Comprehensive Evaluation of a Two-Dimensional Configurable Array

O. Menzilcioglu and H. T. Kung

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

S. W. Song

Institute of Statistics and Mathematics
University of São Paulo
São Paulo, Brazil

Abstract

This paper presents the evaluation of a highly configurable architecture for two-dimensional (2D) arrays of powerful processors. The evaluation is based on an array of Warp cells, a powerful processor developed at Carnegie Mellon and manufactured by General Electric, and uses real application programs. The evaluation covers the areas of configurability, array survivability, and performance degradation. The software and algorithms developed for the evaluation are also discussed. The results based on simulations of small and medium size arrays (up to 16x16), show that a high degree of configurability, and array survivability can be achieved with little impact on program performance.

1. Introduction

We have proposed a general architecture which provides a high degree of configurability for 2D processor arrays of powerful processors [5]. In this paper, we evaluate this architecture.

Many configurable, two-dimensional (2D) processor arrays have been proposed in the literature [4, 9, 11]. However, most 2D configurable array architectures have been considered for arrays of simple processors with VLSI/WSI implementations to improve the yield in fabrication [7, 10] and do not provide sufficient configurability. The architecture evaluated in this paper targets arrays of powerful processors, such as the Warp array [1] which uses programmable processors with 10 MFLOPS computation power each. The architecture intends to provide a high degree of configurability for efficient utilization of the processors, and can be used to provide increased array survivability in mission-oriented applications.

Our evaluation of the architecture is based on an array of Warp cells and uses real application programs and extensive simulation. We have shown that high configurability and survivability can be achieved with a modestly complex switch design. We also found very little degradation in performance of the simulated programs over the lifetime of small and medium size arrays.

The research was supported in part by Defense Advanced Research Projects Agency (DOD), monitored by U. S. Army Engineer Topographic Laboratories under Contract DACA76-86-C-0023.

Section 2 gives an overview of the architecture. Section 3 discusses the evaluation procedure and gives an overview of the software and the sample programs used for this purpose. Section 4 presents the results of the evaluation, and Section 5 contains a summary and conclusion.

2. Overview of architecture

Figure 2-1-a shows a representative 4x4 array. The squares represent the *processors* which are also called cells, and the rectangles on the edges represent the *I/O buffers*. The I/O buffers are also connected to a host which transfers data to and from the array. The circles represent *switches* which are connected to cells or I/O buffers with bidirectional lines called *physical channels*. A switch has six ports, four connected to neighboring switches or I/O buffers and two connected to the local cell, and can establish any connection pattern between its ports.

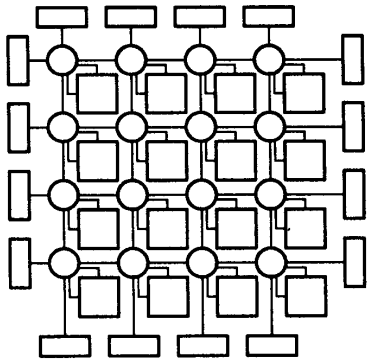


Figure 2-1: A representative 4x4 physical array

The structure of Figure 2-1-a can be applied to implement any 2D processor array, although our intention for this architecture is to focus on arrays of "powerful" processors, for which processor utilization and configurability are more critical. In specific, this architecture was considered for building a configurable and fault-tolerant 2D Warp array. We will give an overview of the Warp cell in the next section.

The array of Figure 2-1-a would have very limited con-

figurability if each physical channel were used to implement only one logical connection required by the program. As the components of the array fails, mapping the logical connections of the program onto the physical array would require multiple channels between neighboring switches. However, providing as many physical channels between neighboring switches as might be needed is a wasteful alternative and can be difficult to implement, especially for large arrays. The other alternative is to multiplex the use of a physical channel among multiple logical connections. This is a more attractive solution since it would provide high configurability with graceful degradation in the array performance. Although multiplexing the use of physical connections has the potential to cause communication bottlenecks, the premise is that *for an array of powerful cells with large local memories the load in logical connections of most programs would be sufficiently low so that a modest degree of multiplexing in physical connections would not lead to a significant performance degradation.* Based on this idea, we have proposed an interconnection architecture for 2D processor arrays with capabilities similar to the virtual circuits used in computer networks [6].

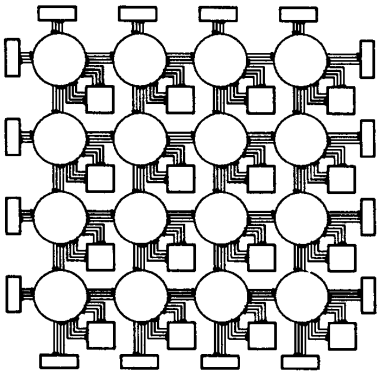


Figure 2-2: The virtual array implemented by the physical array of Figure 2-1

Consider that a physical channel in Figure 2-1-a can implement a number of *virtual channels* in each direction. For example, Figure 2-2 shows the virtual array implemented by the physical array of Figure 2-1-a, assuming the number of virtual channels implementable by a physical channel is four, two in each direction. A switch can connect an incoming virtual channel to one or more outgoing virtual channels. There is a dedicated queue for each incoming virtual channel, which makes it possible to schedule data transfers across the switch locally, independent from other switches. The data transfers across the switch and the use of a physical channel between neighboring switches is scheduled dynamically, by a round robin among logical connections having transferable data, in order to utilize the physical connections efficiently. Details of the switch architecture and issues in dealing with multiplexed channels have been discussed in an earlier paper [5]. Each logical connection required by the program is mapped onto a dedicated virtual path that begins and ends at cells or I/O buffers, and consists of a chain of virtual channels connected by switches. The data transfers in one virtual path are totally independent from data transfers in all other virtual paths; therefore, for the program a dedicated virtual path is no different than a dedicated physical path.

2.1. Warp cell

The Warp cell is a 10 MFLOPS processor developed for the Warp computer [1]. The cell communicates with other cells via two I/O paths (X and Y) where each path can transfer up to a total of 40 Mbytes data per second to and from the cell's neighbors, and has a 512-word input queue to buffer incoming data. Multiple functional units, high internal and external data bandwidth, and large local data memory (32K words) make the Warp cell a powerful computation engine and distinguish it from most processors used in systolic arrays. Each Warp cell is equipped with its own program memory (8K instructions) and microsequencer with wide instructions of 272 bits. Since each cell can be programmed independently, a heterogeneous program model where cells in the array execute different programs can be supported. The cells are programmed in a high-level language called W2 and code is generated by a highly optimizing compiler [2]. The Warp cell is implemented on a 15″×17″ board with over 250 chips.

3. Evaluation procedure

There are several significant questions about the architecture which need to be answered for a comprehensive evaluation. How many virtual channels per physical channel are needed to ensure a successful mapping of the logical array onto a faulty physical array, and whether this varies with array size, with the amount of redundancy, or with switch/cell reliability ratio? How much more reliable the switch must be compared to the cell so that the impact of added complexity on array survivability is acceptable? What is the impact of the interconnection mechanism on program performance? Do multiplexing the use of physical channels and routing data across the switches cause communication bottlenecks and a degradation in performance?

To answer these questions realistically, the architecture was evaluated through extensive simulation for arrays of Warp cells and using real application programs developed for the Warp array. In this paper, we present some of the results of this evaluation.

3.1. Simulation and mapping

It is assumed that all components of the array can fail, including cells, switches, I/O buffers, and physical channels. A fault generator is used for simulation, using an exponential fault model for each array component. It takes as inputs a physical array description along with the MTBF of each component type, and generates a sequence of component failures over time for the array. Let T_C, T_S, T_B, and T_L represent respectively the MTBFs of cell, switch, I/O buffer, and physical channel. Absolute values of MTBFs used for each component type are not relevant for the evaluation, rather their values relative to each other are significant since they affect the distribution of faults among various types of components

in the array. Let $R = T_S / T_C$ represent the ratio of switch vs. cell MTBFs. The following two assumptions are made about the relationships between the MTBFs of components:

- $T_C = T_B$. For simplicity, the I/O buffers are assumed to have the same MTBF as the cells. Indeed, in the case of a 2D Warp array an I/O buffer would have comparable hardware complexity to the cell.
- $T_L = 5T_S$. The physical channel is estimated to be five times more reliable than the switch. This estimation is based on our switch architecture and design. For this purpose, a switch port is treated as part of the physical channel it is attached to, since the functionality of the port is more closely coupled with the physical channel than with the internal switch.

Based on these two relationships, for a given value of R, T_S, T_L, and T_B can be specified in terms of T_C. Therefore, to characterize the set of MTBFs used in a simulation, only the value of R is specified.

Starting with a non-faulty array, the logical array used by the program is mapped onto the physical array after each fault until a mapping can not be found. The mapping of the logical array onto the physical array is done by a mapping program. The mapping program supports a general graph model for both the logical and the physical arrays and uses a graph to graph mapping algorithm. Therefore, programs using various logical array structures can be mapped onto the same physical array. Since it is assumed that all components of the physical array can fail, the physical array is also modeled as a graph. Although the logical and physical arrays need not be rectangular, the arrays used in the evaluations described in this paper are restricted to square arrays for simplicity and uniformity. The mapping algorithm can use several evaluation criteria to determine the quality of a mapping, such as maximum load on a physical channel, longest path, or total path length, which are referred to as evaluation metrics. In the evaluations presented in this paper, the primary objective was to find mappings which would minimize the maximum number of virtual channels used per physical channel. In the next section, we present the mapping algorithm; a detailed discussion of the mapping algorithm will be the subject of another paper.

Given a mapping of the logical array onto the physical array, the execution of the program on the physical array for that mapping is simulated using a performance simulator. Performance degradation of the program over the lifetime of a physical array is determined by a sequence of performance simulations corresponding to the sequence of mappings used during the lifetime of the physical array.

3.2. Mapping algorithm

For simplicity of explanation, assume that the logical array and the physical array consist of only cells and there are no I/O buffers. Let m and n be the number of cells in the logical array and the physical array respectively, where $n \geq m$. Let l_i represent a node in the logical array graph, and p_i represent a node in the physical array graph, in general.

The logical nodes are placed onto the physical nodes using the following ordering. The logical node with the greatest number of descendants in the logical array graph is selected as the starting node. Let the starting node be l_1. Once l_1 is selected, the order to place the remaining logical nodes is determined by a breadth-first traversal from l_1: all the neighbors of l_1 are placed first, followed by their neighbors, and so on. Assume that this ordering corresponds to $l_1, l_2, \ldots, l_m$.

All possible placements of the logical nodes onto the physical nodes form a *placement tree*, which represents the search space of the mapping algorithm. The starting logical node l_1 can be placed onto any of the n physical nodes $p_1, p_2, \ldots, p_n$. For each placement of l_1, l_2 can in turn be placed onto any of the remaining $n-1$ physical nodes, and so forth. The total number of leaf nodes in the placement tree is $n!/(n-m)!$. Each path from the root of the placement tree to one leaf node corresponds to one possible placement of the logical array onto the physical array. The placement tree is traversed in the following order to search for mappings.

For each logical node l_i, the physical nodes are ordered starting with the most promising candidates for the placement of l_i, referred to as the *candidate list* of l_i and denoted by C_i. The ordering of candidates is based on the similarity of interconnection properties between l_i and each of the physical nodes p_i. The algorithm used for forming the candidate lists is described later in the section. The ordering of candidate lists is dynamically updated as the mapping proceeds. When the logical node l_i is to be placed, the most promising candidate physical node in C_i is tried first, then the next, and so on.

Let l_i be a logical node currently being placed on physical node p_r, and let l_j be a logical node connected to l_i. Once l_i is placed on physical node p_r, the following actions are taken.

- If l_j has already been placed, say, on p_s, then the shortest routing path between p_r and p_s is assigned as the connection between l_i and l_j. If a path can not be found, then a mapping is not possible, and that search branch of the placement tree is abandoned. Moreover, the evaluation metric is computed and used to decide if search should continue in that search branch further. For example, if the current evaluation metric for that branch becomes worse than a previously found mapping, a better mapping can not be found by searching that branch further, and that branch should be discarded. Efficient pruning of non-promising branches is thus achieved.
- If l_j has not been placed, then its candidate list C_j is reordered as follows: since l_j is a neighbor of l_i which has been placed on p_r, the physical nodes closest to p_r become the most promising candidates for l_j.

Due to the immensity of the placement tree for even moderate size graphs, a traversal of the entire tree is impossible for practical reasons. How far the placement tree is searched for a mapping depends largely on the desired quality of the mapping. Several stopping criteria have been incorporated into the mapping program for terminating a search.

They include stopping after the first mapping is found, after a mapping satisfying a quality constraint is found, or after a specified number of branches have been searched.

3.2.1. Candidate list algorithm

Let v be a node in a graph, and let $D_{v,j}$ represent the set of nodes in the graph at a distance j from v. Let:

$n_{v,j}$: number of nodes in $D_{v,j}$
$e_{v,j}$: number of edges between $D_{v,j}$ and $D_{v,j-1}$
$d_{v,j}$: difference $e_{v,j} - n_{v,j}$

where j varies from 1 to k, k being the distance of the farthest node from v.

The *connectivity matrix* M_v for node v is defined as:

$$M_v = [m_{i,j}]$$

where:

$$m_{1,j} = e_{v,j}$$
$$m_{2,j} = n_{v,j}$$
$$m_{3,j} = d_{v,j}$$

for $j = 1, \ldots, k$

Let L_i and P_i, respectively, refer to the connectivity matrix of a node in the logical array graph, and a node in the physical array graph. Consider a logical node u of the logical array with connectivity matrix $L_u = [l_{i,j}]$, $i = 1, 2, 3$ and $j = 1, \ldots, k$. The "goodness" of a physical node v with connectivity matrix $P_v = [p_{i,j}]$, is measured by the number of times the following condition is satisfied.

$$l_{i,j} \leq p_{i,j} \text{ for } i = 1, 2, 3 \text{ and } j = 1, \ldots, k$$

The candidate list C_u of a logical node u is formed by computing the goodness of each physical node for u, and ordering the physical nodes in a decreasing order of their goodness. Since the physical nodes of the best mapping appear mostly at the beginning of the candidate lists, during the search for a mapping, most promising branches of the placement tree are searched first. Therefore, good mappings are obtained although a very small portion of the placement tree is searched.

3.3. Sample programs

The programs used in this evaluation were image processing programs developed for a vision research library at Carnegie Mellon [8].

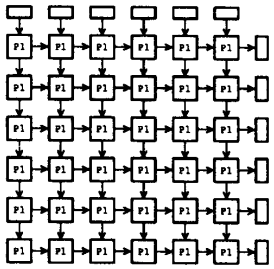


Figure 3-1: Representative 6x6 logical array

Thirty-four programs were selected from this library which vary in complexity from simple thresholding to sophisticated edge detection operators. One of the edge detection programs using the Frei and Chen operator (EGFC) is used as a specific example in this paper. The programs were written in an architecture-independent image processing language called Apply [3] which can be compiled for 1D and 2D Warp arrays of various sizes. For 2D Warp arrays, the programs use the mesh-connected logical array structure shown in Figure 3-1, where the inputs flow from top to bottom and outputs flow from left to right. The dimensions of the logical array can be varied at compile time.

4. Results

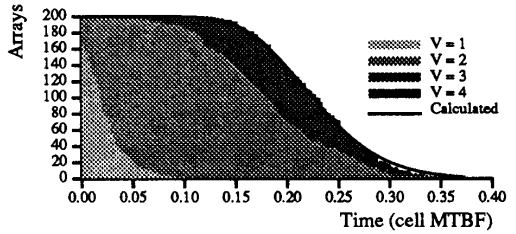
4.1. Array survivability

This section presents simulation results on how two critical parameters of the architecture, number of virtual channels available and switch/cell reliability ratio (R), affect the array survivability.

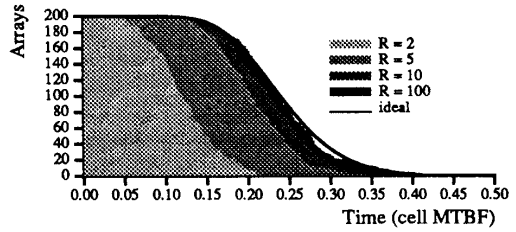
Variation with number of virtual channels available. A physical array survives as long as a mapping can be found to implement the logical array of the program on the physical array. Therefore, in addition to the condition of the physical array, whether a physical array survives or not depends on the success of the mapping program in finding a mapping. One premise for using virtual channels was that finding a mapping would be easy and successful in most cases provided that there are sufficiently large number of channels between switches. Let V represent the number of virtual channels available in each direction in a physical channel. Figure 4-1-a shows how the success of mapping an 8x8 logical array onto a 9x9 physical array depends on V. For these simulations it is assumed that the switch is 10 times more reliable than the cell ($R = 10$). Using the same sequence of faults in each case, the graph shows the number of arrays survived vs. time in 200 lifetime simulations for $V = 1, 2, 3$ and 4. As can be seen, the success of mapping is very low for V=1, which means one virtual channel in each direction per physical channel, and improves significantly for V=2. As can be expected, very small number of virtual channels makes finding a mapping difficult. Figure 4-1-a shows that V=3 provides sufficient number of virtual channels for successful mapping. The curve for $V = 3$ comes close to the analytically calculated curve where it is assumed that mapping is always successful as long as there are sufficient functional components left in the array, i. e. routing is never a problem. Although not clearly observable in Figure 4-1-a, $V = 4$ provides only a slight improvement over $V = 3$, and it was found that $V > 4$ provides no further improvement. An important observation from Figure 4-1-a is that a moderate size array such as 9x9 requires a relatively small switch supporting 3 or 4 incoming virtual channels per switch port. Since there is a dedicated queue in the switch for each incoming virtual channel, the hardware complexity of the switch is highly dependent on V.

Variation with switch/cell reliability ratio. A serial reliability relationship exists between a cell and the switch it is attached to. The ratio of switch MTBF over cell MTBF was defined as R. If R is large, then the combined reliability

of a cell-switch pair is not significantly worse than that of the cell. If R is small however, then the benefit of using a complex switch may be questionable since the reliability of a cell-switch pair would be significantly less than the cell alone. Moreover, a higher percentage of switch failures requires a larger number of virtual channels per physical channel for successful mapping, since routing becomes more difficult. If the virtual channels supported by the switch is not sufficiently large, this has a more detrimental effect on array survivability for small R than for large R. Figure 4-1-b shows the results of 200 lifetime simulations for mapping the 8×8 logical array onto the 9×9 physical array for several values of R where $V=4$. It can be seen from the figure that the curve for R=10 is close to the curve for R=100, i.e., having a much more reliable switch than the cell does not improve the array survivability significantly. The mean array lifetime for R=100 is 0.241, and for R=10 it is 0.228 (in terms of cell MTBF). Note that the curve for R=10 also comes reasonably close to the curve representing the ideal case where interconnections never fail ($R=\infty$). The mean array lifetime for R=10 is 8.2% less than what it would be for the ideal case. Therefore, it is probably sufficient to have a switch that is approximately 10 times more reliable than the cell, if the switch supports sufficient number of virtual channels. In Figure 4-1-a, it was shown that a switch supporting 3 virtual channels in each direction (V=3) was sufficient for mapping of the 8×8 array onto the 9×9 array for R=10. According to Figure 4-1-b, a ratio of $R=5$ may also be acceptable.



a) Variation with V ($R=10$)



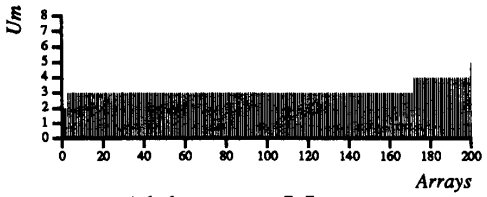
b) Variation with R ($V=4$)

Figure 4-1: Number of arrays survived vs. time in mapping an 8×8 array onto a 9×9 array for various values of V and R

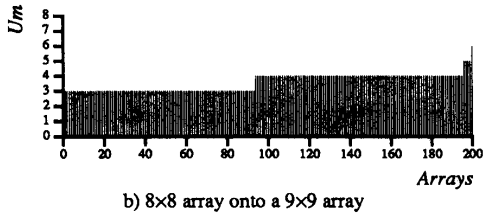
4.2. Number of virtual channels used

This section presents some results on the maximum number of virtual channels used in a physical channel for mapping, and how this varies with array size, amount of redundancy, and switch/cell reliability ratio.

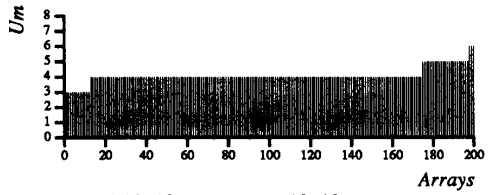
Variation with array size. Let U represent the *maximum number of virtual channels used in a physical channel* in a mapping. Note that $U \leq 2V$, since a total of $2V$ virtual channels are available per physical channel. Also, let U_m represent the maximum value of U in all mappings used during the lifetime of a physical array. Figure 4-2 depicts how U_m varies with the array size. Figure 4-2 consists of four graphs showing the values of U_m in mapping 6×6, 8×8, 12×12, and 16×16 logical arrays respectively onto 7×7, 9×9, 13×13, and 17×17 physical arrays (R=10 for all cases).



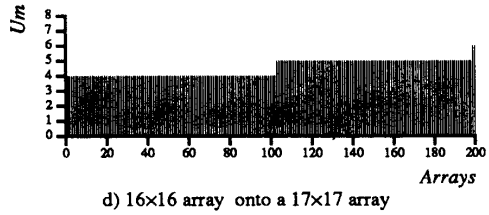
a) 6×6 array onto a 7×7 array



b) 8×8 array onto a 9×9 array



c) 12×12 array onto a 13×13 array



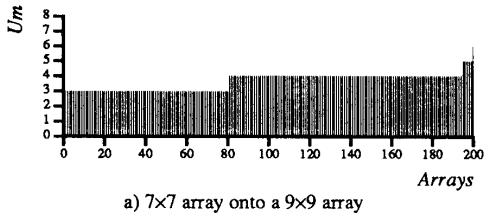
d) 16×16 array onto a 17×17 array

Figure 4-2: Maximum number of virtual channels used per physical channel in array lifetime in mapping different size 2D-mesh arrays (R=10)

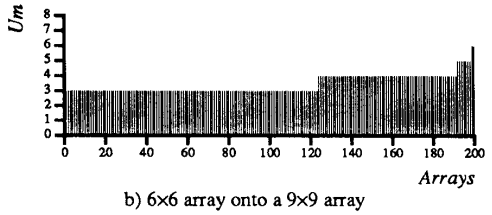
As can be seen, on the average more virtual channels are used per physical channel as the array size increases. For the 7×7 array, 99% of array lifetimes use up to 4 virtual channels per physical channel. The percentage drops to 97%, 86%, and 51% respectively for the 9×9, 13×13, and 17×17 arrays. However, the highest values of U_m do not vary significantly for different size arrays; both the 9×9 array and the 17×17 array use up to 6 virtual channels. Both the number of cells in the logical array to be mapped and the number of faults to be tolerated increase with array size. On the other hand, the total

number of channels available for routing also increases with array size. This may explain why U_m does not increase significantly as the physical array size is increased from 7×7 to 17×17. This is a significant result, since it shows that a switch supporting a modest number of virtual channels can be used for implementing $n \times n$ logical arrays on $(n+1)\times(n+1)$ physical arrays for various array sizes up to $n = 16$.

Variation with redundancy. Another factor to investigate is whether the number of virtual channels needed varies with the amount of redundancy in a physical array. Figure 4-3 shows two graphs depicting the values of U_m in 200 lifetime simulations of a 7×7 logical array and a 6×6 logical array mapped onto a 9×9 physical array. The graph for the 8×8 logical array mapped onto a 9×9 physical array is already shown in Figure 4-2-b. The amount of redundancy relative to the logical array in implementing 8×8, 7×7, and 6×6 logical arrays on a 9×9 physical array is respectively 26%, 65%, and 125%. However, as can be seen from the figures, the increase in redundancy does not have a significant impact on the value of U_m. This is not an obvious result since there are two opposing factors which affect the difficulty of mapping. A smaller logical array means fewer nodes have to be mapped thus mapping would be easier. On the other hand, a smaller logical array also means there will be more faults in the physical array which makes mapping more difficult. The observation that U_m does not change significantly with the amount of redundancy may be explained as the cancellation of these opposing effects.



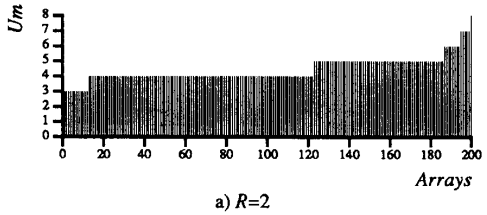
a) 7×7 array onto a 9×9 array



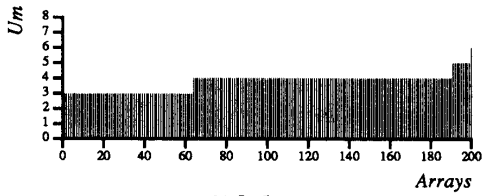
b) 6×6 array onto a 9×9 array

Figure 4-3: Maximum number of virtual channels used per physical channel in 200 array lifetimes, mapping 7×7 and 6×6 arrays onto a 9×9 array (R=10)

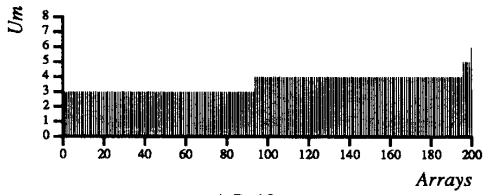
Variation with switch/cell reliability ratio. The switch/cell reliability ratio (R) also affects the number of virtual channels used in a physical channel. Figure 4-4 shows four graphs depicting how U_m varies for several values of R, for mapping an 8×8 logical array onto a 9×9 physical array. As expected, a higher ratio of faulty switches (smaller R) requires the use of more virtual channels per physical channel. It can be seen that for the ideal case, where switches and channels are assumed not to fail (R=∞), up to 4 virtual channels are used. For R=10, although 5 or 6 virtual channels are used in several array lifetimes, up to 4 virtual channels are used in 97% of the array lifetimes. The results are similar for R=5, in which case up to 4 virtual channels are used in 95% of the array lifetimes. This shows that in mapping an 8×8 logical array onto a 9×9 physical array, up to 4 virtual channels will be used in general, if the switch is moderately more reliable than the cell (5 times or more). Note that for R=2, the percentage of array lifetimes using up to 4 virtual channels drops to 61%.



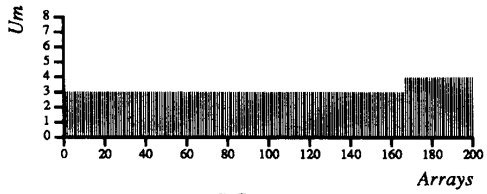
a) R=2



b) R=5



c) R=10



d) R=∞

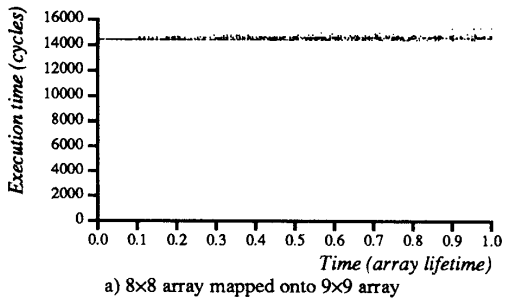
Figure 4-4: Maximum number of virtual channels used per physical channel in 200 array lifetimes, mapping a 8×8 array onto a 9×9 array for various R

4.3. Performance degradation

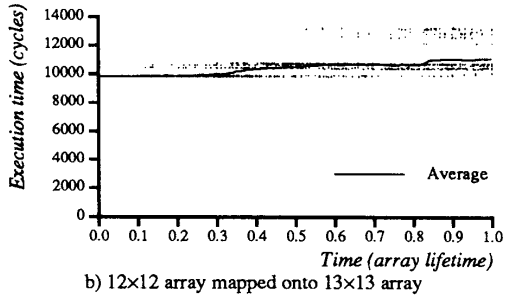
Implementing multiple logical connections on a physical channel and routing data across switches have the potential to cause communication bottlenecks and lead to a degradation in the performance of programs as the physical array deteriorates. However, it was observed that there is little or no performance degradation for the sample programs running on small and medium size arrays ($\leq$ 13×13). This is primarily due to the fact that the I/O to computation ratio of most programs for arrays of this size is sufficiently low so

that the modest degree of multiplexing in physical channels (4-5) does not cause communication bottlenecks. Moreover, the dynamic scheduling mechanism used for routing data across the switch prevents the switch from becoming a bottleneck on its own.

Figure 4-5-a shows an accumulated plot of execution times for EGFC (an edge detection program) using an 8×8 logical array in 25 lifetimes of a 9×9 physical array. The horizontal axis represents time which is normalized to the lifetime of the array so that all lifetime simulations map to the same range. For each mapping in the lifetime of an array, a dot is placed at the beginning and at the end of the time during which the mapping is in effect, corresponding to the execution time of the program for that mapping. As can be seen, there is no significant performance degradation for EGFC in the 25 lifetimes of a 9×9 physical array.



a) 8×8 array mapped onto 9×9 array



b) 12×12 array mapped onto 13×13 array

Figure 4-5: Execution times for EGFC in 25 lifetimes of a 9×9 and a 13×13 array

Let the *load* of a logical connection be defined as the number of data words transferred over the logical connection divided by the execution time of the program on a perfect logical array. In general, the load in logical connections and the amount of multiplexing in physical channels determine the performance degradation of a program. If the amount of data to be transferred across a physical channel exceeds the bandwidth of the channel, then a degradation is expected in the throughput of the program. For example, EGFC has only two types of logical connections, input and output, with respective loads of 0.20 and 0.18 for an 8×8 logical array. Since the maximum number of virtual channels used per physical channel (U) does not exceed 5 in any of the mappings used in 25 array lifetimes, there is no significant performance degradation.

As an example of the case where performance degradation occurs, consider the graph in Figure 4-5-b, which shows the performance of EGFC, in this case using a 12×12 logical array, in 25 lifetimes of a 13×13 physical array. The input and output loads of EGFC for a 12×12 array are 0.27 and 0.23. Although U does not exceed 5 in these mappings, because of the increased load in logical connections, some performance degradation is observed, especially toward the ends of the array lifetimes as U increases with increased difficulty in mapping. A curve representing the average execution time was plotted to give an idea of the distribution of dots and the expected performance.

Figure 4-6 summarizes the estimates for performance degradation of the sample Apply programs for $n \times n$ logical arrays implemented on $(n+1)\times(n+1)$ physical arrays for $n = 6, 8, 12$. Let $D = T_e/T_b$ represent the potential performance degradation of a program, where T_e is the execution time of the program at the end of the array lifetime and T_b is the execution time of the program at the beginning of the array lifetime. The figure shows the expected values of D for 34 Apply programs. The expected value of D for a program is calculated by averaging the values of T_e for 200 lifetime simulations. Since T_e normally corresponds to the worst performance of the program during the array lifetime, the values of D shown in Figure 4-6 can be considered as the average of the worst-case performances.

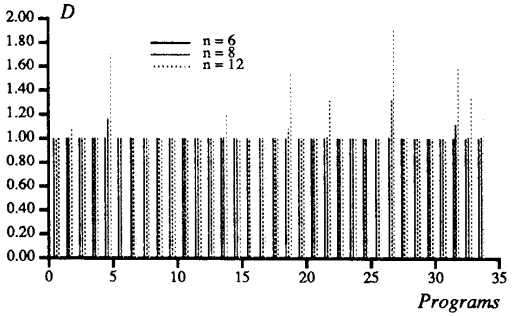


Figure 4-6: Expected lifetime performance degradation of 34 Apply programs for three different size arrays

Figure 4-6 shows that $D = 1.0$, i. e. there is no performance degradation, for most of the sample programs and less than 2.0 for all the programs for $n \leq 12$. However, even those programs which show some performance degradation at the end of array lifetime are likely to show little or no performance degradation for a significant portion of the array lifetime. The loads in logical connections of the majority of Apply programs are sufficiently low such that there would be little degradation in the performance of these programs for mappings with $U \leq 3$, which normally constitute more than 50% of all mappings used during an array lifetime. In general, if the problem size does not change, mapping a program to a larger logical array increases the loads in the logical connections of the program. This is because the amount of computation per cell decreases proportionally to the number of cells in the array, while the I/O per cell decreases proportionally to the array dimensions. Therefore,

there is a higher potential for performance degradation if a program uses a larger array.

5. Summary and conclusion

We have proposed a highly configurable array architecture for arrays of powerful processors. The architecture uses a 2D switch network where physical connections are multiplexed to implement multiple logical connections using the virtual channels mechanism. To evaluate the architecture we have developed simulation tools and an efficient and general mapping program. The architecture was evaluated assuming an array of Warp cells and using a number of image processing programs developed for the Warp array.

Based on our evaluation, it can be said that the architecture provides a high degree of configurability and allows efficient utilization of redundant processors in the array. To implement a near-optimal architecture for a moderate size array, such as 9×9, the number of virtual channels supported by a physical channel in each direction can be as small as 3 or 4, and as a result, the switch can be fairly small and reliable. However, the switch does not need to be extremely reliable compared to the cell, and a switch/cell reliability ratio of as low as 5 can be acceptable. Moreover, the number of virtual channels used per physical channel to ensure a successful mapping does not seem to vary significantly with array size or the amount of redundancy. Therefore, the switch can be used as a building block to construct various size configurable arrays with as many redundant processors as may be needed for the application.

The potential for performance degradation of a program increases with the array size as the I/O to computation ratio of a program increases when mapped to larger arrays. However, for small and medium size arrays, the modest degree of multiplexing in physical connections does not pose a problem for the performance of most programs, since the I/O to computation ratios of most programs are sufficiently low. Little or no performance degradation was observed for a number of image processing programs running on physical arrays up to 13×13. Therefore, the performance implications of the architecture appears to be acceptable for programs using small and medium size logical arrays, such as 8×8 or10×10, which are commonly considered for arrays of powerful processors.

It is becoming relatively easy to implement powerful processors and switching elements due to the rapidly increasing logic densities provided by the VLSI technology. However, the capacities of off-chip and off-board interconnections do not increase proportionally. Therefore, it may not be feasible to implement many redundant physical connections to provide a high degree of configurability in 2D processor arrays where a processor is implemented by one or more chips. The approach taken in this architecture was to use a moderately complex switching element to implement multiple logical connections by multiplexing a physical connection. Our evaluation of the architecture suggests that this approach would be effective in building 2D configurable arrays of powerful processors in general, if the processors have large local memory and do not require tightly synchronized communications. A modest degree of multiplexing in physical connections is sufficient to provide a high degree of configurability, and this would not pose a significant problem for program performance with highly powerful processors.

References

1. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O. and Webb, J. A. "The Warp Computer: Architecture, Implementation and Performance". *IEEE Transactions on Computers C-36*, 12 (December 1987), 1523-1538.

2. Gross, T. and Lam, M. Compilation for a High-performance Systolic Array. Proceedings of the SIGPLAN 86 Symposium on Compiler Construction, ACM SIGPLAN, June, 1986, pp. 27-38.

3. Hamey, L. G. C., Webb, J. A., and Wu, I. C. Low-level Vision on Warp and the Apply Programming Model. In *Parallel Computation and Computers for Artificial Intelligence*, Kluwer Academic Publishers, 1987, pp. 185-199. Edited by J. Kowalik.

4. 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.

5. Kung, H.T. and Menzilcioglu, O. A General Switch Architecture for Fault-Tolerant VLSI Processor Arrays. Proceedings of SPIE, August, 1987, pp. 37-44.

6. Kung, H.T and Menzilcioglu, O. Virtual Channels for Fault-Tolerant Programmable Two-dimensional Processor Arrays. Tech. Rept. CMU-CS-87-171, Carnegie Mellon University, December, 1986.

7. Negrini, R., Sami, M.G., Stefanelli, R. Fault Tolerance Approaches for VLSI/WSI Arrays. Proc. IEEE Phoenix Conf. on Comp. and Comm., 1985, pp. 460-468.

8. Ribas, H. and Webb, J. User's Guide to WEB. Department of Computer Science, Carnegie Mellon University.

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

10. Singh, A.D. An Area Efficient Redundancy Scheme for Wafer Scale Processor Arrays. Proceedings of Intl. Conf. on Computer Design, October, 1985, pp. 505-509.

11. Snyder, L. "Introduction to the Configurable, Highly Parallel Computer". *Computer 15*, 1 (January 1982), 47-56.