12

September 3, 1993

# Design of Parallel Systems for Signal Processing

REPORT FOR ONR/NRL PROJECT N00014-92-K-2018

Edward A. Lee
Associate Professor
University of California at Berkeley

DEPARTMENT OF ELECTRICAL ENGINEERING
AND COMPUTER SCIENCES
UNIVERSITY OF CALIFORNIA AT BERKELEY

## SUMMARY

This report concludes phase I of the project. Two significant accomplishments are reported, the first relating to the design of heterogeneous parallel systems, and the second related to the synthesis of parallel programs for the CM-5 from Thinking Machines.

A methodology for design and evaluation of parallel architectures with heterogeneous components has been developed. Specifically, the "communicating processes" (CP) domain in Ptolemy has been used to model heterogenous parallel hardware systems. Our first demonstration design uses the "ALPS" concept (Alternative Low-level Primitive Structures). Use of the CP domain is a departure from our proposal, where we had proposed to use a discrete-event model. We have determined, however, that the CP model is considerably more convenient for high-level design of heterogeneous parallel hardware. The CP domain in Ptolemy has a multitasking kernel managing concurrent processes. A process, or thread, is created for each computational and communications system resource in a particular design. The CP model of computation makes it easy to write and interconnect a variety of computational resources and simulate the execution of an application to evaluate the particular architectural configuration.

We have developed a Target in Ptolemy for the CM-5 from Thinking Machines Inc. The processing nodes of the CM-5 (Sparc processors with dedicated network interfaces) match well Ptolemy's large grain parallelization approach. Ptolemy will map an application fitting the synchronous dataflow (SDF) model of computation onto the processing nodes and schedule the computation and interprocessor communication (IPC). A variety of different algorithms may be used to perform the mapping and scheduling. Ptolemy generates C code for the computation and IPC; this code is then compiled and executed. The IPC is performed using Active Messages (CMAM) from Prof. Culler's group at Berkeley.

Both results are preliminary in some sense. The hardware design methodology needs a more systematic methodology for evaluating particular hardware configurations. The CM-5 code generation needs better expression of data parallelism in the original graphical application specification and adaptation of our schedulers to exploit this data parallelism.

18 PG

## A. The Design of Heterogeneous Parallel Systems

An architectural approach, called ALPS (Alternative Low-level Primitive Structures)[Wu84], consists of several system resources connected via a "message circus" and a "data circus" as shown in figure 1. A system resource consists of an ALPS primitive structure coupled to a standard system interface called the Interface Control Unit (ICU). Such a standard interface makes it possible to have a wide range of hardware resources within a single application - the ICU handles the discrepancies in the requirements of different resources.The ICU implements the control protocols necessary for data communication, primitive activation, and deactivation. This distribution of control, made possible by a self-synchronizing network architecture, avoids the bottlenecks associated with centralized control and synchronization.

The signal flow graph (SFG) describing the application is loaded into each ICU in the form of a routing table, which acts as a look-up table for task activation. Data associated with an arc between two nodes of the SFG is characterized by its source and destination node pair — called the virtual circuit number (VCN). This VCN, along with an identifier as to the 'type' of resource needed by this data, constitutes a message control token (MCT).

A source node, upon completing execution of its task, broadcasts the MCT(s) to all the system resources via the message circus. A 'free' resource of the requested type grabs this token and

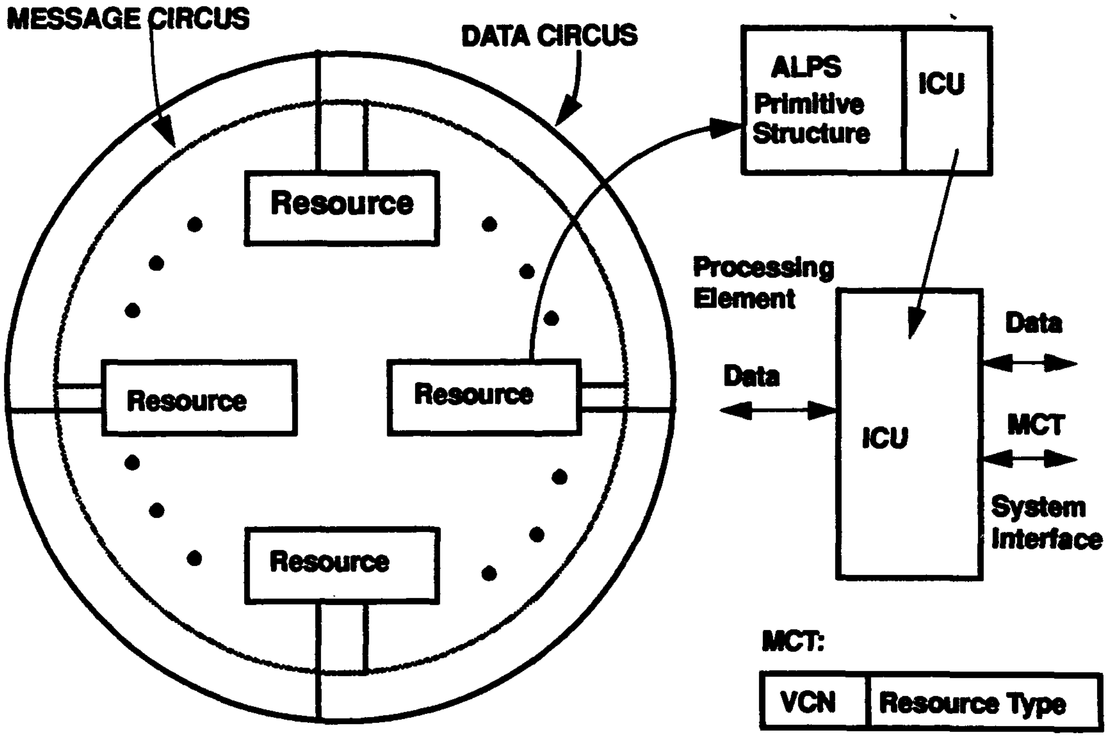


FIGURE 1. System Interconnections and Resource Structure for the ALPS Architecture.

sends an acknowledgment to the source, which then transmits the data on the data circus. After transmitting the data, the source becomes free for subsequent computations. The sink processes the received data and initiates the next transfer.

The protocol supports multiple sinks (multicast, or forking of data), variable-length data, and multirate processing. If an appropriate sink is not found, the MCT is re-transmitted some time later. The design is highly modular due to the standard interface, and can accommodate more or less arbitrary combinations of processing elements.

In comparison with the SPRiNG [Pic89] architecture, where the tokens pass 'through' each resource along the ring instead of being broadcast over the circus as in ALPS, ALPS has a better fault-tolerant behavior. In the case of SPRiNG, if a resource is damaged it can not send the data further along the ring; hence, even if there is another resource of the appropriate kind, the processing halts. With ALPS however, if there is another resource, the source re-transmits the MCT and the processing continues when the resource becomes free.

## A.1 Requirements of Heterogeneous Application-Specific System Design

A key difficulty in exploiting concepts such as ALPS in application-specific systems is the complexity of the design process itself. It is not appropriate in such designs to disassociate algorithm, hardware, and software, as is traditionally done. Instead, all three should be designed and evaluated together. The performance of the hardware should be assessed for variations of the algorithm. And the behavior of the algorithm should be assessed in the presence of hardware faults.

To accomplish this holistic design style, we require a software environment that contains at least the following capabilities:

- Simulation for high-level modeling of hardware and communications.
- Ability to execute signal processing algorithms on the high-level hardware model.
- Ability to simulate hardware faults and evaluate the impact on the algorithms.
- Support for detailed hardware design of the components of the system.
- Ability to synthesize software for programmable components.
- Ability to simulate the detailed hardware design executing the synthesized software.

Another desirable capability would be integrated hardware synthesis, to support the design of specialized hardware resources in an ALPS configuration.

All of the above capabilities are consistent with the basic design of Ptolemy. Some already exist, while others have a fairly clear development path.

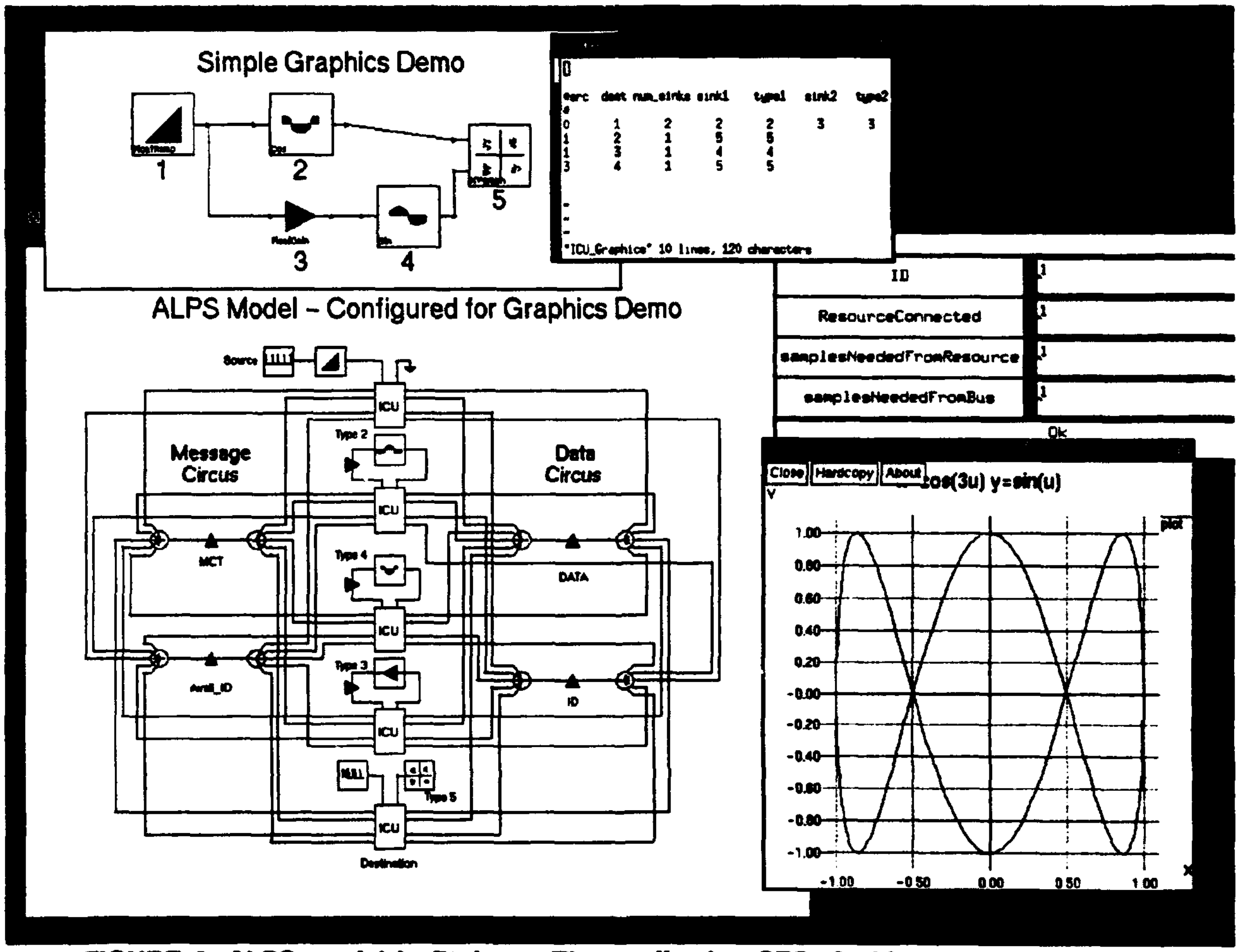
## A.2 First Version — Discrete-Event Simulation of ALPS

The first version of our simulation environment for ALPS architectures used the discrete-event (DE) model. Figure 2 shows a five-element ALPS system executing a simple algorithm specified as a signal flow graph. The algorithm, shown in the top left corner window, generates the plot in the lower right window. The table shown next to the SFG that specifies the algorithm defines the mapping of computations onto computational resources.

A-1

The DE model of the ALPS architecture is shown in the lower left window. Each ICU executes the interface protocol as described above. It has a number of parameters that can be controlled; for example, the ID of the block, the type of resource (Source, Sink, or Processing primitive) it is connected to, the number of samples it receives from the bus, and the number of samples produced by the computing resource. This modest amount of programmability in the ICU is justified because of the high modularity it supports via various types of primitives connected to it. On the Message circus, in the left half of figure 2, each ICU sends out MCTs and acknowledgments. Data transfers are managed by the Data circus that appears on the right side. The plot in the lower right window results from executing this algorithm on the hardware model.

This over-simplified example illustrates the various components involved in developing a model for the ALPS architecture. Different types of functional resources can be connected to each ICU. Each ICU has a pre-settable parameter that identifies the type of computational resource it is connected to. Using this information, the ICU responds to MCTs if the requested



```
0
#src  dest num_sinks sink1   type1  sink2  type2
#
0      1      2       2       2      3      3
1      2      1       5       5
1      3      1       4       4
3      4      1       5       5
-
-
-
"ICU_Graphics" 10 lines, 120 characters
```

| | |
|---|---|
| ID | 1 |
| ResourceConnected | 1 |
| samplesNeededFromResource | 1 |
| samplesNeededFromBus | 1 |

Ok

FIGURE 2. ALPS model in Ptolemy: The application SFG, Architectural Model, Hash Table, and Simulation Results. This implementation uses the discrete-event domain.

resource type matches the type of resource it is associated with. In this demonstration system, the compute resource is modeled as a block in the Synchronous Data Flow (SDF) domain, while the overall interconnection of the ICUs at the system-level is managed by the timed DE domain. The wormhole concept of Ptolemy is used to maintain the interface between the SDF resource and the DE architectural model. A wormhole is a block nested within a parent application, that outwardly appears to be a monolithic block belonging to the same domain as the parent domain, but internally contains an application that belongs to a different domain. The differences in scheduling mechanisms for the two domains are handled at the wormhole interface. This multiparadigm simulation capability permits the coexistence and interaction of different models of computation. In the ALPS model, the various resources are SDF wormholes in the DE parent domain. As a result, it becomes possible to simultaneously simulate the hardware (data and control flow between various ICUs) and software (the functionality of each resource) for architectures such as ALPS.

Our experience with this simulation methodology, however, led us to another approach. The discrete-event model of computation turned out to require rather complex code to specify the functionality of the ICU and computational nodes. The basic reason is that in an event-driven model, the model of a component is invoked in response to events in the simulation environment without concern for its history. The history must be stored locally in the model for each component, making the designer responsible for managing complex, low-level control that has little to do with the functionality he or she wishes to model. Consequently, we developed an alternative model using a "communicating processes" (CP) model of computation. The resulting component models are much simpler and more intuitive.

## A.3 Communicating Processes Simulation of ALPS Architectures

The Communicating Processes (CP) domain in Ptolemy has been implemented by Seungjun Lee (in Prof. Rabaey's group) to provide a hardware-software co-simulation environment. It has a multi-tasking kernel that simulates concurrent processes using the Sun lightweight process library[1]. An autonomous task, or thread, is created for each functional module in the simulation. The execution of multiple threads are supervised by a customized process scheduler.

The architecture described in the report consists of a set of ICUs and network links connecting the ICUs. They execute concurrently, and communicate with one another by sending and receiving data synchronously. Consequently, they are most naturally modeled as a set of concurrent processes communicating with one another by message passing. Because each process runs independently from the other processes, except where there is explicit inter-process communication, the components of the system are easier to design and more intuitive than the discrete-event version. Conceptually, the simulation for each component runs continuously, reading and writing to its communication channels. Unlike the discrete-event versions, no code is required to required to explicitly store state and manage startup. The reason is that when a process suspends and restarts, it restarts exactly where it left off. In the DE domain, by

1. The design of the CP domain is isolated from the details of the Sun Lightweight process library using object-oriented principles, so that other multitasking kernels can be substituted instead.

contrast, when a process restarts, it restarts at the beginning, and it is up to the designer to figure out where it had left off when it was suspended. We now give a detailed description of this implementation.

### The Interface Control Unit

The ICU connects a processor to the rest of the system. It communicates with other identical ICUs and with the processor itself. Each ICU is divided into two discrete modules, the Network Interface Unit (NIU) and the Processor Interface Unit (PIU) as shown in Figure 3. All packets passing into or out of a node are handled by the NIU. The NIU uses the clocking scheme specified by the physical layer. Each NIU is comprised of three queues; an output (transmission) queue, an input (acceptance) queue, and a bypass queue, as shown in Figure 4. The input queue and the bypass queue are strictly FIFO, but the output queue is capable of reordering its members so that high-priority packets are not delayed by lower-priority packets.

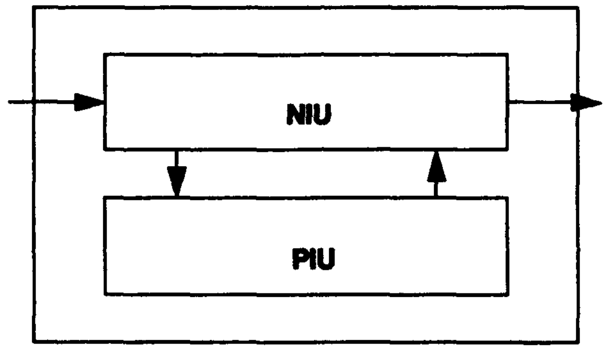


FIGURE 3. The Interface Control Unit (ICU)

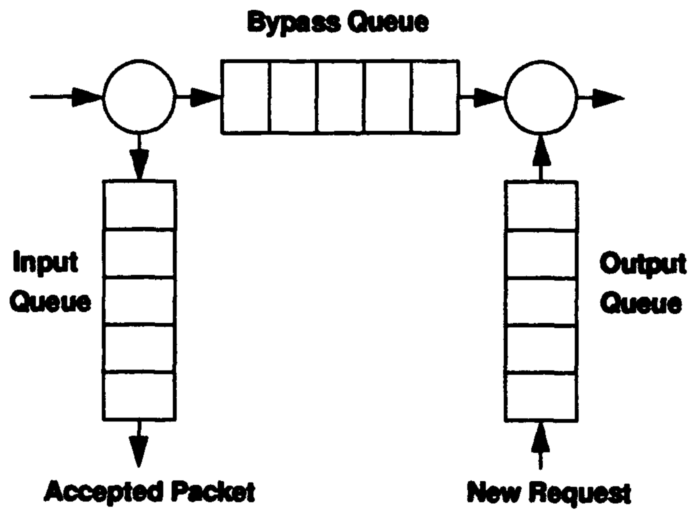


FIGURE 4. Queuing model of a node.

The queue size of the input queue and the bypass queue is configured at 32 words in our simulation, the maximum length of the packet size.

When a packet traveling along the ring arrives at a node, it is routed into either the input queue (if destined for that node) or the bypass queue. In the former case, a short echo packet is conditionally generated and submitted to the bypass queue. Packets in the bypass queue contend with those queued for transmission in the output queue for the link to the next node. The bypass queue is given non-preemptive priority over the transmission queue. Each PIU runs at the clock frequency of the processor to which it is interfaced.

The PIU removes and decodes packets from the input queue of the NIU and appropriately handles the data contained therein. Also, the PIU packetizes data generated by the processor, associating it with the proper control information, and placing it into the output queue of the NIU for transmission. The PIU is programmable.

## Physical Layer

The physical layer is modeled logically, not electrically. The relevant signals are:

- Din<0...15>: incoming data arrives on these lines in packetized form. Since the network protocols are synchronous, idle symbols arrive on these lines when no data packet is present.
- Dout<0...15>: Outgoing data is sent on these line. When there exists no packet to send, idle symbols are transmitted.
- Fin: Incoming flag signal to detect packet boundaries. It is LOW when the incoming link is idle. It goes HIGH to indicate the start of a packet on the Din line. It goes LOW again at the end of arrival of the fourth to last word of a SEND PACKET or last word of an ECHO PACKET. Note that this will increase hardware complexity.
- Fout: Outgoing flag signal.
- Cin: Incoming clock signal.
- Cout: Outgoing clock signal.

## Clocking Mechanism

Because inter-processor communication is achieved through the synchronous interaction of the network circuitry on each node, the clock frequency of each network interface unit must be the same. One node (the clock master) generates the clock signal and each NIU forwards this signal to its downstream neighbor.

## Logical Layer

The ICU supports the following transaction classes:

- Echo-less: This is a non-robust communication mechanism, in that no error recovery is possible. However, it also entails less overhead. A "scrubber" removes packets with erroneous Target IDs from the ring. If a packet arrives at a node that does not have enough space in its input queue, the packet is removed from the ring. No source node buffering is required.

- **Echoed**: This provides a more robust transaction mechanism. Upon receipt of the packet at the target node, an echo is returned. Upon receipt of a "busy echo", the source node must retransmit the packet. If the packet or echo becomes corrupted, the source node generates a time-out and retransmits.
- **Multicast**: To initiate multicast, a node transmits to itself. Participating nodes (those belonging to the multicast group) then copy the packet as it passes but do not strip it from the stream.
- **Resume_Multicast**: When a busy echo is returned from a echoed multicast, the multicast may be re-transmitted, but it is directed to the node which sent the echo. Upon accepting the re-sent packet into its input queue, the target node replaces the Target ID of the packet with the Source ID, and converts it into a normal multicast.

## Arbitration

The ICU assigns packets to any of eight groups. The group to which a packet belongs will be specified in the first word of the packet. Eight bits of this word are used for the destination address (up to 256 nodes) and each of the remaining eight bits may be used to uniquely represent a group. A packet belonging to a given group would have the corresponding group bit set in the first word of its header. A packet should not belong to more than one group. Send packets are always appended to idle symbols that contain a go bit corresponding to the group to which the packet belongs. While emptying its bypass FIFO, a node records the arrival of an idle symbol containing this (and any other passing go bits) in its idle_accumulator. This register must eventually contain the required go bit before the node will be allowed to transmit its packet. When the node's bypass FIFO becomes empty and it has a packet to transmit, it may do so only if its idle_accumulator contains the required go bit. In this case, the contents of the idle_accumulator are merged (using logical OR) with the go bits of the next idle symbol and it may follow that idle with its packet. If the required go bit has not been recorded, group bits of a passing packet must still be recorded, and the relevant go bits of arriving idles blocked. Any go bits contained in the idle_accumulator that are not represented in the block_register are released. Upon transmission of its own packet, a node's block_register and idle_accumulator are cleared and, until it wishes to transmit another packet, idle symbols are passed along unchanged. If a previously idle node suddenly wishes to transmit a packet, it may do so if the last emitted idle contains the required go bits. Otherwise, the node is considered blocked and the protocols described previously are employed. If, upon acceptance of a packet into its input FIFO, a node neither has a packet to transmit nor has a full bypass FIFO, the preceding idle symbol is multiplied. This idle extension stops when the next packet or idle symbol arrives.

## Packet Formats

A packet consists of a three-word header, a cyclic-redundancy code, and sometimes a main body, as shown in Figure 5. Information contained within the main body of a packet is specified in the transport layer. The fields are:

- **target_id**: Local ringlet destination.
- **arb_grp**: The packet's arbitration group.

- **command**: The packet type, send class, multicast information, length, and some flow-control and status information.
- **source_id**: Local ringlet ID.
- **ret**: Incremented when retransmitting a packet.
- **seq_#**: Sequence number. Useful for proper ordering of piped data.
- **main body**: Packet sizes 4, 8, 16, or 32 words. This corresponds to main_body sizes of 0, 4, 12, and 28. Echo packets do not have a main_body and are thus 4 words.
- CRC: Check sum.

## Command Word Formats

A command word format is shown in Figure 6. The fields are:

- **old**: Initialized to 0 by the source node and set to 1 by scrubber. It is the responsibility of the scrubber to remove from the ring any packets it receives with old bit set to 1.
- **echo** (bit 8): Set to 1 for echo packets and to 0 for send packets.
- **len**: 00, 01, 10, 11 respectively mean 4 words, 8 words, 16 words, or 32 words.
- **bsy**: Set to 1 if the issuing node's input queue was full upon receipt of the corresponding send packet.

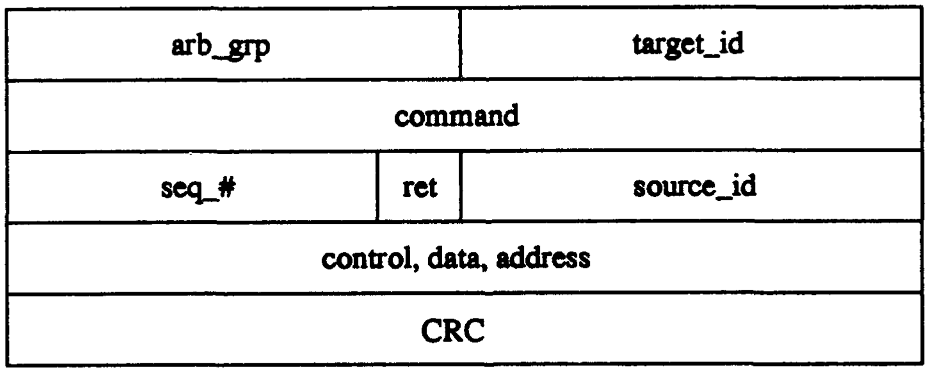


FIGURE 5. General packet structure.

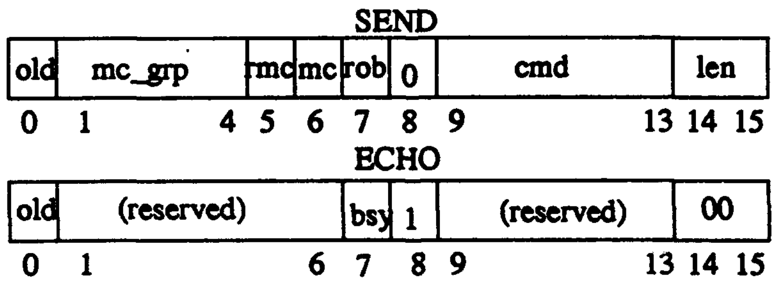


FIGURE 6. Command word formats.

- **cmd**: Transport layer command being carried out specified in the cmd field of a send packet.
- **rob**: Set to 1 by the issuing processor if echo is desired.
- **mc**: Set to 1 to send a packet to multiple destinations. When set, the mc_grp field contains the multicast_group to which the packet is directed.
- **rmc**: Set indicates resumes_multicast. Target node clear this bit, set the mc bit, replace the target_id with the source_id.

### Transport Layer

Each node maintains a pipe_directory. This is necessary for static operations. The pipe_directory contains the following:

- An indication which SFG tasks it can perform;
- The input and output pipe_identifiers associated with each task;
- The nodes between which the pipe is currently established (i.e. source and sink nodes).

The pipe_directories are sometimes used during system initialization or during dynamic scheduling or reconfiguration. All should be implemented as echoed transactions.

## A.4 Architecture Simulations

Figure 7 shows a simulation of a network with 4 ICUs. The Link block connects two adjacent ICU nodes. The signal flow graph representing the signal processing application to be executed on the simulated architecture is attached to the PIU of the ICU node. The NIU, the PIU, or Link is considered as an atomic block (process) for simulation. Processes interact with each other by message passing through ports. The behavior of a process is described in C++.

### Specification in CP domain

All the processes are assumed to run concurrently. A process keeps running until any one of following occurs:

- It blocks trying to communicate with another process.
- It suspends itself for a certain amount of simulated time.
- It terminates.

A blocked process resumes as soon as the condition which caused the blocking of the process is resolved, for example the communication channel becomes available, or an input arrives. A suspended process resumes after the given simulated time period has elapsed. This is used to model computation times and delays in the system.

Processes communicate with each other by passing message through ports. Each Ptolemy star (a process) may have several input ports and output ports. A channel connects an output port to an input port. Only one-to-one connections are allowed. Channels can be buffered or unbuffered. A buffered channel has a FIFO queue with either finite or infinite capacity.

A port is characterized by a data type that it carries and a port protocol. The port protocol specifies the behavior when a channel is full or empty. Four different protocols are supported

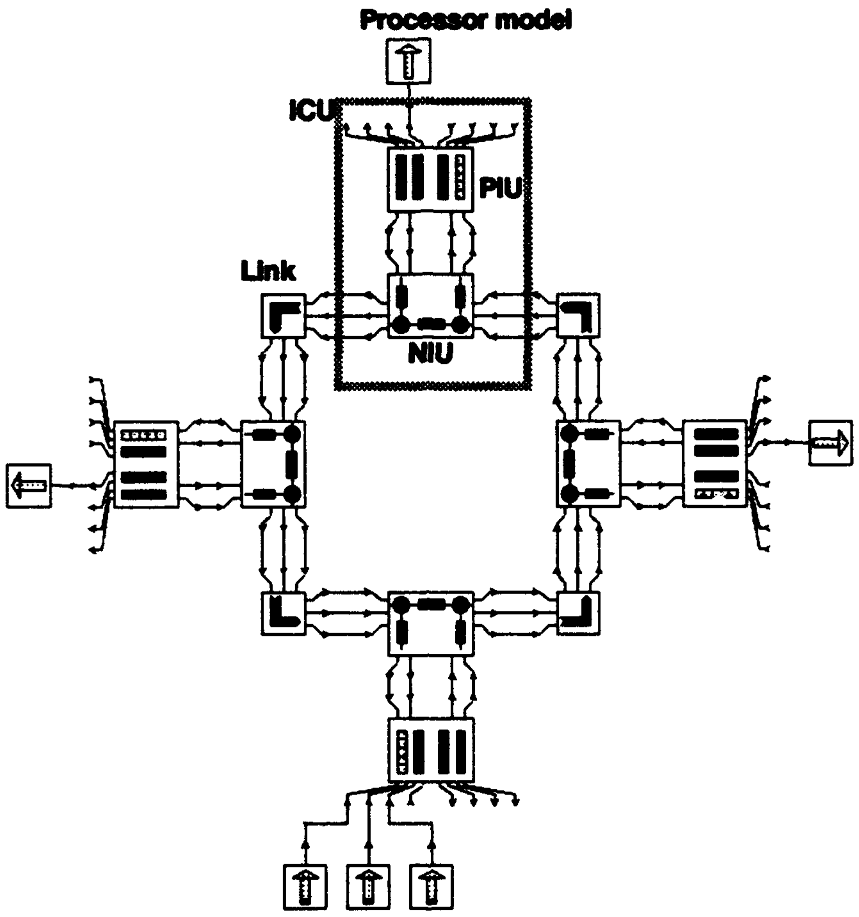


FIGURE 7. A simulated network with 4 ICU nodes.

for each input and output port. An output port can either block on full, block on full with time-out, overwrite on full, or ignore on full. An input port can block on empty, block on empty with time-out, re-use the previous packet on empty, or ignore on empty. The port protocol is fixed at setup time and cannot change dynamically.

The following primitives provided by the CP domain are used to build up the simulation:

- The method **msgSend** (data, outPort) is used to send data through the outPort, while **msgReceive** (data, inPort) receives data through inPort. If the data transaction cannot be made immediately, the process may either block or proceed, depending on the protocol specified on the port.
- **TMsgSend** (data, outPort, time-out) and **TMsgReceive** (data, inPort, time-out) can be used to describe the behavior, "block on full/empty with time-out". They return after a time-out, indicating whether the communication has succeeded. They are only meaningful when they are called on the ports with BLOCK mode protocol.

- **waitFor** (timePeriod) suspends the process until timePeriod passes. It is used to model execution delay.
- **waitAll**() and **waitAny**() wait on a set of input or output ports, and return only when all or any of them become ready for communication. They take an arbitrary number of ports as their arguments, and return the number of ports available for data transaction.
- **waitOne**() describes non-deterministic behavior. It returns any port which is ready for a transaction. When there is more than one port ready at the same time, one is selected randomly and returned.
- **TWaitAll**(), **TWaitAny**(), and **TWaitOne**() have the same functionality as **waitAll**(), **waitAny**(), and **waitOne**() respectively, except that they take one more argument as time-out. They will return when the time limit expires even if there is no ready port.

## Component Star Implementation

The NIU star has two receive sections and two transmit sections. In between these two sections, an execution section is implemented to perform network protocol. The NIU receives data from the Link star (Din, Fin, Cin) every cycle, synchronously, and receives data from the PIU (Din, DinValid) only if DinValid is active. The data valid signal is required in order to synchronize because the cycle time is typically much longer with the PIU than the NIU. The NIU star transmits to the Link star (Dout, Fout, Cout) in every cycle and transmits data to the PIU (Dout, DoutValid).

Similarly, the PIU has two receive sections and two transmit sections. Since the PIU can connect to processing elements with different execution cycle times, the PIU communicates with the processing elements using a non-blocking mode. Simple processing elements are attached in the simulation shown in figure 7, but more complex processors can be used. The cycle time of the PIU is implemented using the waitFor(interval) construct. The cycle time is static throughout a simulation, in this implementation.

The Link star simply passes the data through. Its only purpose in the simulation is to monitor the network utilization.

Communication between the NIU and the Link star is non-blocking. Communication between the PIU and the NIU, and the PIU and the processing elements is also non-blocking, but with a **transReady** construct to prevent the process from hanging when there is no input or output available

## Operation

A simulation example with four ICU nodes is shown in Figure 7. Each ICU is connected through the Link star which synchronizes data transfer between the nodes. At the PIU side of the ICU node, different types of processing elements are attached to generate and consume data in order to simulate the signal processing application.

During initialization, each ICU reads in routing information and the pipe directory information from a schedule file provided by the user for the particular simulation. In this file, there are eight fields to be completed by the user in order to start simulation. Our plan in the future is that this file will be generated automatically from a high-level description of the signal processing function to be performed. For the simulation shown in figure 7, the contents of this file are:

| ID | Pipe | Dest ID | Trans Type | multicast group | Source/ Sink | Packet Size | multicast number |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 0 | 0 | 0 | 16 | 0 |
| 1 | 2 | 3 | 0 | 0 | 0 | 16 | 0 |
| 1 | 3 | 4 | 0 | 0 | 0 | 16 | 0 |
| 2 | 5 | 1 | 0 | 0 | 1 | 16 | 0 |
| 3 | 5 | 1 | 0 | 0 | 1 | 16 | 0 |
| 4 | 5 | 1 | 0 | 0 | 1 | 16 | 0 |

The ICU reads in schedule information only if node ID matches with the ICU node ID. In the above example, there are three lines of schedule information for node 1 and one each for the others. The pipe number indicates the which pipe is used for the connection given by the connect ID. There are four input pipes, 1-4, and 4 output pipes, 5-8. The transaction type indicates what type of packet transaction should be done (0: echoed transaction, 1: echoless transaction, 2: multicast, and 3: resume multicast). Processing elements belong to the multicast group have same multicast group ID. In case of echoed or echoless transaction, the multicast group ID is

```
Processing Element Type
        Pipe Number: 1
        Pipe Type: SOURCE
        Connecting Node: 2
        Transaction Type: ECHOED
        Packet Size: 16

Processing Element Type
        Pipe Number: 2
        Pipe Type: SOURCE
        Connecting Node: 3
        Transaction Type: ECHOED
        Packet Size: 16

Processing Element Type
        Pipe Number: 3
        Pipe Type: SOURCE
        Connecting Node: 4
        Transaction Type: ECHOED
        Packet Size: 16
```

**FIGURE 8. Schedule information for node 1 generated by Ptolemy**

ignored. In the source/sink column, 0 indicates that the pipe is a source and 1 indicates that the pipe is a sink.

The NIU stars read routing information and the PIU stars read pipe directory information. The routing information includes arb_grp, mc_grp, and transaction type. The node_id is set by the user.

The cycle time is set by the user for individual components before the simulation. The NIU cycle time value is set to 1 under the assumption that it is designed to not be a system bottleneck. All of the units are operating in non-blocking mode; that is, the units do not wait for data if data is not available. Once the schedule information read in by the ICU, Ptolemy generates the confirmed schedule information as shown in Figure 8.

Every queue in the ICU can be monitored. Using the simulation, the actual queue size needed in a hardware implementation can be determined as shown in Figure 9. The pipe capacity is directly related to the amount of memory required in PIU. Hence the pipe capacity monitor provides the required memory size. The link monitor displays the network utilization for the particular application mapped onto the simulated architecture, as shown in Figure 10. Here, a "1" indicates that the link is busy, so that the burstiness of the traffic can be read at a glance.

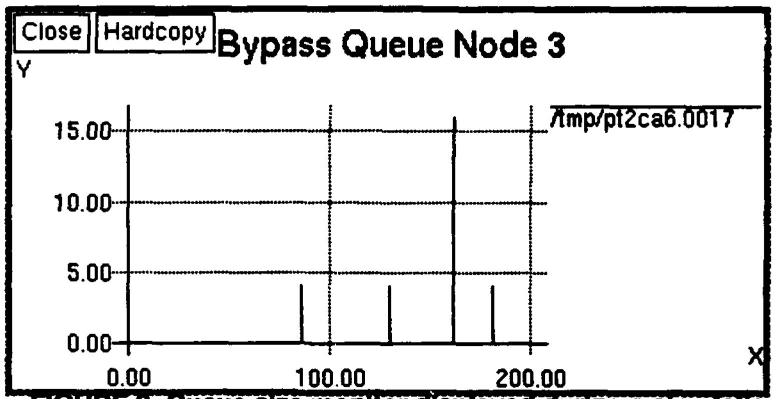


FIGURE 9. Queue size monitor displayed during a simulation

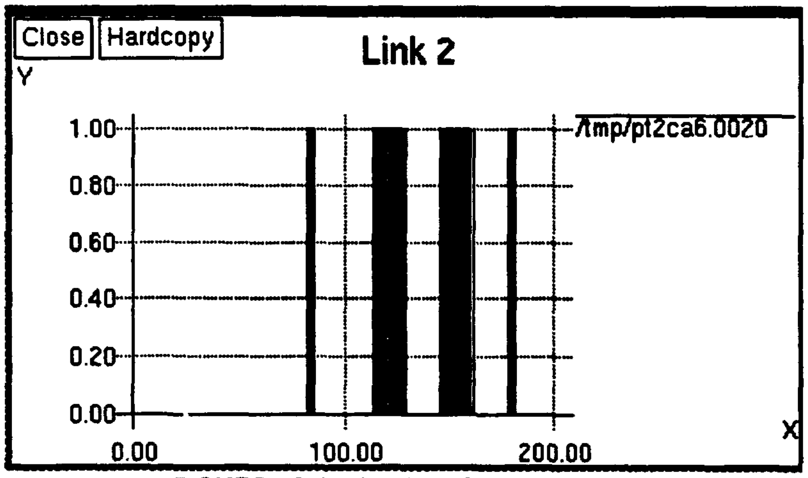


FIGURE 10. Link utilization display

## B. Parallel Code Generation

A code generation Target in Ptolemy for the CM-5 from Thinking Machines Inc. has been developed using the parallelizing schedulers developed in our group [Sih93a,b]. These schedulers easily support the variable communication latency present in the "fat-tree" communication architecture of CM-5. The synchronous dataflow (SDF) model of computation is used. Ptolemy generates C code for the computation and IPC; this code is then compiled and executed. The IPC is performed using Active Messages (CMAM) from Prof. Culler's group [Eic92]

The first major test application is shown in figure 11. This is a "perfect reconstruction" filter bank, in effect performing a wavelet transform and inverse transform. We are currently using this application to measure communication overhead and assess the efficacy of the parallel scheduler. This application, however, runs fast enough for many applications on a single sparcstation, and therefore serves only as an instrumentation test.

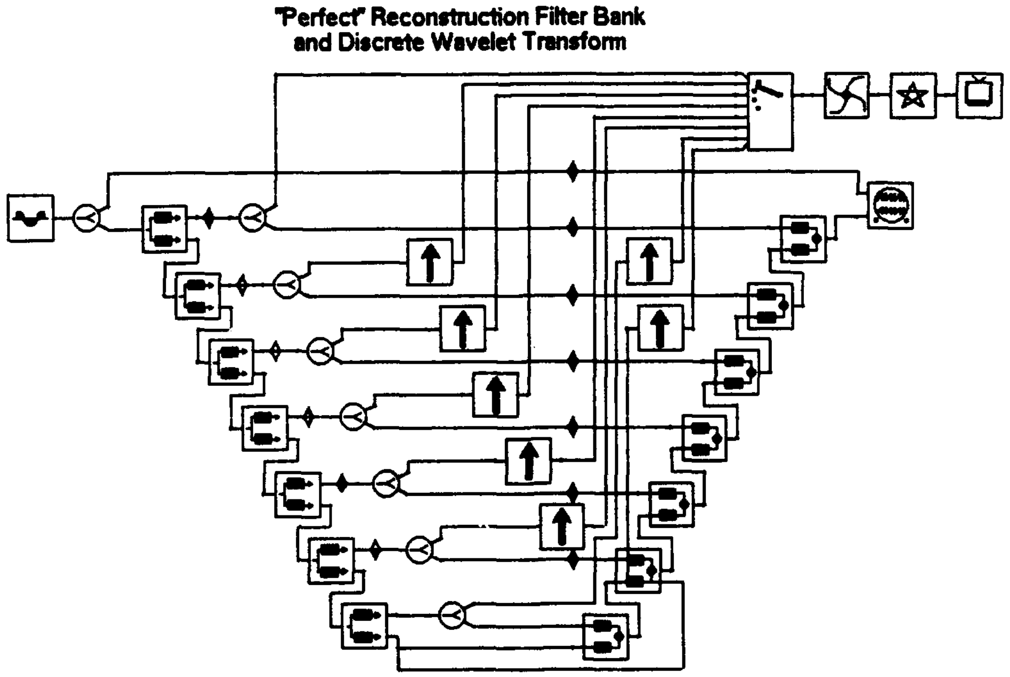

"Perfect" Reconstruction Filter Bank
and Discrete Wavelet Transform

**FIGURE 11. A Ptolemy application mapped onto the CM-5.**

## C. Further Work

### Parallel Programming

The major weakness in our CM5 code generation is its limited ability to exploit data parallelism. Part of the problem is in the program representation, and part in the scheduling. We propose to concentrate on these issues in the next phase of the project. Future plans also include general research of parallel scheduling algorithms, research into fine grain parallelization, and CM-5 specific research on alternative IPC protocols and finding weaker IPC synchronization methods.

### Representation of applications — the programmer interface

Graphical representations of high-level program structure encourages reusable, modular software, and in principle supports visualization of parallelism and real-time performance. But the efficacy of these techniques has only been proven in rather narrow domains. Some constructs, such as parameterized iteration or parallelism, are awkward to express in current graphical environments. We will explore this issue, and find graphical, textual, or hybrid representations that are natural and easy to work with. The principal deliverable for the next portion of the project will be a report comparing alternative approaches and software enhancements in the Ptolemy system that support the best of these.

### Scheduling for Heterogeneous Parallel Systems

Although application-specific processors do not require a scheduler in the traditional sense, compile-time partitioning must be performed. Since partitioning is part of any compile-time scheduling strategy, we propose to begin by adapting existing schedulers to perform the partitioning. The scheduling algorithms will be then be customized to the class of architectures. The new schedulers will be included in future Ptolemy releases.

### Heterogeneous combinations of domain-specific schedulers

An environment will be created where schedulers can be selected and mixed at will, even mixed within a single application. The objective is to support applications containing a variety of algorithms or subsystems that interact, but which have different degrees of compile-time predictability, and different models of computation. Instead of generalizing a single scheduler to support all possible subsystems, our approach will be to mix schedulers that are highly tuned for a particular domain of operation. A principal challenge is that outer level schedulers must be able to manage objects that consume multiple parallel resources under the control of inner schedulers.

### Improved Simulation and Evaluation of Embedded Systems

Our work to date with the ALPS simulation has had the primary benefit of refining our methodology for designing and evaluating parallel architectures. Very little of the result is specific to ALPS architectures, in that the same methodology can be applied to the design of other parallel application-specific systems. However, our work so far has underscored some key weaknesses in our methodology. The principal one of these is that the application description and architecture description are disjoint, and the mapping from one to another is ad-hoc. We propose to concentrate on this linkage in subsequent work.

This architectural concept of distributed resources and control can be further extended for Ptolemy simulations for the analysis of multiprocessor configurations, interprocessor communication, and scheduling. In figures 2 and 7, each computing resource is a specific Ptolemy subsystem implemented in the SDF domain. Although the sequence and timing of the execution is not predetermined, the capability of each processing unit is. This mechanism cannot be conveniently used to model programmable processors as components in the system, since the computation in the resource is hard-wired into the block diagram specification. Reconfiguring the system to execute another signal processing application would require rewiring the hardware model.

To model programmable components, we propose to identify each computing resource with a C++ object capable of executing arbitrary SDF subsystems. One simple mechanism is for the computing resource to be an instance of the Ptolemy interpreter. The SFG is divided into smaller subgraphs, each of which is executed by a copy of the interpreter. A more efficient mechanism is to define a "Target" and a "Scheduler" (C++ class names in Ptolemy) for each processor configuration, where the Scheduler partitions the SDF graph and generates the hash table at compile time. Each processor then invokes schedulable subunits via calls to the Ptolemy kernel. Again, the same concept can be applied to any parallel embedded system, not just ALPS systems.

In Ptolemy, every simulation executes via an object called a Target. The Target, in turn, invokes one or more compile-time and run-time Schedulers. For the high-level hardware model, the Scheduler is the standard Ptolemy discrete-event scheduler. However, we require that the high-level hardware model itself be the Target for the signal processing application. This will require an extension to Ptolemy, allowing one Ptolemy application to serve as the Target for another. A scheduler associated with this Target will be responsible for determining the capabilities of each programmable computing resource. A first-order solution applicable at ALPS and related architectures is to leave these capabilities unconstrained, in effect deferring all scheduling decisions until runtime. An ICU with a free computing resource will grab the first MCT to arrive on the data circus, and begin executing the SDF subsystem associated with that MCT.

Even with the above simple first-order scheme for scheduling executions, a compile-time decision has to be made about how to partition the algorithm. The finest level of granularity (one SDF block per MCT) may be much too fine, overloading communication resources with needless messages that will not enhance execution speed. Hence, the partitioning should probably be guided by a more intelligent scheduler. Fortunately, we already have a suite of three parallelizing schedulers in Ptolemy that can be used.

Simulating hardware faults using the above capability is quite easy. Execution resources can be disabled at random times, even during a computation so that data in process is lost. The robustness of signal processing algorithms in the presence of such degradations can therefore be measured. If, for example, an ALPS configuration has two dedicated FFT processors and one of the FFT modules fails, then the processing can continue with a single module. The effect of this hardware loss on the performance and throughput of the application can be studied. More interestingly, if the hardware module fails in mid-computation, the effect on the application can be assessed.

### Detailed Hardware and Software Design

The hardware model in figure 2 is a very high-level model. No circuit details are represented. Execution times are estimated. Although very valuable information can be gained about architectures and their match to particular algorithms using this type of model, a natural next step would be to support the design of hardware that implements an architecture. Fortunately, this fits very well with ongoing work in Ptolemy (sponsored by SRC) on hardware/software codesign.

The existing Ptolemy Thor domain can be used to implement a programmable DSP, or other circuit element, at each ALPS node. The SFG is partitioned as in the earlier case; however, instead of generating just a partitioning, DSP assembly code is generated for each portion of the subgraph. The DSPs at the nodes of the ALPS configuration execute this code. The run-time of the different portions of the SFG can be determined and different partitions of the SFG can be examined.

Thor, however, is not a widely used hardware specification language. For this reason, we propose to build a VHDL code generation domain in Ptolemy, coupled with a VHDL simulator. Because of Ptolemy's support for heterogeneity, it will be possible to freely intermix these VHDL descriptions with more abstract stochastic hardware models in the DE domain and specialized algorithm specifications in the SDF (and other) domains. The VHDL domain is well under way, but not yet ready for distribution.

## D. References

[Eic92] T. von Eicken, D. E. Culler, and S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communications and computation," *Proc. of the 19th Int. Symp. on Computer Architecture*, Gold Coast, Australia, May 1992, also available as TR UCB/CSD 92/675, CS Division, University of California, Berkeley, CA 94720.

[Pic89] D. Picker, Ronald D. Fellman, and Paul M. Chau, "Performance Analysis of a Ring-Based Multiprocessor System", Submitted, *J. of Parallel and Distributed Computing*.

[Sih93a] G. C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 4, No. 2, February, 1993.

[Sih93b] G. C. Sih and E. A. Lee, "Declustering: A New Multiprocessor Scheduling Technique," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 4, No. 6, pp. 625-637, June 1993.

[Wu84] Y. S. Wu and Les J. Wu, "'Signal Flow' Architecture", COMPCON 84, Wash. DC, Sept. 84.