of the switch as before (to indicate the directionality of virtual links between switch ports, the terms source and destination have been used). Since there can be at most one word coming in from the physical channel every cycle, it is sufficient to provide one input port for the internal switch for each port of the switch.

Since the switch has 6 ports, in the steady state transfer of 3 words across the switch every cycle is sufficient to provide the necessary bandwidth, so that the switch is not a bottleneck. This can be achieved by having three physical links in the internal switch which are implemented as buses. A bus can transfer one word and its tag from an input queue to one or more output transfer queues in one cycle. To avoid the scheduling problems for output ports of internal switch, three output ports exist for each switch port. Therefore, the internal switch has a total of 6 input ports and 18 output ports.

The dynamic scheduling for the internal switch is realized by a *dynamic round robin* mechanism. For each input port, there is an *active queue list* which indicates the active input queues in that input port. Each successive transfer from that input port is made from the input queue, which is determined by a round robin on the active queue list. The active queue list is updated as status of the input queues change. When an output transfer queue gets full, all input queues destined to that output transfer queue are removed from the active queue lists. An input port which has one or more active input queues is said to be an *active input port*. Active input ports are maintained in a list similar to the active queue list. If the number of active input ports is less than four, each input port is allocated a bus every cycle. If there are four or more active input ports, three buses are allocated to active input ports in round robin, so that each input port gets an equal opportunity to transfer. Active input port list is updated as the status of input ports

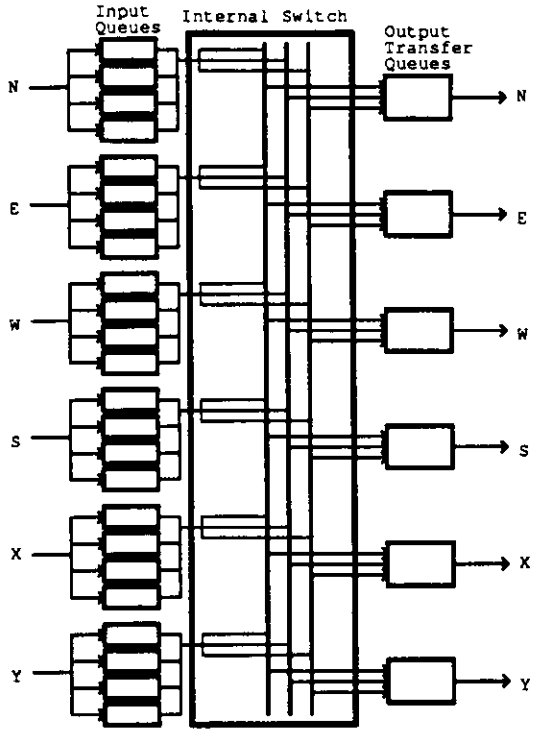

Figure 11. Internal switch

change.

6.4. Routing data

Figure 12 shows major components of the switch architecture involved in routing data from one virtual channel to another. For simplicity, only one input and two output ports are shown in the figure. When a word is received from the channel, its tag is checked and the word is stored in the input queue corresponding to the virtual channel indicated by the tag. Note that since the sender does not send a word unless the destination input queue is guaranteed to accept it, an input queue will never be full when a word is received for it. If the input queue was empty, its active status is updated after the write.

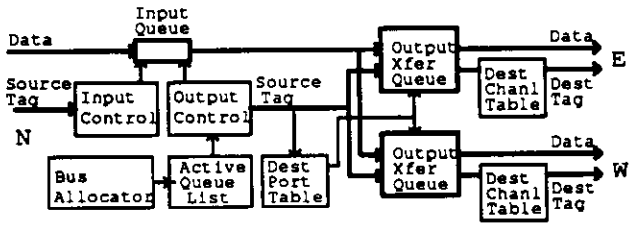

Figure 12. Routing data in the switch

When an active input queue is designated for transfer by the dynamic round robin mechanism, the destination ports for the virtual links are found by a lookup in the destination port table. For each input queue, there can be up to 5 destination ports if the input queue is broadcasting. The word is then transferred to the output transfer queue of

the destination port along with its source tag indicating the source port and source input queue of the word. When the word is removed from the output transfer queue for transfer across the physical channel, its source tag is replaced by a destination tag designating to which virtual channel the word now belongs. This is realized by a lookup in another table called the destination channel table. The destination port and destination channel tables together define the virtual links to be implemented by the switch. Therefore, the combination of these tables will be referred as the *virtual link table*. Programming the switch consists of writing the virtual link table.

7. Implementation of switch architecture

Two implementations are considered for the switch, a board implementation using off-the-shelf components, and a semi-custom chip implementation using gate-arrays. Following are the implementation considerations for major components of the switch architecture presented in the previous section. The cycle time for the physical channels is taken as 200 ns.

7.1. Board implementation:

The number of virtual channels which can be implemented by a physical channel is mostly limited by the number of dedicated queues. For a 10×10 processor array considered for the fault-tolerant Warp array, it is believed that 16 virtual channels per physical channel is sufficient. With 16 virtual channels between any adjacent cells, it appears that one can implement all possible mappings.

7.1.1. Switch interface

The switch interface and the physical channel will utilize time multiplexing of 16-bit connections to transfer 32-bit data with similarly multiplexed tag and status lines. Consequently, there will be a total of 23 lines per switch port assuming synchronous communication.

If switches are operated asynchronously, the switch interface would include small input and output queues, to isolate the asynchronous nature of transfers from the internal synchronous operation of the switch. For an initial implementation, synchronous communication will be preferred to simplify prototyping and development.

7.1.2. Queues

All input queues in a port can be implemented by a single RAM, since there is one data transfer across the port every cycle. It is possible to implement this memory using static memory chips with access times less than 50 ns. (1K-word chips with 25 ns. access time are commercially available.) A total of 1K words of memory will be statically divided among 16 queues with 64 words per queue. The output transfer queues will be implemented using faster (15 ns) but smaller static memories (256-word) to achieve 3 writes and 1 read in one cycle. The destination status queues will also use small (16-word) static memories. There will be an *input queue manager* for each port which will manage the current read and write addresses for each input queue and their active status, including the generation of segment crossing signals. The input queue manager will be implemented using 16-word fast

memories to store each of these tables.

There will be a *destination status monitor* to monitor the empty or full state of all destination input queues in the neighbors of the switch. Among the 16 possible input queues in a port, there can be at most 16 segment crossing signals generated in 128 cycles, since a segment crossing signal is generated every 8 cycles. Since there are 6 neighbors to a switch, there could be at most 96 signals pending at any 128-cycle period. Therefore, it is sufficient for the destination status monitor to check the status of one segment crossing signal every cycle (this would be valid if a segment crossing signal is generated every 6 words instead of 8, but generating the signal every 8 words will have an easier implementation). This will also assure that the destination status queues will never overflow since they can each hold 16 signals.

7.1.3. Internal switch

It appears to be feasible to implement the three buses by using a single bus with a cycle time three times faster than the channel cycle time.

The dynamic round robin mechanism can be implemented using linked lists. An active queue list consists of at most 6 separate doubly linked circular lists, one list for all input queues destined to the same port (since virtual links within the same port are not considered, there can be at most 5 destination ports from one port), and a separate list for all broadcasting input queues. For implementation reasons, a broadcasting input queue is assumed to be broadcasting to all 5 ports, even if it is not actually broadcasting to all. Figure 13 shows an example where the numbers represent 16 input queues in port N. If an active input queue becomes inactive, it is deleted from the linked list. Similarly if an inactive input queue becomes active, it is inserted in the linked list. All 6 linked lists in Figure 13 are implemented together using two 16-word fast memories, which can be accessed four times every cycle to accommodate insertions and deletions. Each word in the memory corresponds to an input queue and one memory stores links in one direction as addresses to the word next in the list, while the other memory stores the links in the other direction.

Each linked list has a pointer showing the next input queue from which to transfer. The pointer is advanced to the next in the list after each successive transfer from the list. The pointers themselves also form a circular linked list to implement a dynamic round robin mechanism on the pointers. In Figure 13, successive transfers will be made from input queues 5, 4, 13, 12, 8, 14, 9, 11, etc. If the data transfer queue to a destination port becomes full, the pointer for the list corresponding to that port should not advance. This is achieved by deleting the corresponding pointer, along with the pointer for broadcasts, from the pointer list and reinserting when the destination port is no longer full.

An input port is said to be active if it has one or more active input queues. Since there are 6 input ports, and a port can be either active or inactive, the bus allocator can be implemented as a state machine with 64 different states using 134 words of memory. (0,1,2 or 3 active ports need 1 word/state, 4 active ports need 4 words, 5 active ports

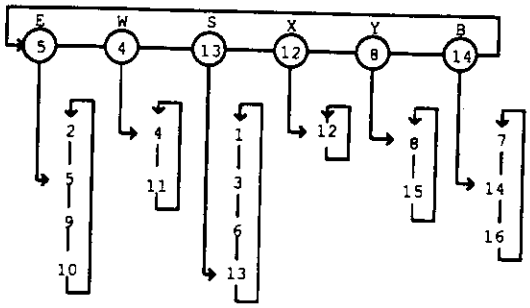

Figure 13. Implementing dynamic round robin mechanism

need 5 words, and 6 active ports need 2 words.) Once a state is entered, the state machine loops in that state allocating the buses to active ports in a round robin manner. This looping continues until the state changes.

7.1.4. Controlling the switch

Various control signals internal to the switch, i.e., those used to control the components described above, will be supplied by a built-in controller implemented by a fast PROM. Programming the switch requires writing the virtual link table externally. Unless virtual links need to be created or deleted dynamically, the switch needs no further control from outside once initialized. One virtual channel in each port is permanently assigned to carry control instructions to the switch itself. Therefore, no extra control lines are needed.

7.2. Semi-custom chip implementation

If the entire switch is implemented by a chip, 144 pins would be sufficient, assuming 23 pins per port and 6 pins for power and ground. However, it is doubtful that a gate array chip can implement all the memory needed for the switch. Therefore, a reasonable alternative is to leave the memories for input queues outside, which can be implemented using commercial chips. It is important to note the following requirement for the switch. In the steady state, the bandwidth needed to and from all input queues is 3 words to read and 3 words to write. Therefore, between the switch and input queue memories, 3 ports are sufficient, each port being able to read and write a word every cycle. Assuming the memory interface of the chip and the memory can be run 6 times faster than the channel cycle time (4K-word static memories with 25 ns access time are available), the following scheme can be implemented.

The switch chip will have a small input transfer queue (16-word) at each port. Data will be transferred from input transfer queues to the memory interface using three input buses. These buses will be allocated to each input queue using the dynamic bus allocation scheme. At the memory interface, words will be written to the corresponding input queues in the external memory one at a time. Similarly, three words will be read from the input queues determined by the dynamic round robin mechanism and brought to the memory interface. The words will be input to the internal switch and routed to output transfer queues with three output buses, as explained in the previous section. The memory interface would require additional 45 pins for the switch chip bringing the total number of pins close to 200.

8. Summary and concluding remarks

With the virtual channels mechanism, an application or fault detection program can assume that every logical path is implemented by a dedicated physical path, although in reality every logical path is implemented by a dedicated virtual path.

The implementation of the virtual paths on the physical array is transparent to the program. After a faulty cell is detected, the reconfiguration master reconfigures the array by re-assigning logical nodes to physical nodes, and each logical path to a dedicated virtual path. The reconfiguration involves setting the virtual link tables in all the switches. The scheduling of switches and physical channels to implement the virtual links and virtual channels are handled by separate mechanisms, which can be either static or dynamic.

The reconfiguration and scheduling mechanisms are independent from each other, and from application or fault detection programs. Thus each can be optimized independently according to a separate set of criteria, without concern to the others. For example, we can use a sophisticated "placement and routing" program to perform the reconfiguration, custom-made hardware to perform the schedulings, and elaborate shadowing methods to detect faults. Various fault tolerance schemes are now readily implementable without programming difficulty. In particular, it is straightforward to have concurrent execution of application and fault detection programs on the same 2D array.

The virtual channels mechanism is orthogonal to many other important fault tolerance considerations such as methods to ensure a speedy and complete fault recovery, and reliability analysis to determine the minimal number of redundant cells. For example, to provide fault tolerance for a square array, the reliability analysis may indicate that some larger non-square array could be sufficient. The virtual channels mechanism can be applied regardless of the shape of the array.

This paper concentrates only on 2D arrays. However, it should be obvious to see that the virtual channels mechanism extends naturally to other interconnection topologies beyond the 2D array.

Another possible extension is to allow the mapping of logical paths to virtual paths to change dynamically during program execution. After a logical path has finished its use of a virtual channel, the channel may be released for other logical paths to use. This can be useful when the number of available virtual channels is a limiting factor in achieving an efficient reconfiguration. However, dynamic mapping of logical paths to virtual paths requires additional hardware support to perform on-the-fly message routings, and also requires carefully designed routing protocols to avoid deadlock and starvation. This is a research topic beyond the scope of this paper.

References

1. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O., Sarocky, K. and Webb, J.A. Warp Architecture and Implementation. Conference Proceedings of the 13th Annual International Symposium on Computer Architecture, June, 1986, pp. 346-356.

2. Annaratone, M, Arnould, E., Cohn, R., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O., Sarocky, K., Senko, J., and Webb, J. Warp Architecture: From Prototype to Production. Proceedings of the 1987 National Computer Conference, AFIPS, 1987, pp. 133-140.

3. Fussel, D. and Varman, P. Fault-Tolerant Wafer-Scale Architectures for VLSI. Proceedings of the 9th Annual Symposium on Computer Architecture, April, 1982, pp. 190-198.

4. Hedlund, K.S. and Snyder, L. Systolic Architecture: A Wafer-Scale Approach. Proc. IEEE International Conference on Computer Design, Oct., 1984, pp. 604-610.

5. Hwang, J.H. and Raghavendra, C.S. VLSI Implementation of Fault-Tolerant Systolic Arrays. Proc. International Conference on Computer Design, Oct., 1986, pp. 110-113.

6. Koren, I. A Reconfigurable and Fault-Tolerant VLSI Multiprocesor Array. The 8th Annual Symposium on Computer Architecture, IEEE & ACM, May, 1981, pp. 442.

7. Kung, H.T. and Lam, M. "Wafer-Scale Integration and Two-Level Pipelined Implementations of Systolic Arrays ". *Journal of Parallel and Distributed Computing 1*, 1 (1984), 32-63. A preliminary version appeared in *Proceedings of the Conference on Advanced Research in VLSI*, MIT, January 1984, pp. 74-83..

8. Laprie, J.C. Dependable Computing and Fault Tolerance: Concepts and Terminology. 15th International Symposium on Fault-Tolerant Computing Systems, 1985.

9. Moore, W.R. Review of Fault-Tolerant Techniques for the Enhancement of Integrated Circuit Yield. Special Issue on Fault Tolerance in VLSI, May, 1986.

10. Preparata, F.P., Metze, G., Chien, R.T. "On the Connection Assignment Problem of Diagnosable Systems". *IEEE Transactions on Computers C-16* (Dec. 1967), 848-854.

11. Sami, M. and Stefanelli, R. Reconfigurable Architectures for VLSI Processing Arrays. Proc. of 1983 National Computer Conferenece, 1983, pp. 565-577.

12. Shombert, L.A. *Using Redundancy for Testable and Repairable Systolic Arrays*. Ph.D. Th., Carnegie-Mellon University, 1985.

13. Siewiorek, D.P. and Swarz, R.S.. *The Theory and Practice of Reliable System Design*. Digital Press, 1982.

14. Yang, C.L. and Masson, G.M. An Efficient Algorithm for Multiprocessor Fault Diagnosis Using the Comparison Approach. 16th International Symposium on Fault-Tolerant Computing Systems, 1986, pp. 238-243.