Proteus System Architecture and Organization

Arun K. Somani, Craig Wittenbrink, Robert M. Haralick,
Linda G. Shapiro, Jenq-Neng Hwang, Chung-Ho Chen,
Robert Johnson, and Kenneth Cooper

Dept. of Electrical Engineering
University of Washington
M.S. FT-10
Seattle, WA 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 G-flops (80 G-flops peak). It accepts data via multiple serial links at a rate of up to 640 megabytes/second. The system employs 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 1 M byte 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.

1.0 Introduction

Proteus is a sea god who changes his shape at will. The Proteus Supercomputer is a reconfigurable network of processors that can change its configuration to perform optimally on a variety of large granularity tasks. It is an MIMD, multiple instruction multiple data, machine unique in many respects. Special features are *enhanced hypercube* circuit switched communications [3], *read* and *write allocating* caches [17], and *system level fault diagnosis* [15]. These features have not been incorporated into existing architectures.

Proteus uses tightly-coupled clusters connected in groups. Communication within a group is through a crossbar connection. Communication between groups is through circuit switched enhanced hypercube connections. A separate control network of buses within each group, and ethernet among groups allows additional control and communication.

Proteus is designed for large granularity pipelined, distributed, or parallel processing applications. Tasks can be partitioned to a single processor, a subset of processors or all processors in a pipeline or distributed fashion. It is also possible to set up multiple processor pipelines to perform different tasks. Example applications include image processing, fast Fourier transforms, and low to high-level vision. We present unique features of Proteus system in Section 2, the architecture in Section 3, applications in Section 4, and conclude in Section 5.

2.0 Unique Features

The special features of Proteus can be demonstrated via a comparison with other recently developed Supercomputers. We focus on communication and processor clustering. Proteus has been designed to balance the interconnectivity and the processor clustering so that maximum utilization of both processor and communication network is achieved. We give a brief overview of the unique developments in Proteus below.

2.1 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 consists of clusters that each contain 36 processors. The nodes are connected in an enhanced hypercube structure. An enhanced hypercube (EHC) contains two links in any one dimension of a regular hypercube, as shown in Figure 1. 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 EHC 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 [9]. By clustering processors at each node the

TH0363-2/91/0000/0287$01.00 © 1991 IEEE

Proteus architecture improves the link utilization. Detailed descriptions of the communication network and the EHC are given in section 3.2.

2.2 Allocating Caches

Clustering of processors together, 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 [17]. 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 % [16]. The allocating cache is a high performance interconnect that is much more general than the register memories used in the Orthogonal multiprocessor (OMP) [10], 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 architecture section.

2.3 Fault Tolerance

Initial design goals focused on the incorporation of limited fault tolerance. By requiring general connectivity of clusters, and the arbitrary assignment of jobs to processors, system level fault diagnosis [15] 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.

3.0 Architecture

3.1 Goals

Proteus design is medicated on the use of higher granularity. This constrains data movement to be in large blocks between processors. Blocks of data are typically images, which should be routable in varied and arbitrary graphs. The performance is related to the load balancing and the utilization of processor resources.

The research goals are: architecture, partitioning, load balancing, and algorithm mapping. The system was designed to accommodate 256 to 1024 processors. We present an overview of the architecture followed by a discussion of the design.

3.2 Organization

Proteus is a scalable EHC based computer system with a large number of processors at each node. Unlike the NCube, iPSC 860, and the FPS T Series [5] Proteus has 36 processors at each node. The system 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 EHC nodes. The communication across the hypercube, within groups, and control of all processors is described in this section.

3.2.1 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 EHC 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 1 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 also 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 to ease contention and multiprocessing performance degradation.

3.3 Communication

The communication structure is hierarchical to share resources and distribute control overhead. Currently, com-

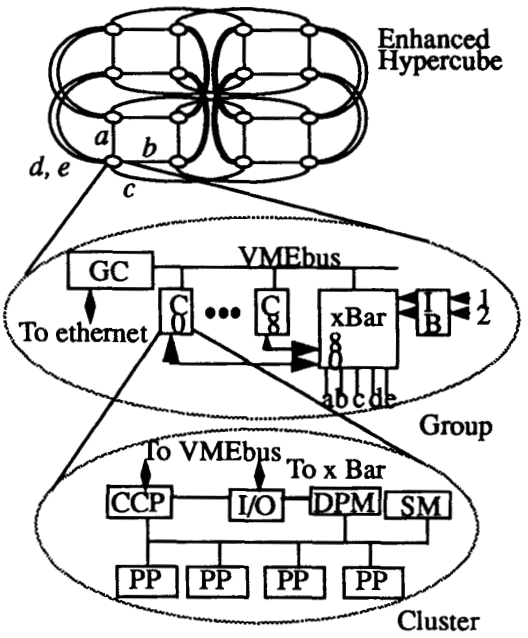

Figure 1. Exploded View of Proteus System

munication 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 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 2. If $j_1=j_2$ (then $k_1=k_2$) and cluster j_1 arranges for data transfer through the shared memory. If $j_1 \neq j_2$ 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 EHC. 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.

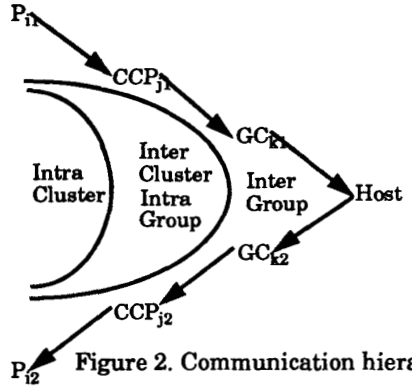

Figure 2. Communication hierarchy

3.4 Control

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 3.

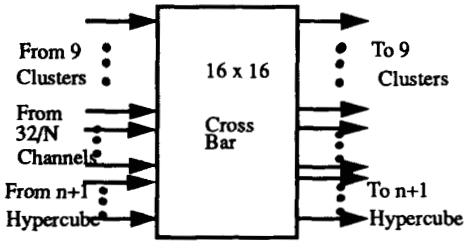

Figure 3. Crossbar Connections

The group controller is a single processor Unix board equipped with the VMEbus and ethernet interfaces. It 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. 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 [17]. 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.

3.5 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 as investigated include 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 64 bit data bus and 12K bytes of on-chip cache memory. Analysis by Levy [11] showed the i860 to be poor for operating system work, so extra care was taken in design to minimize interruption of the i860's processing. Specifically, another control processor was chosen to take care of scheduling and interrupt handling, i960.

Interconnection and I/O. Investigation into interconnection schemes and I/O to handle the high input rate 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 took place at the I/O interface. Input data was stored in shared memory using 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 [6] and a fast cross bar chip from Gigabit logic [7] 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 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	EHC 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 3.

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 while the data were transferred and the local memory is a fixed size. However, local memory was advantageous when processing created 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 [17], 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.

3.6 Software

Software design proceeded with the hardware design.- Hardware and software groups worked closely to optimize performance. The application software includes an interpreter, translator, and debugger. System software was developed to load programs, and control communication. Figure 4 shows interaction among the modules.

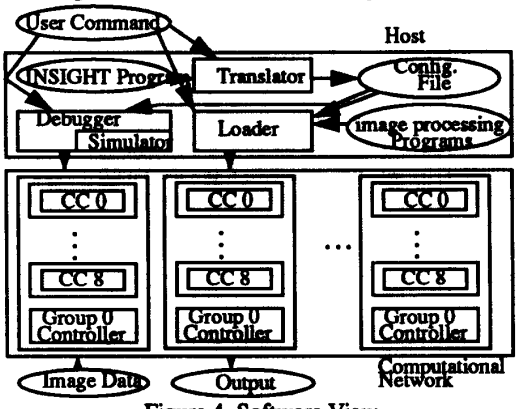

Figure 4. Software View

INSIGHT is a dataflow language in the LUCID family of languages developed by Shapiro and Haralick [13][14]. INSIGHT operates on sequences of values. In the original INSIGHT, the values were expected to be pixels of images or cells of data structures. In the version of INSIGHT running on the Proteus system, the values are whole images or whole data structures. The INSIGHT program describes the flow of a sequence of images and resultant structures through the Proteus network. Each node of the network performs one or more operations on its input image(s) and/or structure(s) to produce output image(s) and/or 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 5 illustrates the graph structure for a program. This graph must be mapped onto the Proteus hardware.

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 are decomposed into sequences of nodes that use smaller structuring elements [18]. 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 processing elements, with a specified interconnection network and interprocessor communication costs, how can the operations be partitioned among the processors to gain maximum throughput. Initially we chose a greedy technique which is similar to 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 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

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

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 maximum load that this new assignment would create is generated. If any of these loads are less than the current maximum load, then one of the movements that reduces the maximum load is completed 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.

An additional software module, the loader, is necessary to set up Proteus for execution. The loader runs on the host, accepts the user's requests, downloads the pre-stored linked object codes, the scheduling of jobs, and the network configuration control codes to the pipeline hardware units, and finds the mapping from symbolic processor names to physical processors.

For program and hardware debugging a parallel debugger is, PBUG was developed. PBUG is a window-based application that runs on the host. It allows the user to control the executions of both the simulator and the subject INSIGHT program (on the Proteus system), and to visualize and verify the results produced by the Proteus system. It is being implemented in Ada, using VADS on a Sparcstation under Unix. PBUG is implemented as two communicating processes, one on the host and the other one on the Proteus system and interacts with the user through the host's window system.

The low-level software support for the high-level programming environment is the processing library. As an example, the image processing library contains the Intel i860 code for the operations in the INSIGHT application program. The initial set of functions in the library include arithmetic and logical operations on images, geometric spatial transforms, morphological operations, neighborhood operations, connected components, and masking.

4.0 Applications

Computer vision and image processing have task graphs that may be pipelined, and parallelized. Such problems can be partitioned into large chunks of data, which is what Proteus is targeted for. To translate the system vision application task to the associated task graph, the proteus programmer encodes his algorithm in INSIGHT, a relational dataflow language. The INSIGHT program is translated to a graph structure which is mapped to the Proteus architecture. The symbolic mapping produced by the translator specifies the symbolic processors to which image processing operations are assigned. The code for the operations is stored in a library on the host. A loader program performs the mapping from symbolic processors to physical processors in the Proteus network and downloads the code for each processor and the appropriate control codes for configuration of the network to execute the algorithm.

Software controller programs running on the cluster controllers synchronize the flow of data between processors and control the execution of the processors in that cluster. Software controller programs running on the group controllers handle transfer of data from the input, between clusters, and between groups.

Figure 5 illustrates an INSIGHT program for the Proteus system. The input to the program is 512x512 gray tone image G0, and the output is a 512x512 binary image B4. Intermediate graytone 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 graytone 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 [8] with a structuring element that is a box (rectangle) of dimension 5 x 5, with the result becoming gray tone 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.

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 going to successive processor blocks until all processors have been used. If the number of proces-

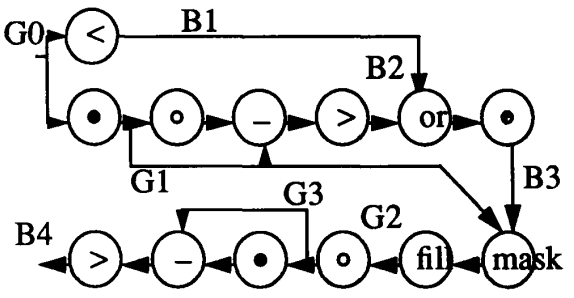

Figure 5. Insight Program

sors that the algorithm is divided among is not a multiple of four, them the replicated mapping may not be exactly the same.

The processing and computation in Proteus uses a variety of software and hardware control mechanisms. Each pixel processor 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 completion record and interrupts the cluster to report the results. After receiving the interrupt signal from the pixel processor, the cluster controller reads the completion record to get the information the pixel processor, updates the status of data regions due to the task just completed, and continues to activate sleeping processors.

5.0 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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] H. Levy, Personal Communication.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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