Proteus: a reconfigurable computational network for computer vision

Robert M. Haralick, Arun K. Somani, Craig Wittenbrink,
Robert Johnson, Kenneth Cooper, Linda G. Shapiro,
Ihsin T. Phillips, Jenq-Neng Hwang, William Cheung,
Yung Hsi Yao,Chung-Ho Chen, Larry Yang,
Brian Daugherty, Bob Lorbeski, Kent Loving,
Tom Miller, Larye Parkins, and Steve Soos

Department of Electrical Engineering, FT-10
Applied Physics Laboratory, HN-10
University of Washington
Seattle, Washington 98195

ABSTRACT

The Proteus architecture is a highly parallel MIMD, multiple instruction, multiple-data machine, optimized for large granularity tasks such as machine vision and image processing. The system can achieve 20 Giga-flops (80 Giga-flops peak). It accepts data via multiple serial links at a rate of up to 640 megabytes/second. The system employs a hierarchical reconfigurable interconnection network with the highest level being a circuit switched *Enhanced Hypercube* serial interconnection network for internal data transfers. The system is designed to use 256 to 1,024 RISC processors. The processors use one megabyte external *Read/Write Allocating Caches* for reduced multiprocessor contention. The system detects, locates, and replaces faulty subsystems using redundant hardware to facilitate *fault tolerance.* The parallelism is directly controllable through an advanced software system for partitioning, scheduling, and development. System software includes a translator for the INSIGHT language, a parallel debugger, low and high level simulators, and a message passing system for all control needs. Image processing application software includes a variety of point operators, neighborhood operators, convolution, and the mathematical morphology operations of binary and gray scale dilation, erosion, opening, and closing.

1. INTRODUCTION

There are a variety of parallel architectures that can be used for image processing applications. To cut short the review of the variety of these architecture types, we make some political parallels which have been informally made by some noted researchers. The array can be likened to a Fascist dictator leading a march. The pipeline can be likened to a capitalist assembly line. The pyramid can be likened to the cell block hierarchy of Communist dictatorships. The multiprocessor systems can be likened to parliamentary committees at work. The network can be likened to political anarchy. And the reconfigurable network just cannot get its act together. Particular examples of these architectures include processor arrays such as Illiac IV, CLIP4, CLIP7, DAP, MPP, and GAPP, systolic arrays such as WARP, pipelines such as Cytocomputer, Genesis, and VAP, multiprocessor systems such as PASM, POLYP, ZMOB, GOP, PICAP, TOSPICS, DIP, FLIP, PM4, and pyramid systems. Scholarly reviews of these architectures can be found in the papers listed in the reference section.[6]

The purpose of this paper is to discuss an architecture by which the act of the reconfigurable network can be put together for computer vision in a large grain parallelism mode. We do this in the context of knowing that no use of reconfigurable networks for image processing has been made and that there has been little discussion of architectures which are simultaneously suitable for low level image processing, high level computer vision, and the computation tasks required to direct robots and material handlers. All this must be done in near real time in the factory setting.

Our viewpoint will be different from the usual discussions of computer architecture which tend to concentrate around hardware design issues in a hardware language. We believe that all paradigms which map image data onto processors like the arrays and pyramids or paradigms which map specialized tasks onto different processors like the multiprocessor systems, must of necessity create specialized and inflexible systems. We believe that the watch-

0-8194-0813-1/92/$4.00

word of computer vision is flexibility. There must be the *image in and image out* operations of low level vision. There must be the *image in and data structures out* operations of mid level vision. There must be the *data structures in and data structures out* operations of high level vision. And in any computer vision system whose purpose is to be economically useful in the factories of the society in which it functions, there must be the capability for performing numerical calculations, data formatting operations, communication reporting operations, and real time control of external devices such as material handlers and robots. This suggests that the approach needs to be an integrated one. To approach the design, we must step back and understand that the low level neighborhood operators discussed in today's archival literature can be much more complex than the Roberts and Sobel variety. We must understand that the manipulation and processing of the high level vision data structures may be as complex as the symbolic processing required by artificial intelligence computation.

Within our universe, where can we take a stand so that our viewpoint can unravel the inherent complexity of this computer vision question? The required flexibility suggests that the architecture should be able to naturally execute algorithms of a general nature. The quantity of data processed in a computer vision system suggests that it seeks to have a higher input data rate and that the architecture must be in some sense optimized to spend a substantial amount of its processing time doing uniform pixel pushing operations. If it can have high efficiency in performing a regular pattern of operations on a large data set, it can afford to have a lower efficiency in performing less regular operations on small data sets. Or said in another way, the architecture must spend its time performing a variety of activities. If it can configure itself so that it has high efficiency for the most computationally intensive activities, it can afford the overhead required to reconfigure itself to perform the less computationally intensive and more irregular activities. High efficiency for computationally intensive activities suggests the algorithm driven systolic network. Flexibility suggests reconfigurability. The combination of the two suggests a data flow architecture, a reconfigurable network capable of systolic or non-systolic computation.

However, the kinds of data units that are processed in computer vision change as the processes proceed from low to high level. Pipelines and systolic architectures which are optimized for the pixel data unit will not be efficient for mid-level data such as the digital arc, or high-level data units such as a set of corresponding model image feature points. This suggests that instead of thinking that the architecture processes small data units such as a pixel, we can visualize an architecture which processes more complex data units such as the image, digital arc, sets, and relations. Here the idea of systolic computation must dissolve for the units are too large.

In factory applications of machine vision, the same vision algorithm is applied repeatedly to a succession of images. The input is not an image, but a sequence of images. The output to each image processing operation likewise is a sequence of images. The clock tick of the systolic array or video rate pipeline gives way to the time chunk taken to process an entire image. The processor, instead of only processing one simple operation such as an add or multiply on the primitive data unit now must perform an arbitrarily complex sequence of operations on the large data unit. The code run on the processor now does not have to be the kinds of specialized code used for vector processor, pipelines and systolic arrays, or digital signal processors. Rather the code can be the same kind of code written in languages such as C or Ada and which can be tested on standard workstations.

For high input data rate and simple algorithms, such an architecture runs in a single program multiple data stream mode. For low input data rate and highly complex algorithms, such an architecture can reconfigure itself to function in a pipeline network or full multiple instruction multiple data stream mode. We call this architecture the reconfigurable Proteus architecture.

To understand what must go into the Proteus data flow architecture, we must have a language in which to discuss its configuration possibilities. Hardware programming languages like VHDL or N2 and graph description languages are at too low a level. The interesting thing about the data flow in a network is that a high level specification of the configuration of the network is a specification of the program the network is executing. This is different from von Neumann architectures in which a specification of the architecture tells nothing about what program may be executing on the hardware. Now low level specification of a network, or more formally, a graph having labeled arcs and nodes, has nothing about it which is sequential or procedural. Likewise, a high level specification need not be sequential or procedural. A high level specification of a network is just a specification of the relations which hold in the network. So specification of the configuration of a network amounts to specifying relations, and since the specification is the program which the network executes, the language used to program a data flow network is naturally a language of relations. The language must be inherently non-procedural. From a high level perspective, *the semantics of the language specifying a computational network describes the essence of the architecture.*

In section II of the paper we describe what we mean by a reconfigurable computational network and its underlying distributed control mechanism. In section III, we describe the language INSIGHT, a language in the LUCID family of data flow languages (Wadge and Ashcroft, 1985) which we have developed to be used both for specification of data flow architecture configurations, and for the high-level expression and coding of our computer vision algorithms. In section IV we describe the architecture from a hardware point of view.

2. RECONFIGURABLE COMPUTATIONAL NETWORK

In this section, we give a perspective of the reconfigurable computational network which emphasizes those aspects of the computation that the network must handle and which a programmer using the network does not have to think about. Such a perspective illuminates the division of the hardware domain from the software domain. It provides the hardware-software interface conventions by which the semantics of the language INSIGHT can describe the essence of the architecture of the reconfigurable computational network.

The operation of a reconfigurable computational network involves the flow of sequences of high-level data units through a network of architectural primitives. Architectural primitives are of two types: processors and connections. Processors have one or more inputs and one or more outputs. They produce high-level data units of the same or different kind than the input data units for output lines after some finite execution time. The amount of time taken to execute may be proportional to data unit size or it may be worse than linear time as it might be for a search algorithm.

More formally, a network *configuration* consists of a set of *processors* P and a specification C of the interconnections between the processors. Each processor $p \in P$ is a pair $p=(I_p, O_p)$ where I_p is a named set of input lines and O_p is a named set of output lines. Each connection $c \in C$ is a quadruple $c=(o, p_1, i, p_2)$ specifying that output line o of processor p_1 connects to input line i of processor p_2. A top level view is shown in Figure 1.

Since different processors may take different amounts of time to process their input data structures and produce their output data structures, the control insures that a processor will only begin processing its input data when its input buffer contains valid data. For this purpose, there is a state s associated with each buffer. Legal values for the state s are:

1) ready and unconsumed: the buffer has valid new data but not all processors which require it have accepted it;

2) ready and consumed: the buffer has valid data which has already been used by the processors that require it and the new data which is to be loaded into the buffer is not yet ready; and

3) not ready: the buffer has no valid data in it.

A process can execute when all of its input buffers are in the ready and unconsumed state and the output buffer it has been assigned has had its data consumed by every process for which it is an input.

The execution of the process takes some finite amount of time. When the execution is just starting, each of the previous output buffers are in the state ready and unconsumed. As soon as an output buffer reaches this state, it is available to the processes that wish to consume it. It reaches the state ready and is consumed only when all of its potential consumers have consumed it.

3. INSIGHT

Figure 2 illustrates a top level view showing how the software relates to the Proteus hardware. whose proces-

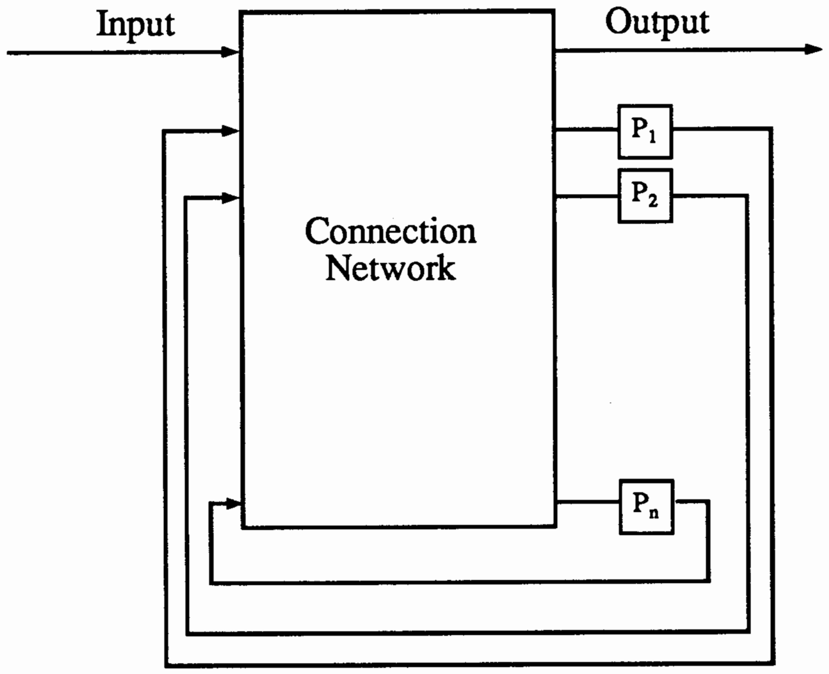


Figure 1 shows a top level view of the reconfigurable computational network.

sors are partitioned into groups and within groups partitioned into clusters. High-level programs for Proteus are written in the INSIGHT programming language. The INSIGHT program describes the flow of a sequence of images, and other data structures, and their resultant data structures through the Proteus network. Each process of the network performs one or more operations on its input image(s) and/or structure(s) to produce output image(s) and/or other data structure(s).

The most important aspect of the INSIGHT language is that it expresses relationships, not commands. The order in which the relationships are stored in the program has no effect on the results. Instead the relationships dictate a graph structure that defines the flow of data through the system. Figure 3 illustrates the graph structure for a program given in Figure 4. This graph must be mapped onto the Proteus hardware.

The input to the INSIGHT program shown in Figure 4 is a 256x256 gray scale image G0, and the output is a 256x256 binary image B4. Intermediate gray scale images G1, G2, and G3 and intermediate binary images B1, B2, and B3 are also produced during execution of the program. The first relation says that gray scale image G0 is to be thresholded using threshold T1 (a constant), and the result is to become binary image B1. The second relation says that G0 is also to be the input to a morphological closing operation [11] with a structuring element that is a box (rectangle) of dimension 5 x 5, with the result becoming gray scale image G1. The third relation specifies the production of another binary image B2 that is the result of performing an opening in G1, subtracting the opening from G1 itself and thresholding the result of the subtraction. The other relations can be analyzed in a similar fashion.

The INSIGHT translator maps the algorithm onto the hardware. The INSIGHT translator has two main parts: the scanner/parser module and the linker/partitioner module. The scanner/parser module uses standard translation techniques. It employs a finite machine for lexical analysis and a recursive descent parsing mechanism with look ahead by one, augmented by a precedence parser for expression. The output of the scanner/parser module goes to the linker which replaces single nodes of the graph representing INSIGHT library routines by prestored subgraphs that came from previous translations. Also, nodes representing morphological operations which use possibly complex structuring elements may be decomposed into sequences of nodes that use smaller structuring

Software View of the System

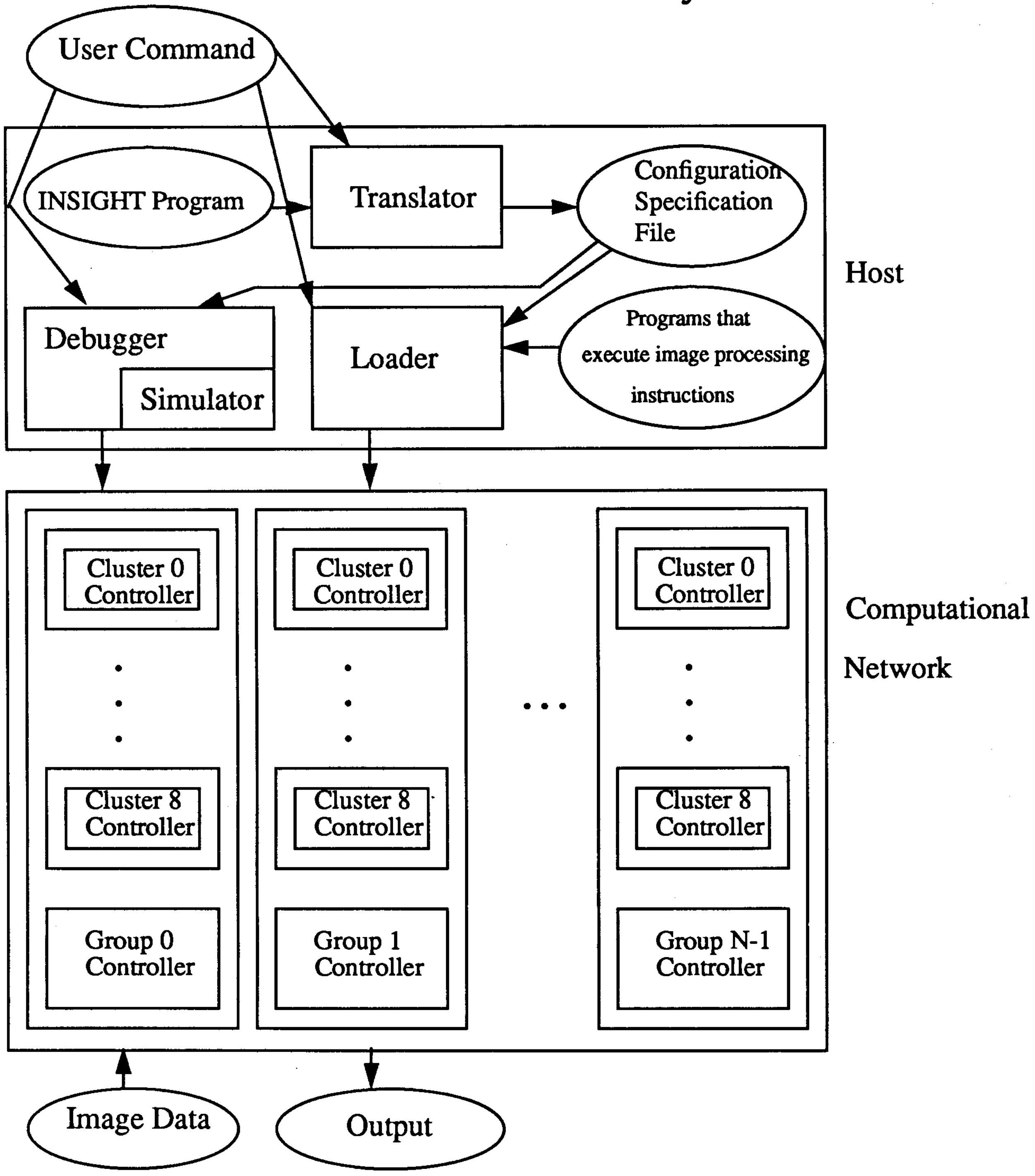


Figure 2. Software View

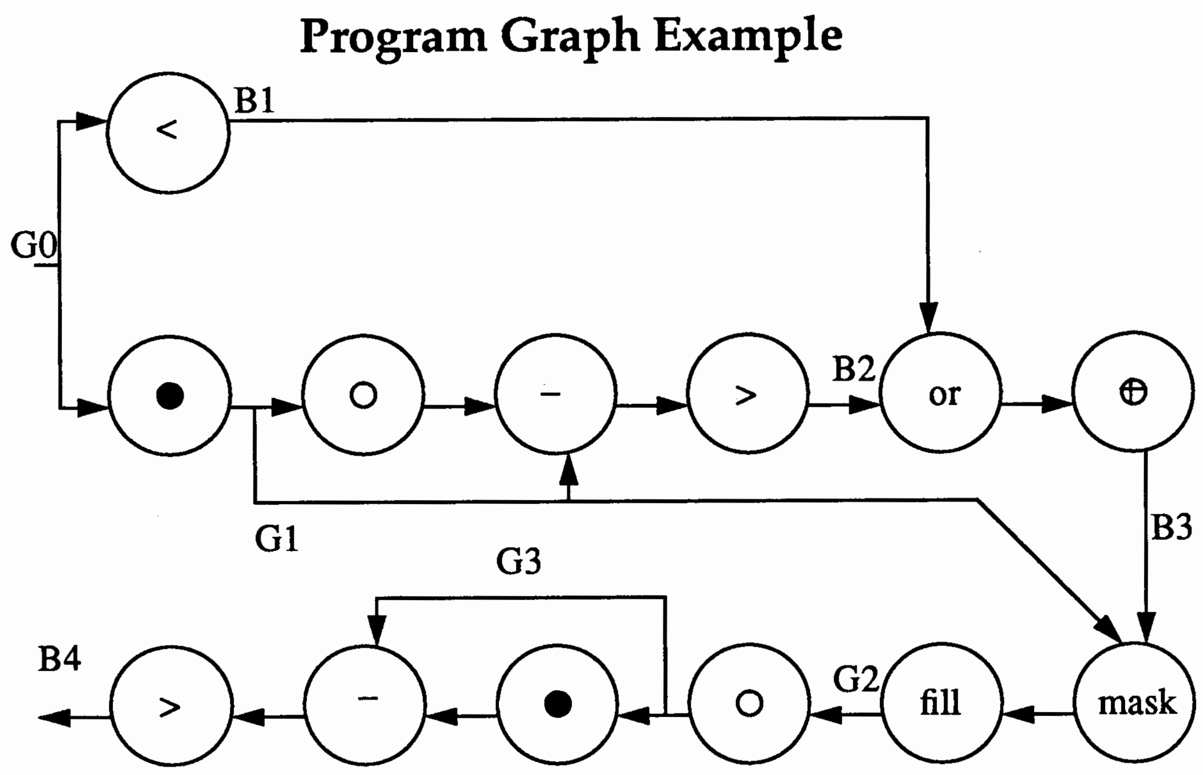


Figure 3. INSIGHT program graph generated automatically by the INSIGHT translator

elements [27]. This decomposition is beyond the scope of this paper. The partitioner is the only nonstandard part of the translator. Its job is to map the operations in the final dataflow graph onto the reconfigurable network. The goal is to produce the mapping with the highest throughput, so that as much data as possible can be handled by the reconfigurable network.

The problem of the partitioner can be stated as follows. Given a dataflow graph with K nodes with an estimation of the amount of processing time each takes, and a multiprocessor shared memory system with N_{proc} processing elements, with a specified interconnection network and interprocessor communication costs, determine how the operations be partitioned among the processors to gain maximum throughput. Initially we chose a greedy technique as in [1].

To control the load balancing, each processor has all of the nodes it will process assigned to it. The algorithm keeps a list of all nodes that have had all of their ancestors allocated. This is called the ready list. A heuristic is generated for each of the nodes in the ready list at each step estimating the cost of assigning that node to the current processor. The heuristic is based on the expected computation time of the node, the load so far on the processing block, and the communication required by assigning this node to this processing block. The lowest heuristic cost is assigned to the processor, and a new ready list is determined and the process repeated until no node has a heuristic below a threshold value. At this point, nodes are assigned to the next processing block.

After allocating the nodes in this fashion, a relaxation procedure is used to determine if one or more nodes can be shifted between processors to lower the maximum load. The first step of this procedure is to determine which processing element has the largest load of computation + communication determined by

$$L(\max) = \max_{n = 1, \ldots, N_{proc}} \sum_{i=1}^{N_n} (t_{i,n} + c_{i,n}) \qquad \text{(EQ 1)}$$

where N_{proc} is the number of processors, N_n is the number of nodes assigned to the nth processor, $t_{i,n}$ is the computation time of the i^{th} node assigned to the n^{th} processor, and $c_{i,n}$ is the communication time required by the n^{th} processor due to the i^{th} node.

When the processor with the largest load has been determined, then each node assigned to that processor is checked to see if it can be moved to the previous or the next processor. A node can be moved to the previous processor if none of its input arcs are generated by nodes that are on the processor that this node is currently assigned to. Similarly, a node can be moved to the next processor if none of its output arcs are consumed by nodes that are on the processor that this node is currently assigned to. If a node can be moved, then the new maximum load (see Eq. 1) that this new assignment would create is calculated. If this new maximum load is less than the current maximum load, then the movement that reduces the new maximum load is taken and the process is repeated. If none are found that reduce the maximum load, then the relaxation is complete. Two methods of selecting the modification have been used: 1) Maximum Optimization Rule: The node that lowers the maximum load the most is selected, and 2) Minimum Disturbance Rule: The node that lowers the maximum load the least is selected.

The low-level software support for the high-level programming environment is the processing library. As an example, the image processing library contains the processor code for the operations in the INSIGHT application program. Each image processing operation in the following is a verb in INSIGHT. The initial set of functions in the library include arithmetic and logical operations on images, geometric spatial transforms, convolution and morphological operations, neighborhood operations, connected components, and masking.

In the Proteus system, users are allowed to choose the number of processors they wish to partition the algorithm between. This partition is then replicated with successive inputs images routed to successive processor blocks until all processors have been used.

3.1 System software

The system software consists of a loader and debugger which run on the Proteus host, message processing and control modules which run on the group and cluster controllers, and an interrupt handler which is the only system-level software resident on the processor.

As specified by the INSIGHT program and the number of processors the user chooses to use for each task, the translator generates an assignment file. Each assignment file defines a set of generic processors and the jobs partition between them. Each assignment file defines a task class. The Proteus system can execute multiple instances of a single task class or single instances of multiple task classes.

The user issues a task(s) request to the host to start loading and task execution. A task(s) request indicates which task class(es) should be executed, the external data sources for each task class if they need to be defined. As specified by the user's task(s) request, the loader retrieves the assignment file(s) produced by the translator. It determines if the Proteus hardware is capable of executing the specified assignment. If all is well, the loader performs the mapping of the logical processor names in Proteus to the symbolic processor names used by the translator to define the generic processor set. The mapping can be many to one or one to one. Physical processors are the true working horses for computation. For reasons of flexibility and fault tolerance, the loader does not assign jobs to physical procesors directly. The cluster controller has that responsibility. At run time it makes a one to one mapping between a logical processor name and the true physical processor.

Throughout the system, the logic and physical mapping is decided hierarchically. The host decides the mapping between the logical groups and physical groups. Each group controller decides the mapping between logic clusters and physical clusters. Each cluster controller assigns physical processors to logic processor names. The containing

A Typical Program

```
function Detect
        (integer array[256, 256] G0;)
        (binary array[256, 256] B4;)
where
        declare
                integer array[256, 256] G1,G2,G3;
                binary array[256, 256] B1,B2,B3;
                integer constant T1=195,T2=20,T3=25;
                integer constant W1=5,W2=15,W3=42,W4=126,W5=3;
        relations
                B1 = G0 < T1;
                G1 = G0 closedby box(W1,W1);
                B2 = (G1 - (G1 openedby box(W2,W2))) > T2;
                B3 = (B1 or B2) dilatedby box(W5,W5);
                G2 = fill(G1 maskedby B3);
                G3 = G2 openedby box(W3,W3);
                B4 = (G3 - (G3 closedby box(W4,W4))) > T3;
endwhere
```

Figure 4. An INSIGHT Program

relationships between logical processors, logical clusters, and logical groups are predefined, i.e., a logical processor belongs to a certain logical cluster which, in turn, belongs to a certain logical group.

Each logical processor inherits the assignments from its corresponding symbolic processor. According to the inheritance relationship, each logical processor has a list of jobs to be executed. Each job contains the following attributes: Program_id, input_arcs, output_arcs, constant parameters. Each job has a uniqe job ID assigned by the loader. Each arc has a unique arc ID assigned by the loader. Buffers are assigned to each arc. How many buffers should be assigned is decided by the translator. The cluster controller assigns actual memory addresses to each buffer.

The loader packs all the schedule information needed by a logical cluster into a scheduling file and all the constant parameters needed by the jobs executing within that logical cluster into a constant file. At run time, the cluster controller transfers task scheduling information from the cluster to the pixel processor via a task control block. The task control block contains the identification tag, starting address of the program to be executed by the pixel processor, and pointers to the buffers that will be used for input and output. According to the jobs assigned to the logical pixel processors with each logic cluster, the loader determines which programs should be loaded to that cluster and assigns memory space in shared memory to each program. Then, the loader creates a transfer request file for each group. Each transfer request file has a list of file transfer requests. Each request has the following format:

<request> := <source><destination list>

<source> := <file_name><file_size>

<file_name> := full path name of the file to be transferred.

<destination_list> := <destination> | <destination><destination_list>

<destination> := <logic_cluster_name><physical_starting_address>

After all the required files are generated properly, the host uses the socket facility of the Unix system to send a loading request message to each group via Ethernet. These request messages include two parts. The first part specifies that the action is loading. The second part is the full path name of the transfer request file. According to the received message, each group controller retrieves the transfer request file through NFS from the disk. Each group controller reads in its file specified in the request file, writes it to the VME buffer of the destination cluster(s) and requests the destination cluster controller to move it to the shared memory starting from the physical address specified in the request file. The cluster controller uses the check sum stored in each file to check for any transmission errors.

When all the files specified in the transfer request file have been moved to the clusters, the group controller sends a file transfer complete message to each cluster in the group. If the file transfer complete messages have been received and every file is properly stored and every processor in the group is ready to work, the cluster controller sends a ready-to-work message to the group controller. After receiving a ready-to-work message from all the clusters wihin the group, the group controller sends a ready-to-work signal to the host. After receiving the ready-to-work message from all the groups, the host synchronizes the external data sources and the Proteus system begins the task(s) execution.

The processing and computation in Proteus uses a variety of software and hardware control mechanisms. Each pixel processor in a cluster and the cluster controller have shared-memory mail boxes. They also communicate with each other via interrupts. At run time, the cluster controller dispatches a job to each idle pixel processor by interrupting the pixel processor to indicate the task control block is ready to be read. When a pixel processor finishes its assigned job, it creates a task completion record and interrupts its cluster controller to report the results. After receiving the interrupt signal from the pixel processor, the cluster controller reads the completion record to get the information from the pixel processor, updates the status of data regions due to the task just completed, and continues to activate sleeping processors. When busy processors complete a task, they consult their task control block area to determine their next task. In this manner, busy processes do what they have to do without ever having to be interrupted once they begin their processing.

The debugger of the Proteus system is implemented as two communicating processes, one on the host and the other on the Proteus system. The debugger interacts with the user through the host's window system. It has the capability to support the development of system and application programs. The debugger provides the user with the capabilities to control and monitor the execution of all the pixel processors in the Proteus sytem. Therefore, the user must have full knowledge of the system architecture, how the system operates, and its physical sources of input images during the run-time.

On the host, the debugger is simply a front-end interface that interacts with the user, and manages the bulk of debugging information (e.g. symbol tables) that allows it to map symbolic information to physical addresses in the hardware. However, the physical laydown of code breakpoints, memory accesses, and modification of processors' execution states must be done by system services provided by the cluster controllers and the pixel processors from the hardware side.

Other than the execution of the hardware, the debugger is also in control of three other system applications on the host; they are the loader, the high-level simulator, and the low-level simulator. Each of them supports the debugging of image processing application programs at a different level.

At the application level, the subject language is called INSIGHT. At this level, the debugger allows the user to control execution by setting image watchpoints (i.e. data breakpoints) in the INSIGHT data flow graph. When a pixel processor picks up a task that will be producing an output image associated with an image watchpoint, it will be interrrupted and the debugger will take the control of its execution. At this debugging level, the user may invoke the high-level simulator to execute the corresponding INSIGHT program. The simulator, which also runs on the host, will produce a trace of images that can be compared against those that are generated by the hardware. In order to allow the user to visualize the results, the debugger can compare two images and display the corresponding images in a window.

At the system level, the subject language is the assembly language of the i860 pixel-processor. The user may trace through the execution of a program by a pixel processor by setting code breakpoints inside the program and single-stepping through the program. The user may invoke the low-level simulator to execute this program in the same sequence that would be executed in the hardware. By comparing the execution states of the pixel processor against those generated by the simulator at certain points of the course of execution, the debugger is able to locate any error that could occur.

The high-level simulator, written in ADA and executed on the SparcStation host, is designed to verify that INSIGHT algorithms and the Proteus image processing library are correctly implemented in Proteus. Under the control of the Proteus debugger, the high-level simulator executes vision algorithms exactly the same algorithms are being executed from the Proteus hardware. A comparison between the two results can then provide information about the correctness of Proteus's result.

There are three major components in the high-level simulator: a debugger interface, a controller, and an application library. The debugger interface parses the command-line for the simulator passed down from the debugger. The controller figures out the appropriate tasks to carry out while the application libary gets accessed by the controller for the number-crunching. The high-level simulator could be instructed to start a new application by loading in an INSIGHT program. It is able to find out the exact steps involved to perform the calculation for each node in the INSIGHT data-flow diagram. It also provides a session-save feature so that intermediate results and system status can be loaded back into the system in a future session with the session-reload capability. Although it functions under the control of the Proteus debugger, the high-level simulator is a full-fledged image processing system on its own.

Finally, to provide an aid for execution tracing that can be used in performance monitoring, tuning, and debugging, Proteus system software has been provided with the Portable Instrumented Communication Linrary (PICL)

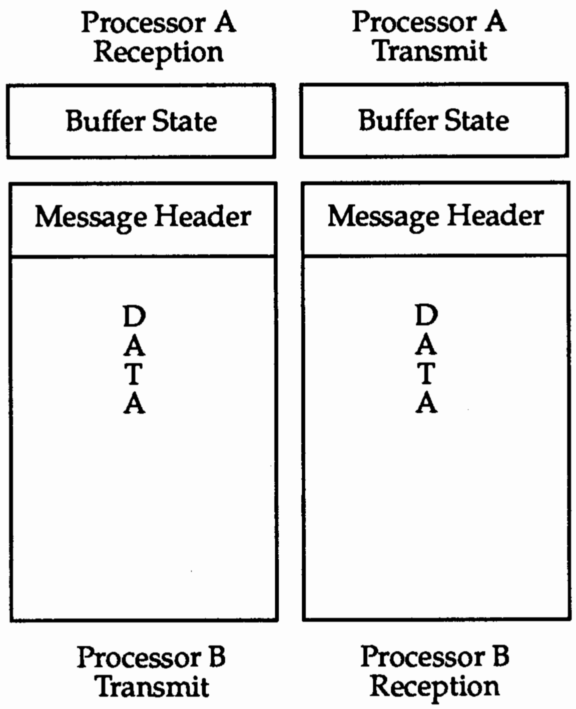


Figure 5. Message buffer layout.

developed by Oak Ridge National Laboratory.[8,9.] The PICL communication library was originally developed to provide portability and easy parallel program development. It has a good facility for execution tracing which is its major use in Proteus.

All messages passing between processors consists of copying the message (header and data) from the source processor memory to a memory location that the receiving processor can access. This memory is accessed in a synchronized manner to prevent confusion. The sending processor and receiving processors view the buffer in a complementary fashion as shown in Figure 5.

In Figure 5, processors A and B are shown sharing a set of message buffers. Notice that Processor B Transmit buffer is processor A reception buffer and vice versa. Data from A to B is copied into the buffer shown on the right and data from B to A is copied in the buffer on the left.

The general steps processor A goes through to send a message to Processor B are:

1. Processor A checks its transmit buffer state and if in the *Unread* state waits until a later time to send the message.

2. When the buffer state is set to *Read* by processor B, Processor A has control over the contents of Processor A transmit buffer. Processor A then copies bother the header and data of the message into the Processor A Transmit buffer.

3. Processor A then sets the buffer state to *Unread* indicating a new message is in the buffer.

4. Processor A completes its portion of the message transfer by interrupting Processor B to cause it to check its reception buffer.

When a processor B receives a message, it performs the following processing:

1. It receives an interrupt that causes the processor to stop its current processing and check to see if it has received a message.

2. The message check consists of making certain its reception bufer state is in the *Unread* state.

3. When Processor B is done with the reception buffer (either by copying it into a local buffer or processing it directly) it sets the buffer state to *Read*.

4. PROTEUS HARDWARE DESCRIPTION

As illustrated in Figure 6, our implementation of the Proteus architecture has tightly-coupled processor clusters connected in groups. Communication within a cluster is through shared memory. Communication within a group is through a circuit switched cross-bar connection. Communication between groups is through a circuit switched enhanced hypercube connection. A separate control network of buses within each group, and ethernet among groups, allows additional control and communication. A top level view of the system is shown in Figure 6.

4.1 Circuit Switched Enhanced Hypercube

The binary hypercube-based computers, cosmic cube, Ncube, and FPS T-Series [5.], use packet switching to communicate from node to node. Proteus uses circuit switching. A Proteus node, what we call a group, consists of 9 clusters, each having 4 processors, making a total of 36 processors. The groups are connected in an enhanced hypercube structure. An enhanced hypercube contains two links in any one dimension of a regular hypercube, as shown in Figure 7. The primary advantage of the enhanced hypercube architecture is the permutation embedding capability. A centralized algorithm at the host may route any arbitrary permutation. The 32 groups in a full scale system can thus communicate with each other in an arbitrary permutation for rapid exchange of data. By not buffering the data at the intermediate nodes, the transmission across the diameter of the hypercube are negligible.

The enhanced hypercube is scalable from a 3 cube to a 5 cube with 8 to 32 nodes, or groups. The primary advantage of the large number of processors in each group is for large grain parallelism problems which may communicate efficiently using large blocks of data. The external input is received on 32 parallel channels which are equally distributed to the enhanced hypercube nodes.

The enhanced hypercube of Proteus is also a special case of the generalized folding cube [2.]. Direct application to algorithms is provided by trivial embedding of meshes, rings, tori, etc. The general interconnections available allow many algorithms to be directly mapped into Proteus with optimal performance. The generalized cube has multiprocessors at each node. Studies have shown that efficiently coded algorithms on the hypercube underutilize the available bandwidth [13.]. By clustering processors at each node the Proteus architecture improves the link utilization. Detailed descriptions of the communication network and the enhanced hypercube are given in section IV.3.

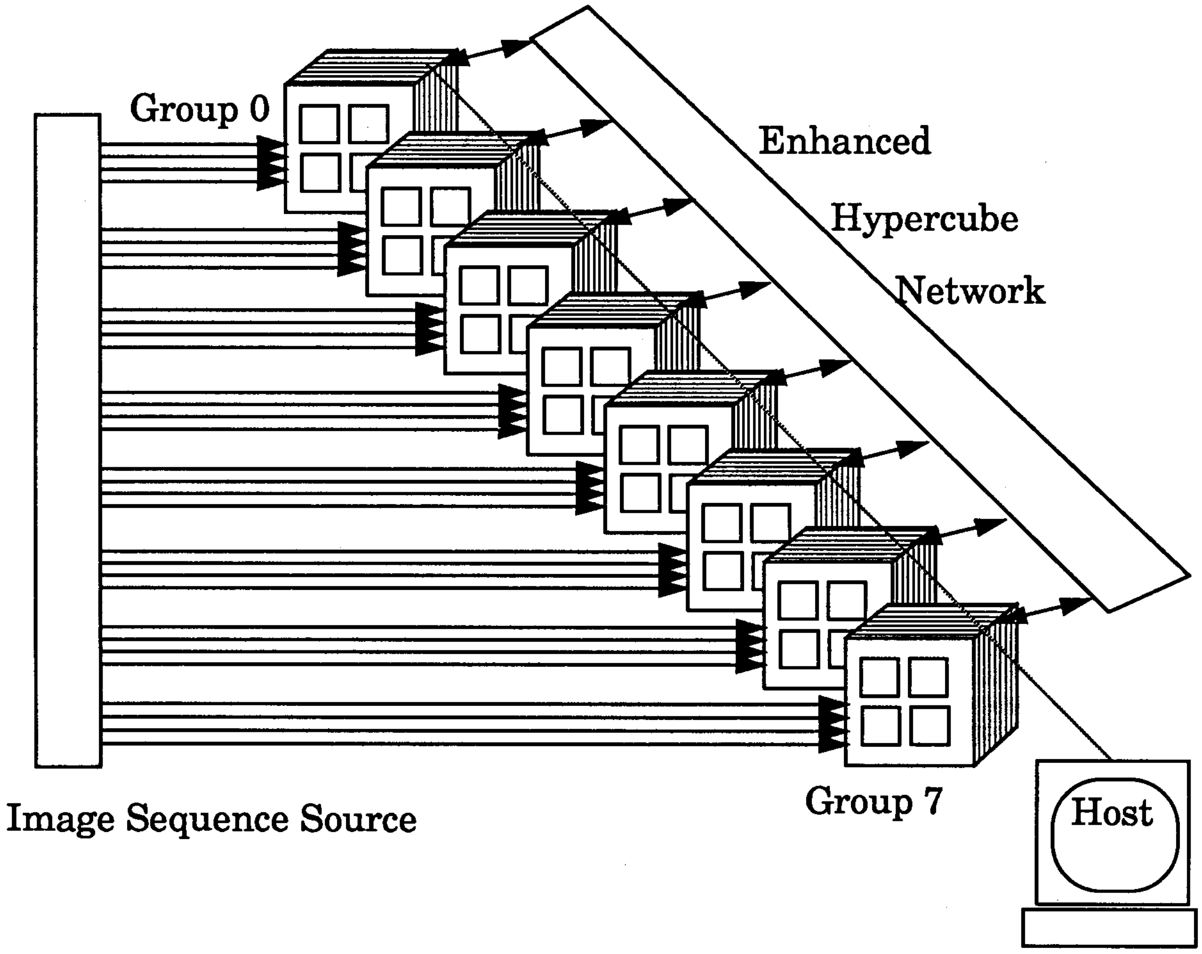
4.2 Allocating Caches

Clustering of processors, while cost effective, may cause contention for shared resources. Detailed simulation, program studies, and architectural trade-offs allowed us to optimize the use of the shared memories at clusters. In effect, the advantages of local memory and cache memory have been combined by using an innovative implementation of read and write allocation [25.]. Read and write allocation force cache accesses to hit, thereby reducing shared memory accesses, and limiting multiprocessor contention. For initial applications read/write allocation has shown shared bus accesses to be reduced by 6.6% in the single program, multiple-data stream mode [24.]. The allocating cache is a high performance interconnect that is much more general than the register memories used in the Orthogonal multiprocessor (OMP) [14.], which requires explicit loading and unloading of register variables. Proteus caches may be set to different modes by using mode bits in the address, so any combination of modes may be used in pages which map to unique positions within the cache. The caches are described fully in the design section.

4.3 Fault Tolerance

Initial design goals focused on the incorporation of limited fault tolerance. By requiring general connectivity of

Proteus Hardware System



- 32 Parallel Channels for image transfer
- 8 Groups Expandable to 32
- Each Group has up to 9 Clusters
- Each Cluster has 4 Processors
- Peak input rate of 25 512x512 image frames per second on each channel

Figure 6. A top level view of the Proteus architecture.

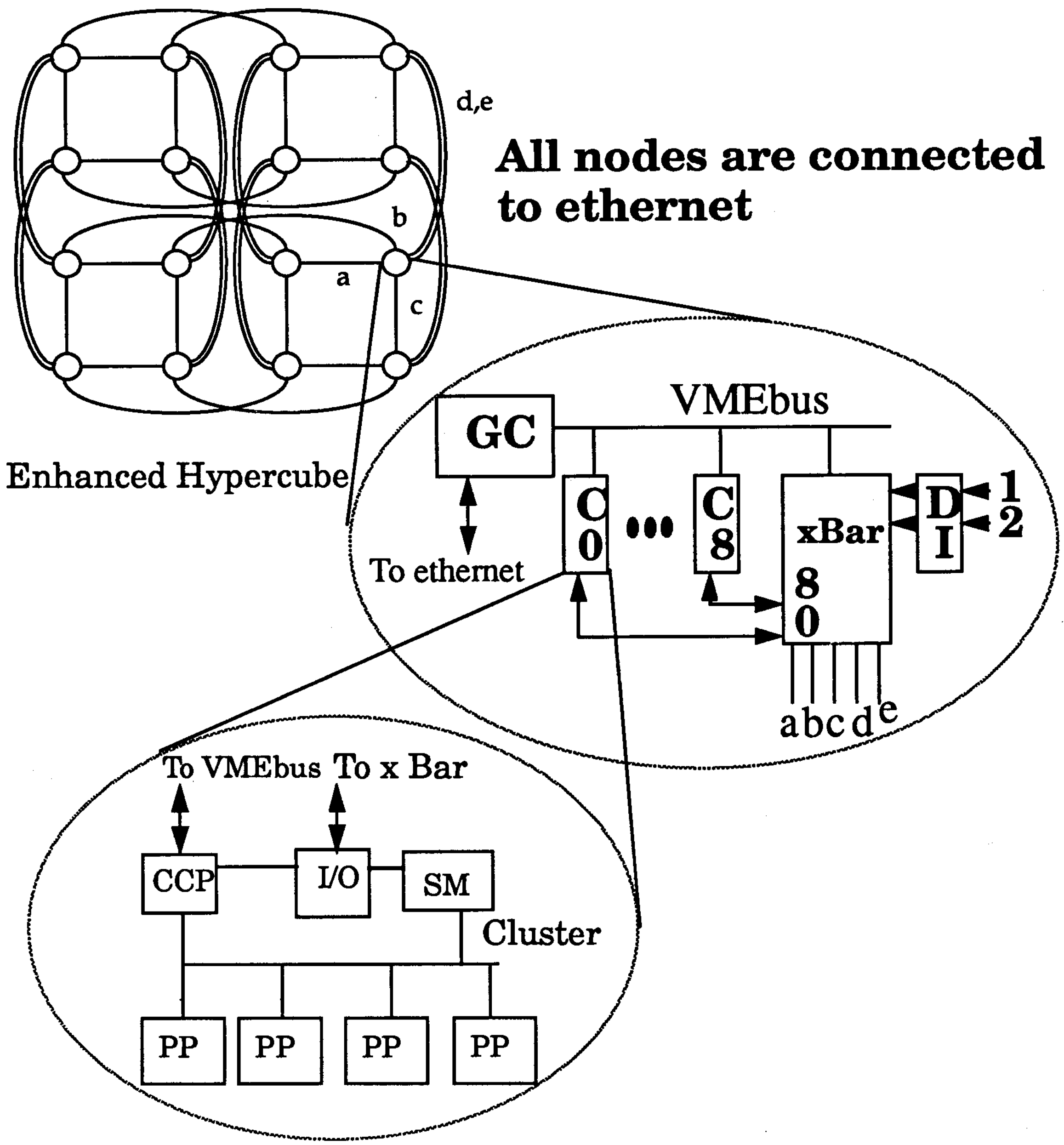


Figure 7. Exploded View of Proteus System

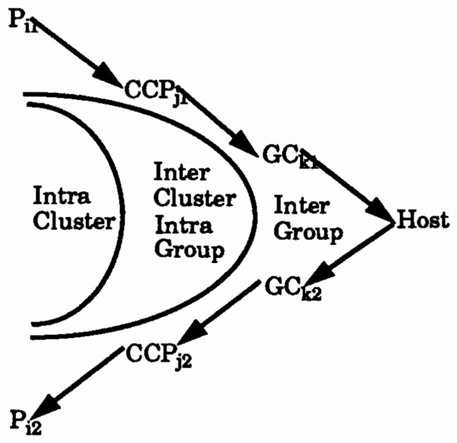


Figure 8. Communication hierarchy

clusters, and the arbitrary assignment of jobs to processors, system level fault diagnosis [21]. can be performed at the cluster level. Proteus incorporates a small amount of spare processing capacity which is used for roving tests and redundant computation, to create on line fault diagnosis. The fault diagnosis strategy is discussed further in the architecture section.

These unique aspects of Proteus create a research computer that advances current architectural thought. The Proteus architecture is a test bed for hypercube communications, allocating caches, and system level fault diagnosis. Simulation shows these features give higher performance and reliability than other architectures.

4.4 Enhanced Hypercube

The hypercube is an undirected graph of 2^n vertices where each vertex has n links, or edges to other vertices. A 3 dimensional cube has $2^3 = 8$ vertices, and each vertex has 3 links. A permutation in the hypercube is a connectivity set used to represent the communication to occur. For example a 2-cube permutation is [3,2,0,1] so that vertex 0 connects to 3, 1 to 2, 2 to 0, and 3 to 1. Arbitrary permutations may be possible in any dimensional cube, but it has not been proven.

Proteus uses the enhanced hypercube static network for which it has been proven that arbitrary permutations can be embedded [3]. The Enhanced Hypercube uses two links instead of one in any one dimension of the original binary cube for $n > 3$. This gives us complete reconfigurability. Figure 7 shows Proteus with $n = 4$, and the extra links connecting all nodes in the vertical dimension.

The links marked *a*, *b*, *c*, and *d* are the high speed serial links input and output for one group. The *e* link is the additional link which allows full permutation capability. The exploded view of the group contains the Unix board group controller (GC), the clusters (C0 to C8), and the communication interface or crossbar (xBar). Clusters are connected by crossbar to each other and to the enhanced hypercube. I/O from external sources is fed through the I/O buffer marked as IB. An exploded view of a single cluster is shown, and consists of the cluster control processor (CCP), the shared memory (SM), the I/O buffer and memory (I/O DPM), and the RISC processors (or pixel processors, PP). Pixel processors in a cluster share memory and a serial I/O link. External caches and control processors help ease contention and multiprocessing performance degradation.

4.5 Communication

The communication structure, shown in Figure 8, is hierarchical to share resources and distribute control overhead. Currently, communication through hypercube links is arranged by the host. Communication within groups is set up by the group controller, and communication within a cluster is set up by the cluster controller. All links to cross-bar are optical serial links which transmit/receive data at 250 Mbits/second. When a path has been set for cube communication, data passes directly from the source cluster to the destination cluster in another group. No

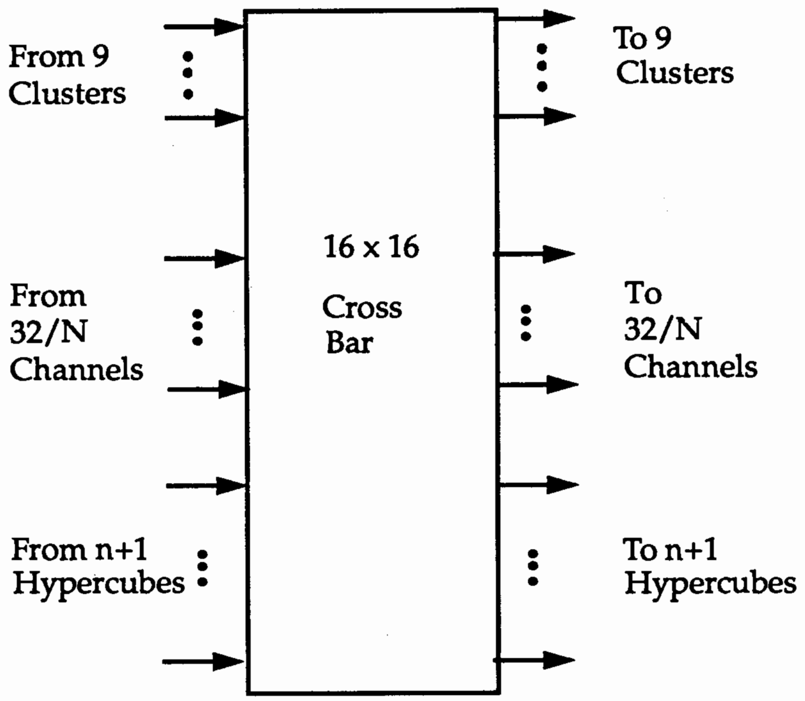


Figure 10. Crossbar Connections

store and forwarding is done with the circuit switch connection.

Within the group, a crossbar connects serial links to and from sources and destinations. In parallel with cube communication, additional clusters within the group may be transmitting and receiving data. At any time, k clusters in a group may be using cube connections, so that $9 - k$ clusters may communicate amongst themselves. The cluster's four processors share a serial I/O link which is accessible through a dual port memory buffer. The shared memory provides intra cluster communication, and the dual port buffer provides highest I/O performance. The control of communication, and the control network are described in the following section.

When a PP i_1 in a cluster j_1 in group k_1 wants to send a block of data to another PP i_2 in cluster j_2 in group k_2, the path is set up under the control of cluster controller j_1, j_2, group controller k_1, k_2 and host in a tree fashion depending on the location of PP(i_1, j_1,k_1) and PP(i_2,j_2,k_2). This is depicted in Figure 7. If j_1=j_2 (then k_1=k_2) and cluster j_1 arranges for data transfer through the shared memory. If $f_1 \neq f_2$ but k_1=k_2 then cluster controller j_1 request group controller k_1 (=k_2) to set up the path through the crossbar. Group controller also informs the receiving cluster j_2 to be ready to receive data. If j_1=j_2 and k_1=k_2, then the group controller k_1 requests to host to set up a path through the enhanced hypercube. When the path is available, the host informs all GCs which include GC k_1, GC k_2 and intermediate GCs. All GCs set up their X-Bars. GC k_1 and GC k_2 inform their respective clusters which in turn sets up their respective transmission and receive DMAs.

4.6 Control

The system is hierarchically controlled as illustrated in Figure 9. Both the Enhanced hypercube and the crossbar connections within a group are managed by the generalized communication interface, GCI. The link connections to the cube and clusters are provided in a crossbar within each group. The GCI consist of a 16×16 cross point switch. Each input can transmit up to a 1000 Mbits/sec fiber link but the actual speed to be used in the current system is 250 Mbits/sec. The 16 links on the input side are used by the nine clusters in the group, $32/N$ input channels and the enhanced hypercube links. A block diagram showing the crossbar connection is depicted in Figure 10.

The group controller is a single processor Unix board equipped with the VMEbus and ethernet interfaces. It

System Control Hierarchy

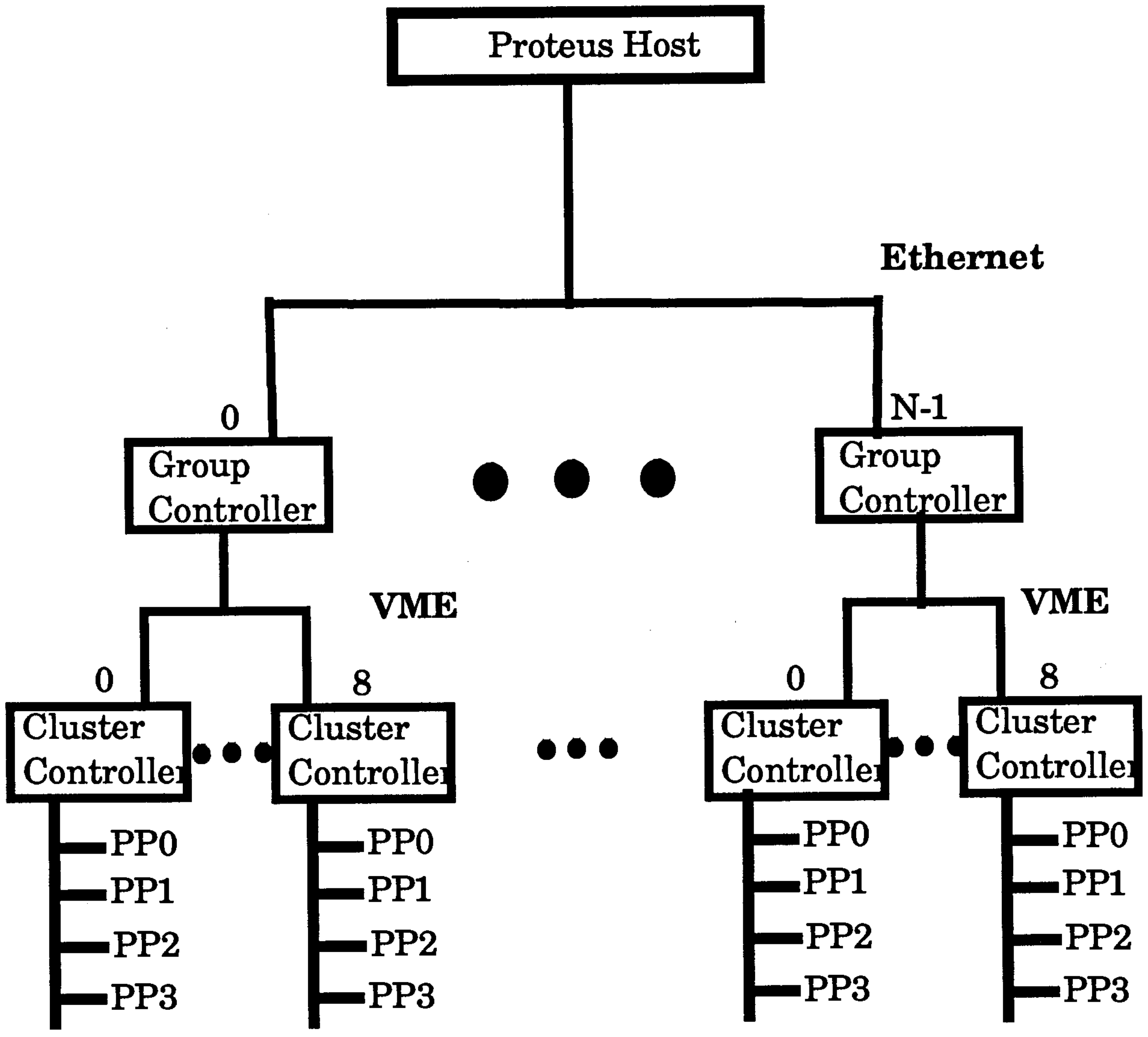


Figure 9. The system control hierarchy

operates under a real time UNIX operating system environment. Each group has a single VMEbus accessible to all of its clusters. The group controller coordinates all activities within the group. It assigns tasks to each cluster and sets up communication paths. Possible paths are from input to cluster within the group, intra cluster within the group, and intergroup. Paths are set by writing to the GCI.

The Proteus host sets initial configurations and manages cube links between groups. It is a general purpose Unix work station. It is responsible for system operation, user interaction, and output collection. Algorithms are developed at the host and mapped on to the system. Under the host, $N = 8$ to 32 groups are connected to the Proteus host through ethernet.

Within the group 8 clusters (with 1 extra for fault tolerance) are controlled via VMEbus. The group controller reads sending requests and activates destination clusters through VMEbus control registers. The movement of data is synchronized and each image frame transmission is completed within a fixed time. The set-up for all GCIs is synchronized. If the communication is to be restricted within a group, then the GCI allows asynchronous communication under the control of the group controller.

The lowest level in the hierarchy is the cluster. This is shown in Figure 11. The cluster has a dedicated control processor, the Intel i960. The cluster controller schedules tasks on the pixel processors, manages shared memory, arranges for receiving and dispatching data by serial I/O, and monitors performance by using a hardware timer. The 4 RISC processors, i860s, share memory and have their own cache. The Intel i860 is a high performance 64 bit microprocessor. It supports parallel and pipelined execution with a RISC paradigm, using independent core/integer unit and a floating point/graphics unit. These units may operate in parallel, and may access on-chip caches in a single cycle at 40MHz.

Custom external caches tie directly to a shared 64 bit data, 32 bit address bus which services the 8 – 32 Mbyte shared memory and the 1Mbyte I/O buffer. The shared bus allows locked accesses for semaphore, test and set, and compare and swap operations, and burst fetches, of four 64 bit words.

The external cache memory holds both data and instructions. The external cache is organized as a 1 megabyte direct mapped cache with a line size of 32 bytes. This matches with the internal line size of the Intel i860. The cache is designed for efficient multiprocessing with adaptable modes dependent upon the data: cached locally, cached shared, or uncacheable. Normal caching modes include write through and write back. New modes allow for validation of tags without reading that line from the shared memory[25.]. Cache write allocation forces a hit upon a write. This reduces the shared bus cycles and improves the overall performance of the system. In addition to the novel use of the above modes, line flushing, flush and invalidate, invalidation, and labelling are used to control individual lines in the cache. The cache modes are established by using multiple virtual addresses for the same physical memory. Software is responsible for cache management.

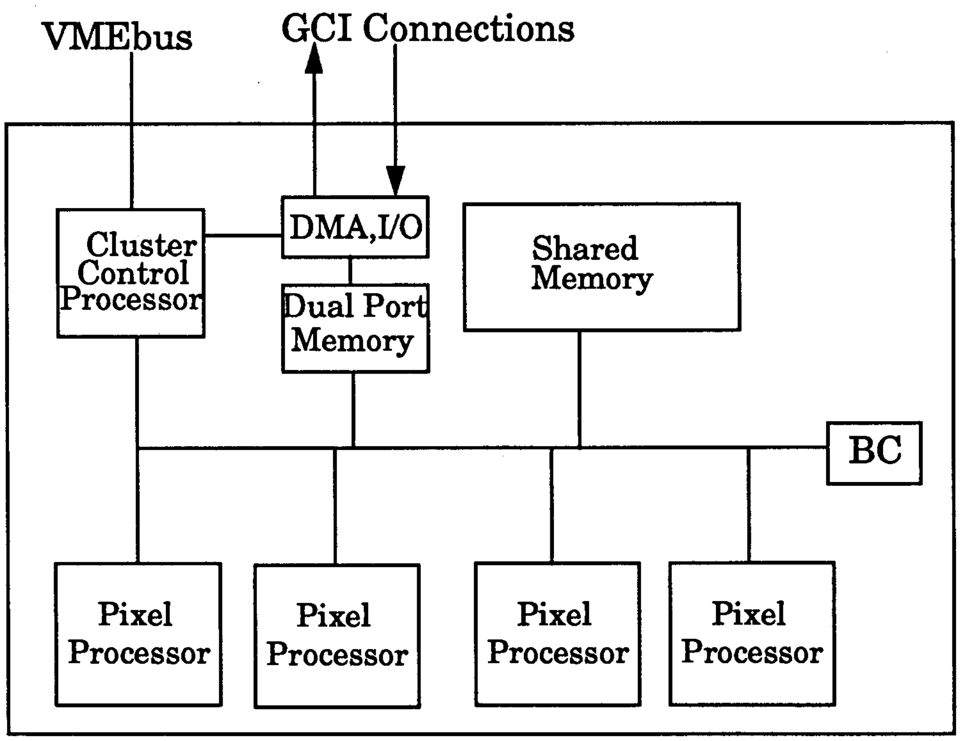
For performance monitoring, the hardware provides one 16-bit counter and one 32-bit counter for each cluster controller, each driven by a 1 Megahertz clock.

4.7 Design

The design process started with initial discussions on approach, performance, and applications. The design was to be restricted to one circuit board, if possible, to reduce layout and debugging time.

Processor Selection. A processor survey was done to determine the most applicable microprocessor. A representative algorithm, morphological dilation, was used to "paper" code programs to compare their performance and features. Important features used for comparison were arithmetic speed, number of registers, on chip memory (or cache) size, external bus bandwidth, and floating point capability. Processors investigated included the Intel i860, the MIPS 3000, the Motorola DSP 96002, the Texas Instruments TMS 320C30, the AT&T DSP 32C, the Motorola 88000, the Motorola 68040, the Intel 80486, and the Inmos T800. The i860 proved to be the clear choice for design, because of a combination of a 64 bit data bus and 12K bytes of on-chip cache memory. Analysis by Levy[16.] showed the i860 to be poor for operating system work, so extra care was taken in design to minimize interruption of the

Cluster Overview



Cluster with Pixel Processors And Cluster Control Processor

- Four Pixel Processors (PP)
- 8 Mbyte shared memory + 1 Mbyte Dual Port Shared Memory
- 1 Mbytes Cache memory for each PP
- 160Mbyte/second shared memory bus
- I/O Channels transmitting at 250Mbit/second

Figure 11. An overview of the Proteus cluster.

i860's processing. All system-level functions were put on the control processor whose principle job is to take care of scheduling and interrupt handling.

Interconnection and I/O. Investigation into interconnection schemes and I/O to handle high input rates revealed a variety of options. The basic requirement was to allow data to be sent directly to any of the 256-1024 processors. To support processor pipelines and distributed processing data were also to be moved from processor to processor.

The only feasible option to support the high-input data rate was to divide input data over multiple I/O channels. Parallel data transfer would imply large numbers of cables, not a desirable feature. So fast serial I/O channels were considered. Serial to parallel data conversion takes place at the I/O interface using recently available gallium arsenide derivatives. Input data is stored in shared memory using a DMA. A separate 64/256 bits wide fast parallel bus for data exchange within a group was considered. A 64 bit wide bus with available technology could handle the average data load, but performance would suffer if a peak load was experienced. A 256 bit width bus was would have forced us into a tight design space as it would have required larger board area and wide memory word size. Another option was to use switched fast serial lines between clusters. High speed parallel-to-serial and serial-to-parallel chips from Gazelle [7.] and a fast cross bar chip from Gigabit logic [10.] were available from off the shelf. This was an attractive design option and these chips form the backbone of the communication network within each group. In addition the enhanced hypercube connection could be realized using the same crossbar chip. With 32 I/O channels and 8 clusters per group (plus one for fault tolerance), there were 7 ports left for managing input/output channels and enhanced hypercube connections. An enhanced n-cube (n dimensional) requires $n+1$ links at each node for $n > 3$ and n links for $n \leq 3$. At the same time 32 I/O channels were to be equally distributed among the groups. The distribution of channels is as follows.

Table 1. Channels

n	*I/O channels/node*	*Enhanced hypercube link*
5	1	6
4	2	5
3	4	3

This suited us very well, and we used a 16x16 crossbar at each node as shown in Figure 7.

Cluster Design. The most detailed analysis for design was performed on the cluster board. With the available technology, it was reasonable to fit four processors on one board. To support embedding of more general program graphs, we searched for a more general design. One possibility was to split the memory into several banks and provide a crossbar interconnection among processors and memory banks. This could do well with processor pipelines, but embedding arbitrary program graphs would cause blocking. Thus other options were considered. These included 1) a shared memory with a 256 bit wide bus with 4 x 256 bit data buffers (memory interface unit, MIU); 2) shared memory and a local memory with each processor; and 3) a shared memory with processor caches. In each case, it was possible to share the memory for I/O through DMA, direct memory access, or provide separate buffering for I/O. The bus could be 64 or 256 bits wide. These alternatives were compared using Network II.5 simulations and then low-level HDL, Hardware Description Language, simulations.

Several things were learned from the simulation. A high amount of conflict resulted whenever the input data was being transferred into shared memory. Because of this, closer attention was paid to the I/O design on the board. With the use of an I/O buffer the input and output data could be removed from the shared bus. Therefore, a dual port memory was added to manage the I/O. The MIU model suffered because the processor could not cache all of its data in its on chip cache and higher contention resulted. In addition a 256 bit bus was thought to be an implementation risk. The local memory model suffered because no processing occurred when the data is being transferred to shared memory and because the local memory is a fixed size. However, local memory is advantageous when processing creates large results which were to be used again by the same processor. The cache solution computed while reading initial data, did not fix the size of programs and data, and allowed a 64 bit bus to achieve acceptable performance. However, depending on the algorithm, the bus could still be saturated.

Additional optimization of caching was investigated. When blocks of data are to be generated as a result of computation, reads do not have to be done for caching. The processing of blocks of data lead to the idea of allowing pages of the cache to be controlled in a local memory mode, so local data could be forced to stay off the shared memory bus. Through allocation a section of the cache was to allow allocated writes. These writes would hit irrespective of the address tag present in the cache. If valid data was previously in the cache that needed to be flushed, this would be done, and then the write would be performed.

Further investigation showed that clever coding on the processor allowed results to be cached which dramatically reduced traffic on the bus. Since the i860 allowed 64 bit transfers, using the bus for less than 64 bit transfers results in under-utilization. In particular, if the transfer happens to be a byte, which was the case for our first vision application, the performance loss is severe. Therefore a scheme in which write data are cached and transferred to main memory in chunks of 64 bits yields much improved performance. Table 2. shows the results of this study. Three models were studied which are: A) a statistical read/write model, B) a deterministic read/write model, and C) a statistical read/write model that caches the writes. The same program was running on all four processors, and processes a 64 K byte image and creates a 64 K byte image in an optimistic 45 milliseconds. In the first two models, A and B, the byte pixel writes go directly to the shared memory, so that all four processors writes may cause conflicts. Model C reduces write traffic by writing words of 8 pixels which would be flushed from the on chip cache. The Delay of getting the bus (nanoseconds), the number of processors queued up waiting for the bus (processors), and the percentage of time that the bus is busy are shown (average/maximum).

Table 2. Byte writes vs. reads and flushes

Model	Delay	Queue	% Busy
A	15/465 ns	0.11/3 proc.	41.6%
B	7/701 ns	0.04/3 proc.	30.8%
C	10/378 ns	0.011/2 proc.	12.2%

One way to force a hit on writes was to modify cache tags, a feature available in the i860. However, that required extensive modification in program development. An alternative was to read result locations before writing. This happens naturally in many applications where the computation is of the form $A \leftarrow A \otimes B$ where $\otimes$ is any operation and A and B are two operands. Otherwise, the compiler (or programmer) could do so for operations like $A \leftarrow B \otimes C$. In the second case it does not matter what data is read for A as they are overwritten. If possible then read allocation [25], or forcing a hit on reads in external cache, was found to be useful when preparing the processor to cache results on chip. The processor reads the buffer from the cache without going to shared memory. This read is done to validate the on-chip cache tag, so subsequent result writes hit in the cache. The addition of optional read and write allocation further improved the cache solution, and provided a unique solution to the memory bandwidth matching without changing the microprocessor itself.

The final shared memory design prevents byte, 16 bit, and 32 bit writes. This is done so that inefficient use of the shared memory bus is not allowed. Programmers must use the external cache and explicitly flush their results from the external cache, or use read allocation and flush the on-chip cache to write-through to the shared memory.

5. SUMMARY

We have presented an innovative architecture designed for processing applications where large granularity may be used. The separate communication and control allows for high communication and I/O rates. By utilizing Choi's recent theoretical developments in hypercube theory [2,3,4], Proteus creates complete permutation capability. This allows embedding of arbitrary graphs, and the circuit switched links provide guaranteed rates of communication. Shared memory multiprocessors contention problem is addressed by clustering processors, and by using innovative cache designs to allow for the ideal cache and local memory behavior. With the general interconnections and reassignment of clusters, System Level Fault Diagnosis is achieved for all applications running on Proteus

We have discussed how the system software easily permits the efficient control of large grained parallelism without having to handle the general concurrency problem. We have described how the user can write high level algorithms which get efficiently mapped to the Proteus hardware by the INSIGHT translator. We have shown how the reconfigurable computation network can get its act together.

Our first image processing application will be running by the Fall of 1991. As soon as this occurs, we will expand the application software to include higher-level computer vision operations as part of the INSIGHT language.

6. REFERENCES

1. M. L. Campbell, "Static Allocation for a Data Flow Multiprocessor," *Proceedings of the 1985 International Conference on Parallel Processing*, 1985, pp. 511-517.

2. S. B. Choi and A. K. Somani, "The Generalized Folding-Cube Network," *NETWORKS, An International Journal*, in press.

3. S. B. Choi and A. K. Somani, "Rearrangeable Hypercube Architecture for Routing Permutations," Accepted for publication in *JPDC*, December 1990.

4. S. B. Choi and A. K. Somani, "The Generalized Hyper-Cube," in *Proceedings of ICPP-90*, August 1990, pp. 372-375.

5. Jack J. Dongarra and Iain S. Duff, "Advanced Architecture Computers," Technical Mem. No. 57, Argonne National Laboratory, Sept. 1989.

6. R. Duncan, "A Survey of Parallel Computer Architectures," *IEEE Computer*, Feb. 1990, pp. 5-16.

7. Gazelle "Preliminary HOT ROD High Speed Serial Link Gallium Arsenide" GA 9011, and GA 9012, Gazelle Microcircuits, Inc., Owen Street, Santa Clara, CA 95054.

8. G.A. Geist, M.T. Heath, B.W. Peyton, P.H. Worley, "A User's Guide to PICL: A Portable Instrumented Communication Libnary," ORNL/TM-11616, Oak Ridge Natioal Laboratory.

9. G.A. Geist, M.T. Heath, B.W. Peyton, P.H. Worley, "A User's Guide to PICL: A Portable Instrumented Communication Libnary, C Reference Manuak" ORNL/TM-11130, Oak Ridge Natioal Laboratory.

10. Gigabit Logic "16x16 Crosspoint Switch 2.6 Gbit/s Data Rate/1.8 ns Reconfiguration Time", 10G051 Gigabit Logic.

11. R. M. Haralick, S. R. Sternberg, Y. Zhuang, "Image Analysis Using Mathematical Morphology," *IEEE Transactions On Pattern Analysis and Machine Intelligence*, Vol. PAM1-9, No. 4, July 1987.

12. M.T. Heath, "Visual Animation of Parallel Algorithms for Matrix Computations," *Proceedings of the Fifth Distributed Memory Computing Conference*, IEEE, 1990.

13. J. M. Hsu and P. Banerjee, "Performance Measurement and Trace Driven Simulation of Parallel CAD and Numeric Applications on a Hypercube Multicomputer," *17 Annual Int. Symp. on Comp. Arch.* May 28-31, 1990, Vol. 18, No 2. pp. 260-269.

14. K. Hwang, et. al. "OMP: A RISC-based Multiprocessor using Orthogonal-Access Memories and Multiple Spanning Buses," *Int. Conference on Supercomputing*, Vol. 18, No. 3, June 1990, pp. 7-22.

15. B. Kruse, "State-of-the-Art Systems For Pictorial Information Processing", *Fundamentals in Computer Vision*, O.D. Faugeras (Ed.), Cambridge University Press, Cambridge, 1983, pp. 425-442.

16. H. Levy, Personal Communication.

17. C. F. Olson, "Load Balancing in Dataflow Multiprocessors, A Project for EE 595," Technical Report, University of Washington, 1990.

18. T.A. Rice and L.H. Jamieson, "Parallel Processing For Computer Vision", *Integrated Technology For Parallel Image Processing*, S. Levialdi (Ed.), Academic Press, Inc., London, 1985, pp. 57-78.

19. L. G. Shapiro, R. M. Haralick and M. Goulish, "INSIGHT: A Dataflow Language for Programming Vision Algorithms in a Reconfigurable Computational Network," *International Journal of Artificial Intelligence and Pattern Recognition*, Vol. 1, No. 3/4, 1987, pp. 335-350.

20. L. G. Shapiro, "Programming Parallel Vision Algorithms: A Dataflow Language Approach," *The International Journal for Supercomputer Applications*, Vol. 2, No. 4, 1989, pp. 29-44.

21. A.K. Somani and V.K. Agarwal, "Distributed Syndrome-Decoding for Regular Interconnected Structures," in *Proc. of FTCS-19* held at Chicago, pp. 70-77, June 1989.

22. S.R. Sternberg, "Parallel Architectures For Image Processing", *Real-Time/Parallel Computing*, M. Onoe, K. Preston, and A. Rosenfeld (Eds.), Plenum Press, N.Y., 1981, pp. 347-359.

23. W.W. Wadge and E.A. Ashcroft, *LUCID: The Dataflow Programming Language,* Academic Press, London, 1984.

24. C. Wittenbrink and A. K. Somani "Algorithm Based Cache Design for High Performance Morphological Image Processing," submitted to *ACS Transactions on Computer Systems,* Nov. 1990.

25. C. Wittenbrink "Directed Data Cache for High Performance Morphological Image Processing," Masters Thesis, University of Washington Dept. of Electrical Engineering, Oct. 8, 1990.

26. S. Yalamanchili, K.V. Palem, L.S. Davis, A.J. Welch, and J.K. Aggarwal, "Image Processing Architectures: A Taxonomy and Survey", *Progress in Pattern Recognition*, L.N. Kanal and A. Rosenfeld (Eds.), Elsevier Science Publishers, B.V. (North Holland), Amsterdam, 1985, pp. 1-37

27. X. Zhuang and R. M. Haralick, "Morphological Structuring Element Decomposition," *Computer Vision, Graphics, and Image Processing,* 1986, Vol. 35 pp. 370-382.