RAPIDbus: Design of an Extensible Multiprocessor Structure

John C. Willis
Arthur C. Sanderson

CMU-RI-TR-84-13

Department of Electrical and Computer Engineering
The Robotics Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

May 1984

This work was supported in part by The National Science Foundation under grant number 7923893. Portions of the work described here, particularly the RAPIDbus II design, were supported by the Air Force Office of Scientific Research under contract number F49620-83-C-0100.

Table of Contents

List of Figures

Abstract

Research in areas of robotics such as machine vision and control systems can benefit from appropriate increases in the available computational power. If algorithms can be structured to take advantage of task level concurrency, a multiprocessor design can provide cost-effective enhancements to the computational resources while decreasing the impact of subsystem failures.

RAPIDbus is described in this report as two evolutionary steps in the development of a system to support research in advanced, integrated, sensor based robotic systems. RAPIDbus I, a four processor architecture, is evaluated based on a two processor implementation fabricated in the laboratory. Building on the first design, RAPIDbus II is described as an extensible, high performance, packet switched structure supporting a multitude of heterogeneous processor-memory nodes.

Both RAPIDbus architectures assume a single address space populated by a moderate number of comparatively powerful processors. RAPIDbus II goes beyond the breadth of the earlier architecture by assembling groups of fifteen processors into ensembles called societies. Packet repeaters between societies allow up to sixteen ensembles to be assembled in a problem-dependent configuration within a single shared address space. Although not realized in the current proof-of-concept system, an object layer interface was suggested to maintain cache coherency, support strong data typing, and assist in dynamic memory management.

The RAPIDbus II implementation independently allocates bandwidth in each society using redundant, time-multiplexed busses and an efficient bus transfer mechanism. Both single and multiple word transfers are supported to match the needs of different tasks and processors. An upward compatible implementation is suggested which replaces the busses in each society with a cross-point switch, increasing performance while decreasing complexity.

Realization of prototype hardware led to exploration of asynchronous interface design, system complexity, and integration level issues. Embodiment of the architecture and implementation in hardware allows comparison with other designs, helping to locate RAPIDbus within the multiprocessor design space.

Acknowledgements

This project report reflects the assistance of many people who, while not necessarily endorsing the structure, have helped to bring the structure to life. RAPIDbus I is derived in part from design concepts developed by Mario Zoccoli and Rafael Bracho.

Dave Coleman, Rob Emmons, and Paul Oppedal are providing essential assistance to transform drawings into working hardware. Under IBM sponsorship, Pat Snyder of the University of Minnesota Micro group is bringing parts and supplier relations together in record time. Design automation was made possible by Dario Giuse's prompt and patient support of early versions of drawing package. Thanks are also due to Howard Wactlar and the engineering lab for providing the kind of support that makes *almost* anything possible.

Complementary to the hardware structure, Nanda Alapati and Jim McQuade are bringing a software environment together with the assistance of Industrial Programming Inc.

RAPIDbus I was supported by the National Science Foundation. Architectural design and partial fabrication of RAPIDbus II is being supported by the Air Force Office of Scientific Research as a tool in the exploration of space based image analysis algorithms. Colin Harrison and Dale Krutchten with IBM's Instrument's Advanced Technology Division are supporting fabrication of the RAPIDbus II implementation. SKY Computer is providing essential hardware and support for the floating point processor nodes. H. T. Kung supported importation of LLL's SCALD into our environment, and design of the ECL bus interface chip in conjunction with A. Nowatzyk.

The advice and helpful suggestions made by many researchers, designers, and architects at IBM Instruments, Motorola MicroSystems, Motorola Semiconductors, and ESL/TRW are gratefully acknowledged.

Chapter 1
Why RAPIDbus?

Two hundred kilometers above the earth's surface, a space platform is responsible for the analysis of imagery representing events below. Within a factory of the future, individualized electronic packages are assembled by a team of interacting robots. In the laboratory, a new computer system is taken from design drawings to gate level emulation with minimal human intervention. Hypothetical settings such as these represent goals motivating current robotics research. Each such application is rooted in specific theoretical and implementation questions. They share a common need for appropriate and significant increases in the computational power available to support future development.

RAPIDbus is directed at exploring one approach to providing computational resources for the development of advanced robotic systems. It is both a project in application-directed computer engineering, and a potential research tool. Conceived with such dual purpose, it is a blend of the freedom provided by theoretical computer architecture, and the realities best embodied by the label: tool. By providing a research multiprocessor which is attractive and practical for a select user base, valuable feedback can be generated, improving our understanding of both the application environment, and practical approaches to concurrent programming.

This document describes RAPIDbus as an evolutionary foundation upon which an application dependent configuration and software base can be built. Although both the

software environment and the implementation of the application are critical to a system success, this document explicitly discusses neither beyond motivating the chosen machine architecture.

1.1. Defining the Application

RAPIDbus is intended to provide a system architecture which transcends a particular robotics environment, and yet no attempt has been made to create a "general" computing resource. Many special purpose processors have been designed to effectively support a narrow subsystem need. As a multiprocessor, RAPIDbus is intended to combine the efficiency of such processors with the requisite general purpose processor element. Like a visit to a fine tailor, there is no "size" (specific environment) cut into the design, yet a narrow "style" (robotics research) is defined.

1.1.1. Model of a Robotic System

From the standpoint of the computer architect, it is useful to create a model for the computational engine within a robot system. Each environment will place slightly different requirements on each subsystem, yet there is an underlying similarity in the structure of each. Figure 1-1 illustrates a variety of internalization tasks interacting with a knowledge base to form an integrated representation of the external environment. A dynamic goal manager relies on the model generated by the knowledge base to provide control directives for packages that modify the external environment.

1.1.1.1. Internalization

The internalization stage is one of the most difficult parts of the system, both conceptually and computationally, since the external world is often weakly constrained. In a robust system, this stage may be composed of a multitude of different packages which rely on both sensor devices and the knowledge base for a bottom-up, top-down analysis of the external world. Two-way interaction with a knowledge base functionally separates different internalization media, and while allowing access to a time history of the internal model of the outside world.

Machine vision represents an appealing internalization medium in many applications, both because of the available bandwidth, and by analogy to human strategy. Yet the bandwidth, and the confounding of useful information by a multitude of factors often makes vision the most computationally challenging of the internalization media. Vision represents a primary

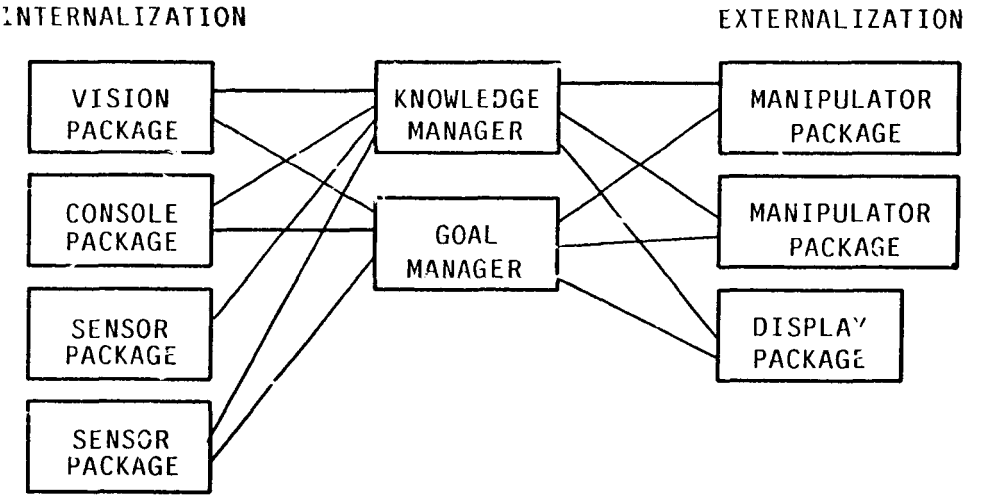

Figure 1-1: Computational resources supporting many robotics applications can be summarized as a system with internalization, knowledge management, goal management, and externalization packages.

application area guiding the RAPIDbus design.[1]

Auditory input can be represented as a linear input stream sampled in time, often with much lower bandwidth than three dimensional visual signals.[2] Once sampled, audio processing often is computationally intensive with some of the same qualitative processing requirements as vision [58].

Other internalization media include contact [18], proximity, position, or force sensing. Internalization tasks must convert the sensor input to a representation useful for the knowledge manager.

In contrast to more general computing systems, a user console is very much an ancillary input task. In the laboratory, a console is useful for interacting with the system under development using highly constrained, relatively low bandwidth paths. Seldom is there a

---

[1] See the subsection following.

[2] Two spatial dimensions and one of time.

need for multiple consoles beyond hardware diagnosis. It is conceivable that a fielded system in the future might have no console at all.

1.1.1.2. Knowledge Base Management

Current research directed at a single portion of a robot system often embodies the knowledge base manager within the internalization and externalization packages. As multiple internalization and externalization packages are added, the concept of a central information repository becomes more important. Packages represent the talents of many different researchers who require a common means of communicating. For instance, tactile, auditory, or other cues might be useful in the selection of relevant visual or other media, yet exposure of one internalization package to another's data structures increases both the conceptual and implementation difficulties. The structure of both such a knowledge base, and the means by which it interacts with internalization packages is an open research topic.

1.1.1.3. Goal Management

The goal manager package motivates both the internalization and externalization stages based on the internal state representation maintained by the knowledge manager. For instance, Weiss's visual based servo control might be conceived of as a goal manager which asks the knowledge base to maintain a stream of **X** and **Y** coordinates of specific features relative to a reference frame [60]. The goal manager might then instruct the externalization packages globally so as to narrow an arm down on an object of interest. At a higher level, the goal manager might plan the assembly of an object based on a description maintained by the knowledge manager.[3]

1.1.1.4. Externalization

The externalization packages attempt to affect the external environment in accordance with the directives of the goal manager package. Generally each package can be expected to merge the internal model of the environment, as depicted by the knowledge base package with the directives from the goal manager to create control signals to specific actuator mechanisms.

In some systems, such as an assembly cell, the externalization packages may regulate position or process control parameters according to a control scheme. In other systems auditory output or visual display may provide feedback to the user.

---

[3]Perhaps this is a second knowledge base manager distinct from that monitoring the immediate state of the internalization packages.

1.1.2. Machine Vision

Algorithm development for the analysis of time-varying space based imagery is a prime motivator for the fabrication of a RAPIDbus system. This particular application serves to illustrate some of the characteristics of vision internalization and knowledge base packages, although console and display packages are also involved.

Vision packages are typically characterized as a pyramid, beginning with large iconic images, which are typically processed to extract a concise representation of the external environment relevant to the internal knowledge base. The knowledge base may guide the vision task by providing information from a time history of past image frames or by describing objects or situations known to the system [74, 50].

Most approaches to early vision processing, at the base of the pyramid, are highly parallel, accommodating data rates which may exceed ten million, eight bit samples per second for high resolution aerial image sensors. Many different special purpose architectures, both digital [53] and electro-optic [40], have been designed to implement operators such as convolution [15], correlation [56], spatial filtering, moments [14] and edge enhancement [15].

At higher levels in the pyramid, data rates decrease, but the computations becomes less regular. Concurrency is still possible, though at the task level, either by working on separate portions of the image, or by working on different hypothesis of system structure. For instance, in a graph matching approach [21], several tasks could start at different points on the search tree, working to convergence on a particular representation. A multitude of other high level analysis approaches are discussed in volumes such as Ballard & Brown [6], many of which are potentially adaptable to programmer dictated parallelism.

As a high level representation of the external environment forms within the knowledge package, the goal manager can begin to exercise reporting criteria, making decision as to how and when to report the results of analysis.

1.1.3. Assembly Assistance

Research in sensor based robot assembly cells represent a second potential application for a RAPIDbus system [61]. Within such cells, one or more manipulators may be used to accept incoming parts, join them, and pass subassemblies on.

Automated assembly may require a wide range of internalization packages including binary or grey level vision [62], acoustic, optical, electromagnetic, force, or tactile. Each of the sensors may be introduced to the system expediently using a standard interface protocol such as RS-232, GPIB, current loop, or parallel port. Preprocessors may already exist using one or more I/O bus standards such as STD bus, Qbus, or Multibus. Other than vision systems, such sensors are often low bandwidth, but numerous. Their internalization packages can often be productively formulated with a separate package bound to each sensor or group of sensors.

The centralized knowledge management package is especially useful with very diverse internalization packages. As in the vision application accented above, flexible assembly may rely on two knowledge base packages, one describing the immediate assembly environment, the second describing the assembly procedure.

At a higher level, several such assembly cells may interact under the control of a goal manager package within one tightly coupled system. Irregularities or individual job characteristics may then be passed from cell to cell through the knowledge base package.

At the externalization stage, a variety of different actuators may be required, controlling position, flow, or force at many different points. Support for standardized interface protocols is again useful in the rapid integration of existing controllers.

1.1.4. Design Assistance

Design automation represents a third application for a tightly coupled processor system. Project's such as Demeter at CMU are working to develop integrated design environments to assist in the translation from design specification to fabrication documents [34, 66, 8]. Although the majority of such systems fit well onto workstation or general purpose computing environments, the gate level simulation of large digital circuits presents special performance problems.

Prior to expending the effort required to fabricate either custom silicon or board level designs, it is useful to analyze their behavior. Such analysis can help to verify correctness, evaluate performance tradeoffs, and study the response to system faults. Although many design automation tasks for a large project can be handled in small pieces, simulation, by nature, often involves the entire design. Current uniprocessor-based simulators are frequently too slow to get meaningful performance statistics, or to integrate actual silicon devices with dynamic memory elements.

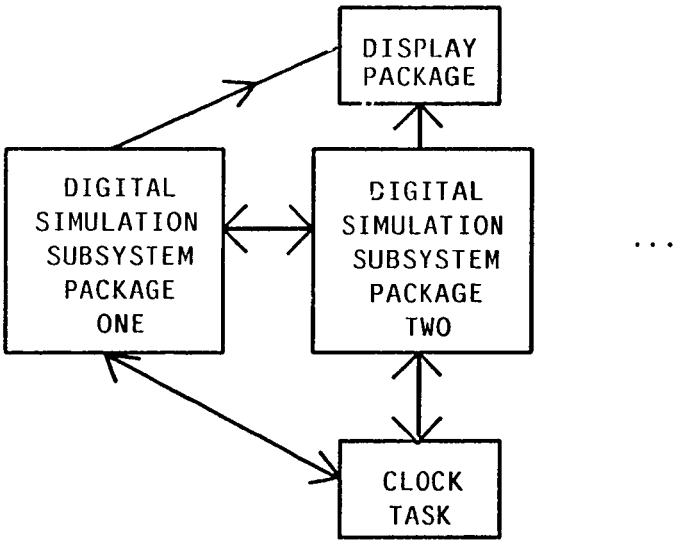

Figure 1-2: Mapping subsystems to separate simulation packages with inter process communication along well defined interface paths can improve simulator throughput.

Based on the pioneering work done by Lawrence Livermore Labs with SCALD [63], many design automation systems support hierarchical design descriptions with a multitude of different modules at all but the top layer, as shown in figure 1-2. At the second highest layer, communication between modules can readily be made very explicit, and is usually along clean interface boundaries.

This suggests the possibility of mapping each such module at a convenient high level to a different task, and potentially processor, for simulation. Each task simulates an event queue for a specific step interval. When the queue is exhausted, outputs are exported to other simulation tasks for event scheduling, designated signals are sent to a display task, and a clock task is given an interval completion signal. In turn the clock increments the timing and initiates the queue for the next step in each simulation task.

In this way, simulation of large digital circuits can be productively cast as a highly concurrent multitasking problem with programmer controlled parallelism and tight coupling. In the context of RAPIDbus, simulation is a support task for the efficient fabrication of new digital hardware.

1.2. RAPIDbus

1.2.1. Defining a Structure

The model of a robot system described above has strong implications for the design of an effective, very high performance machine architecture. Even at the package level, a high degree of concurrency is suggested. As three example environments indicated, each package is often further divisible into concurrent tasks. Coupling the need for high, cost effective performance with this degree of concurrency led to the choice of a multiprocessor architecture for RAPIDbus. The strong dependency between tasks suggested that low latency communication was imperative. The variety of different tasks suggested a heterogeneous processor structure.

1.2.2. The Goal

It is the goal of this project report to describe a multiprocessor structure supporting a multitude of diverse, heterogeneous tasks grouped into packages with a tight locality of reference. Parallelism is explicitly designated by the programmer, assisted by architectural features to support modular, strongly typed code. Underlying this goal is the hypothesis that such a structure can effectively support the development of advanced robot systems.

Chapter 2
RAPIDbus I:
Architecture and Realization

2.1. Why Build?

When the time comes to create a machine intended for *use* in a particular application, one begins to appreciate the observation that there are two kinds of computer architects [12]. The first spend their lives studying how to build machines, the second build machines. Each approach has strengths, each is symbiotic with the other. The first kind of architect lives in a world of paper and models, the second in a world strung together with semiconductors and unforgiving electrons. RAPIDbus I is our first step into the perils, and lessons of this second realm.

A time-multiplexed common bus was chosen as a cost-effective initial configuration connecting a small number of high-end microprocessors. At the suggestion of Rafael Bracho, we took advantage of commercial Versabus monoboard computers connected through individual interface cards onto a time-multiplexed physical bus [13]. The basic design for the time-multiplexed bus was adapted in part from an earlier design by Zoccoli [81].

A two processor system, shown in figures 2-1 and 2-2, was constructed and evaluated to provide a realistic basis for the qualitative and quantitative description presented in this

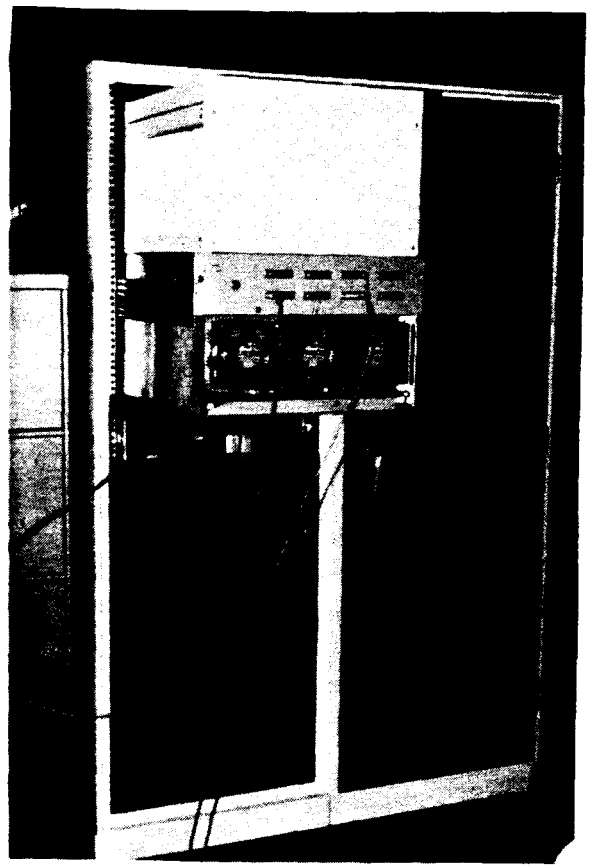

Figure 2-1: A dual height Versabus card cage was constructed to house the two processor RAPIDbus I prototype.

chapter. This background forms the basis of the evaluation described in the following chapter. More extensive details of the RAPIDbus I design can be found in CMU Robotics Institute Technical Report 82-13 [77].

2.2. Systems Architecture

2.2.1. Choosing Versabus as a Host

Versabus is a circuit-switched common bus protocol developed by Motorola to the 68000 family of microprocessors. Using an asynchronous handshaking protocol, data can be transferred on up to 32 data lines and selected by 32 address lines in a series of upward compatible steps. Multiprocessors are supported using a bus arbitration scheme controlled by a central arbiter. Vector interrupts are implementing by extending the 68000 interrupt structure to the bus.

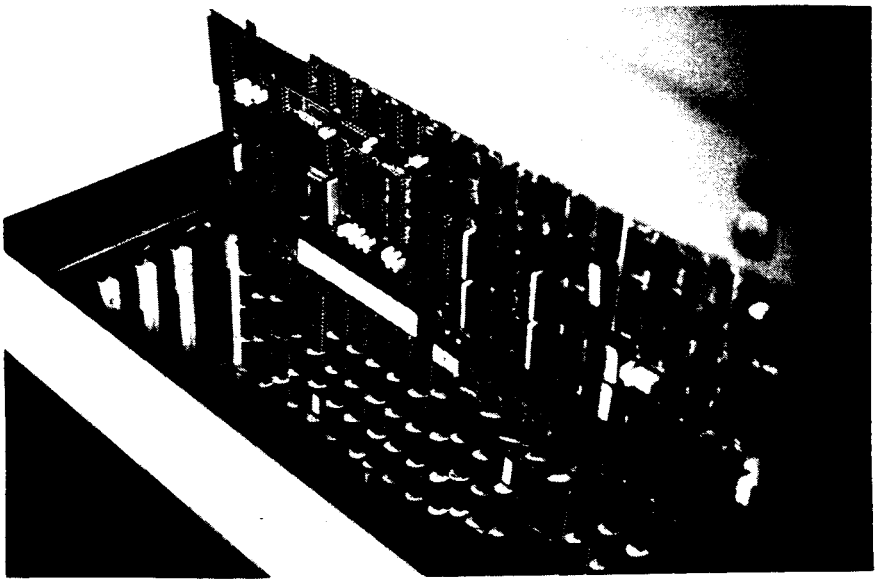

Figure 2-2: The RAPIDbus I interface was implemented on a mother- daughter board inserted between the physical bus and a commercial monoboard computer.

In late 1981, when the processor was being selected, the 68000 microprocessor was a reasonable choice for the kind of numerically intensive computation that was foreseen in our application code. The 68000 allows use of 16 bit data transfer capability, with a 16 megabyte (24 bit) addressing range.

Similar results are likely using the Intel 80X86 line of microprocessors running on a Multibus with suitable modifications. Although the original Multibus was not designed for 32 bit compatibility, a new draft standard, MULTIBUS II, is expected to provide 32 bit synchronous data transfer capability [70].

Versabus is specified in draft form as Motorola document M68KVBS-D4 [46]. An IEEE committee has been formed to consider IEEE standardization of a similar bus protocol. Since the initial release of Versabus, market pressures have lead to the design of a VME bus specification, which has a protocol similar to that of Versabus, but is usually implemented on dual DIN cards using the 16 bit bus option.

Other industrial bus standards were rejected as a prototype host interface for a variety of reasons. Card size restrictions and limited bus widths eliminated the S100. Qbus did not provide a commercially available processor card with the price/performance that the VM02 offered. TI's 9900 bus did not have a seem to have enough support or performance to be interesting. Both the VAX SBI and the Gould SEL bus were considered and rejected based on the inavailability of appropriate single board processor cards.[4]

2.2.2. PMS Level Alternatives

Once the decision was made to use a monoboard computer as the processor node, a variety of processor-memory-switch (PMS) [64] level structures were possible using a common bus. The simplest approach was to assign processors on one set of cards, and the memory on another set of cards (figure 2-3). All memory references are subjected to bus latency, increasing load on the interchange network.

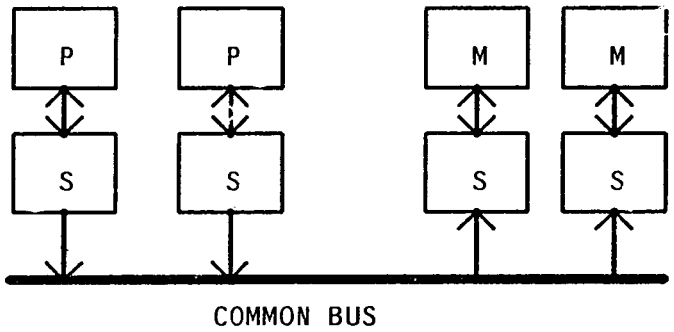

Figure 2-3: Separation of processor and memory functions on the bus maximizes bus loading and average memory reference latency.

In order to reduce inter-processor bus contention, many monoboard computers provide local memory, either ROMS containing system software, or RAM for private data or instruction segments. Shared memory segments are still located on separate RAM cards accessible via the common bus. Access to such shared memory requires that both a read and a write operation take place on the system bus to communicate a word of data between processors (figure 2-4).

By dual-porting memory onto both the local processor and system bus, the number of

---

[4]The SBI and the SEL bus are synchronous, a appealing characteristic when time-multiplexing a host port.

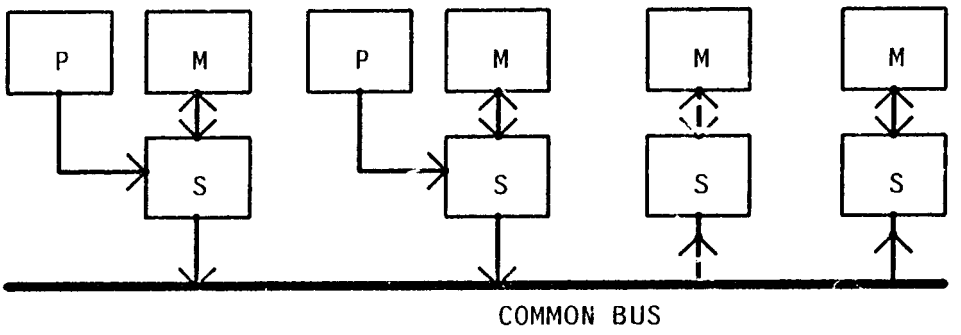

Figure 2-4: Addition of local memory on the processor card permits quick access to private data and instructions.

references required on the system bus decreases (see figure 2-5). The communication of data from one processor to another requires only a single bus transfer to share each word of data.

A circuit switched interchange network, which supports only one circuit path for the duration of a memory access, such as Versabus, has no risk of deadlock resulting from the sharing of a single interchange port between outgoing processor requests and incoming memory accesses. Performance can decline resulting from memory contention, but since the Versabus host port is only assigned to one master at a time, a situation cannot arise where respective local dual ports are simultaneously assigned to the local processor engaging in a bus transfer that requires the other dual port memory, resulting in deadlock.

Several months into the design of RAPIDbus I, after we were committed to implementing a dual VM02 prototype with dual ported memory, it became clear that the circuit switching assumptions built into Versabus would greatly complicate the overlaying of a time multiplexing interchange with multiple simultaneous paths. Since multiple Versabus hosts could simultaneously receive conflicting Versabus port grants, a serious deadlock problem was introduced. Coupled with the 68000's inability to rollback on all instructions, we were left with the possibility of either accepting bus error (time out) traps for legal accesses, or modifying the Versabus protocol on each host.[5]

The efficiency of each of these approaches is analyzed in the following chapter, both using a straight Versabus implementation of the common bus, and with a RAPIDbus interface card interposed between host and bus to create the illusion of several physical busses through

---

[5] RAPIDbus I accepted the possibility of traps on valid requests.

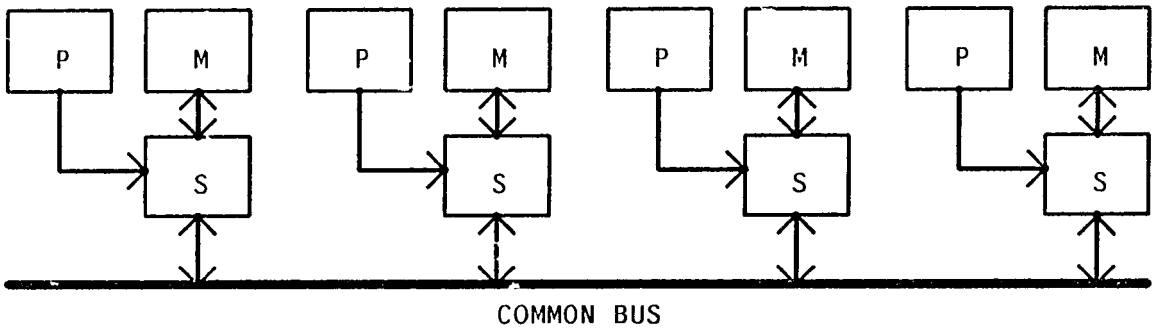

Figure 2-5: Dual porting of the local memory can economize on bus cycles required to access shared memory.

time-multiplexing. As one expects, performance comparisons can only be made based on application dependent parameters. It is clear however that going from the first to fourth configuration above, the hardware complexity increases. All tests involving actual hardware were based on a pair of VM02 processors and RAPIDbus I interface cards.

2.2.3. The RAPIDbus I Interface

2.2.3.1. RAPIDbus I Concept

A primary goal of the RAPIDbus I protocol is to take advantage of the bandwidth differential between a single block of bulk dynamic memory, and the theoretical bandwidth which the system bus is capable of. By time multiplexing the physical backplane, each Versabus port capable of initiating a data transfer (master) is assigned a virtual bus, along which all other available RAPIDbus ports in the system may be accessed as slaves. Each master appears to have a private link from the RAPIDbus port on the interface card to the RAPIDbus ports of all other system cards. Versabus processor which share a dual-port with local memory may have to arbitrate use of their port with other masters accessing the dual-port memory.

The virtual busses connecting the RAPIDbus port on each interface card are implemented using time-domain bus windows. Each interface which supports a processor is sequentially given a window during which the master interface may send a data transfer request to one or more slaves, and/or receive a response from an already activated slave. System timing is illustrated in figure 2-6. At least three windows are required to complete a data transfer. Experience with the VM02 card suggests that most Versabus cards will require several more. Thus the RAPIDbus interface card transforms between the time-multiplexed RAPIDbus windows that are pertinent to a task, and the time-static Versabus host.

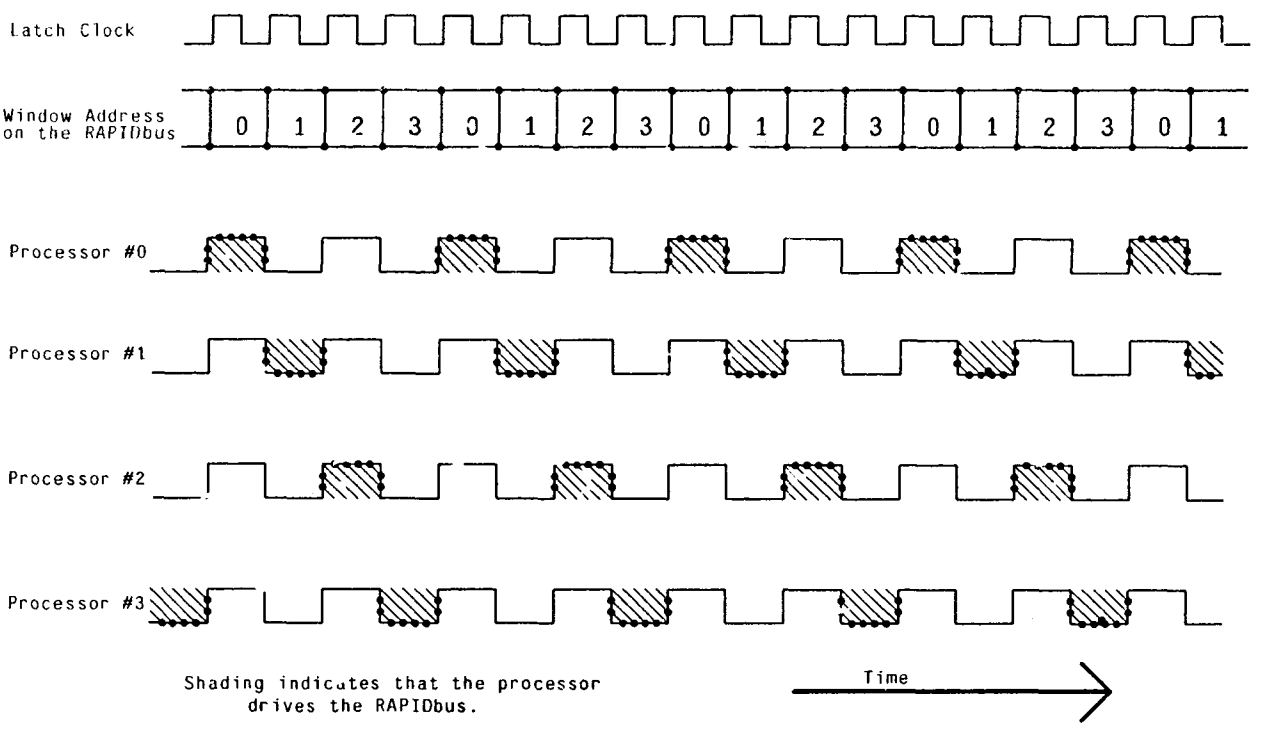

Figure 2-6: The virtual bus system is implemented using bus windows to link several masters and slaves simultaneously.

Our FAST TTL implementation indicates that up to four windows can reliably be supported using an eight to ten megahertz processor. A variety of signal lines are required to implement data transfer, some of which are time-static, and some of which are time-dynamic, communicating during bus windows. Data and address lines are examples of lines sent on bus windows, having different values for each of the virtual busses. Other lines are time-static and are identical for all virtual busses. The interrupt lines are examples of time-static lines, interrupting a processor handling a given level independent of the virtual bus that the interrupt handler is assigned to.

A multicasting capability is supported on the interface which allows one master to write blocks of data simultaneously into multiple memory locations. Prior to a multicast transfer, the system multicast capability must be assigned to a master, and the required address generators initialized.

2.2.3.2. RAPIDbus functional modules

Each RAPIDbus I interface card is composed of a series of functional blocks as shown in figure 2-7. This compartmentalization of function is intended to improve the readability of a design, simplify debugging, and identify functions that lend themselves to packaging integration. The interface is composed of a window handler, drivers, latches, an address translation unit, a multicast address generator, a parity check section, a chip select section, a timing generator, an interrupt control section, and an interface controller. The Ibus links the Versabus port on the top of each interface card with the RAPIDbus port at the bottom. It comprises the address translation section, the drivers, and the latches. The Versabus port, the RAPIDbus port, the multicast address generators, the parity section, and the chip select section drive or monitor the Ibus.

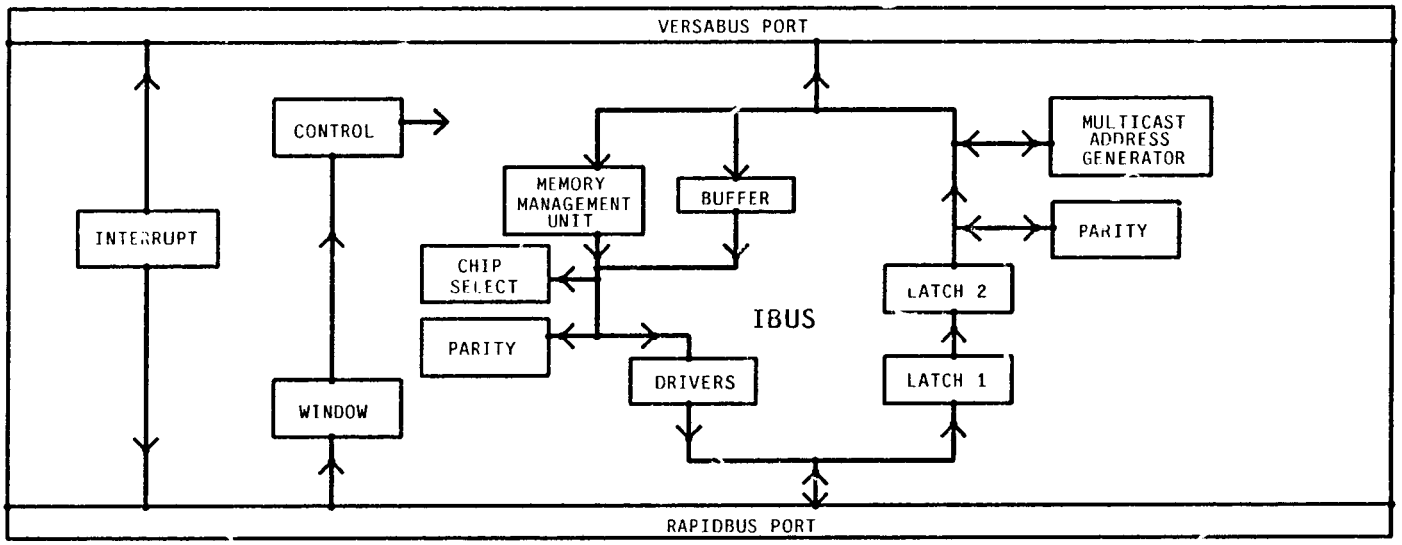

Figure 2-7: Each RAPIDbus interface card is composed of multiple modules, centered around the Ibus.

Window Handler

The heart of the time-multiplexed bus is the **window handler.** Each interface port off the RAPIDbus is assigned a unique window identification number, which determines the home address, timeout interval,[6] and in the case of a slave, the control page address within the RAPIDbus physical address space. The window handler scans the window address bus looking for either the home window address or that of a master for whom this interface is working as a slave. Through the control register the interface can be configured so as to recognize only one window address for slave accesses (masking).

[6] to reduce deadlock

Driver

The **driver** section is used to gate the address, data, and control lines onto the RAPIDbus during a window as directed by the control sequencer. When the control sequencer does not indicate that this interface is to drive a given backplane line in the current window, the lines are to be tri-stated. Line drive is to commence as soon as the appropriate window is recognized and continue until the next window is recognized.

Latch

The **latch** section holds data at two levels. The first latches all time-multiplexed RAPIDbus lines at the end of every window. During the following bus window, these latched lines are examined for a slave reference to this interface card. If the Ibus is not being requested or is in use, and a slave reference to this interface is detected, then the contents of the first level latch are held by the second level latch. The second latch also holds the bus window if the Ibus is already allocated to the virtual bus currently sending the bus window held at the first level latch. Under the direction of the control section, lines can be selectively gated onto the Ibus according to the function of the current Ibus master.

Address Translation

Data transfer exchanges must always be initiated by a master, and each master interface may optionally have an **address translation unit** positioned between the host processor and the RAPIDbus drivers. The function of the address translation unit is to map A8 - A23 to physical addresses PA8 - PA23. In any interface that incorporates a address translation unit, the translation must be capable of being circumvented through the control register so as to map the Versabus address directly to the RAPIDbus drivers. The lower seven address lines, A1 - A7 are supplied directly to the driver section without translation. The control register also allows switching between primary and secondary memory maps on interfaces that support memory management.

Multicast Address Generator

The **multicast address generator** is required for all interface cards that must function as a slave during a multicast data transfer cycle. The master desiring to multicast must set the multicast request bit in its control register, asking the processor's interface to try for ownership of the multicast capability. Line twelve of the interface status register is asserted low by the interface to indicate that the multicast capability has been secured by this interface.

Prior to sending multicast transfers, the multicast address generator registers of all interfaces that are to be destinations must be loaded with the base address and the number of words that are to be multicast. For interface cards which support master and slave functional hosts, the master occupying the Versabus port must be asked to initialize his multicast address generator since the MAG registers for a master interface only appear in that master host's memory map. For slave-only interface hosts, such as a memory card, the MAG registers are mapped into the RAPIDbus physical address space.

Following write instructions to the multicast reference address will be multicast to each activated slave card until a slave's word count is exhausted or the master stops writing to the multicast address. Each activated slave depends on its multicast address generator to supply the memory address and maintain the count of words still to be transferred. The address counter is not incremented and the word count not decremented if multicast retry is asserted by any interface being multicast into before the multicast data acknowledge has gone high.

Parity

The **parity** section generates and checks parity during master and slave transfer operations over both data and address lines. A parity error results in a retry of the transfer cycle unless the aborted instruction was part of a read/modify/write operation.

Chip Select

The **chip select** section serves to direct references by the Ibus owner to the interface control page, Versabus, or RAPIDbus port. If a master reference is not to the control page, then the reference is directed to the RAPIDbus drivers. Control page references are subject to further decoding to identify the reference as a multicast, multicast control, interface control/status register, or memory management unit reference, selecting the appropriate device or in the case of the multicast, the interface RAPIDbus multicast server.

Timing Generation

The **timing generation** section controls the timing of the interface state sequencing. This section generates the multicast and regular address strobes when the respective address lines are ready for the RAPIDbus drivers, and generates decode enables and DTACK for the interface mapped resources.

Interrupt Control

The **interrupt control** section supports interrupt generation and interrupt handler vector requesting. This section presents to the Versabus port on the interface only those interrupts that the interface host is strapped to uniquely handle. If installed, the memory management unit interrupts at one of the levels handled by the local host. The interface host is able to generate any one of the seven levels of RAPIDbus interrupts.

Interrupt vector acknowledge operations are correctly supported, with the hardware supported restriction that only one master at a time may run an interrupt acknowledge cycle. Within an interrupt level, physical priority in the card cage dictates interrupt handling priority.

Control Section

The **control section** knits together the other functional blocks and controls the logical state of the interface. The ownership of the Ibus is decided by the control section. The interface bus error, retry, data acknowledge, and halt are generated by this section, as are timeout and retry limits. The control section contains the control/status register which allows dynamic configuration of interface parameters such as the masking address, the address translation path, and the auxiliary memory map.

2.3. Communications Protocol

A variety of different data transfer protocol are used in the RAPIDbus I architecture. This section serves as both a summary, and a pointer to more detailed literature.

2.3.1. The 68000

The Motorola 68000 forms an entire micro-coded processor on a chip. The chip must communicate with external hardware however, in order to accept instructions, read and write data, generate and handle interrupts, and respond to irregular termination conditions such as reset, bus errors, and retries. This section is concerned with summarizing the protocol for communication off chip. For detailed information, see the relevant Motorola manuals, and technical bulletins [48, 35]

2.3.1.1. Memory Reference Cycles

The 68000 uses a handshaking protocol to access either byte or word operands from either memory or peripherals. Twenty-three explicit address lines are used, supplemented by three bits of function code, and two data strobes. The two data strobes, upper (A0 is low), and lower (A0 is high) are used to select either or both of the least significant bytes addressed by the sixteen megabyte address range. Long-word transfers are implemented as a pair of atomic transfer operations. The 68000 defines five significant function codes, supervisor program and data, user program and data, and an interrupt cycle. Either supervisor or user function codes, when paired with an asserted address strobe, and at least one asserted data strobe indicate a valid data transfer cycle.

A given data transfer cycle can read an operand; write an operand; or read an operand, modify it's value, and write it back as part of an atomic bus cycle. A write operation is indicated by the write line being asserted low while both address strobe and at least one data strobe are low.

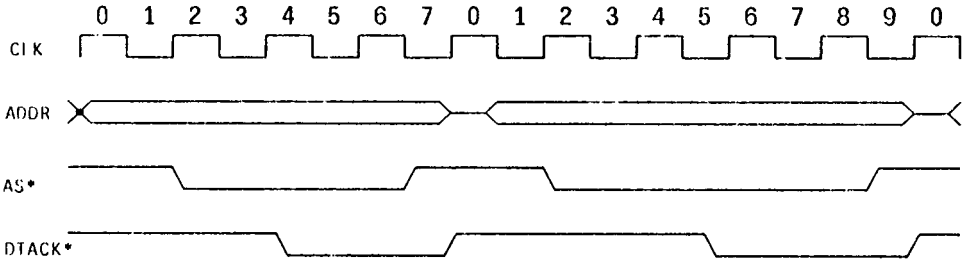

Figure 2-8: Bus transfer timing relative to the processor clock for two cycles, the first in zero wait states, the second requiring an extra clock cycle.

Any data transfer operation is terminated either with data acknowledge, bus error, halt, or reset being asserted low. Normal bus cycles conclude, or read-modify-write cycles are punctuated by the assertion, and later removal, of data acknowledge. Bus error is an irregular termination which is sampled once on each microcycle during the transfer. If bus error is asserted, and halt is still inactive, then the transfer is aborted, and the microengine traps to a handler routine. When bus error is asserted in conjunction with halt, then the current memory fetch cycle is terminated, and if possible, rerun. In the current 68000, all instructions except

for test and set are capable of being rerun. Assertion of halt alone puts the microengine into a state of suspended animation from which only a reset will recover.

Motorola specifies the data acknowledge, bus error, halt, and reset for assertion a setup time before they are sampled by the internal microengine. It is interesting to note that some designs violate this setup time, with apparent impunity. A penalty is, however, paid in processor throughput, and potentially in system reliability. Since the internal micro-engine samples the data acknowledge only once during each clock cycle at and following the end of microcycle four, assertion of data acknowledge randomly with respect to the clock potentially slows down every memory reference by an average of half a clock period, or about twelve percent (relative to a zero wait state operation).

2.3.1.2. Interrupt Cycles

In order to efficiently respond to events asynchronous to the current instruction stream, the 68000 is designed to support vectored interrupts. Three external interrupt lines are sampled at the end of each instruction cycle. Subject to possible internal masking, an interrupt acknowledge cycle will be initiated on the data bus, denoted by an interrupt acknowledge function code. The low address represents the interrupt level which the processor is currently responding to. A pointer to the proper handler routine is returned by the interrupting device, or an interrupt vector generator working on behalf of the interrupter on the low byte of the data bus.

2.3.2. The Versabus Data Transfer

Shortly after the introduction of the 68000 processor, Motorola released Versabus as a bus protocol optimized for their processor interface.[7] Versabus was intended to support a tightly coupled system of "data processing, data storage, and peripheral data control devices" [46].

---

[7]Manufacturers and those using their parts have often designed bus protocols around a new processor, The S100 bus is based on an 8080, the Apple bus on the 6522, the Qbus on the LSI-11, and the Multibus on the 8086, to cite a few.

2.3.2.1. Bus Arbitration

Each Versabus system has a system controller which is responsible for Versabus arbitration among other duties. A requester desiring use of the bus pulls one of five prioritized, open-collector bus request lines **BR0-BR4**. A special requester, the emergency requester, has an additional, high priority request line. If a request is submitted at a higher priority than the current bus master, the system controller's arbiter can assert the **bus clear** line, instructing the current user to free the bus. Some Versabus masters automatically release the bus after a single data transfer, others hold the bus for multiple transfer cycles, and some even hold onto the bus until ordered to release by the **bus clear** signal.

After the current bus master has removed the **BBSY** line, the arbiter grants the bus to the highest priority active request level following an arbitration delay. In some implementations this arbitration process can occur while the preceding data transfer operation is concluding. Since multiple potential bus masters may have made a bus request on the same line, the arbiter sends the **bus grant** signal down one of several daisy chains where it is passed along until it encounters a card which requested the bus at this level. Such a card traps the bus grant, and begins to drive the **BBSY** signal. The arbiter receives this **BBSY** signal, and removes its bus grant.

2.3.2.2. Data Transfer

After bus mastership has been granted, a processor begins by asserting the address, function code, address strobe, and data strobes as appropriate to the width of the data transfer (see a description of the 68000 above for details). All potential slaves scan the address and function code lines, waiting for a reference that addresses their resources. The selected slave arbitrates for use of the dual-port RAM if needed, conveying the address bits to the memory or peripheral logic.

If the write line is high following assertion of data strobes, then a read operation is in progress. After the operands are available, and gated onto the Versabus, the slave drives the data acknowledge line low, causing the master to latch the data and release the lines it is driving. In the case of a write operation, with the write line low, the memory latches in the data, and returns data acknowledge to the processor, causing the processor to release the lines that it is driving. In the case of a read/modify/write operation, the address strobe remains assert after data acknowledge comes back from the read operation, continuing through the write cycle. Any data transfer cycle can be terminated if the **BERR** line is asserted.

Optionally, parity is checked for both address and data transfers, although no current Versabus processor cards are known to support this capability. The halt line is no longer carried out to the Versabus, making retry operations impossible from the bus with unmodified Versabus cards. A variety of bus width options are enumerated in the specification, covering data paths of eight, sixteen and thirty-two bits, and address bus widths of sixteen, twenty-four or thirty-two bits.

2.3.2.3. Interrupt Handling

The 68000 vectored interrupt capability is brought out to the Versabus, using a cycle similar to a read data transfer. The handler is granted bus mastership, asserts address and data strobes, and conveys the interrupt level being serviced on the least significant bits of the address lines.

In order to support multiple potentially interrupting Versabus cards at the same level, the interrupt lines are daisy-chained similar to the bus grant lines. Only the first card which interrupted at the level being acknowledged by the daisy chain generates a vector. Thus within a given interrupt level, physical location of the interrupter determines priority of service.

2.3.3. The RAPIDbus I Data Transfer

The RAPIDbus I data transfer protocol is based on an adapter which sits between the Versabus port of a standard Versabus card, and the physical bus, converting between the time-static bus seen by the Versabus card and the time-multiplexed busses implemented on the physical RAPIDbus backplane. On each interface the Ibus connects the Versabus and RAPIDbus port. This bus resembles a Versabus to the Versabus host port, with suitable arbitration between host and requests that may come from the RAPIDbus port.

2.3.3.1. Data Transfer

Once the Versabus port has requested and been granted mastership of the Ibus, some versions of the interface card translate the address using a Motorola 68451 memory management unit. Later versions of the processor card incorporate the memory management unit at the processor, and the interface MMU is not supported. The potentially translated address is decoded for routing either to MMU control registers, the interface control register, multicast address registers, or RAPIDbus resources.

A decoded address which maps to the RAPIDbus waits until the fixed time-slot allocated to

this interface card, then drives the backplane with the full address, function code, and control lines. Other interface cards examine the backplane for references to their resources. Until the full transfer operation is complete, all data transfer will occur on the home window of the originating processor.

At some point the proper memory or peripheral card becomes available, latching in the data transfer request so as to present it in a time- static manor to the slave Versabus card, where it appears as a normal transfer operation. The data is similarly exchanged using the initiating processor's home window. To the two Versabus ports involved, the transfer operation completely resembles a standard Versabus transfer.

As the evaluation net model which follows will make clear, the current implementation permits four such exchanges to occur simultaneously given four processors, four memory or peripheral cards, and an appropriate addressing pattern. Each transfer is however slightly longer than a single conventional Versabus transfer would be as a result of the additional switching that is taking place. Data and address parity is checked on all transfers, with errors handled via a processor rerunning the transfer for all but a 68000 test and set instruction.

2.3.3.2. Interrupt Handling

Interrupt vector operations are more complex than with a straight Versabus. The interrupt vector request and return are handled as a data transfer operation, with a valid address corresponding to the interrupt level which the interrupter generated. Problems arise in the interrupt acknowledge daisy-chain. Unfortunately there is no simple way of multiplexing a daisy chain at high speed. Since the interrupt acknowledge is not a very frequent bus cycle, an interlock mechanism allows only one of the four virtual busses to run an interrupt vector request cycle at one time.

2.3.3.3. Multicasting

The Multicast capability is an enhancement not supported under Versabus, which allows one master to simultaneously write into multiple destination memory locations assisted by address generators located on the interfaces of the destination cards. Only a single virtual bus can own the multicast capability at any one time in order to simplify the complexity of the address generators and share multicast acknowledge lines.

Hardware arbitrates the multicast capability among any busses desiring use of the

capability, based on interface control register requests. As soon as the previous multicast master releases the capability, a bit in the interface control register will indicate a grant to the next requesting interface, allowing the new multicasting master to acquire the required capability.

The master must then initialize the multicast address generators in each of the destination cards, specifying both the number of words to be transferred, and the base address. If the destination is a memory card, the register can be written directly. If the destination is located on a dual port processor card, then the initialization must be requested from the local processor, perhaps through an interrupt.

As soon as all requisite address generators are initialized, the master can initiate transfers by writing to a special multicast control address which is decoded on the local interface card. With the overhead described above, it is clear that the multicast capability is only useful when large blocks of data are to be transferred. As with standard data transfer operations, parity is always checked. Parity failure results in a retry operation affecting all active multicast cards.

2.4. Evaluation Net Modeling

2.4.1. Use of Evaluation Nets for Hardware Modeling

Evaluation network [*ENET*] notation is a convenient method of representing control flow in both hardware and software systems. Here it is used to provide a model of the RAPIDbus I multiprocessor system. Use of enet notation allows critical timing paths in the control schema to come to the forefront in preparation for performance analysis and system optimization. Once an enet model has been devised, translation to a computer simulation can be made largely automatic. All timing information noted on the graphs was derived from logic analyzer traces run on actual hardware with the exception of the memory-only card, which used times extrapolated from a working VM02 and a memory card print set.

Enet notation was devised by Nutt [54], [52] based on experience gained in the practical application of Petri nets to computer systems. The network is made up of transitions which link locations in a unidirectional manner. Each location contains a vector of state information (possibly zero length), called tokens. Several locations and transitions proposed by Nutt are shown in figure 2-9.

Any particular transition has a set of input locations, on the left of the vertical line, and output locations, to the right of the vertical line. A transition is said to fire if a set of locations on the left specific to the transition type are occupied by a token, and none is present on the relevant output. Firing a transition initiates a transition process which fills the proper output location on it's expiration.[8]The transition process is often a function of the vector of state information present at the input.

Although transition functions were not used explicitly in the modeling of the RAPIDbus I system, transformations of the input vector to produce a new output vector can provide an effective modeling of control path effects on data flow. Further development of hierarchical CAD tools for parallel implementations could benefit from such capability.

Figure 2-9 illustrates the enet components used in the RAPIDbus I model. On the far left, the circle is used to represent a location. The location is assigned a reference number to simplify the pairing of text and diagram. Each location also has an initial marking indicating the presence or absence of a token in a given location on reset. Here the initial marking for a given figure is described in the accompanying text. At most a single token can occupy a given location at any one time.

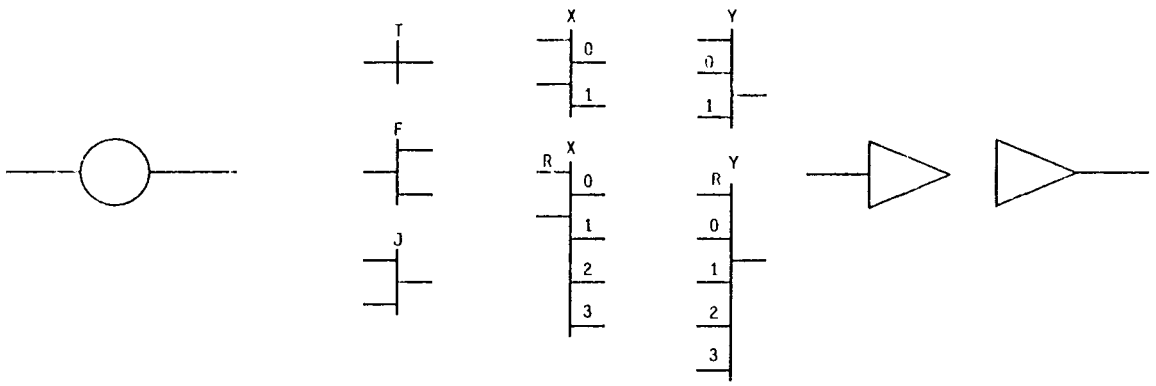

Figure 2-9: Locations and transitions are used to represent control flow using evaluation net notation.

On the far right of a diagram are two arrows used to illustrate connectivity of flow in disjoint

---

[8]In Nutt's original work, the transition process was formally divided into four phases: pseudo-enabled, enabled, active, and terminating. The pseudo-enabled stage was only used by X and Y transitions where resolution was required following the marking of input nodes and the clearing of output nodes. The enabled phase indicates that all nodes were ready for firing of the transition. The active phase indicated transition was in progress. Finally the terminate stage resulted in the marking of the required output nodes and the clearing of the required input nodes.

diagrams. Two connected arrows are assigned the same reference number in both diagrams. In cases where one arrow may connect with several other matching arrows, as in a driver gating onto a bus, the arrow number is suffixed with a letter.

The formal enet notation also specifies a resolution location which is used to direct the flow of control to or from multiple paths, according to the value of the location. Here the function of the resolution location is informally supplanted by text which indicates the arbitration strategy. Consideration of the enet notation for performance evaluation in our work led to the concept of a random variable resolution node, which took on random values according to some specified parameters. The model implemented in the following chapter uses a poisson process to model flow based on our first order application parameters.

The remainder of the symbols shown in figure 2-9 are transitions. The most basic is the *T* transition. When the location at the right of the transition is unoccupied by a token, and the input on the left is filled, then the transition fires. Firing of the transition causes a shift of the input token to the output location after a transition time. Here the transition time is given by a value under the transition's vertical bar.

The *F* or fork transition causes the input token to be conveyed to both output locations a transition time after firing. The complementary function is the *J* or joining function, which fires as soon as both input locations (to the left of the vertical bar) are filled, and the output location on the right is empty. Note that when actually implemented in hardware, the proper joining of two asynchronous events can make or break the correct functioning of a system. The ramifications of this asynchrony are discussed in greater detail in the next chapter.

Both the *X* and *Y* transitions are shown in two forms, indicating the flexibility with which new composite operators can be fabricated [52, 79]. The *X* operator allows the direction of the input to a particular output node. When the input is filled, then the resolution node is evaluated. If the destination selected by the resolution node is unoccupied, then the transition fires, transferring and perhaps modifying tokens at the end of the delay. If not explicitly shown, the resolution node is taken to be the upper left hand branch of both *X* and *Y* nodes, as shown for the four output *X* node.

The *Y* node functions conversely to select one of a variety of inputs for the transition output. The *Y* node does not require the evaluation of the resolution node if only one input location is marked. If more than one input location is marked, then resolution is initiated.

2.4.2. Caveats

With some transition delays, the timing could not be determined accurately, and is prefaced by a "~". A notable example of this is the monostable used to control the refreshing timing. The presence of several active time bases made determination of some parameters difficult, and reflects itself in hardware reliability as discussed in the following chapter. Zero transition delays are often artifacts of the notation's symmetry, in most cases representing the fact that the transition function was actually lumped into another time delay. The slash between two different transition delays conveys different timing for token vectors that designate a read or a write data transfer operation. The read value is always given prior to the "/", the write value afterward. Where the resolution function is a priority encoding of inputs, a token on the zero input is taken to be highest priority.

2.4.3. The VM02 Dual Port Processor

Figures 2-10, 2-11, and 2-12 describe the data transfer timing of the Motorola VM02 monoboard computer card. The noted delays were implemented on the card using combinations of a synchronous state machine (processor), delay lines (eleven actively used), monostable multivibrators and gate delays. As delivered from the factory, the card operates at eight megahertz with the time base derived from a thirty-two megahertz clock. When run in a ten megahertz configuration, two time bases are used, one at twenty megahertz for the processor (divided by two), another for the board, running at thirty-two megahertz. All measurements here were made at eight megahertz.

Although these diagrams only illustrate memory reference paths, the VM02 and interface are capable of a variety of alternate cycles including interrupt vector fetches, I/O bus references, and in the case of the interface, of multicast operations. Since a survey of the intended application indicated these were infrequent bus operations, mostly bound in performance by external factors, they have not been included in the enet model depicted here.

The initial marking of figure 2-10 puts a token on **L1**, with all other locations in figure 2-10 blank. The delay time from **L1** to **L2** is highly dependent on the instruction mix being run by the processor. The shortest delay results from the second half of a long word fetch, requiring 190 nanoseconds. Much longer delays are experienced when operations such as multiply and divide are run with few external memory references required relative to the internal microcycle count. The stated delay, 260 microseconds, was derived from both observation of

a variety of instruction mixes running on a 68K, and validated using hand calculation.[9]

The birth of a memory reference is signaled by the processor asserting address strobe. Since the address lines are precharged a fraction of a clock cycle before address strobe, only twenty nanoseconds are required for a PLA to select the addressed resources, transferring the token on **L2** to **L3**, **L4**, **L5**, or **L6**. The ROM access is predicated on the use of < 250 nanosecond memory, with no contention possible. Access to RAM, via arrows **1/2** references the dual port located in figure 2-11. Access to Versabus resources is through arrows **3/4** to figure 2-12. A reference is terminated when date acknowledge is returned to the processor, again marking node **L1**. As a result of the resource allocation schema taken here, the address decode resolution delay is effectively lumped with the resource utilization delay.

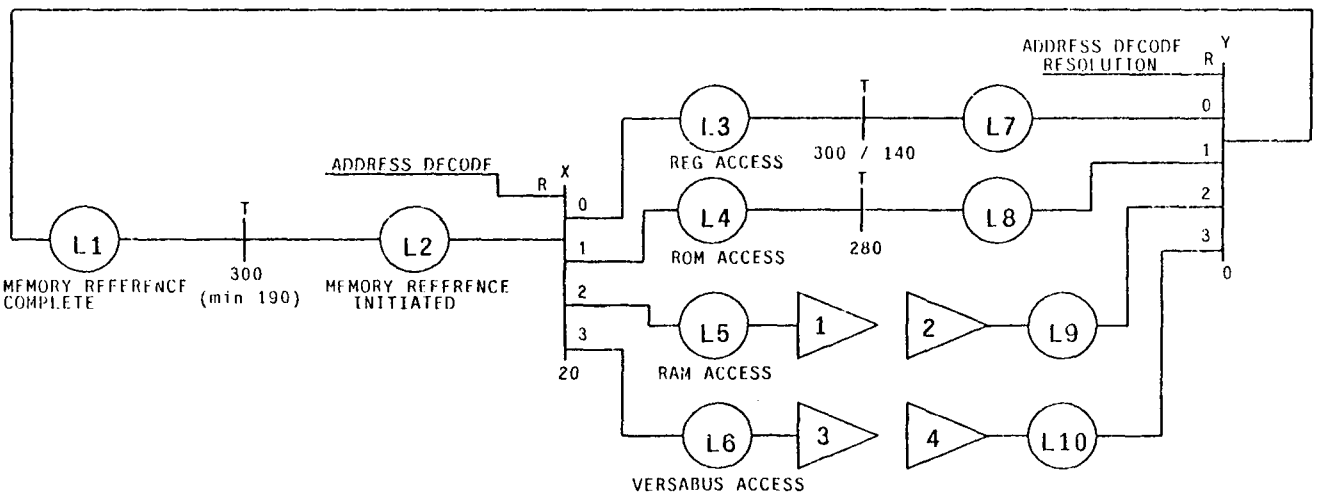

Figure 2-10: The processor's local bus allows access to local memory, board registers, and the Versabus port.

The local ram is dual-ported, as noted earlier, to both the local processor bus and the Versabus (figure 2-11). Since only one port can have access to the resource during one RAM cycle time, a critical section is implemented. In the VM02 design, references that are resolved by the priority decoder come from three asynchronous time bases, the refresh timer, the local

---

[9] A scientific instruction mix was used based on the work of Marathe, converted to 68K "equivalents", and then projected onto the Motorola microcycle statistics. The mapping of PDP-11 instructions to 68K "equivalents" is not a highly accurate procedure, but it seemed a reasonable index into an otherwise unavailable number, since no hardware monitor was available [47].

bus, and the Versabus. The refresh timer is initialized on reset by marking location **L15**, and the critical section of the dual port is initialized by marking location **L12**.

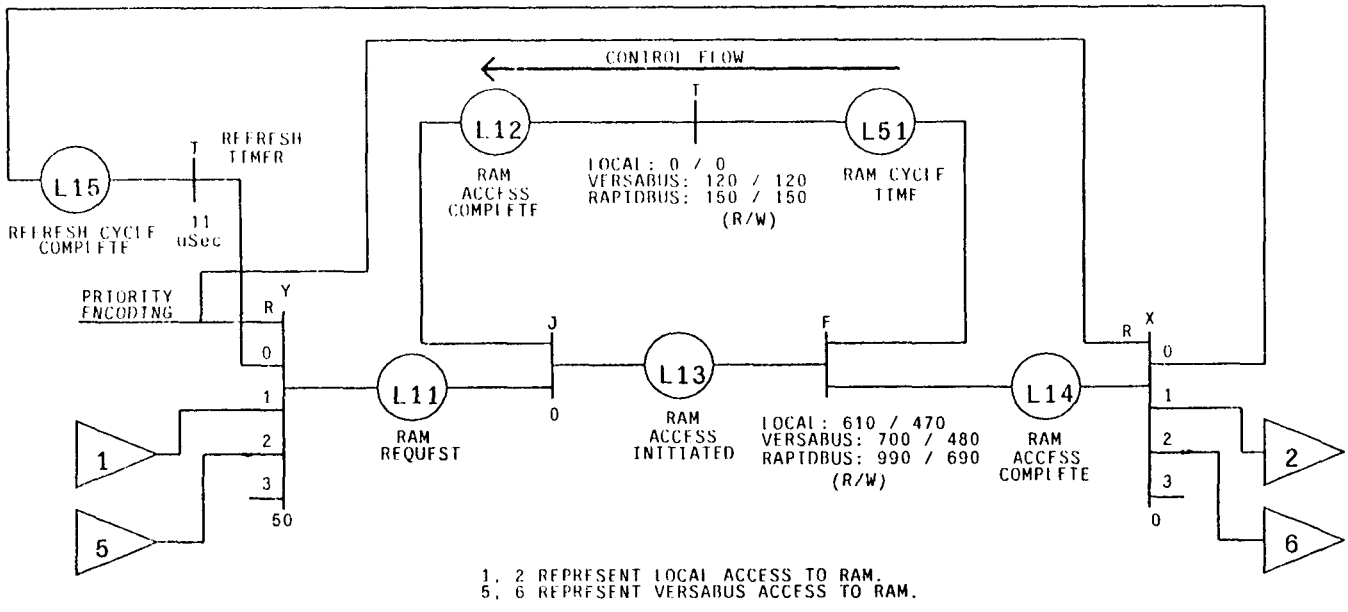

Figure 2-11: The local ram is dual ported to both the Versabus and the local processor bus.

Once a port has been granted access, the service time is dependent on the source and type of reference (read/write). The source dependency results from the need to send data acknowledge to the initiating processor, and receive the address strobe removal before releasing the resource. This allows the proper operation of read/modify/write operations. The dependency on the operation type reflects the ability of hardware to return a write operation data acknowledge earlier in the ram cycle than could a read data acknowledge. Read/modify/write operations are not analyzed here as part of the performance since they are a very infrequent bus operation.

After data acknowledge is returned to the processor, there is still a period of latency before the memory array can handle another request. This is reflected in the delay between **L51** and **L12**.

The Versabus dual port arbiter, figure 2-12, allows either the local processor to access the Versabus port, or another card to access the dual port memory from the Versabus port. As in figure 2-11, a critical section is front-ended by an arbiter which has asynchronous inputs. Note that if arrow 3 is granted access, then the arbiter assigns flow through the 7/8 arrow pair to the Versabus diagram, located in figure 2-14. If arrow 9 is granted access, then a request from the Versabus is granted access to the dual-ported memory. Arbitration between the local bus and Versabus port for the local memory must then take place, before the reference begins service.

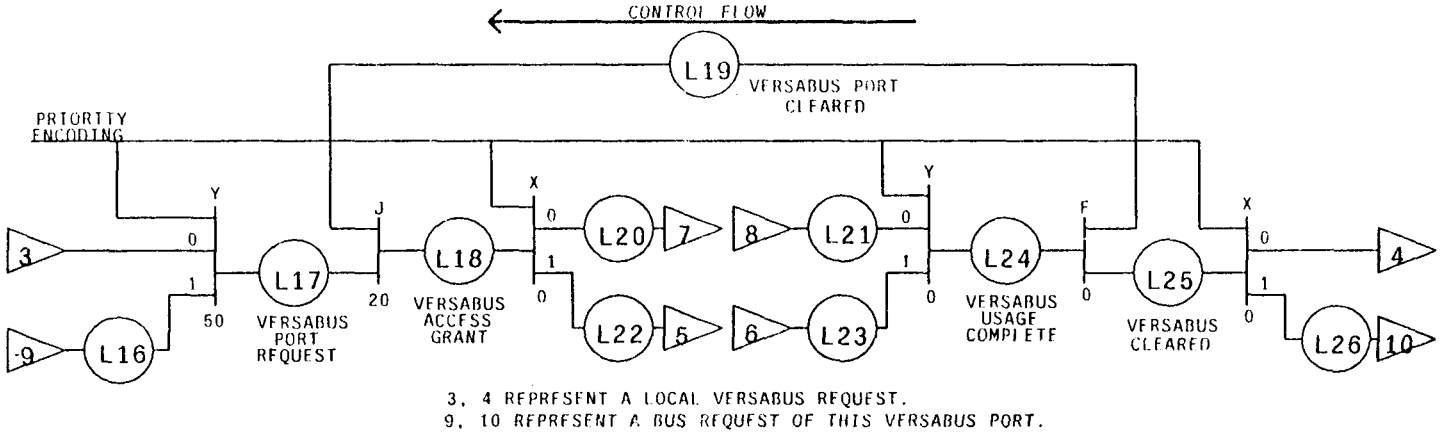

Figure 2-12: The dual port arbiter allows a connection between local bus and Versabus or between local ran and the Versabus.

Since the RAM resource or bus resource service times include the return of data acknowledge to the initiating processor, transition delays are set here to zero. At the expense of suggesting a non-causality in the notation, accuracy could be improved by modeling a slight negative transition time, or using a more complex resource model. The Versabus specification allows arbitration of Versabus ownership during the end of the last data transfer cycle, leading to similar remarks about an instantaneous bus clear. The diagram is initially marked on location **L19** to activate arbitration of the dual-port critical section.

2.4.4. System Memory Cards

The model of a Versabus memory-only card with no processor on board is based on the VM02 design strategy and timing model. The local timing here refers only to the cycles devoted to the refresh timer. The diagram is initially marked on **L47** to activate the critical section used for RAM service, and **L46** to activate refresh.

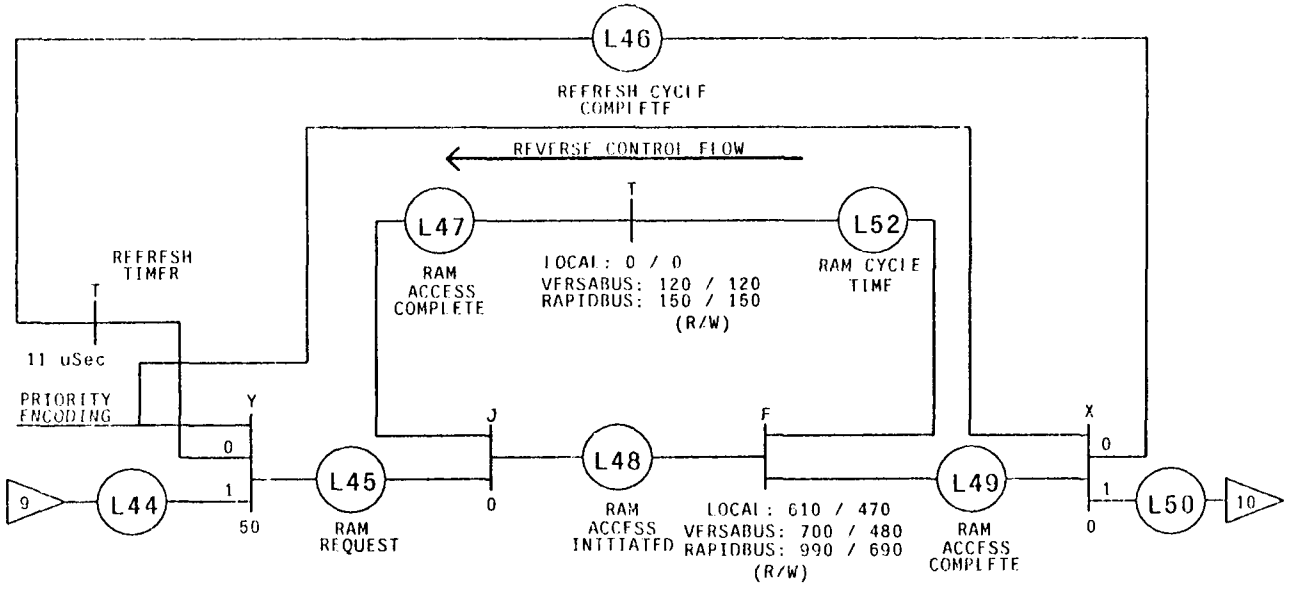

Figure 2-13: System memory cards resemble VM02 ram cards without the dual-port arbitration.

2.4.5. The Versabus Interconnect

The VM02 processor card was designed to function in a multi-card, multi-processor configuration using the Versabus protocol. Control-flow for a sample Versabus system is illustrated in figure 2-14, with four masters with dual-port memory, and four memory-only cards. Arrows marked **xA**, **xB**, **xC**, and **xD** correspond to masters with dual-port memories. Memory-only Versabus cards are designated by arrows **xE**, **xF**, **xG**, and **xH**.

The arrow pair **7x/8x** represent requests for the Versabus and the termination of requests respectively. As indicated by the critical section (locations **L27**, **L28**, and **L29**), only one of the four masters, designated by **xA**, **xB**, **xC**, and **xD** is able to access the bus during any one memory reference cycle. The memory reference can be serviced by any Versabus port other than the port granted access to the bus. In order to prepare for proper operation, location **L28** must initially be marked.

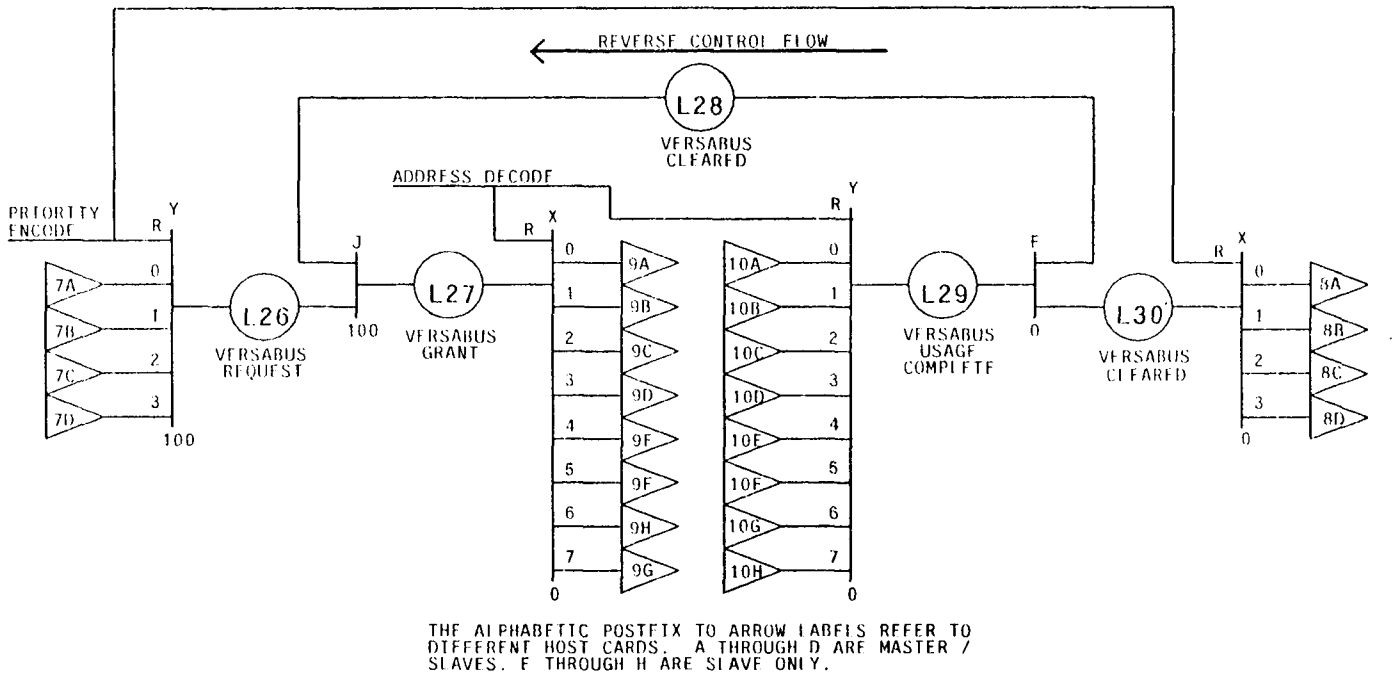

Figure 2-14: The Versabus interconnect protocol assigns the single physical bus to a particular bus for the duration of of a data transfer operation.

2.4.6. The RAPIDbus Interconnect

In contrast to the single figure 2-14 which represented the entire Versabus interconnect scheme for four processors, figure 2-15 is replicated for each of the virtual Versabuses, four in this example. Each of the four memory-only cards operating using a similar flow diagram without the arbitration process for the Ibus required to choose between incoming and outgoing references (arrows 7/8).

Address decoding is restricted from the Ibus. If the instantiation of a diagram represents an interface with a processor, then the control register space, memory management unit, and RAPIDbus are only visible to the processor directly attached to the interface. If the interface represented by the diagram is being accessed via flow arrows 11x/12x from the bus, then the only accessible resource is the Versabus port, providing access to the dual-ported memory. If the diagram is an instantiation of an interface used in memory-only mode (slave) then the interface register, and Versabus port are available for access through the RAPIDbus port.

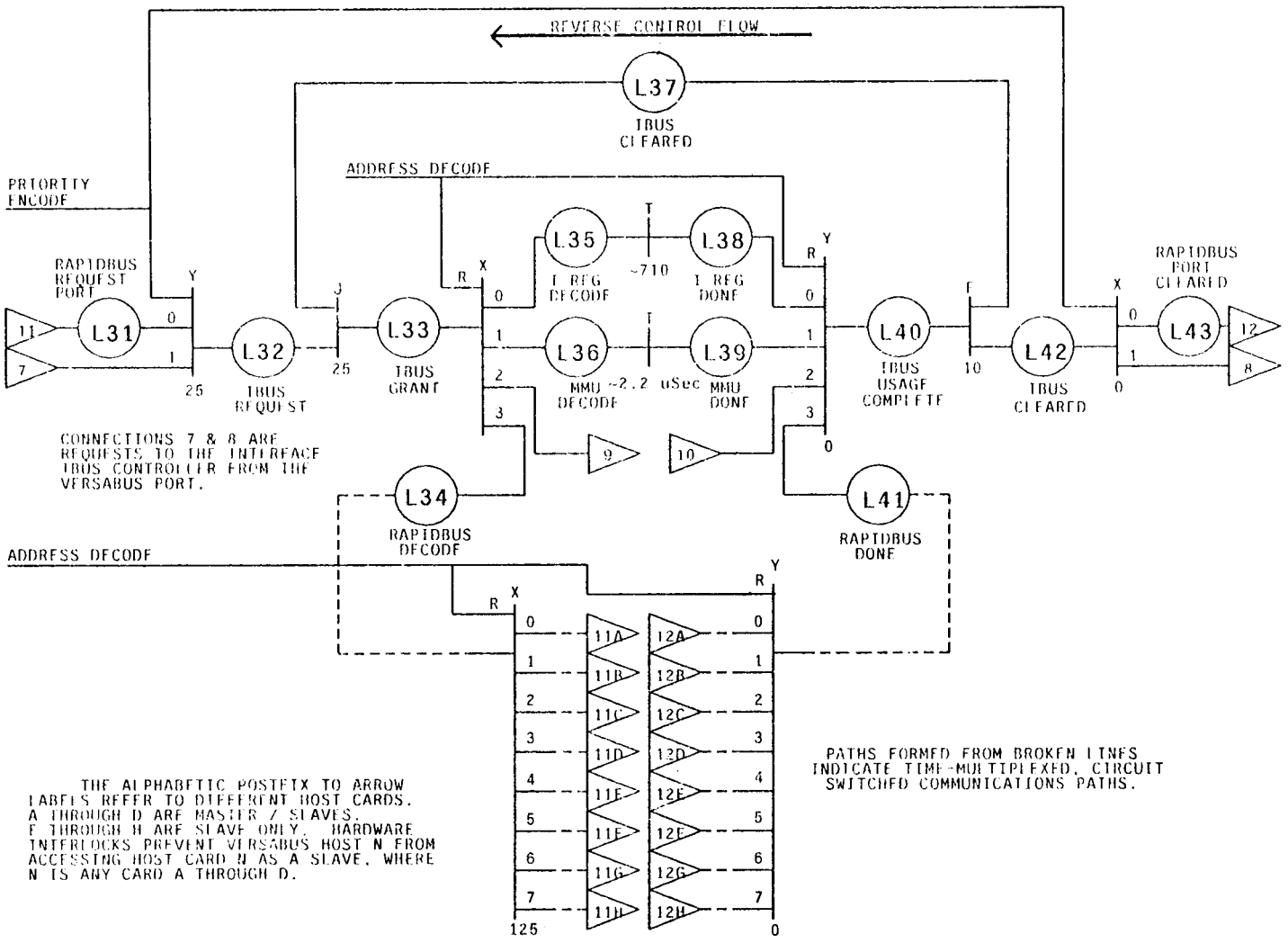

Figure 2-15: The RAPIDbus interconnect provides four virtual bus paths, each of which is described by the graph above.

The enet model developed above forms the basis for a simpler queuing model developed in the following chapter for use in the computer simulation of a variety of Versabus and RAPIDbus system configurations.

Chapter 3
RAPIDbus I: An Evaluation

3.1. Evaluation Methodology

Building on a description of RAPIDbus I, this chapter identifies and evaluates critical design decisions. A convenient structure is imposed through vertical stratification at three different levels of abstraction. Following the style of Blaauw and Brooks, analysis is divided into three sections covering system architecture, implementation, and realization [9].

The three levels of abstraction represent different project goals. At the highest level of abstraction, the architecture represents the functional character of the programming environment visible to the user. Supporting the architecture is an underlying logical structure referred to as the implementation, which is described in terms of protocol and functional diagrams. At the lowest level the machine depends on its realization, composed of elements such as gates and transmission lines which provide a physical substance.

The machine architecture represents an investment in both familiarization and software development by any potential user. Users are more likely to accept a machine to which they can apply what they have internalized from other machines. In order to design a valid research machine which will attract users in a free market environment, the architect needs to

carefully choose those aspects of the machine which are experimental at the user interface level. These changes must show potential for increasing performance, reliability, or quality of the programming environment. In order to derive valid scientific data from any new architecture, particularly one inside previously charted regions of the design space, the experimental and control aspects must be clearly located and delineated with respect to other data points.

Once the architecture is accepted by a user base it becomes a very difficult layer in which to make non-upward compatible changes. In the development of a research machine which is exploring the interesting fringes of the design space, experience dictates that it is critical to budget several fabrications and redesigns of a machine before any presentation to a user community should take place.[10]

The implementation layer is more amenable to changes than the architecture since by definition, this layer must remain transparent to the user. Reduction of compatibility constraints opens the implementation layer to the unbridled creativity of the designer, with the potential for enhancements which race ahead of device technology. Through an understanding of the critical facets of each subsystem, new protocols are developed which become visible to the user as enhanced performance and reliability.

The realization layer exists to embody the architecture and implementation within devices and interconnect technology. At best, this layer will use appropriate technology to bring the machine to life. The quality of that life; its reliability, and performance, are vulnerable to compromise at this point. This section will survey some of the real-world lessons we have learned from RAPIDbus I.

Quantitative performance evaluation represents a important and yet highly controversial field. Characteristics of all three levels of abstraction are combined with the uncertainties of the measurement process, resulting in a metric which one hopes to correlate with performance. The ultimate test of any realization of RAPIDbus lies in its performance within the intended robotics application environment. Since implementation of an application and operating system were deferred to run on the RAPIDbus II design, comparison between benchmarks run on RAPIDbus I, and performance on other machines running a full operating system would be misleading, and have been omitted in this report.

---

[10] A glowing example of this is the evolution of the MIT LISP machine into the Symbolics 3600.

3.2. Architecture

Our initial goal, achieved with the fabrication of RAPIDbus I, was to gain experience in the systems design of a small multiprocessor communicating through shared memory. Based on what we learned from RAPIDbus I, and an improved understanding of the application environment, the need for major alterations in the RAPIDbus architecture became evident. In order to justify the effort required to create a new tool for research in machine vision and robot control, the resultant architecture needs to qualitatively extend the state of the art in performance, reliability, and programmability which can be brought to bear on robotics applications in the laboratory.

There are many approaches for accelerating image processing and control algorithms using ensembles of hardware elements. Many of these architectures are highly effective, often by building assumptions about the structure of a particular class of regular algorithms into the architecture. In order to effectively support research into robot *systems*, a very broad spectrum of algorithms need to be both supported, and at a higher level, need to tightly interact to form a cohesive whole.

3.2.1. Supporting Broad Task Concurrency

RAPIDbus is fundamentally based on the hypothesis that an advanced robotic system can be composed of a multitude of small, concurrent processes which are hierarchically structured to create the desired system behavior. Thus the process or task forms the unit of parallelism granularity, with problem decomposition explicitly controlled at some level by the programmer. Many new structured high level languages are being developed with the intent to assist the programmer in abstractly dealing with concurrency.

ADA represents a major example of a high level programming language which supports concurrent code as an integral part of the language structure [5]. Sections of high level code can be packaged into tasks. Each is capable of execution either on a multi-tasking uniprocessor, or concurrently on one of an ensemble of processors. A master task, subprogram, block statement or library package can activate a task, or access entry points internal to a task. Language constructs allow tasks to converge, or continue execution through conditional waits. Variables can either be shared directly between tasks, in which case no assurance of state coherence is made, or parameters can be explicitly passed, invoking normal task synchronization techniques.

Resulting from the many kinds of run time checking which a full ADA implementation is required to do, current implementations generate less dense machine code for standard machine architectures than a simpler language such as Bell Lab's C. As support for ADA and similar languages grow, one can expect to see language subsets which disable run time checking, and the development of machine architectures which provide run-time assistance.

Real-time multiprocessor operating systems, such as MTOS, developed by Industrial Programming, provide task activation, synchronization, and communication in the form of high level calls from existing high level languages such as C [51]. Such external extension of the language is in keeping with C philosophy at the expense of integral structural enforcement provided by ADA.

Although many uniprocessor, multitasking operating systems provide support for multiple, intercommunicating tasks, the programmer is not rewarded for the effort required to learn how to think concurrently. Since the "concurrency" is achieved by swapping the state of one process for another in the core of the processor every few milliseconds, such parallelism is often penalized.

In the search for incremental performance increases through parallelism, the first stepping stone is often a master-slave configuration in which one processor handles I/O intensive tasks, and the other processor handles computationally bound work. Dual porting an MA-780 memory module to two VAX 780 processors created the VAX 782 multiprocessor. Resulting from the partitioning of tasks, very few data structures are simultaneously accessible to both processors. Thus few modifications are required to run existing VAX system uniprocessor code [23].

Taking the next step up to a system with several undifferentiated processors, major changes need to be made in system and application code to protect data structures in the event of cycle by cycle concurrent access to the same system structures. With multiple processors, the application programmer is given incentive to decompose his project into smaller, separately executable pieces. With the move from a uniprocessor to effective multiprocessor code, a substantial software development overhead is paid.

In the case of the RAPIDbus I implementation, it is very difficult to justify additional development time to write an application as a set of communicating processes if the maximum potential speedup is less than four times that of more traditional serial code. Once the

conceptual barrier to writing modular, concurrent code is overcome, proponents of languages such as ADA argue that large programming project are made both simpler, and more reliable than with less modular languages [55].

The four processor limitation of the RAPIDbus I implementation needed to be increased, but there were no guidelines for a "sufficient" number of processors. Most contemporary multiprocessor architectures lead to implementations which are heavily limited in their practical extensibility. Thus a revised architecture for RAPIDbus II could benefit from practically extensibility to almost arbitrary numbers of processors.

Interconnect architectures such as the University of Maryland's ZMOB [59] and Cambridge's FastRing [67] provide a precedent for such extensible multiprocessors, but often at the expense of increasing average transfer latency as the number of hosts is increased. Experience with system such as Carnegie-Mellon's CM* supports the intuitive conclusion that effective use of a tightly coupled system is dependent on minimizing the transfer latency between cooperating tasks [73, 80].

3.2.2. Host Homogeneity

Many extensible multiprocessors are based on replication of the same host node, often referred to as an homogeneous multiprocessor. Use of the same host node architecture does simplify both the hardware realization, and later the design of system software, but often at the expense of performance.

Both vision and control represent a wide ranging set of algorithmic requirements. One processor design cannot be optimized, for instance, to handle large arrays, complex decision making, and I/O operations. With a shared pool of processors, specialization becomes much more practical than with a uniprocessor. Either multiple task queues can be maintained for different classes of processors, or a field can be used within the task control block to indicate which processors can execute a given task.

Such diversity of roles was supported in the RAPIDbus I architecture through the use of a standard host interface (Versabus), for which many different kinds of commercial hosts could be acquired without incurring host development expense. Within a research environment, it is essential that a revised RAPIDbus support the integration of commercial hosts.

Use of existing commercial hosts from different sources opens up questions of architectural compatibility. Different processors recognize a variety of data types and package similar data types differently within a word. To further complicate integration, few processors indicate to the host port what type of data they believe they are accessing. For instance, a Motorola processor may write an IEEE floating point number to a 32 bit memory word. A floating point accelerator based on the DEC floating point format may later read the number, assume a native format, and proceed with an incorrect calculation. The interchange network has no way of knowing the difference between either floating point number or a string of bytes.

3.2.3. Reliability

Robotic systems are increasingly being placed in roles where dependable operation is essential. If the system is operating in an unmanned or unmonitored location, it is important to exploit the parallelism of a multiprocessor so as to gracefully degrade performance in response to system failures. Within the laboratory environment, the success of RAPIDbus is judged on it's ability to act as a tool. Failures which are visible to the user by interrupting or corrupting the execution of an application directly detract from the system's usefulness. However, if incorrect behavior has occurred so that rollback to a known correct state is impossible, it is highly desirable for the machine to detect the situation and gracefully inform the user.

RAPIDbus I provided poor fault tolerance at all levels of abstraction. Even fully functional hardware had the potential to interrupt user activity in the event of deadlock during a processor test and set instruction. At the implementation level, the single common bus provided many points at which a single hard or soft failure could halt operation. In order to support continued system operation in response to single point failures, multiple redundant interconnect paths are essential in the revised RAPIDbus architecture.

3.2.4. Programmability

As multiprocessors develop from being laboratory curiosities into usable tools, the quality of the programming environment, the "programmability", becomes an essential concern [33]. As temporal concurrency is coupled with tight interaction, it becomes critical to appropriately abstract system functionality through modularization. Communication between modules must be made explicit, with assertions on the temporal coherence and semantic compatibility of the data which can be checked at run-time.

Modularity has its roots in the currently accepted tenets of structured programming. Decomposition of an application into small parcels of code with specified input and output allows the programmer to concentrate on a tangible goal. As the complexity of the potential interactions among modules increase for a multiprocessor, it becomes increasingly likely that unintended and incorrect interactions will occur. In order for the multiprocessor system to confirm that the assumptions stated by the programmer are valid, support must be present at both the high level language and the machine architecture levels.

Unfortunately, the vast majority of high level language and microprocessor implementations are inadequate to support a truly concurrent, quality programming environment. Paying with performance, ADA and Intel's 432 provide notable exceptions. Development of both the required languages and supporting processors is currently an important research topic.

Communication of data between processes is aided by object based addressing, in which the name and the location of the object referred to by the name are separate entities. Unfortunately, this frequently requires one or more levels of indirection through main memory to access an operand, resulting in a performance penalty. In order to spread the overhead of such object based addressing across multiple word fetches, several authors have proposed the implementation of hierarchically defined data types which can be manipulated as a single entity [33, 68].

RAPIDbus I was constrained to use an existing microprocessor with minimal support for data type checking, object addressing, or enforced access rights. The 68000 processor used provided four kinds of memory accesses, divided into data and code, as well as user and supervisor segments. Within the shared memory, any user code running on any processor is given the ability to modify any write enabled area of memory. Since any one of several concurrent processes can alter a location, errant code can easily result in behavior which is neither replicable nor correct. Since this was judged unacceptable, the interface card provided a commercial memory management unit which restricted access for each class of transfer to or from one or more variable size memory segments. The same component, a Motorola 68451 memory management unit, allowed translation of virtual to physical addresses for bus memory, but *not for local memory*. Such non-uniform protection and address translation was a undesirable compromise in order to use an existing Versabus processor card. It would be highly desirable to extended the granularity of protection down to a single, variable sized data object, with checking for process access rights as well as data type matching.

3.2.5. Societies of Processors

The architectural evaluation of RAPIDbus I, the related experience of other, relevant multiprocessor projects, and an analysis of the intended application suggests that a minimal ensemble of homogeneous processors is a marginally useful tool. In order to provide a powerful application engine with moderate fault tolerance and a quality programming environment, a more appropriate structure seems to result based on several, tightly communicating societies of heterogeneous processors adapted to support a quality multiprocessor environment.

Such multiprocessor configurations have been proposed before [3, 24]. Revised RAPIDbus designs provide the ability to support such proposals with realizable hardware which can place modified commercial host processors into a highly extensible, modular ensemble limited in performance primarily by the throughput and transfer bandwidth of each host.

3.3. Implementation

The RAPIDbus I implementation was intended to allow evaluation of a time-multiplexed, rotary access communications structure based on existing Versabus cards. Success was achieved through the design, fabrication, and productive evaluation of a four virtual bus system populated by two dual port processor cards.

3.3.1. System Structure

Both Versabus and RAPIDbus I implement a multiprocessor in which a shared address space is visible to all processors in the system (full connectivity). It is useful to consider three different PMS level diagrams for Versabus systems with and without RAPIDbus enhancements.

3.3.1.1. Simulation Methodology

Each system configuration was simulated using a queuing model and parameters measured on the two processor system fabricated in the laboratory. Although RAPIDbus I is limited to four processors, simulation was carried out for systems with up to eight processors under the assumption that alternate protocols could be developed with similar service times accommodating a larger number of processors.

Instruction cycles were initiated using a Poisson birth process with a 2.25 machine cycle parameter between memory references. Twenty-five percent of the memory reference cycles were assumed to be write operations, with the entire workload running from RAM. Both parameters are reasonable estimates derived from several sources including our logic traces of monitor routines, and a much longer set of PDP-11 traces gathered by Marathe and mapped into the 68000 instruction set for this study [47].

The percentage of memory references which involved communication with other processors is a critical parameter determined by both application and data structure design. Rather than make assumptions, ρ was established as a parameter ranging between 95% local and 80% global. Each data point represents 1000 processor instruction cycles for each processor in the data point configuration. Experimentally this lead to stable performance statistics while consuming a reasonable number of CPU hours for simulation.

The effectiveness of the interconnect structure was determined using a system efficiency metric (SEM). The SEM compares the throughput of N processors relative to a single processor running the same code. Since most code adapted for a multiprocessor application requires a larger number of instructions than an equivalent algorithm on a uniprocessor, the SEM probably represents an upper bound on the speedup as processors are added to a system. SEM was determined by measuring the average number of clock cycles among all "installed" processors required to complete a synthetic workload, and then normalizing by the time required by a single Versabus processor with dual port memory.

3.3.1.2. Versabus - Separate Processors and Memory

The simplest, and least efficient Versabus configuration provides for *n* processors and *m* separate memory cards communicating via the Versabus. Illustrated in figure 3-1, such a structure requires every memory reference to request, wait on, and receive service from a central memory server. All bus memory becomes a part of the central memory server with a single unit of parallelism and a service time equal to the memory access time.

Resulting from contention for the central memory server, a separate processor and memory Versabus configuration SEM can at best asymptotically approach

$$[T_p + T_m] / T_m$$

as many processors are added to the system, independent of the number of individual memory blocks (T_p represents the time spent between memory references to the bus, T_m

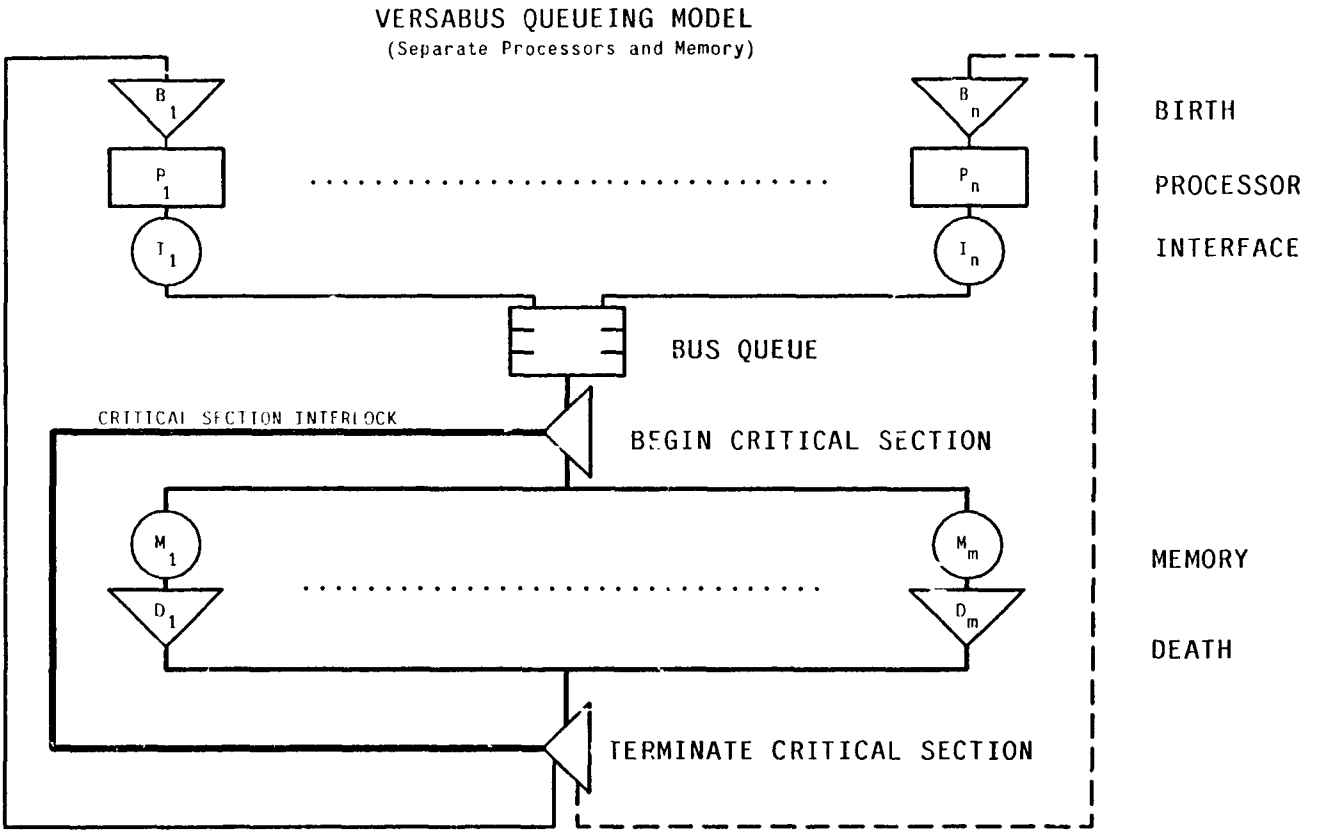

Figure 3-1: Use of separate processor and memory cards forces all processors to be served by a central memory server on all memory reference cycles.

represents the service time of a bus memory unit.). Simulation confirmed a SEM which leveled off as more processors were added at less than twice the throughput of a single processor, as illustrated in figure 3-2.

The full separation of storage and processing functions shown here represent an extreme corner of the design space, unsuitable for a performance multiprocessor with a single circuit-switched interchange system such as Versabus. Practical systems using separate processor and memory units are possible using high speed, very parallel interconnects with simultaneous access to multiple memory units, or through the buffering of shared memory, perhaps in a cache.

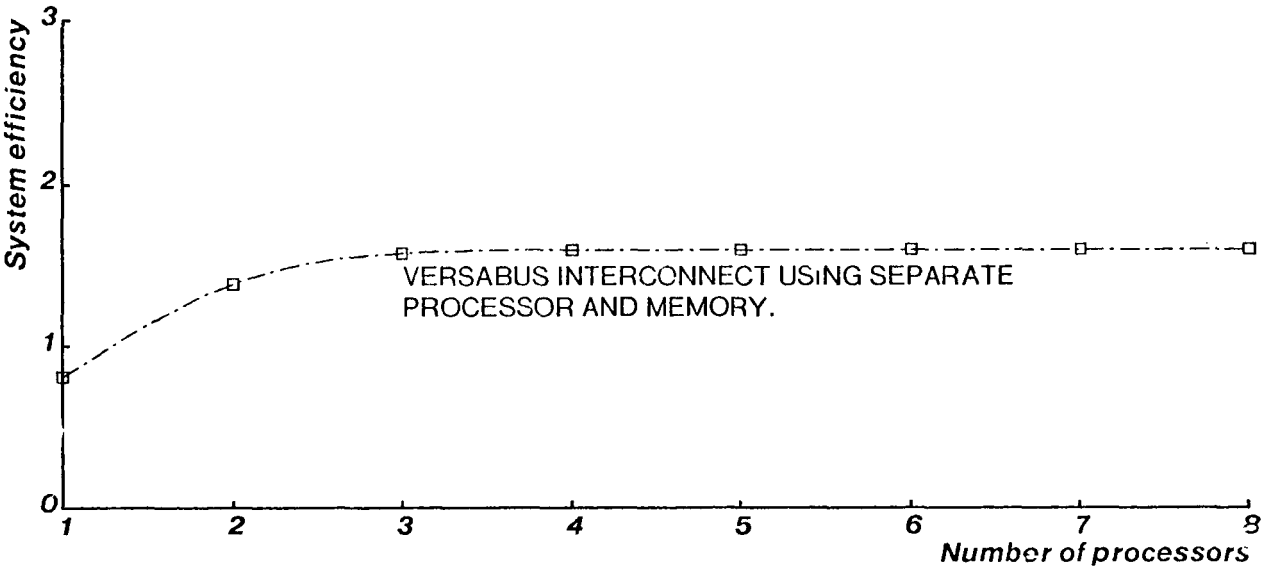

Figure 3-2: With separate Versabus processor and memory cards communicating on the bus, our system would level out at less than twice the performance of a single processor.

3.3.1.3. RAPIDbus I - Bus Memory

Addition of RAPIDbus interface cards to a separate processor and memory system replaces the critical section containing the Versabus with independent queue at each of the system memory cards as illustrated in figure 3-3.

Under simulation, the performance of one or two processors was slightly lower than than in the Versabus control (see figure 3-4), as a result of additional interface overhead. From the third processor on, the results show improvement. With four processors there is a fifty percent increase in system efficiency. In an eight processor system, the increase is more than 170 percent, appearing to increase linearly, although not with unity slope, up to the system implementation limit.

3.3.1.4. Versabus - Local and Global Processor Memory

Contention for system resources can be reduced along with the average memory service time if some memory space is located on the processor card, visible only to the local processor. Such a structure is shown in figure 3-5. This can take the form of either explicitly addressable local memory, or transparently addressable memory such as a cache.

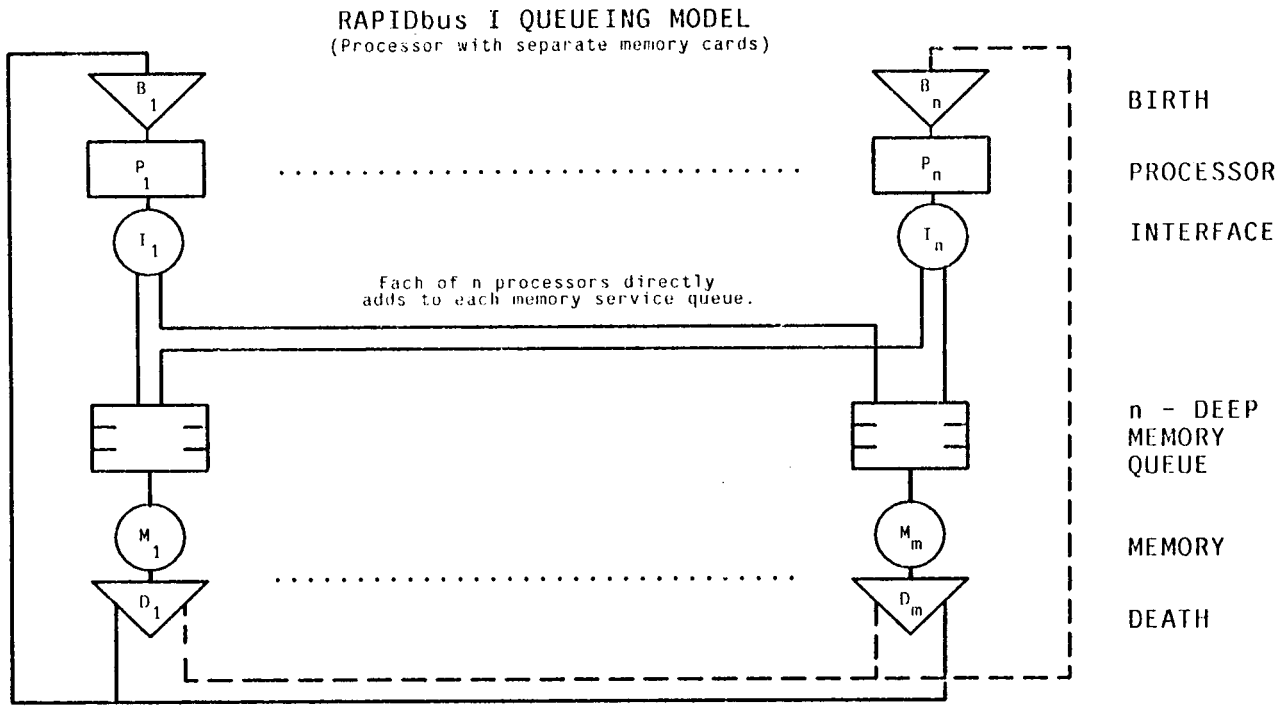

Figure 3-3: Addition of the RAPIDbus interface to a Versabus system with separate processors and memory removes the critical section enveloping bus memory.

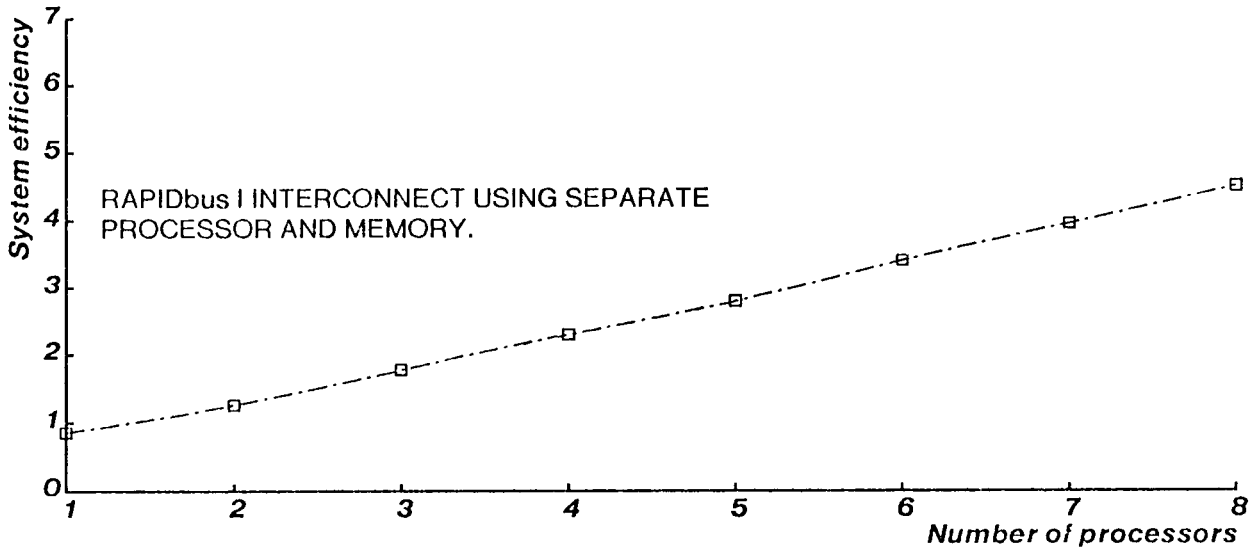

Figure 3-4: Separate processors and memory cards running with the RAPIDbus interface dramatically improve the throughput, but in an absolute sense, still runs poorly.

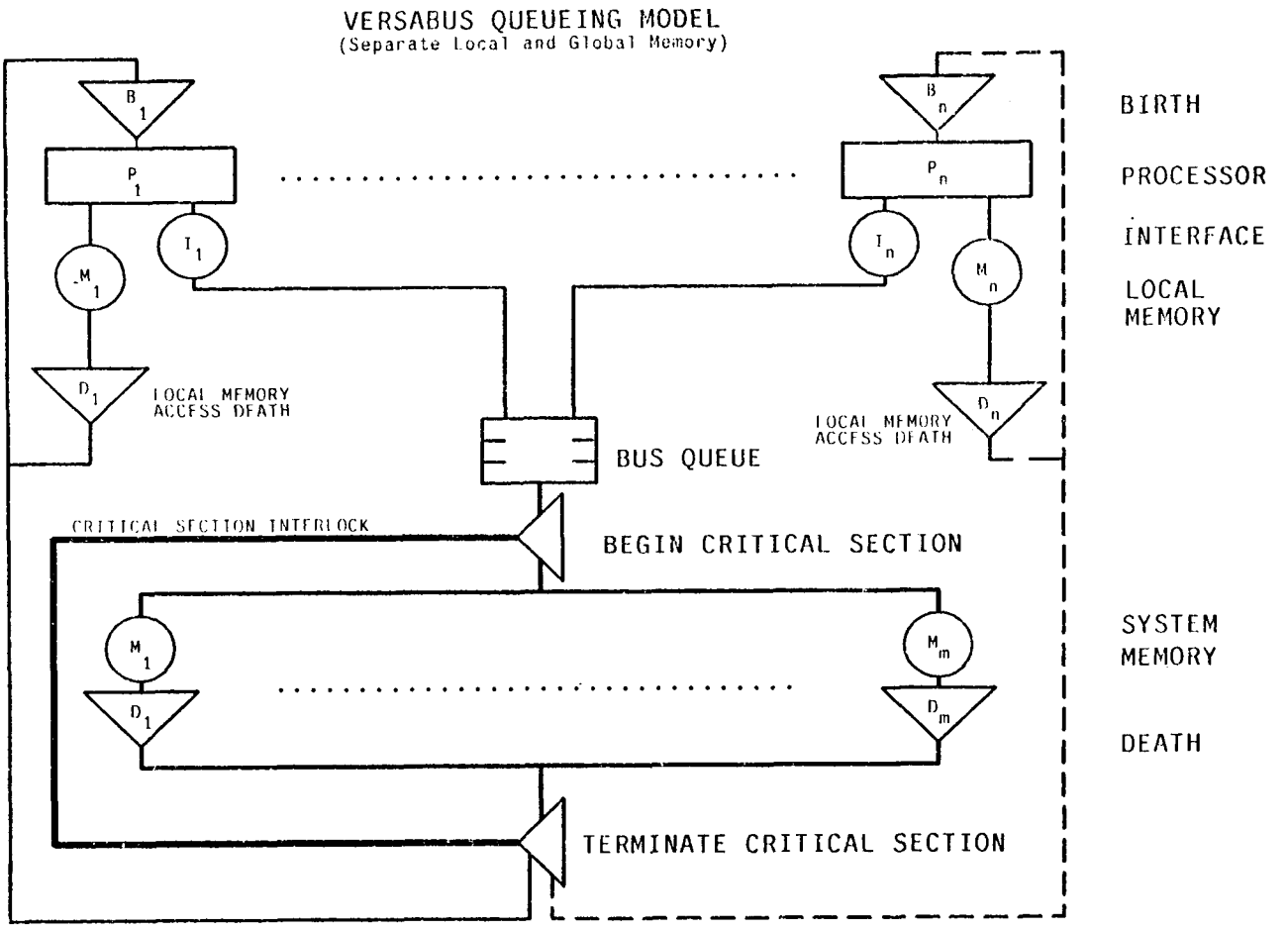

Figure 3-5: Addition of local memory on each processor decreases contention for main memory while increasing the complexity of memory allocation.

If the memory is explicitly addressable, then variables must be partitioned between local and global memory segments. Contemporary multiprocessors often reserve the local storage for system software, where the partitioning burden is felt less often. Alternately the programmer, perhaps with high level language assistance, can separate the variables. If the system is required to suspend a process being executed and restart on another processor, then any required local variables must be moved to another physical location within the address space visible to the restarting processor. If the restart is required because the first processor failed, a problem exists.

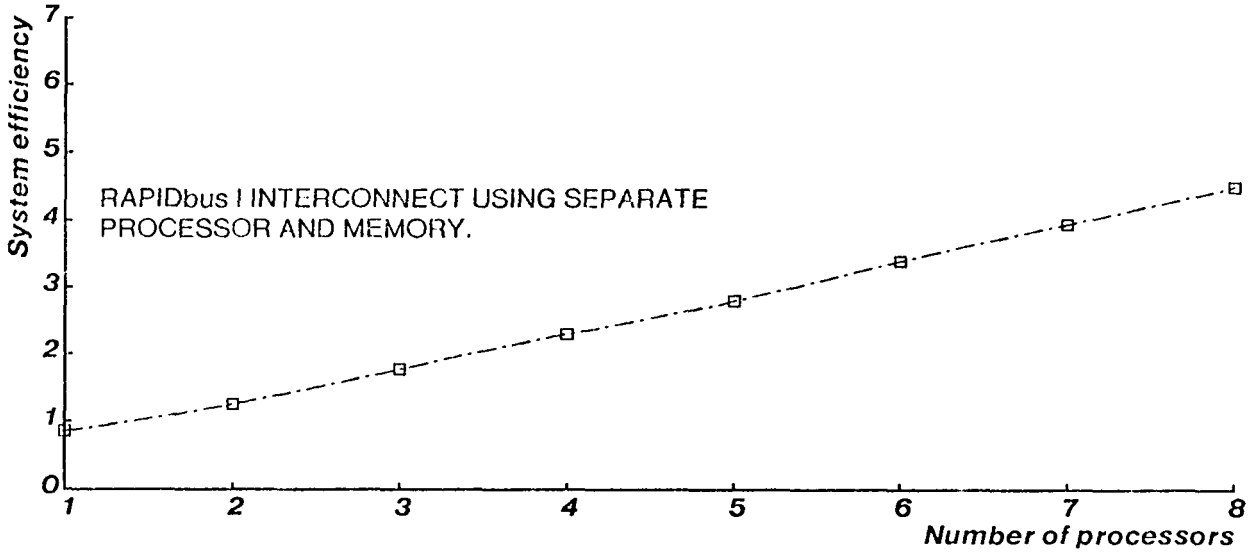

Figure 3-6: Addition of local memory decreases load on the system bus at the expense of a possible increase in the complexity of the programming environment.

If the memory is transparent to the programmer, such as a cache, then the partitioning is taken care of (all variables can be assigned to a single address space), but the problem of cache coherency arises [57]. In common bus systems, each cache will often monitor bus activity for write operations which invalidate an entry in the local cache. The Charles River Universe 68 is an example of a cached Versabus processor which monitors the common bus to stay coherent [17].

In the queuing model, figure 3-5, any memory reference mapped to the bus will still see a single server with unit parallelism. The major difference is that ρ% of the references never become part of bus traffic. If the local memory takes the form of a cache, stable performance values for a properly designed cache range from .80 to .90 (the hit ratio). With explicitly addressed memory, ρ values are less accurately estimated.

As shown in figure 3-6, performance is greatly improved for high ρ values (most references local), relative to the separate processor and memory configuration shown in figure 3-2. The simulation run with ρ = .20 and eight processors required 160,490 clock cycles, or about 39,000 cycles fewer than the system with independent processor and memory cards.

3.3.1.5. RAPIDbus I - Local and Bus Memory

Addition of the RAPIDbus interface to a system with both local and bus memory, as illustrated in figure 3-7 removes contention for the bus from the remaining (1-ρ)% of the memory references. The single memory server is replaced with M separate servers, each with roughly the same service time. Unlike the Versabus case, adding memory cards to the bus will increase performance.

Simulation of this configuration, figure 3-8, showed an increase in throughput for systems with three or more processors and a ρ less than .8. For high ρ values, and few processors, the increased system overhead actually decreased throughput. With eight processors and a ρ of .8, the RAPIDbus interface increases performance by ten percent. For ρ of .4, the four processors see a sixteen percent performance increase. With eight processors, performance jumps 117%. In order to control the effect of addition blocks of memory, a memory card was added to the simulation for each processor added, keeping M equal to N.

3.3.1.6. Versabus - Dual Ported Memory

Some of the disadvantages of strictly local memory on the processor card can be removed by dual porting the memory both to the local processor and the system bus, as shown in figure 3-9, replacing the global memory cards used earlier. Dual porting increases the effective bus bandwidth since communication of a word on the bus can be accomplished with a single transfer cycle compared with the bus write followed by a read required on a bus with separate memory. With a slight performance disadvantage, processes can be suspended and restarted without active assistance from the initial processor.

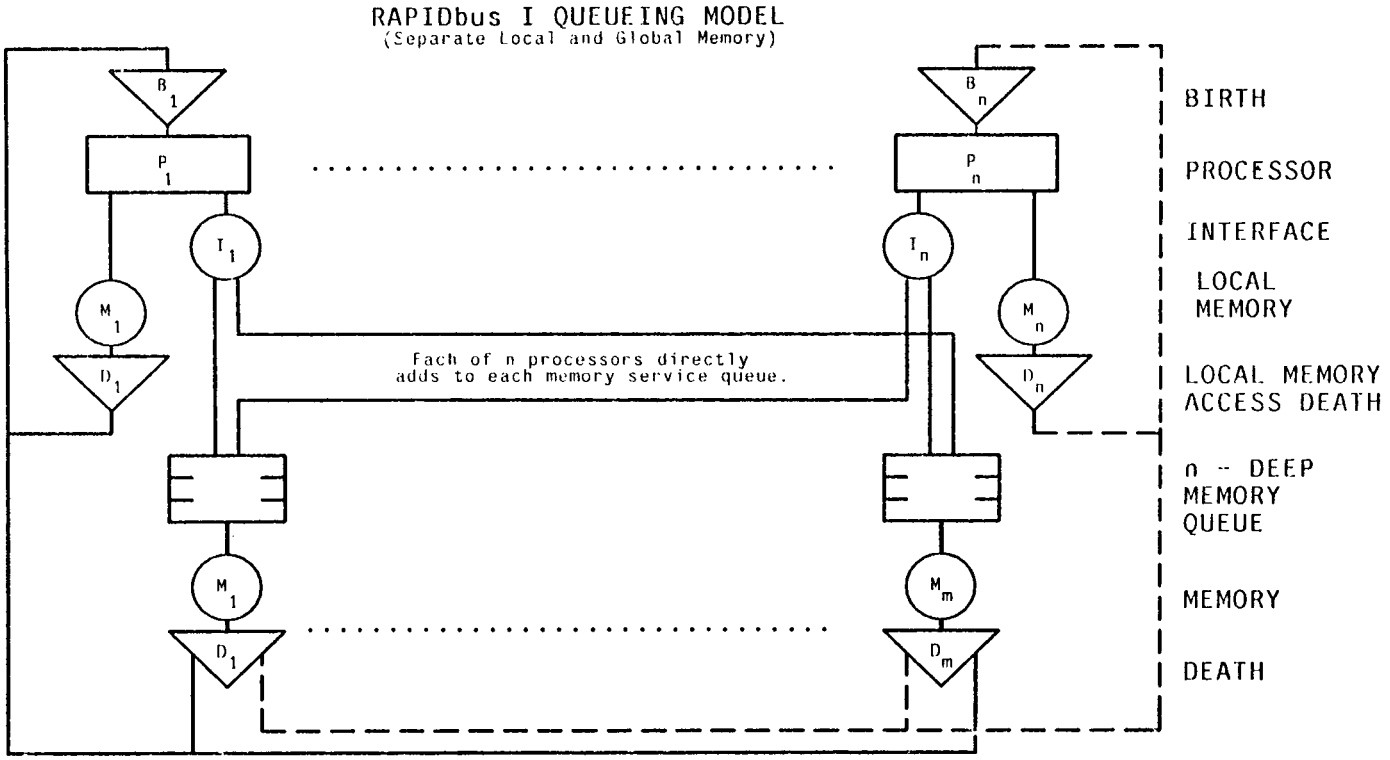

Figure 3-7: Addition of RAPIDbus interfaces to a Versabus system with local memory removes bus contention for those references mapped to the system bus.

Any reference to memory blocks on other processor cards must still contend with the single Versabus server surrounding all bus memory ports. Contention also exists between references by the local processor to local memory, and references by foreign processors to the local memory. Simulation results, shown in figure 3-10, suggest that the effect of contention for each dual ported memory is less than the contention for the bus with separate local and global memory shown in figure 3-6. As expected, the improvement is greatest for systems with much interprocessor communication (low ρ), and many processors. In the extreme case, with a ρ of .2, and eight processors, there is a 97% increase in performance due strictly to dual porting.

In order to keep the use of ρ comparable across configurations, half of all references which the local memory configuration would have mapped to the bus are directed to the dual port memory instead. This assumes that each word communicated is sent only to a single destination. If the communication is read many times for each update, then the improvement due to dual porting would be decreased.

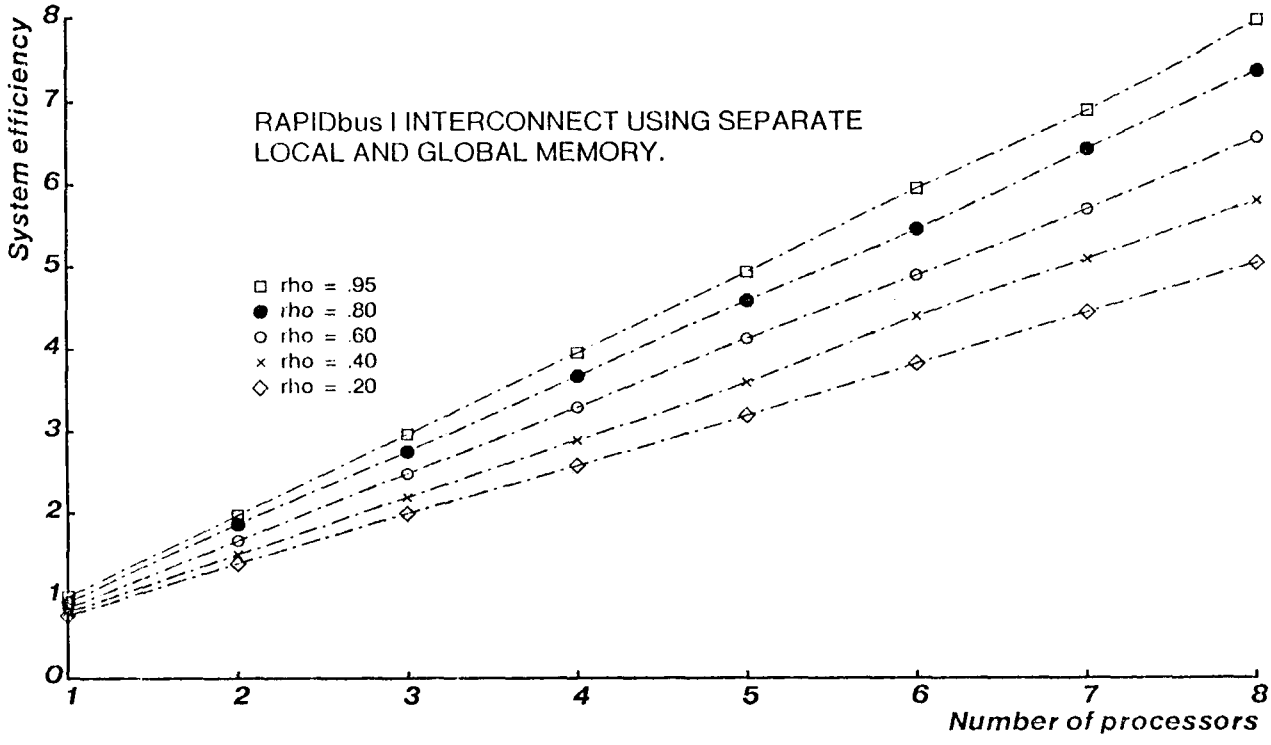

Figure 3-8: Addition of RAPIDbus interface cards decreases contention for the system bus, improving performance in systems with low ρ and more than three processors.

3.3.1.7. RAPIDbus I - Dual ported Memory

Addition of the RAPIDbus interface to a dual ported Versabus system removes bus contention while introducing the possibility of port contention and of deadlock, as shown in figure 3-11. As with each of the above RAPIDbus enhancements, the single critical section containing the Versabus is replaced by independent critical sections around each of the memory blocks, located one to a card. Unfortunately dual porting encloses the host processor's port to the system bus in the same critical section as the local dual port memory. The local processor can't access either local or other bus memory while another processor is accessing the local memory through the dual port, decreasing performance. Relative to a system with separate memory and processor ports, dual porting reduces the number of ports to the common bus by fifty percent.

Deadlock represents a more serious problem in the RAPIDbus I design using a dual ported processor/memory card. Since the local Ibus is arbitrated on each interface card, it is

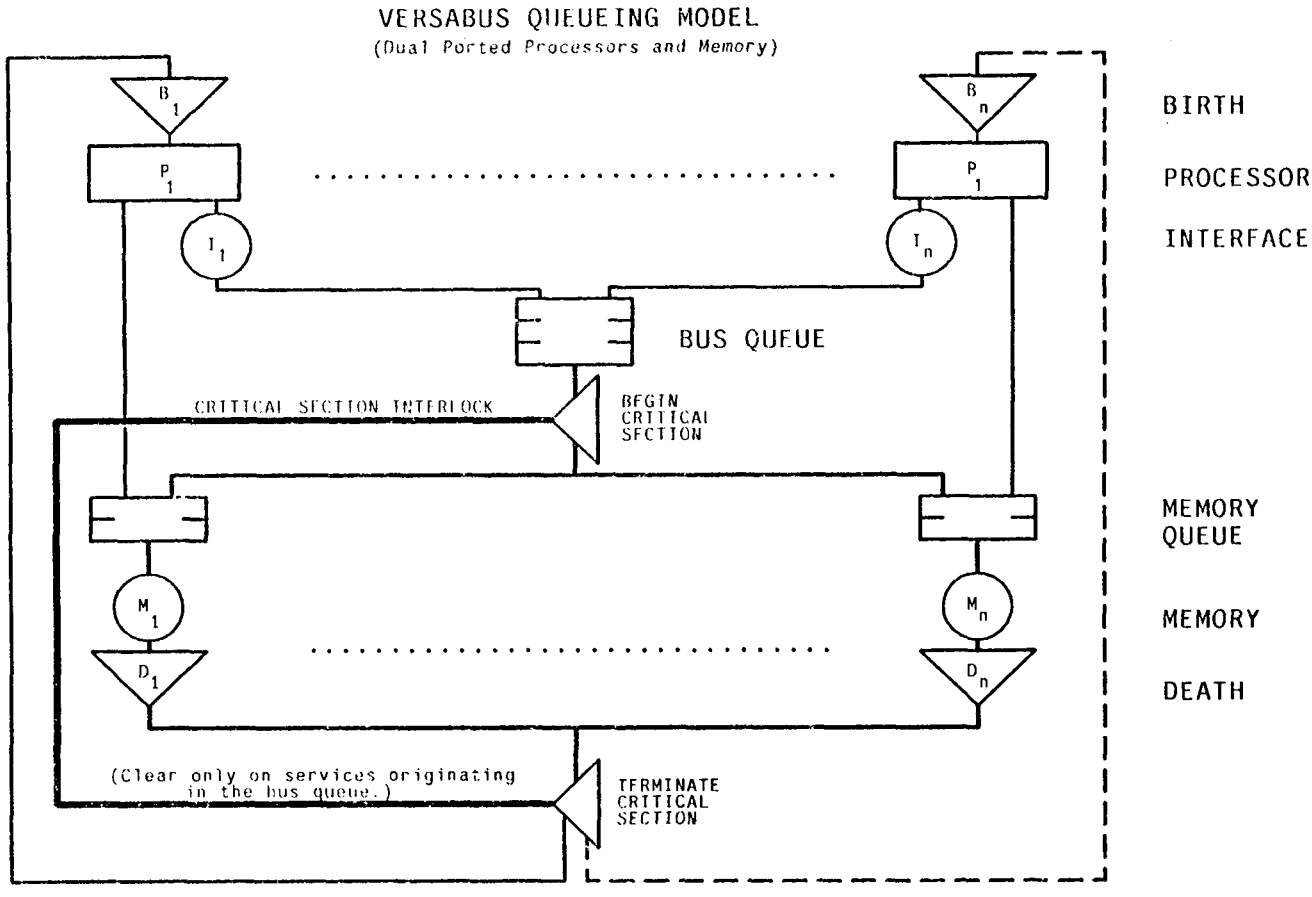

Figure 3-9: Dual porting the local memory to the system bus decreases bus contention relative to separate memory cards without the disadvantages of purely local memory.

possible for two hosts to be granted access to their local Versabus port and Ibus before either has requested the required slave memory port. The retry line, added to the Versabus specification for operation with a RAPIDbus interface allows deadlock to be resolved by timing out and repeating one bus reference at a time, unless the instruction to be retried is an atomic read-modify-write. By design, the 68000 will not retry such an instruction and automatically traps to an exception handler routine. In order to reliably run transfers related to any instruction, a more extensive modification of Versabus is required for use in a practical RAPIDbus system.

Simulation of the RAPIDbus I enhanced dual port system showed an increase in throughput over a similar RAPIDbus enhanced system with strictly local memory only for systems with low

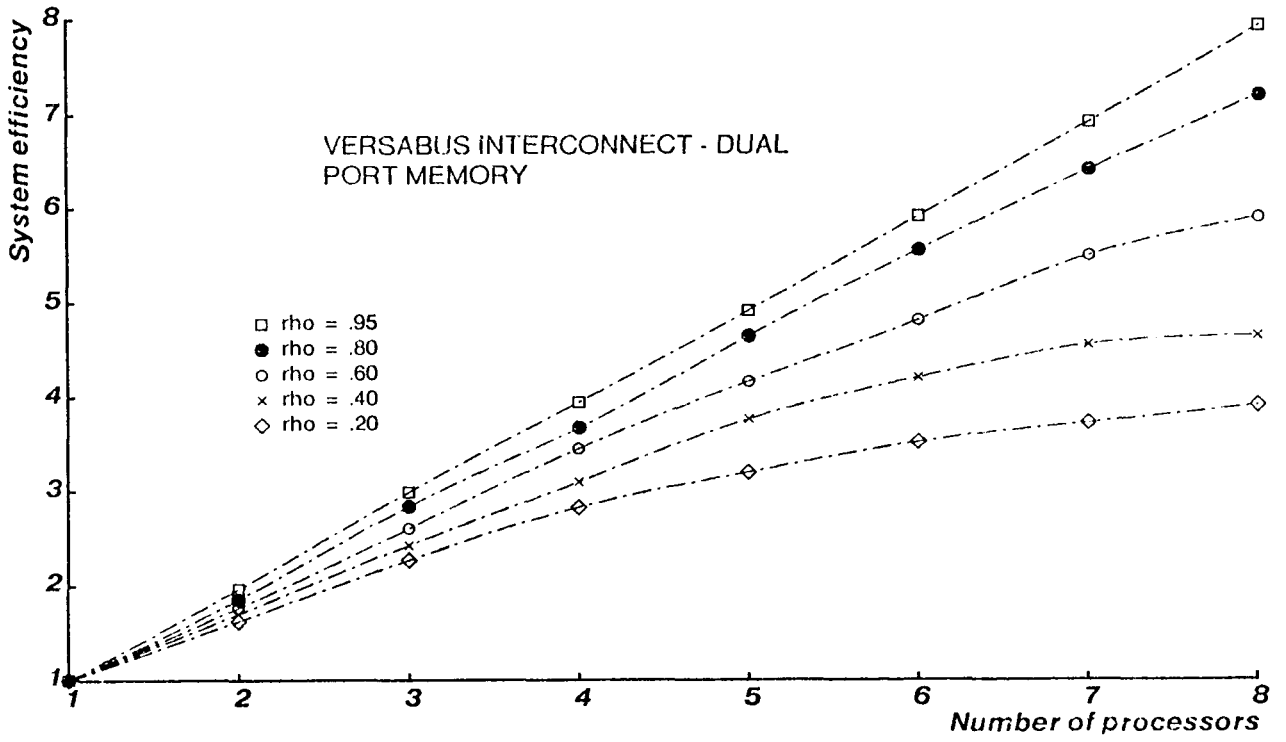

Figure 3-10: Dual porting the memory local to the processor decreases bus contention and simplifies restarting a suspended process.

ρ values. For a ρ of .8, system performance was roughly identical for both configurations. Only when the ρ dropped to .2 did the performance improvement exceed ten percent.[11]

Figure 3-12 allows direct comparison with a Versabus dual port system, shown in figure 3-10. The addition of the RAPIDbus interface increased system efficiency noticeably only for ρ values of .6 or less. At best, with a ρ of .2, and eight processors, performance increased twenty-nine percent.

3.3.1.8. Structural Conclusions

The results of the above discussion and simulation are not too surprising. The RAPIDbus interface is most effective in systems with high bus bandwidth, either because of the number of processors, their structure, or the nature of the code. More efficient structures, such as a dual ported memory benefit least. In the two later configurations, the most efficient in an

[11] With a ρ of .2, performance was enhanced 12% for the four processor system and 13% for an eight processor system.

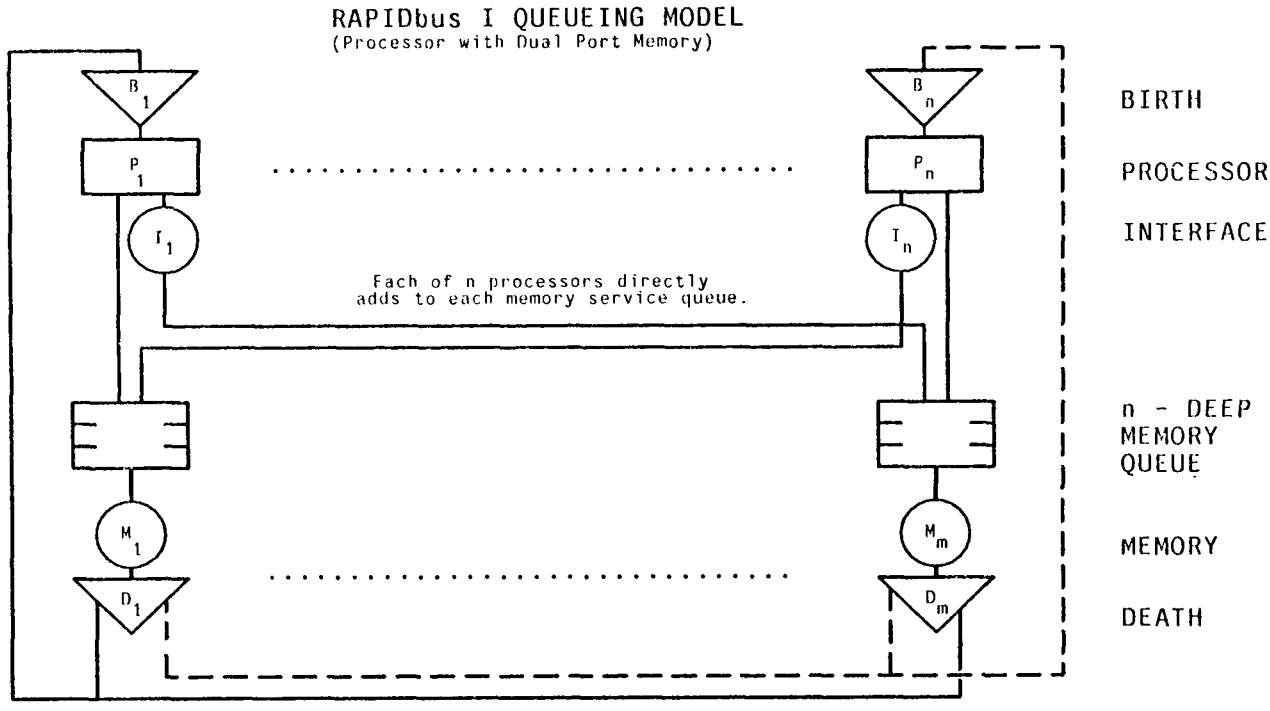

Figure 3-11: Addition of a RAPIDbus interface to a dual port Versabus system decreases bus contention while introducing the possibility of deadlock.

absolute sense, the RAPIDbus I architecture never increased performance more than a factor 1.3 with four processors, and 2.5 with eight.[12]

There is an inherent bias toward Versabus in the performance modeling, resulting from the two interfaces (Versabus and RAPIDbus) traversed in the RAPIDbus configuration versus one in the Versabus configuration. Replacement of the Versabus interface on the processor by a RAPIDbus interface would increase throughput, but probably not dramatically. Revisions need to be explored at all levels in order to provide a more significant performance improvement, and to accommodate systems with many processors, where RAPIDbus appears to have real value.

[12]This is for local and global memory. With dual porting, the improvement was even less significant.

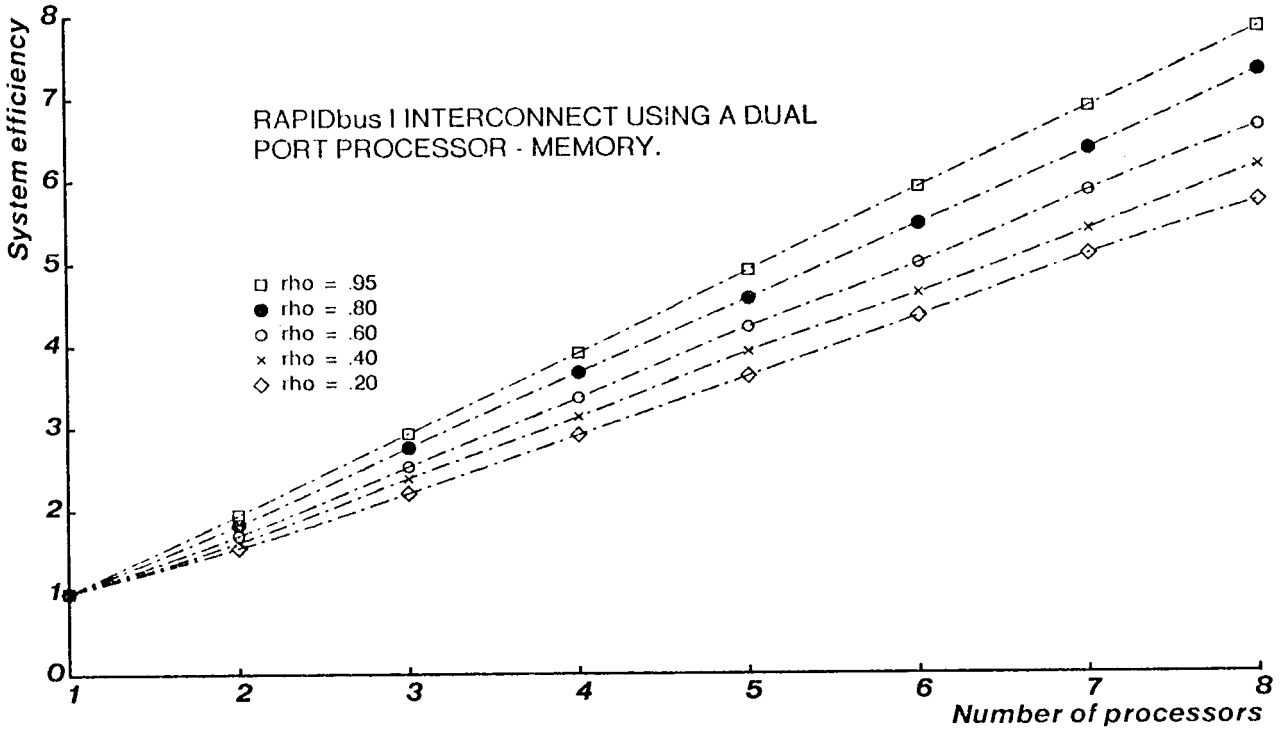

Figure 3-12: Addition of the RAPIDbus interface to a dual port system produces a very limited increase in system throughput for any but the lowest ρ values.

3.3.2. Bus Utilization

In order to increase the number of processors communicating along the same physical bus without decreasing the response to any one processor, it is useful to evaluate the informational content of the bus, seeking strategies for bandwidth compression.

Analysis is facilitated through division of time-multiplexed lines into three categories; the master, data, and slave busses. The master bus is always driven by the processor initiating the data transfer. The data bus can be driven by the master or the slave depending on the state of the write line.[13] The slave bus is always driven by the slave card responding to the transfer master.

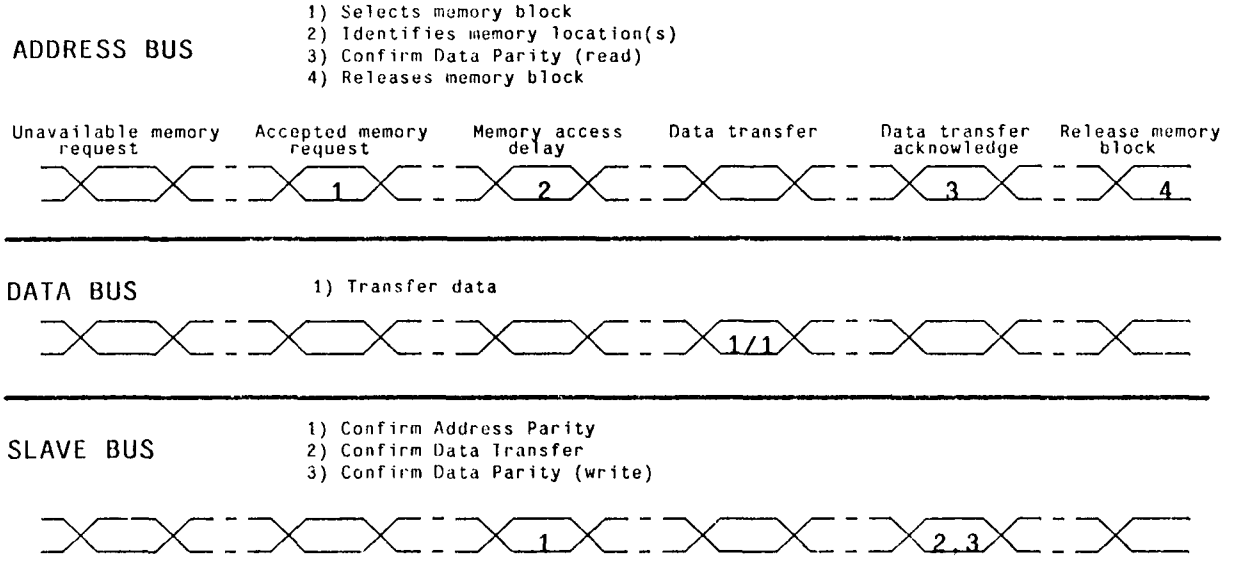

Figure 3-13: Analysis of the information content during each window of a RAPIDbus I data transfer operation suggests more efficient transmission protocols.

Following the virtual Versabus strategy, RAPIDbus I updates the connection between exchanging master and slave on every fourth bus cycle on all lines. For simple bus cycles, most lines are latched at the host receiving end only once during the entire data transfer process. Thus a great deal of redundant information is being transmitted on the bus without increasing the system fault tolerance or performance.

---

[13] In a sixteen bit data path, twenty-four bit address system the master bus consists of address, address modifier, three strobes, write, address parity, and a read parity confirmation line [33 lines]. The data bus includes data and data parity [18 lines]. The slave bus includes a data acknowledge, and bus error line to confirm address strobe and write data parity [2 lines].

Consider a typical data transfer cycle such as illustrated in figure 3-13. During the first bus cycle, the four most significant address bits and the address strobe are required to select a destination card. If the destination card is not available, these lines must be repeated until the destination becomes available. On these cycles, five of 53 lines are used, resulting in 9.4% informational efficiency.

After the destination becomes available, the full address can be sent effectively using 32 of 53 lines for a 60% efficiency. Once primed, the following cycle can be used to transfer data and confirm address parity for a write (a read would probably use the same lines a cycle later). On the data transfer, 19 lines are used for a 36% data transfer efficiency. On the following cycle, before the write takes place, or during the cycle preceding data transfer for a read, no information is transferred, resulting in 0% efficiency. Data transfer acknowledge and data parity confirmation require two lines for a 3.7% informational efficiency. The final bus transfer frees the destination as address strobe is removed with 1.8% efficiency.

Thus the six bus cycles used by current Versabus cards in a RAPIDbus configuration have a combined efficiency of 18.5%. A comparable thirty-two bit address and data path version would achieve a 20.8% efficiency. In contrast, a Versabus system with a similar bandwidth backplane would only achieve a 4.7% efficiency over sixteen bits and 5.2% efficiency over 32 bit paths.

This suggests that decomposing the bus into separately switched groups of lines covering destination allocation, memory location selection within a card, address parity, data transfer, data parity, data acknowledge, and destination deallocation would result in a nearly five times increase in effective bus bandwidth. Relative to the control Versabus configuration, effective bandwidth would climb by a factor of nearly twenty. This increased bandwidth could be utilized by additional processor ports.

Making use of this potential bandwidth suggests increased control complexity required to independently drive and latch lines with six different functionalities and timings. Whereas some additional complexity is undeniable, the relationship between all of the lines is not independent. The address parity acknowledge always follows the window specifying the address location within a memory block by a known interval. Similarly the data parity acknowledge also follows the data by a known delay.

3.3.3. Bus Allocation

RAPIDbus I uses a rotary allocation scheme for the assignment of virtual bus time slices to the physical bus. This approach to bus allocation assigns equal bus bandwidth to each of the four processors in the system, avoiding use of a global bus arbiter with the accompanying two way communication delays.

Considering only bus allocation on RAPIDbus I, the bus communications latency between master and slave on a virtual bus is never more than four bus cycles or two processor clock cycles, with an average delay of one processor clock cycle. Since implementation of a bus arbiter with the same semiconductor technology used on RAPIDbus I would result in a processor cycle delay, there is no performance advantage here to either strategy. In order to appreciate the bus allocation design space it is useful to consider the original RAPIDbus allocation scheme from which Bracho derived the bus allocation scheme used on RAPIDbus I [13].

The RAPIDbus design proposed by Zoccoli was based on 25 of Zilog's Z8001 microprocessors running at four Mhz [81]. Unlike the heavily nano/microcoded 68000 processor used in later RAPIDbus machines, the Z8001 uses a hardwire instruction decoder and sequencer which results in fewer, but longer average clock cycles. Zoccoli took advantage of this clock parsimony and the bandwidth afforded by ECL to allow each of the 25 processors to receive a bus time slot during each processor clock cycle. Since the processor was tightly coupled to the high speed bus, the processor clock phase could easily be synchronously shifted so as to minimize the time between control generation by the processor and the occurrence of a processor's bus slot.

While the possibility of implementing the required ten nanosecond windows is open to question at the implementation level, the viability of the rotary bus access is clear here.[14] The communications latency across any one of the twenty-five virtual busses is largely limited by potential timing skew in the processor relative to the clock, without any arbitration delay component. Performance is bought here at the expense of a tremendous bandwidth differential between processor and bus technologies.

RAPIDbus I was constrained to use a bus implementation technology which afforded much

---

[14] No commercial processor has ever been delivered which ran a 10 nanosecond common bus packet.

lower bandwidth than the 10K ECL used on the original RAPIDbus. At the same time, the switch to a microcoded processor running at higher speeds than the Z8001 decreased the time interval between potential generation of new control signals. Use of the 68000 on an Advanced TTL bus shifted the break even point from twenty-five to four processors for a bus arbiter. As the designers of busses such as Multibus and Versabus are keenly aware, use of a dedicated bus protocol reduces the break even point for bus arbitration down to a two processor system; both use bus arbiters, not dedicated time slots.

3.3.4. Interrupt Structure

The exception structure within a real-time, multitasking system forms a priority scheduler, implemented largely in hardware, which runs asynchronous to the regular software scheduler. The priority scheduler causes currently executing code to be suspended, resuming execution with code from a particular exception handler process.

In the context of a multiprocessor system, exceptions can be broadly divided into processor specific and system exceptions. Processor specific exceptions, such as a hardware bus error or an instruction fault are sensitive to the context in which the processor was executing when the exception was raised, and are best handled on a particular processor. System exceptions occur when the exception handler must be scheduled on a processor separated from the exception generator by the system interchange network. Analogous to the binding of processes by the software scheduler, the system exception may need to run on a particular processor, one of a particular class of processors, or on all processors in the system.

Under Versabus, system exceptions are asserted along one of seven open collector lines available to all processors. In response to assertion of a particular interrupt line, a hardware assigned processor requests a vector along the Versabus from the exception generator describing the requested service through an indirect pointer to an exception handler process. Since more than one Versabus card can raise the same exception simultaneously, the request for vector is daisy-chained until a card is reached which pulled the appropriate interrupt line. The Versabus protocol permits only seven different interrupt handlers, each with 256 possible handler processes. Beyond the hardwired power failure and reset exceptions, which are non-vectored, there is no provision for the exception handler being activated in either a class of processors, or on all processors simultaneously in the Versabus design.

The RAPIDbus concept of four independent virtual Versabuses was complicated by the

Versabus interrupt structure. The open- collector interrupt lines were no problem, however the daisy chain used in the interrupt acknowledge bus cycle could not be practically time multiplexed. The arbitration capability mechanism used to assign the daisy chain to a particular virtual Versabus for the duration of an interrupt acknowledge cycle violated the orthogonality of the busses, and required additional distributed logic.

System extensibility and implementation simplicity would be enhanced if special backplane lines such as the open-collector interrupt lines and the daisy-chain interrupt acknowledge lines were not required. The interrupt acknowledge cycle's addressing based on the three lowest bits of the address, the address modifier lines, and the daisy chain also limits extensibility (a three bit address space for exception handlers), and requires gates not used for any other cycle on the high speed side of the bus interface.

3.3.5. Multicast Capability

In the quest for optimal use of RAPIDbus I host and bus bandwidth, it was attractive to consider directing a single bus write cycle to multiple destinations. Once the overhead required for initialization is complete, this capability effectively reduces latency and increase the apparent transfer bandwidth for the associated cycles. Bracho proposed multicasting in the RAPIDbus I design to facilitate an approach to image preprocessing in which several processors would simultaneously run different operators on the same incoming image [13].

Implementation of a one to many bus transfer is relatively inexpensive when the interchange network is based on a common bus where all interfaces have access to all bus transfers. It is at the realization level that multicasting on even a common bus starts becoming expensive. The originating master must collect transfer acknowledges from each of several potential destinations before finishing the bus cycle. Without providing separate acknowledge lines or an acknowledge arbitration, this requires use of an open collector multicast acknowledge line. Since TTL open collector technology does not permit the same short windows that are allowed by tri-state technology, the designer could either increase the system window size, or implement a multicast acknowledge line which is not time multiplexed. If the window size is increased enough to allow use of a TTL open-collector line, bus throughput will be seriously compromised for all data transfer cycles. Use of a special, non-multiplexed line forces arbitration for the multicast capability. In the RAPIDbus I realization, eight percent of the interchange logic was directly tied to supporting the multicast capability.

Unfortunately, the special provisions required for a one to many routing become an increasing portion of the total system complexity as system performance increases. Without technological changes, the interconnect performance can be dramatically increased only by increasing parallelism so that not all communication between system elements is visible to all system interfaces. Routing information can no longer be encoded as an N bit number for 2^N destinations, but rather increases to 2^N separate lines. Creating a consistent implementation required this overhead cost be paid even for interchange ports which would not use the multicast capability.

3.4. Realization

At the realization level, he sought to learn how to build reliable, complex digital hardware operating at cycle times below 100 nanoseconds. This goal was realized through the successful fabrication of two interface cards capable of reliable digital communication with a 64 nanosecond window.

3.4.1. Asynchrony

The greatest single challenge at the realization level arose from the inherent conflict between reliability and performance at each interface between the asynchronous Versabus and the synchronous interchange network. Our experience, in complete agreement with the consensus of current literature, pointed to the need to avoid asynchronous interfaces wherever possible within a high performance machine [72]. When an asynchronous interface is absolutely required, the lines involved must be identified and properly synchronized, often at the expense of performance.

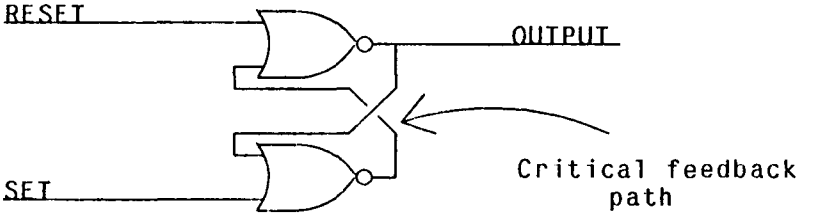

Figure 3-14: Bistable elements, designed conceptually like that above form the basis of the metastable problem.

Synchronization problems may potentially arise any time two digital subsystems running on different time bases attempt to exchange information. Since the incoming information is not

guaranteed to have any particular timing relationship to state changes in the receiving subsystem, a synchronizer is required to permit only intended state changes. Unfortunately there are no known methods for absolutely reliable synchronization in a bounded amount of time.

Some version of a bistable element, or flip-flop, forms the basis of any synchronizer. Using digital logic gate notation, a bistable element can be represented as in figure 3-14. If the set input is asserted the output stabilizes to a high value. Likewise, assertion of the reset terminal will lead to a stabilization at a low output value. Unfortunately, the feedback path between the two elements of the bistable requires a finite amount of time to communicate information. When the bistable is incorporated into a D latch, as shown in figure 3-15, this critical time delay results in a temporal window of vulnerability for each of the two stages in the D latch, often with different critical skew times for the two independent sections [16].

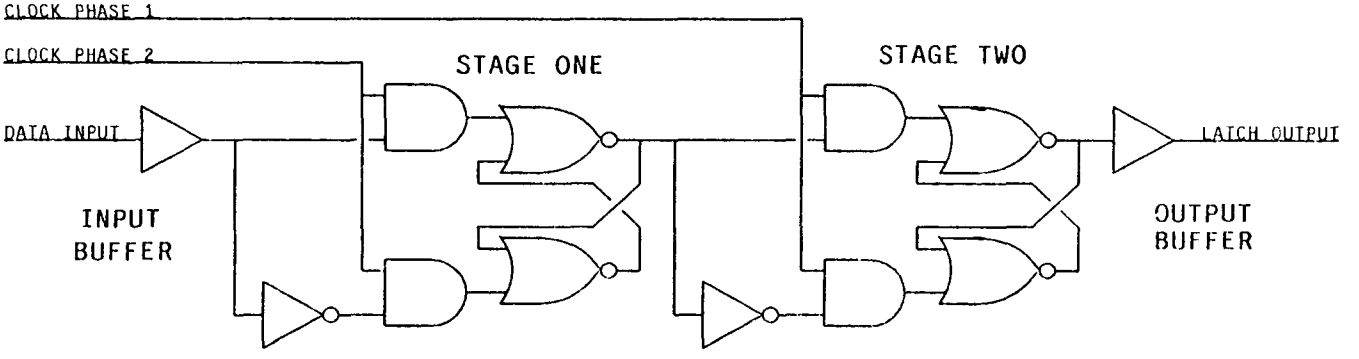

Figure 3-15: A D latch, key to the design of synchronizers, can be represented by structures the two stage structure shown above.

The digital flip-flop, like any physically realizable bistable element, must have an intermediate, or metastable region. In the first stage of the flip-flop, if the clock and data change state well separated in time, the bistable element can reach steady-state across the feedback path. Under steady-state conditions, the bistable changes state in a well defined period of time. However as changes in the data and clock line approach a slightly skewed simultaneity, the feedback path no longer has time to reach a stable state. The two subcomponents of the bistable reach equality at a metastable voltage about half way between a high and low output.

According to analysis by Couranz and Wann, this metastable voltage is surrounded by a

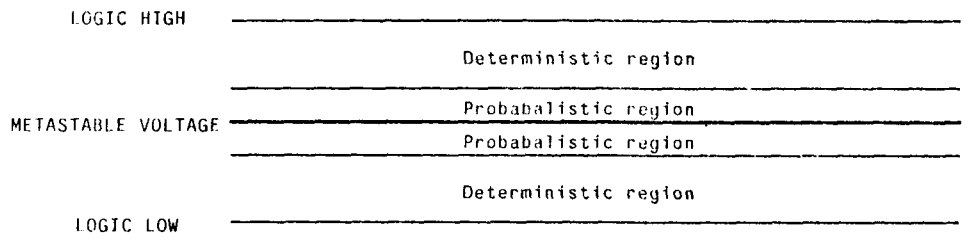

Figure 3-16: The metastable voltage is surrounded by a small probabilistic region, where escape is noise dependent, and a larger deterministic region where the propagation delay is design dependent.

small probabilistic region on either side of the metastable voltage, and a large region on either side which has a deterministic behavior, as shown in figure 3-16 [20]. Once the two components of a bistable have both settled at the metastable voltage, a chance event made more likely as the data and active clock edge approach each other, the time required to escape is largely dependent on the circuit noise. Random noise provides the primary force required to allow escape from the probabilistic region, into a deterministic delay region where transition time to a known output value is dependent largely on circuit design parameters.[15]

In order to quantify the probability distribution function for the duration of the metastable state, researchers have built both accurate simulation models at the silicon level, [75, 20] and developed device level parameterizations for existing devices [72, 31, 16]. The general form for the mean time between metastable states arising from a flip-flop being used as a synchronizer, and still unresolved after time t_{res} is given by...

$$[\exp(t_{res}/\tau]/[T_o * T_{clk} * T_{input}]$$

The flip-flop can be described by τ, the time constant of bistable resolution, and by T_o, given by various researchers as either the "normal" propagation delay [72] or as an experimentally derived parameter [16].

When a designer is aware of the potential problems inherent in communicating across an asynchronous interface, a design similar to that shown in figure 3-17 is used. The first register in subsystem B is allowed the possibility of a metastable state, since the second D latch will not allow the incoming signal to propagate into the B state machine for the duration of the subsystem B timebase period minus the normal propagation delay of the D latch. Inherent in this synchronization scheme is an average communications delay from the output

---

[15]The effective magnitude of the noise voltage can be increased, and the delay of the deterministic region decreased as the load capacitance and series resistance of the resolution node are decreased.

of subsystem A until the communication is allowed to affect the subsystem B state machine of 1.5 times the period of the subsystem B timebase. If the period of the B clock is chosen to be too brief for the resolution time of the chosen bistable element, the possibility of a metastable propagating through to the B subsystem increases.

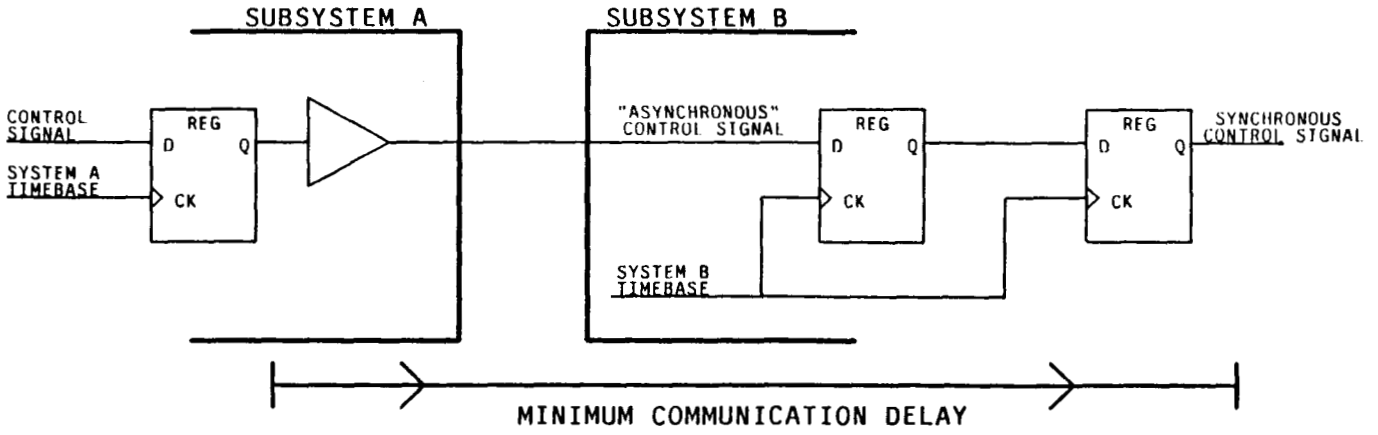

Figure 3-17: Design of a practical system using an asynchronous interface requires a synchronization latency to increase the mean time between metastables that propagate through to the second subsystem.

As complex digital systems become more common and designers strive for higher performance, asynchronous interfaces are often seen as a easy solution. Lurking in the shadows of all too many of these designs is a cavalier attitude toward the pitfalls of the asynchronous interface.

One early implementation of a Versabus single board computer seems to have ignored the synchronization problem entirely. Among other problems, logic was required to arbitrate between asynchronous local and Versabus references to the dual port memory. With roughly 100K transfers trying to compete with the local processor per second, failures which took the effected processor out of the system were observed every two to twenty minutes on five different cards. A two level synchronization circuit was used based on Schottky logic and a twenty-five nanosecond interlatch delay. Using Chaney's data for the 74S373 octal latch, TAU = .91 nanoseconds, T_o = 60 microseconds, and h = 16 nanoseconds [16]. Substituting this into the model, a failure is predicted roughly every minute.[16]

[16]Replacing the sampling latch with an F374 part, and extending the sampling time to fourty nanoseconds, the calculated MTBF for the synchronizer rises to several centuries, clearly a small but important change! Later engineering change orders by the manufacturer are reported to reflect a more reasonable arbitration time.

Asynchronous interfaces are advocated by their supporters as a means of eliminating synchronization delays, reliance on a single clock, and enforced conformity to a particular clock among different subsystems. It is useful to examine the basis for these claims in the majority of applications seeking maximum performance from commercially available logic.

It has often been suggested that use of an asynchronous, "handshake" protocol allows control information to propagate from one subsystem to the next without waiting for the next clock edge. If both subsystems are synchronous state machines (SSM) of some form, then reliable operation requires that all control inputs coming in from a foreign source undergo a synchronization step. With the best TTL technology, this requires about 40 nanoseconds for reliable operation. In contrast, data coming in from another subsystem with a time base synchronous to the receiving SSM will average out to roughly a half clock cycle input delay. Since most performance TTL and MOS system today can support at least a 12 Mhz clock speed, there is no noticeable timing advantage to asynchronous timing. Clock skew between physically distributed components of a system is a potential area of trouble, but through careful clock distribution with well characterized logic, it is possible to distribute a single high speed clock within a moderate size card cage.

Fault tolerant systems where operation must continue following one or more subsystem failures are often forced into use of separate time bases.[17] In most system, the master clock and the distribution logic are both straight forward and are usually reliable. Unless extraordinary means must be taken to prevent a system failure, separate clocks are likely to do more harm than good in a properly designed performance system.

Finally designers often argue that use of an asynchronous interface simplifies the design task through relaxed timing requirements between separately designed subsystems. Whereas it is undenialable that it is easier to make a system initially operate if no set-up and hold times must be met, it is much harder to build such a system with both high reliability and maximal performance. A price is paid for the simple system interface.

---

[17] Zoccoli has proposed the use of a phase-lock oscillator linked to a master time base. Failure of the master clocks would cause separate system clocks to gracefully pick up. Unfortunately such a system would need to incorporate synchronization devices at the interface level to cover the eventuality of global time base loss. Unless these synchronizers could be gracefully added and removed, the performance penalty for separate time bases would be paid even in the primary mode of operation [81].

3.4.2. System Complexity

The few research multiprocessors which have reached the realization stage seem to indicate that there is there is an upper bound on system complexity as expressed by the number of independent components which must be made to work in reliable harmony. An informal survey of projects implemented with commercial, chip level packaging suggests that current design and realization techniques have great difficulty in supporting 10,000 or more active, independent data path packages in a research environment.[18] Advances in computer aided design and fabrication systems may make the implementation of a 10,000 component system easier, however device reliability and characterization problems are not likely to greatly increase this bound in the immediate future.[19]

In order for an architect to increase the complexity of a system, the gate density of each package becomes an important issue. Very high speed logic is severely limited in packaging density. GaAs system designers can hope for 500 gates per custom package, bipolar designers for 2500 gates. With moderate speed logic families, custom MOS designers can hope for a quarter million gates or more per package. Unfortunately the designer of a research machine is often limited by the high cost of developing new packages. Introduction of funding derived from a potential user based reduces freedom to scrap an approach and reimplement.

Accepting an upper bound on the number of component packages in a realizable system, the number of processor societies, and the complexity of each society is directly governed by the ability of the designer to reduce package count per host node. Such a system design is then caught between the need to take advantage of existing, very dense components, such as the 68000, or the 68451 used in RAPIDbus I, and the need for new functionality.

---

[18] Several very expensive projects have tended to suggest that even large influxes of capital don't noticeably alter this upper bound.

[19] Commercially supported machines are able to exceed such a limit at great expense and effort. For instance, the CRAY 1S uses 230,000 gates in the CPU with a packing density of roughly 2 gates per board level package. This complexity comes at the price of tremendous design effort and substantial machine field support. As of 1981, expected MTBF was "more than 100 hours" for the system as a whole [43]. Such support is not realistic for a research project.

Hierarchical computer aided design systems patterned after Lawrence Livermore's SCALD have potential for allowing many mistakes to be discovered prior to fabrication of an implementation, and support the rapid rerealization of an implementation in new media. Thus an early design in discrete logic could in theory be revised and reimplemented with denser, custom or semicustom packaging [63].

Unfortunately, tools which will provide useful assistance for the designer are new, often still in the development stage. As the need for such development tools became evident in preparation for a second implementation, SCALD II was imported from Lawrence Livermore, with the assistance of Dr. Ray Picard at ESL Inc. and Dr. H. T. Kung and group at CMU. Unfortunately SCALD II, while a valid research project, was far from a usable design tool. Shaping a system like SCALD II into a useful tool is a major job which directly subtracts from the effort devoted to the computer architecture. Commercially design systems with modest capability exist, but were not available beyond the schematic capture and net list extraction stage for the design of RAPIDbus I.[20] Experience in the realization of the designs described in this report suggests that a good design environment is *essential*. Use of a less than fully debugged environment, or no environment at all, is a serious mistake if *any* value is assigned to project manpower or development schedules.

RAPIDbus I began with Versabus processor card which required nearly two hundred chips to support a 68000 processor. The Versabus to RAPIDbus interface added over 150 sixteen pin equivalents to the processor node parts count. Whereas we had little control over the host, useful conclusions came from examining the breakdown of chip complexity on the interface, as enumerated by figure 3-18.

The combined latches, drivers, and parity logic represent more than a third of the package count on the RAPIDbus I interface card. Expansion to a thirty-two line address and data bus would have increased contribution for these three sections to more than fifty percent of the the total parts count. These sections also had dominant responsibility for determining bus bandwidth. Recognizing that this was an excellent candidate for custom packaging, independent of the Versabus host used in RAPIDbus I, an interface bit slice was devised as described in the next section.

---

[20]Use of Dario Giuse's drawing package is gratefully acknowledged, but it was never intended to provide a full CAD system with simulation and timing verification capability.

Section Name	Package Count
Window handler	10
Drivers	11
Latches	36
Memory Management	21
Multicast Generator	13
Parity	12
Chip Select	7
Timing Generator	5
Interrupt Control	11
Control	36
Spare	5
Total	167

Figure 3-18: Package distribution on RAPIDbus I interface cards.

The control section and the memory manager were also responsible for large numbers of packages. From analysis of the RAPIDbus I design at the implementation level, it was clear that a new implementation could benefit from an even more complex control algorithm than RAPIDbus I used. Since the control logic was both host dependent, and likely to undergo numerous changes, a custom sequencer package was ruled out. Use of a microcode,

augmented by hardwired RAPIDbus interface logic has potential for both increasing control complexity and decreasing package count. At the architecture level, it was earlier observed that the memory management capability on the interface was another concession to the available Versabus hosts, and was more effectively placed on the processor card.

3.4.3. Bus Interface Integration

Recognizing the importance of the combined driver, address recognition logic, two level latch, and parity logic as a modular package, a custom bit-slice was devised. Use of a custom logic package offered the opportunity to increase bus speed, and decrease package count. The conceptual design is shown in figure 3-19.

ECL logic was recognized as an essential medium for decreasing the length of a bus window, and increasing bus bandwidth. Since the interface host was constrained to a TTL implementation by the available high density commercial components, any bus implemented in commercial 10K ECL would require use of *10124* and *10125* level translators in quad packages. For wide address and data paths, this would have resulted in nearly fifty additional, high powered packages. Realization as a custom module offered the possibility of combining level translators with the other logic without increasing the package count. American Micro Circuits Corporation verified that they could fabricate such a mixed level chip. Initial accessment of the design by AMCC suggested that the sixty-four nanosecond bus window realized in Schottky could be reduced to 12.5 nanoseconds using differential drivers on their ECL gate array.

Referring to figure 3-19, the TTL host address, data, function code, or control bus is attached to an eight line bus at the top of the diagram. Outgoing information is held in the driver latch at the left of the diagram while waiting for a bus cycle grant. For each class of lines on the backplane, only one set of drivers is activated for each bus grant. All bus cycles are held in the first level receiving latch on all cards. During the bus cycle following, either address or data packet tag information can be decoded to identify references to a particular card. Only the card detecting a valid reference will drive the parity acknowledge lines during the second cycle following transmission of the bus. If all goes well, the received bus cycle will propagate through the second and third level latch up to the TTL host bus port on the destination card.[21]

[21]The second and third latching levels are designed to provide sufficient time for address comparisons from all slices in an interface to be joined and propagate out again to each slice.

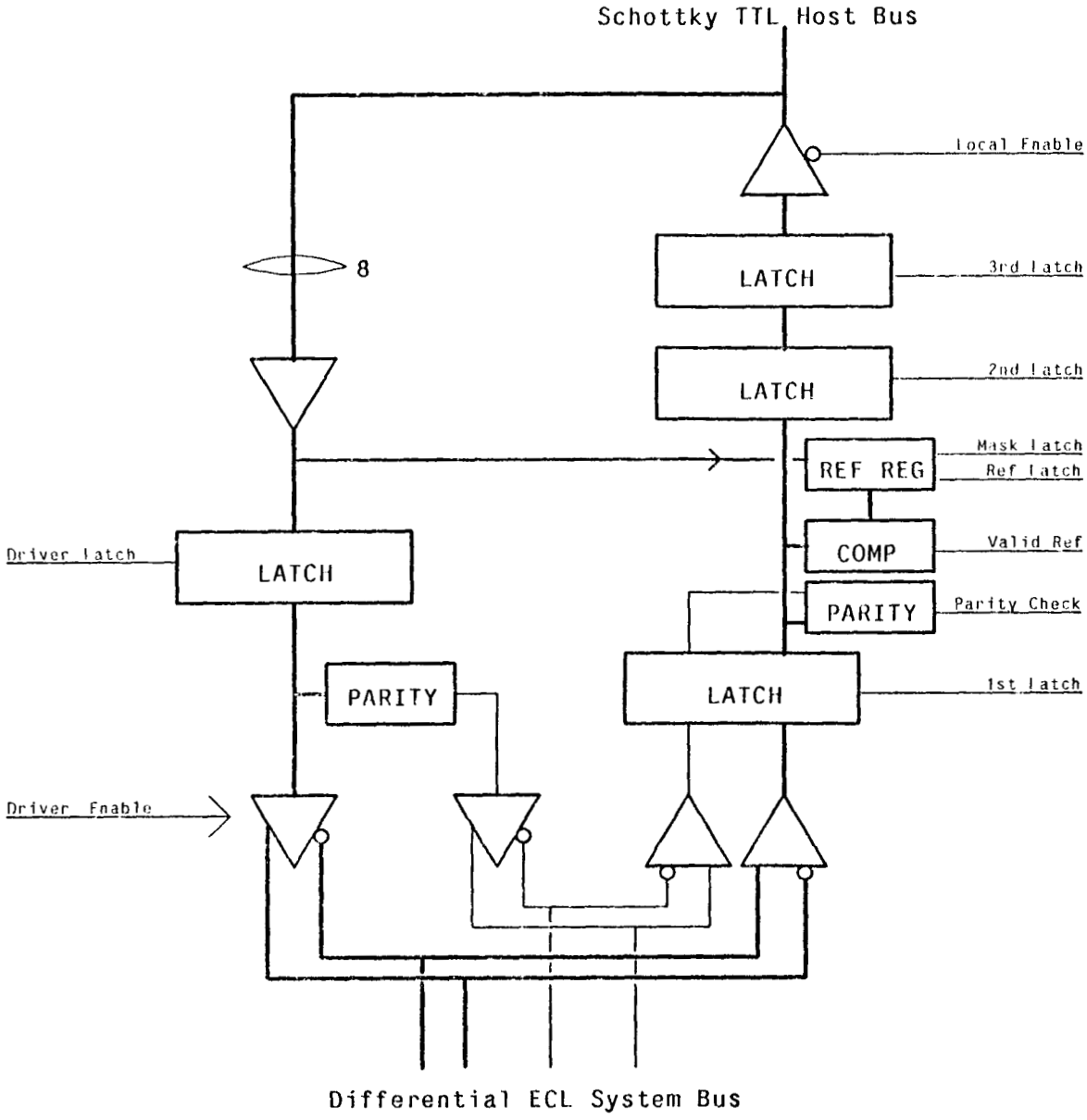

Figure 3-19: Consolidation of latches, drivers, parity logic, and comparators into a translator slice results in a fast, compact time-multiplexed bus.

The high cost of realizing the chip fell beyond the resources available to this project, and was picked up by the CMU VLSI project under Dr. H. T. Kung. With changes to support eight simultaneous outstanding data transfer requests per hosts, and a multiplexed address / data bus, a chip was realized in collaboration with A. Nowatzyk which provided four multiplexed address and data lines per slice. This chip forms the basis for the backplane of the Kung's Universal Host machine. The chip was not available for use with follow-on RAPIDbus designs.

3.4.4. Fabrication Technology

Fabrication represents the last, crucial step in moving from architectural statement to working machine. Requirements for high performance translate at the fabrication stage into use of the highest speed logic logic family available commensurate with system power, size, and cost constraints. In order for the inherent speed of higher speed logic families to become manifest in the realization while not jeopardize reliability, the designer must deal with distinctly analog power supply, interconnect, and thermal issues.

3.4.4.1. Power Supply Engineering

The power supply subsystem extends from the line voltage supplied by the building power system, through conditioning, transformation, rectification, and regulation into a power distribution system which creates the proper power environment at each semiconductor package. Inadequate conditioning of the power environment, or a potential difference between corresponding supply terminals of communicating packages can adversely affect the noise margin, speed, or fanout.

The TTL logic used in RAPIDbus I expects a reference ground and a positive rail supplying between + 4.75 and + 5.25 volts. Within this supply range the noise margin, or minimal voltage difference between an output voltage and the TTL threshold voltage is roughly 450 mV [28]. This noise margin decreases as the positive supply rail dips below 4.75 volts, or with asymmetries in the grounds or positive rails between communicating chips. If the sum of the noise superimposed on the line between driver and any receiver on the same line exceeds the noise margin, unreliable behavior may result. Noise superimposed on the signal line may come from interconnect problems, covered later.

External noise can enter the machine via either electro-static or electro-magnetic coupling to machine wiring. The large variety of power supply and signal interconnects form a myriad of tuned antenna elements. Whereas electro-static energy can be effectively stopped by a tight aluminum enclosure, electro-magnetic energy requires a ferrous metal enclosure with all elements tied to ground.

Noise generated internal to the machine calls for a distributed solution. Most ground and positive rail noise begins with current spikes caused by the synchronous switching of many package output stages. The finite impedance of the current return path through the ground plane results in momentary voltage spikes.

Since RAPIDbus is a synchronous machine using high speed drivers and latches in close proximity, it was important to develop techniques to minimize the voltage spikes that might potentially decrease noise immunity. Ground and positive voltage pins on the dual-in-line (DIP) sockets were tied to corresponding power planes on the wire-wrap card using low inductance, high current copper links. Each high speed package was provided with a .1 μf ceramic capacitor with low series inductance, and adequate capacitance in the radio-frequency region. Slower, but larger variations in the current load are absorbed by Tantalum capacitors ranging from four to fourty μfarads.

Bus drivers are a major source of spikes since by design they have fast edges, sink a great deal of current, and are usually gated on in large numbers. In addition to performance reasons for decreasing the number lines which are simultaneously switched, the realization becomes easier.

The speed and fanout are decreased for high performance components when adequate current is not provided, even momentarily, by the board level design. The rate of voltage change at any given circuit node, either inside the chip, or on external interconnects, is directly proportional to the current available to that portion of the circuit, and inversely proportional to the load capacitance:

$$I = \frac{C\ dV}{dt}$$

Output buffers are typically much slower than gates internal to a chip as a result of the increased loading. If the current available from the power terminals is inadequate, the required slew rate may not be maintained. Since additional inputs tied to an output buffer increase the capacitive load (C), for a fixed current, the slew rate must decrease. In practical terms this underscores the need for both good, low inductance bypassing (instantaneous current available), and the need for low impedance connections back to the power supply sensing terminals. Constraining the fanouts to five for time-critical circuits within a card seemed to work out well with RAPIDbus I.[22] With adequate power engineering, performance is still dependent on the proper handling of interconnects and thermal problems.

---

[22]The available bus bandwidth is constrained by the unavoidable bus capacitance, current limitations on individual drivers in an octal package, and the required fanout to two devices [driver and receiver] on each interface.

3.4.4.2. Interconnect Engineering

The high speed propagation of digital signals between semiconductor packages presents a major challenge to maximizing performance and reliability from a particular performance logic family. High speed bipolar logic such as Advanced Schottky and 10K ECL provide rise and fall times of one to two nanoseconds, with substantial frequency components of several hundred megahertz [71, 25, 27, 10, 7] Transmission of such high speed edges is prone to ringing and crosstalk among lines on a bus. Ringing occurs when the far end of an interconnect is not terminated at the characteristic impedance of the line. TTL logic is at a major disadvantage here since few TTL gates will drive the 100 ohms or less commonly realized with high speed, multilayer PCB technology. A variety of Schottky clamping circuits are now incorporated into high speed TTL in order dampen reflections. ECL logic in contrast can drive lines down to 25 ohms in many cases, allowing termination at the characteristic impedance of the line in order to reduce ringing.

Crosstalk results from fast signals on one conductor inducing large enough voltages in adjacent conductors through capacitive coupling that a state change on the second line is perceived to occur. Use of the relatively random spacing between wires provided by wire wrap minimized this potential concern. In order to avoid problems with ribbon cables transmitting high speed RAPIDbus II signals, alternating conductors are assigned to ground and signal lines.

Wire-wrap was the only prototype medium suitable for the RAPIDbus I interface cards. In order to reliably run a 64 nanosecond cycle, all clock lines were carefully fanned out from a single octal bus driver package, using twisted pair wire wrap. The card layout was arranged to minimize propagation distances for high speed control lines. Supported by good power supply bypassing, reliable waveforms resulted (see figure 3-20).

3.4.4.3. Thermal Engineering

Thermal engineering becomes an important issue as clock speeds and gate densities climb in response to performance requirements. When the thermal environment of a semiconductor package is stressed both the propagation and mean time to failure can be expected to increase [65, 76, 10].

The first order linear coupling between supply current and output buffer slew rate

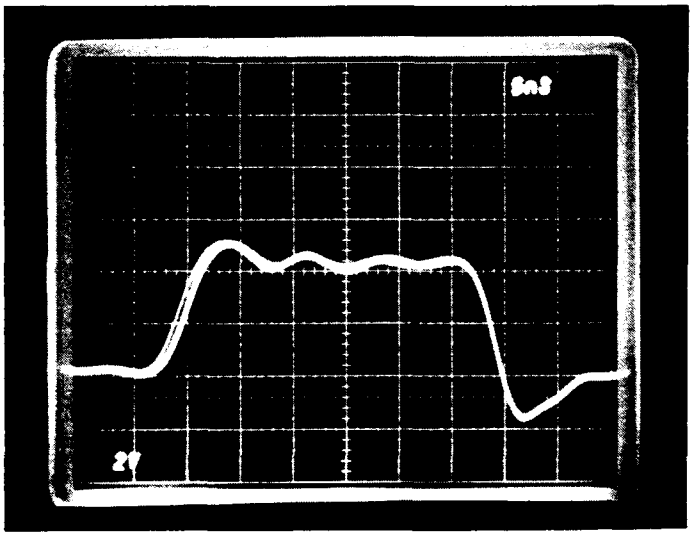

Figure 3-20: Use of a good ground place, bypassing, and short lines, acceptable waveforms were achieved using wire wrap on RAPIDbus I.

establishes our interest in current for high performance logic.[23] Additional large current spikes arise from the momentary simultaneous conduction of buffer pull-up and pull-down stages using TTL logic. Thus with TTL logic, as state changes more rapidly, power dissipation climbs.[24] Virtually all energy supplied by the power supply subsystem as electrical energy must later be removed from the semiconductor package as thermal energy. The rate at which thermal energy is removed from the semiconductor die is directly proportional to the temperature difference between the die and the external air flow, and inversely proportional to the thermal resistivity. Forced air cooling attempts to bring cool air into the package environment, replacing warmer air so as to increase the thermal gradient. Heat spreaders internal to large (hot) semiconductor packages, as well as heat sinks attached to the package help to decrease the thermal resistivity between die and airflow. Although cool air is a

---

[23] New FAST and AS gate design minimizes the amount of additional current which early TTL logic families used to saturate transistor junctions.

[24] Current requirements for ECL are relatively insensitive to clock rates for a given ECL subfamily, and voltage swings are much lower than for TTL, leading to a break-even point for high speed logic where ECL requires less power than TTL.

convenient heat transfer medium, it is highly inefficient, leading to current research into alternate heat transfer media for cooling digital hardware.

3.5. Major Contributions

RAPIDbus I served important and productive roles at all three levels of abstraction. At the architecture level, the first implementation helped point to a more appropriate system specification based on linked societies of heterogeneous processor elements. Analysis of the RAPIDbus I implementation pointed to developments which both dramatically increased the performance and extensibility of the interchange network, but also decreased cost and complexity. At the realization level, many techniques required for the fabrication of large, high performance digital systems were verified. System specifications for RAPIDbus I are summarized in table 3-21.

RAPIDbus I:	Four processor, four slave system
	Two dual port Versabus processor system built
	1.7 microsecond observed transfer latency
	~1 Megabyte per second bandwidth observed in prototype
	~16 Megabyte per second theoretical bandwidth

Figure 3-21: RAPIDbus I system specifications.

Beyond the three primary goals with which the project was entered, RAPIDbus I lead to the evaluation of several computer aided design tools including a schematic capture system, several post processors, and LLL's SCALD system. The evaluation described here directly shaped more practical RAPIDbus designs.

Chapter 4
RAPIDbus II: Architecture

4.1. The Goal

RAPIDbus II was conceived as a architecture to support the developing specifications for an advanced robot system which was formulated as a multitude of concurrent, tightly coupled, computationally intensive tasks. In support of this goal, a multiprocessor structure was designed which provides a quality support environment, moderate fault tolerance, and high system performance implemented in convenient steps.

4.2. Architectural Specification

Literally hundreds of data points exist within the multiprocessor design space, each devised in response to a particular set of application requirements. Thus it is useful to describe the RAPIDbus II system specifications both in relation to the application, and the realities of a large digital system for which an implementation is required.[25]

---

[25] In this case, practicality demands that the desire of the purist to separate architecture from implementation or realization must be moderated.

4.3. Extensibility

A substantial overhead cost is paid at the systems software level when moving from one to two equal processors. Relatively few changes are needed to the software structure as additional processor are added to the initial pair. If a sufficient number of tasks are available, potential increases in system throughput are generally limited by the increasing average communications latency as tasks begin to block each other while accessing common data structures. Through implementation techniques evolved from earlier RAPIDbus designs, RAPIDbus II appears to be capable of minimizing both contention for interchange bandwidth, and with suitable hosts, contention for access to shared memory blocks [81, 77].

Since the number of tasks is an experimental variable, the ideal number of processors appears limited by the maximum acceptable average communications latency. A reasonable metric for such a radius of extensibility is based on the time required for a processor to respond to a context swap request and execute the desired task. If a process can get the results of a task execution faster locally than by communicating with a distant processor, the foreign processor clearly lies beyond the range of practical extensibility.

Consideration of actual RAPIDbus II implementation technology suggests that a task swap has nearly 100 clock cycles of overhead before replacement code can begin execution. Several hundred RAPIDbus II processors could be placed within this communications latency. Thus in order to demonstrate reasonable extensibility without incurring substantial addressing overhead, 240 nodes.[26] are defined by the architecture, with straight forward extension to a larger population.

Within a single task, address space limitations in previous multiprocessor systems have been recognized as a serious limitation to extensibility [41]. Processors such as the DEC PDP-11, and the Zilog Z80 used in several early multiprocessors are limited to a 2^{16} byte address space [41, 59]. Taking advantage of the tremendous leap in semiconductor technology, RAPIDbus II is designed around a 2^{32} byte physical address space. The virtual address space available to the programmer is host dependent, but in the proof-of-concept realization, provides up to 2^{24} bytes in one or more separately mapped segments.[27]

---

[26] The number of potential nodes includes both processor locations, and those used only for system communication [links].

[27] Bankswitching makes the remainder of the address space accessible with greater effort

4.3.1. Heterogeny of Elements

In traversing the depth of an advanced robot system from visual input, through image understanding, to a suitably enunciated output, a myriad of different kinds of algorithms are required. Early vision processing may require very regular operators on large data objects. Later stages of vision may require the manipulation of small objects interwoven within intricate data structures. Reporting or control stages will require still other computational support. Within a research environment, externally supplied input or data output may be required at any point in conjunction with a variety of devices.

Traditionally, multiprocessors have relied on a single general purpose execution element which is replicated as needed throughout the structure. Although such homogeneity simplifies the design, any general processor cannot hope to optimally handle a broad range of array, scalar, and I/O tasks in a cost- effective way. Trade-offs are inevitable with a general purpose architecture. In contrast, designers are increasingly finding ways to create processor structures which achieve very impressive performance over a limited algorithm domain [45]. For a given cost, performance often is inversely proportional to flexibility in the design of a processing element.

A multiprocessor provides potential to take advantage of special purpose processor structures interfaced as one or more host nodes. If a complementary set of nodes is assembled so as to work symbiotically with one another, the performance of special purpose designs can be harnessed without loosing some generality of overall function. However, within a research environment, the cost of designing any processor module is high. Seldom do resources provide adequate support for both the hardware design and later software support of more than one processor architecture.

RAPIDbus thus relies heavily on being able to integrate the vast effort invested in existing commercial hosts, either at the chip set or subsystem level. Typically such hosts have widely diverse, often conflicting interface requirements [19]. Such simple grounds for communication as the location of the most significant bit or the packing of bytes within a quadlet is seldom the same for different hosts. In order for any interface to bring order from the bable, the mapping between bus cycles and data objects must be known to the interface. In the general case, this requires both data typing, and either control over operand alignment, or an indication of the relationship of the bus cycle to the boundaries of the data object.

In the course of algorithm research, it is useful to integrate prototype functional accelerators. Both in the digital domain, and with electro-optic or CCD technologies, very regular portions of a specific algorithm can often be readily implemented to achieve performance well beyond the range of a general purpose computer of comparable complexity. The catch is often in the support logic required to get operands in and out of the regular portion of the box, and handle boundary requirements. Such support logic frequently consumes major amounts of design time while directly contributing little to performance enhancement. Experimentation with new functional boxes becomes more practical if overhead requirements, such as operand stream generation and boundary calculations can be assumed by other, existing, members of the processor society [30].

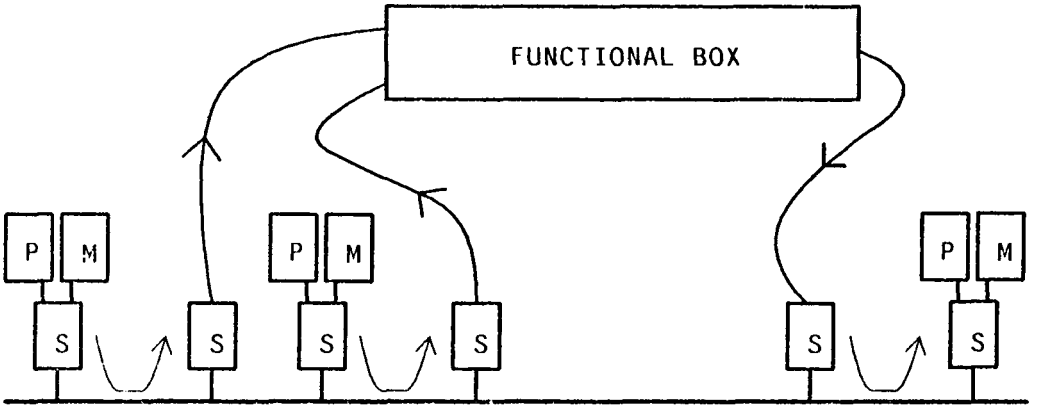

Figure 4-1: Multiword packets can be used to integrate a prototype functional box onto RAPIDbus while existing processors absorb overhead functionality.

The multiword packet is provided within the RAPIDbus II architecture specification in part to provide streams of high bandwidth operands to or from very simple functional hosts. Each port of a functional box is connected to a RAPIDbus node, which acts as a slave to sink or source streams of data coming from other, more general purpose nodes as shown in figure 4-1. Host nodes can be added as required to meet channel bandwidth or address stream interleaving requirements as needed.

4.3.2. Software Support

The concurrency provided by a multiprocessor adds a new spatial dimension to the temporal medium programmers are skilled at dealing with. Because of the additional complexity introduced by the new dimension, hardware support for quality programming becomes more important than on a comparable uniprocessor system. Although contemporary compilers and assemblers provide checking to catch errors prior to run-time, some interactions can only be reliably detected while code is running. Many kinds of run-time checking can be performed with software in line with application code at the expense of both reliability and performance.[28] Earlier discussions established the need to integrate existing processor implementations where possible. Yet with reasonable retrofits to existing VLSI, RAPIDbus II can potentially assist the programmer by restricting access to memory, mapping virtual addresses to physical memory locations in a storage hierarchy, managing of dynamically allocated storage, and assisting in the maintenance of data type coherency.

With relative simplicity, a commercial memory management part can be interposed between processor and the local processor bus, providing both segment level memory protection and virtual to physical address translation. Such coprocessors have memory descriptors paired with each segment of memory for which the task currently executing on the coupled processor has access. In some units, segments can further be decomposed into pages, or the available capabilities made specific to read or write operations.[29]

Ideally, both access protection and relocation of segments would be extended downward to support small objects structured from one or more of the primative data types according to a template. An upward compatible enhancement path, based on one or more object coprocessors for each operand processor is proposed, creating an object interface layer [*OIL*]. *OIL* is intended to support object based access protection, true virtualization of shared data, and effective support for dynamic memory allocation. In order to maintain compatibility with other objectives, *OIL* must be adaptable to existing processors and not exact significant performance penalties.

---

[28]Compiler generated run-time checking is asking software to verify software integrity, a questionable case of self-policing. Software run-time checking must insert additional instructions into the stream, competing for machine cycles with the application task. As the level of verification increases, the performance must suffer.

[29]The proof-of-concept processor node replaces the 68000 processor socket on an existing CS-9000 processor card with a board which includes a 68000, 68451 [MMU], and a single entry translation cache.

Many different algorithms operating in minimally constrained environments, such as robotics, use large data structures whose size and growth patterns are data dependent. For instance, a structure describing a particular situation which the system may find itself in, a frame, is the result of information acquired at run-time. At compile-time, the programmer neither knows how many frames will be used, nor how complex a given frame might be. Dynamic memory allocation is commonly used to parcel such storage on a demand basis. During the course of running a particular application, such storage space may become allocated and at a later time, the application will loose interest in the information. Particularly if the memory is allocated in small pieces, such memory can drop out of sight without being returned to the pool of available memory; storage can leak.

As tasks run for some period of time, loosing small areas of physical memory on each allocation and return cycle, the system will encounter a state where insufficient memory is available to be allocated, even thought substantial pools of memory are inactive. Some form of garbage collector is required to reclaim this memory. Garbage collectors are often run on machines without hardware support, but at the expense of both performance, and periodic intervals when the machine is unable to respond while "collecting". With contemporary machines and a large address space, this may require several minutes.

If memory is packed in large, nondescript segments allocated by the compiler, as is the case with most uses of VLSI memory managers, hardware does not have the information required to assist in garbage collection, leaving the burden to software. Thus the RAPIDbus II architecture does not provide badly needed support for garbage collection. Once again, the object interface layer coprocessor, running in support of a primary processor, provides a hardware basis with the proper level of granularity to consider putting garbage collection into hardware.

Data type checking is also a difficult retrofit to an existing processor / software system. Most commercial processors at the chip or board level do not provide strong data typing. Those that do, such as Intel's 432 family, currently exact an unacceptable performance penalty. The basic RAPIDbus II architecture depends on host adherence to whatever limitations are needed to allow interface hardware to transform shared variables to and from the interchange data specification. This may include alignment restrictions or require the use of external hardware monitoring the instruction stream.

Potential upgrades to the RAPIDbus II architecture based on OIL can support automatic

data type coherence and impose data type checking external to existing processors. This requires that the strong data typing of a language such as ADA be carried to the machine level, assisted by type conversion instructions executed by the coprocessor. Within the object interface layer, data typing operations are performed by the *TYPE BOX*.

4.3.3. Modularity

At the structural level, programmability, performance, and reliability are enhanced by the partitioning of large host ensembles into subsets called societies. Such processor societies are composed of several different kinds of processor hosts chosen to provide complementary capabilities needed to handle a particular package of tasks. By partitioning into societies at the architecture level, the complexity which a programmer must organize at any one time is limited to a single subgoal. Each processor node may own a portion of the total system address space through a dual porting of local memory to the RAPIDbus interchange.[30] As shown in figure 4-2, repeaters are used between processor societies to support access by any processor to memory segments in another society for which memory protection tables offer access.

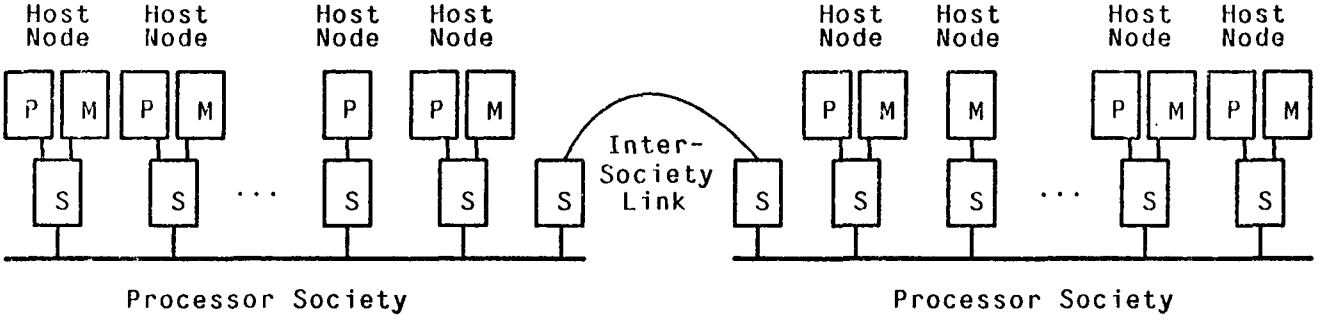

Figure 4-2: The RAPIDbus II architecture is composed of societies with up to fifteen host nodes. High speed parallel links between societies can be configured in response to research requirements.

Performance is enhanced in most implementations by decomposing processors into societies since the bulk of interprocess communication can be expected to fall within a tight locality of processors (the society). If bandwidth is independently allocated for each society,

---

[30] In an extended RAPIDbus II architecture, each host node would also be responsible for supporting objects "owned" by a particular host and resident in the dual ported memory. Such support includes maintaining a list of tasks with local copies of the objects, those with write authorization for their local copies, and dynamically insuring consistency of remote object copies as they are mutated by tasks.

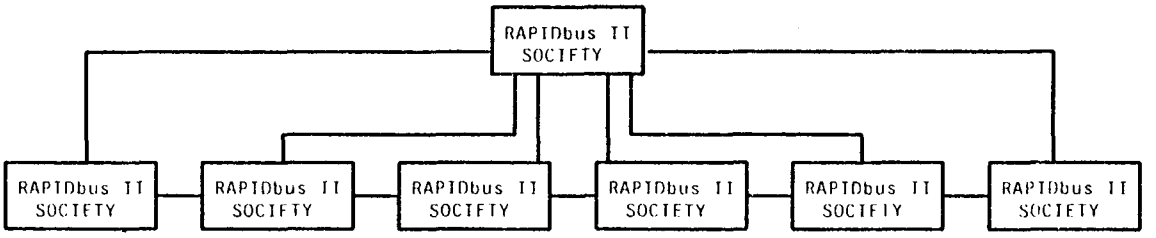

Figure 4-3: A pipeline of societies fits applications where most of the data flow obeys a linear, single input port, single output port relationship.

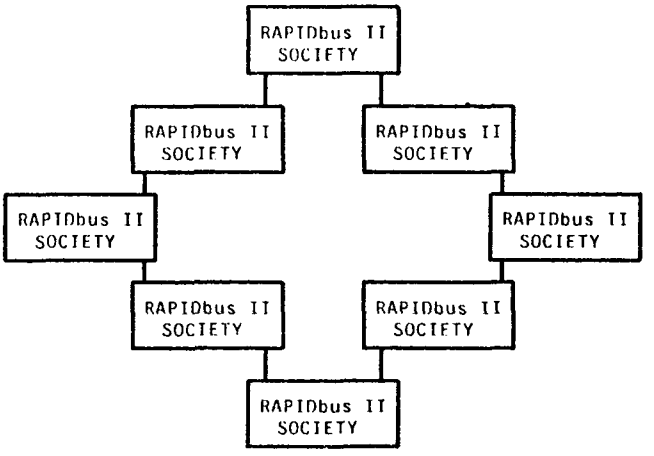

Figure 4-4: A ring of societies provides low latency communication throughout the address space with singly redundant paths between societies.

then the total bandwidth available in all linked societies can be increased with respect to allocation over a single system bus. By partitioning a system into small sections, each of which has redundant hardware, the probability of two uncorrelated failures leading to a system failure is greatly decreased [44]. When a subsystem component does fail, narrowing the neighborhood of the failure simplifies either automated or human diagnosis.

In response to particular application requirements, RAPIDbus societies can be linked in a variety of different topologies to minimize the average number of societies through which a memory reference must pass between master and slave. Applications which are generally structured as a pipeline, with the bulk of information flowing from one society to the next can effectively be linked as shown in figure 4-3. Alternate problem domains might best be served by a ring, as in figure 4-4 or an n-cube such as the 3-cube shown in figure 4-5.

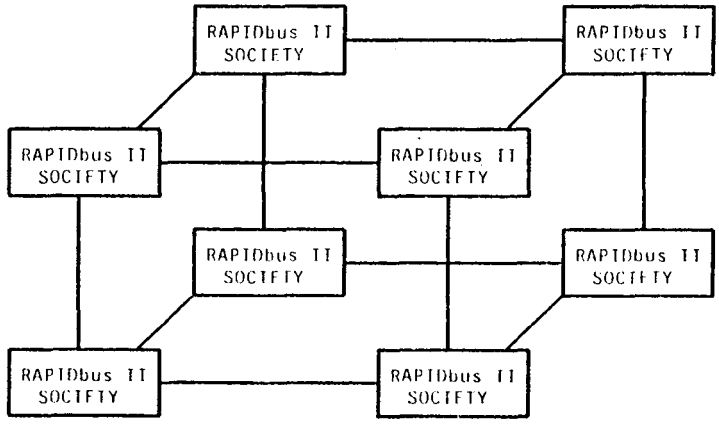

Figure 4-5: Rings of societies can be generalized into N-cube topologies, with arbitrarily many redundant paths between societies at the price of increased overhead.

4.3.4. Specification Summary

Both the application environment, and the realities of the implementation have lead to a structure based on small groups of heterogeneous processors called societies. Inter-society links provide communication paths to create a single shared memory environment for the entire ensemble of societies required by a particular application. Several capabilities are recognized as being essential for optimal support, and yet had to be designated as an upward compatible path for practical reasons.

4.4. Addressing

Communication within the RAPIDbus II architecture is through a sparsely populated physical address space composed of 2^{32} addressable bytes of information. Locations can be accessed as bytes, doublets (16 bits), quadlets (32 bits), or multiple quadlets. Doublet or quadlet transfers which cross a quadlet boundary must be run as two separate RAPIDbus transfers, one on either side of the quadlet boundary. Multiple quadlet transfers must take place aligned to quadlet boundaries. Most processor architectures allow split transfers arising from quadlet boundary crossings to be handled transparent to the programmer interface. Where supported by the processor, RAPIDbus transfers are optimized for quadlet transfers aligned to quadlet boundaries.

Transfer requests can always be made for any of the four supported access widths,

independent of the width of the slave implementation as long as physical memory is available on a single host for each byte being accessed. If the data path of the slave is insufficient to handle the required transfer, the slave RAPIDbus interface will split the access into multiple host cycles.

Access to nonexistent memory locations will be handled differently depending on whether the location fits within a valid memory descriptor for the task in question. If no descriptor exists, the interface defines an error handling protocol which may involve bringing the required segment in from a secondary storage medium (virtual memory). If a valid descriptor exists for a nonfunctional memory location, the processor will be interrupted with a nonvectored bus error exception.

4.4.1. Memory Map Structure

Each installed host node is populated with a contiguous segment of the 2^{32} byte physical address space. As shown in figure 4-6, the physical address space is hierarchically structured. The highest four bits, *A31-A28*, indicate the society in which a memory location is to be found. The next four bits, *A27-A24*, are currently assigned to provide a 2^{24} byte potential address space on each host node. As an upgrade path, *A27-A24* could also be repartitioned to increase the number of societies, the number of host nodes within a society, or the number of virtual processor / memory servers located at a single host node.

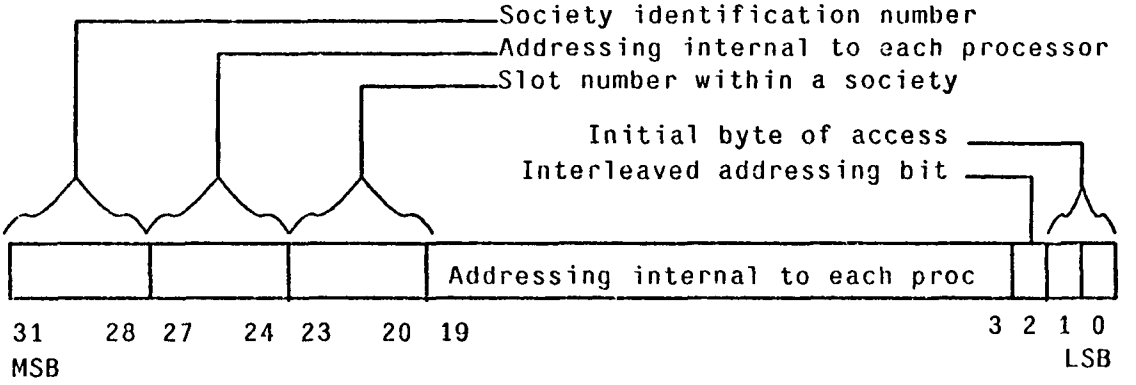

Figure 4-6: The physical address space is partitioned hierarchically into societies and then host nodes within a society.

Within a society, there are fifteen host nodes which could potentially be populated. Address values zero to fifteen on address lines *A20-A23* indicate the node to which a memory reference should be directed. The megabyte belonging to the sixteenth node, independent of

A31-A24 in the address, contains a local control segment to accommodate system ROM, peripheral device and control registers, many of which will vary with the particular host.

The RAPIDbus interface specifies four locations within the local control page; an interface control, a diagnostic, a bank switching register, and the node address. The bit assignments of the control register are specific to the nature of the host interface. In the case of the microcoded interface described in the realization section, five bits are used to extend the number of different host nodes on which a remote interrupt handler can be scheduled. The highest three bits are potentially used to control the conversion of user data flowing through the interface. Diagnostic information specific to the RAPIDbus host interface operation can be read from the same address at which the control register is written. The bank switch register expands the address space accessible to hosts with less than a 32 bit address range. The location of both registers within the local control segment is flexible to conform to existing host memory maps.

The lowest twenty bits of the address access within a potential one megabyte memory block on a host node. Many commercial monoboard computers with dual port memory are shipped with memory located at the bottom of the address space (0xx0) for all cards. The same memory locations appear in higher address ranges from an off-board access. The local address decode on such cards must be modified to make the internal and external addresses identical if all fifteen host nodes are to be populated with dual port processors within a society.

4.5. Data Formats

In order to efficiently and unambiguously support the interchange of data between processor nodes, specification of both data representations, and their assignment to the the interchange structure is essential. The high degree of incompatibility between existing commercial processor nodes forces the RAPIDbus interface to perform host specific format translation to achieve format compliance for any data object traversing RAPIDbus [42, 19, 11]. In formulating interface strategy, data structures were considered for the Intel 80x86 [2], National xx032 [1], Motorola 680xx [48], VAX [23], and IEEE 896 advanced backplane [4]. Even though software considerations may prohibit direct use of more than one processor architecture, compatibility with ancillary hardware designed for different data structures is essential. Analysis of conversion problems confirms Cohen's conclusion [19] that the current proliferation of structures is a costly, time consuming tragedy.

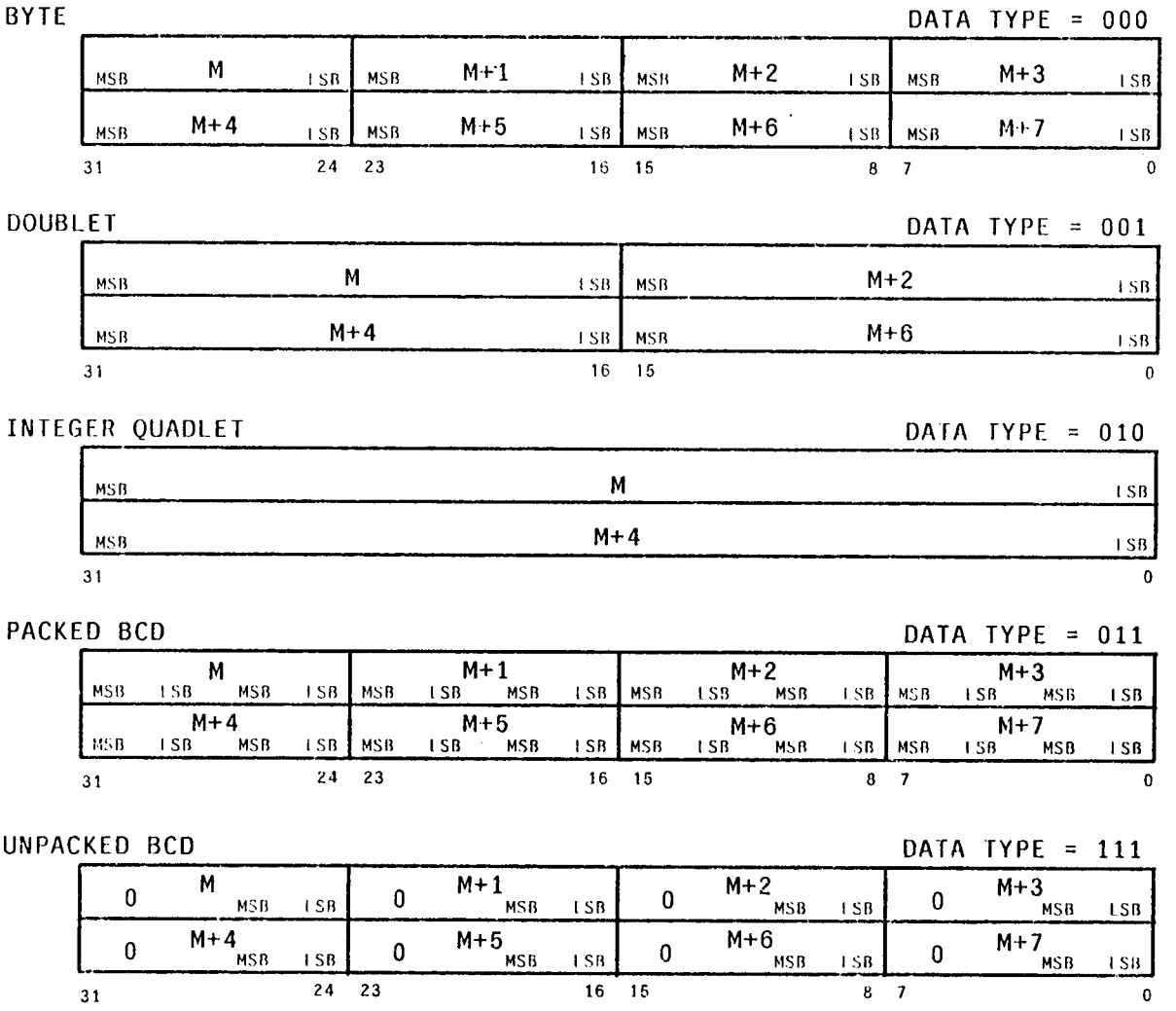

Figure 4-7: Five primative scalar data types are supported based on Motorola 68000 representations.

4.5.1. Scalar Data Types

At the lowest level of structuring, the RAPIDbus interchange handles five different scalar data types. Each is described in relation to a quadlet, or thirty-two bit word around which the memory system is organized. The packing is identical to that of the 68000 processor family, optimizing the system for such processors as a matter of practicality. This packing is suboptimal, since ascending bytes or doublets packed within a quadlet (32 bit word) are stored going from the most significant to least significant side of the quadlet, in direct contrast to the bit numbering scheme. All operands are referenced by the byte address of the lowest

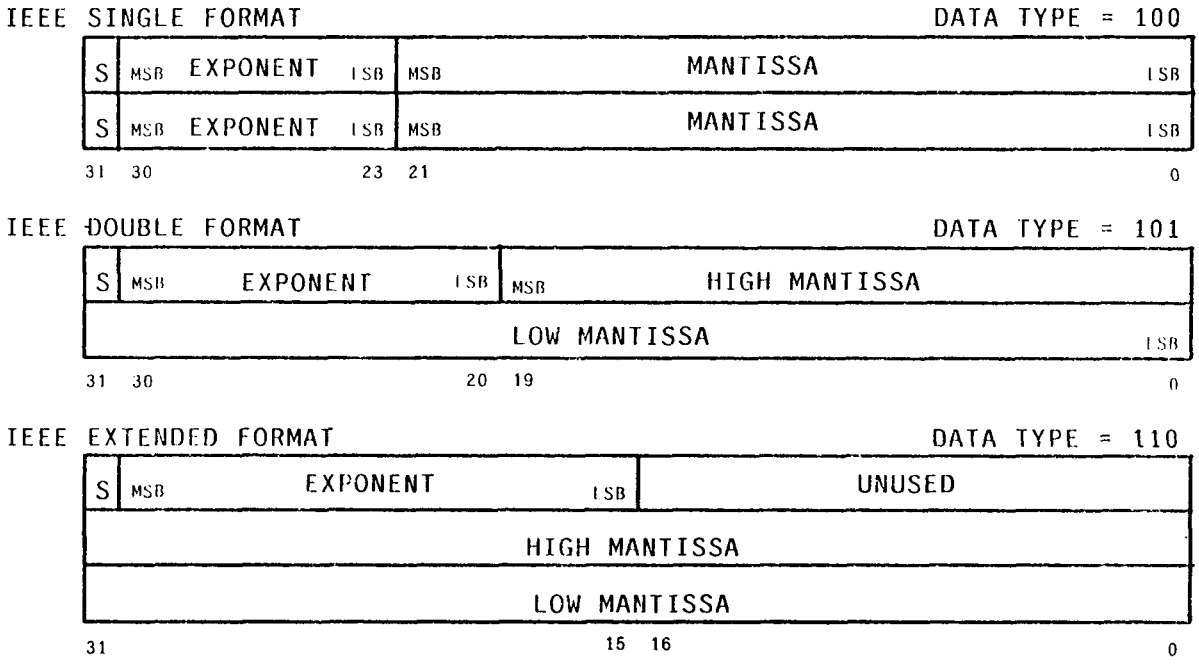

Figure 4-8: Three different floating point representations are supported based on the Motorola packing of the proposed IEEE floating point specification P754.

byte enclosed within the operand. Where possible, operands should be assigned to avoid crossing quadlet boundaries.[31] Packing strategies, and the associated data typing indices are indicated in figure 4-7 for bytes, doublets (16 bits), quadlets (32 bits), packed BCD, and unpacked BCD. As an upward compatible step, type translation logic would be significantly simplified if doublets were assigned only to addresses where *A0* is zero, and quadlets assigned only to addresses where both *A0* and *A1* were both zero. Since this requires modification of existing assembly code and compilers, this requirement is not made by the RAPIDbus II architecture.[32]

---

[31] Performance is improved by avoiding quadlet crossings. Unlike the 68000 architecture, this restriction is not *required*.

[32] Use of a 68000 processor with an internal instruction cache will make this requirement essential since less information is available retrofit instruction monitoring logic. If software enforces this restriction, and avoids the use of shared BCD data types, only references to floating point operands need to be caught by external instruction monitoring logic. Since such floating point references should be executed on a separate floating point coprocessor, data types can still be translated to alternate packing structures even with a processor cache.

4.5.2. Floating Point Data Types

The three IEEE floating point data types are supported using Motorola packing structure, shown in figure 4-8 [38]. Respectively, single, double and extended formats require one, two, or three quadlets. Once identified, the IEEE single format can be readily translated when accessed as a single data cycle into alternate packings. Unfortunately, IEEE double and extended format require more complex translation techniques. Translation into alternate packing structures requires that the order of the successive words be inverted. Since such words are handled by the RAPIDbus interface as independent data transfers, complex logic is required at the host to reorder both words and bytes within a word. Conversion to and from DEC floating point notation also introduces numerical transformations in addition to byte level repacking.

4.6. Upward Compatibility: OIL

The object interface layer is intended as an upgrade path for the RAPIDbus II architecture to support features and implementation performance enhancements which were not in keeping with the needs of a basic proof-of-concept implementation. *OIL* is intended to combine the performance of existing operand based architecture implementations with some of the advantages of an object oriented environment. The interface layer is conceptually interposed between processor and an upward compatible RAPIDbus II interchange network. Basic to the approach taken by *OIL* is the assumption that interconnect bandwidth is plentiful, but that only point-to-point links are supported between nodes.

4.6.1. Objects

The concept of an object oriented multiprocessor is not new. This section builds on ideas from machines like Starlet [33], Intel's 432 [39], CMU's C.MMP/HYDRA [65] and CM* [41], MIT's Lisp Machine [78], and *A MACHINE ARCHITECTURE TO SUPPORT AN OBJECT-ORIENTED LANGUAGE*, a Phd thesis by Alan Snyder [68]. It is not intended as a final design, but rather to suggest that a high performance multiprocessor and an object architecture can coexist productively in the intended robotics laboratory environment.

An object, used in the context of RAPIDbus II, is the name of a location containing a value; either a single primitive data item (for instance a doublet integer), or a more complex structure composed of primitive data items. Support for complex structures as a single object is an

important factor in minimizing overhead in a system built for high performance, largely numeric computation [33]. All objects are dynamically allocated at run-time by *OIL* running as a host coprocessor. Complementing the objects used for shared storage, *OIL* supports variable based segments, allocated by the compiler, for non-shared operands.

With the support of *OIL*, each shared object has a unique name, by which the object is known by all tasks to which it is accessible. At any instant, *OIL* insures that a single value is attributed to the object by any processor running in the environment, even though performance may dictate that multiple "copies" of the object are kept.

4.6.2. Object Support

A group of consecutive objects is "owned" by a particular node within the RAPIDbus system. Although performance is enhanced if objects are allocated on a node within a society making frequent reference to an object, this is not essential. The owner node maintains a record of each object's value, including the required tag, a list of tasks with copies of the object, and a list of tasks with access rights. The owner node is responsible for maintaining the consistency of each copy of owned objects through cache update packets [57]. Mapping between an object's identifying number; the "address", and the location of the owner node is done through a writable routing table on each node. In this way, objects can be migrated from one owner node to another in response to node failures or the paging onto secondary store of a sequential group of objects.

Access rights are given to a task either in response to object allocation, or through a coprocessor instruction run by *OIL* copying access rights owned by the current task to another task. *OIL*, acting as a coprocessor, sends an appropriate packet to the object owner. Tasks which no longer need access to large objects can speed up the garbage collection process by sending a coprocessor instruction to *OIL*, releasing object access rights. If the last access rights are returned to an object owner, and packet skew limitations are met, an object's storage can be returned to the pool of available memory prior to being reached by the garbage collector.

The conceptual structure of an object layer interface is shown in figure 4-9 with the processor at the top of the figure, and the two RAPIDbus ports at the bottom. The processor can fetch from either a cache, which supports small primative data types, or a larger, directly addressable memory accessible through a commercial memory management unit. The

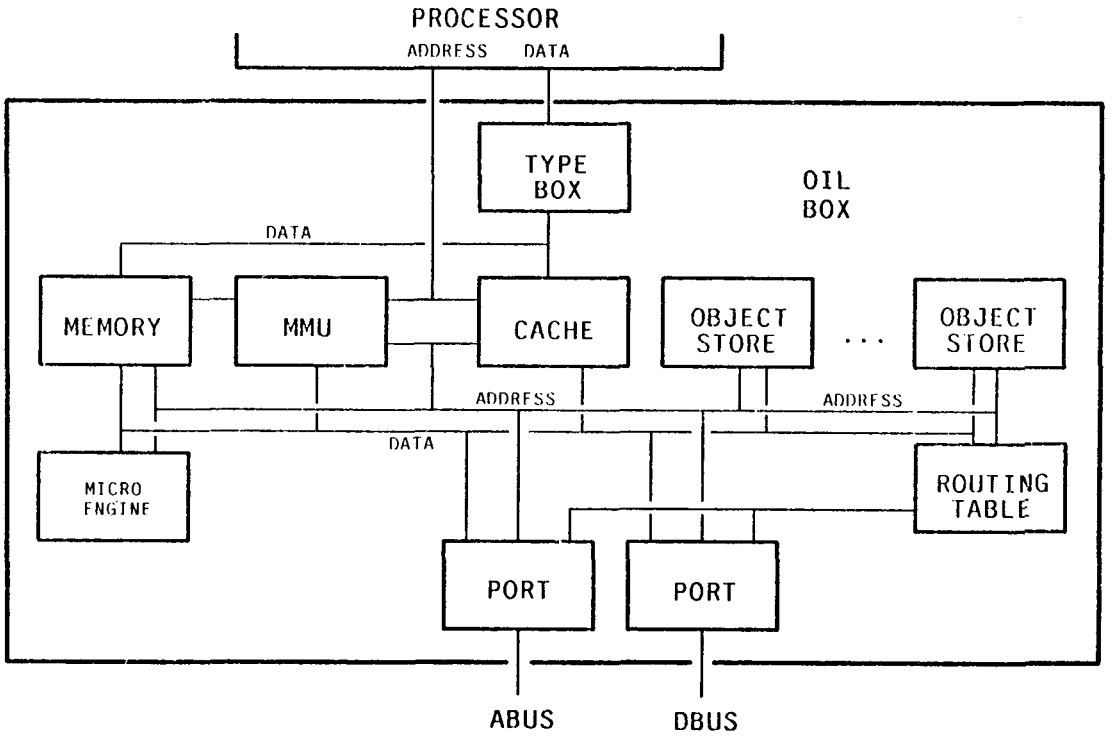

Figure 4-9: The object interface layer is inserted between processor and interchange to assist in operand management.

memory mangement unit maintains descriptors which represent either directly addressable; compiler allocated variable segments, large shared objects, or mailslots.[33] Directly addressable memory is handled as intended by the MMU designers. In the second case, where a descriptor represents a shared object such as an image or a block of code, the MMU / memory is functioning as a cache with a block size equal to a page. The difference between this use of an MMU and traditional virtual memory is evident when the operand is neither available in the cache nor does the MMU have a descriptor. When a descriptor represents a mail slot, as in the third case, the reference is treated as non-cachable, and written or read directly to or from the owner object store via the *OIL* microengine.[34]

Local memory faults are handled by the microengine which forms the heart of the *OIL* on

---

[33] Since segments representing small portions of the local physical memory might be typically used, having a large number of MMU descriptors is essential.

[34] Useful for feeding special purpose pipes or other functional blocks.

each processor node.[35] A memory fault which comes from an MMU descriptor for local, compiler generated storage, is handled as a traditional virtual memory problem with the microengine requesting the segment from another host node with a secondary storage device (such as a disk). The segment is brought into the memory block as a large, multiword packet, possibly causing another descriptor to become "paged out". If a descriptor for a large object is paged out, the owner node of the object must be notified that a copy is no longer present on the paging node.

Cache faults, and faults from MMU descriptors representing large objects are handled differently. The fault initiates a read packet addressed to the object owner as selected by the routing table. The object owner node can either respond with the requested object for placement in the memory or cache (depending on size), or the requester can be refused access because the task requesting the object did not have access rights. Such access violation is intended to prevent one task from affecting another except along communication channels asserted by the programmer through access ownership.

4.6.3. Data Typing

The typing capability provided by the *TYPE BOX* within *OIL* retrofits type checking and type translation to an existing processor implementation. Detailed in figure 4-10, the *TYPE BOX* requires the support of a strongly typed language, a compiler/assembler which retains the strong typing, and the addition of type translation instructions executed by the *OIL* as a coprocessor.

Addition of type logic increases the variety of different processor architectures which can cleanly be interfaced into the multiprocessor structure since conformity to a particular standard is not required, only convertability. Although code segments are obviously not simply translated, shared objects can be readily transformed between the representation used by the host and that of the interchange network. The type logic also detects type violations in an effort to detect processor operations which were not as intended by the programmer.

---

[35] Performance considerations may dictate multiple *OIL* microengines per oil box.

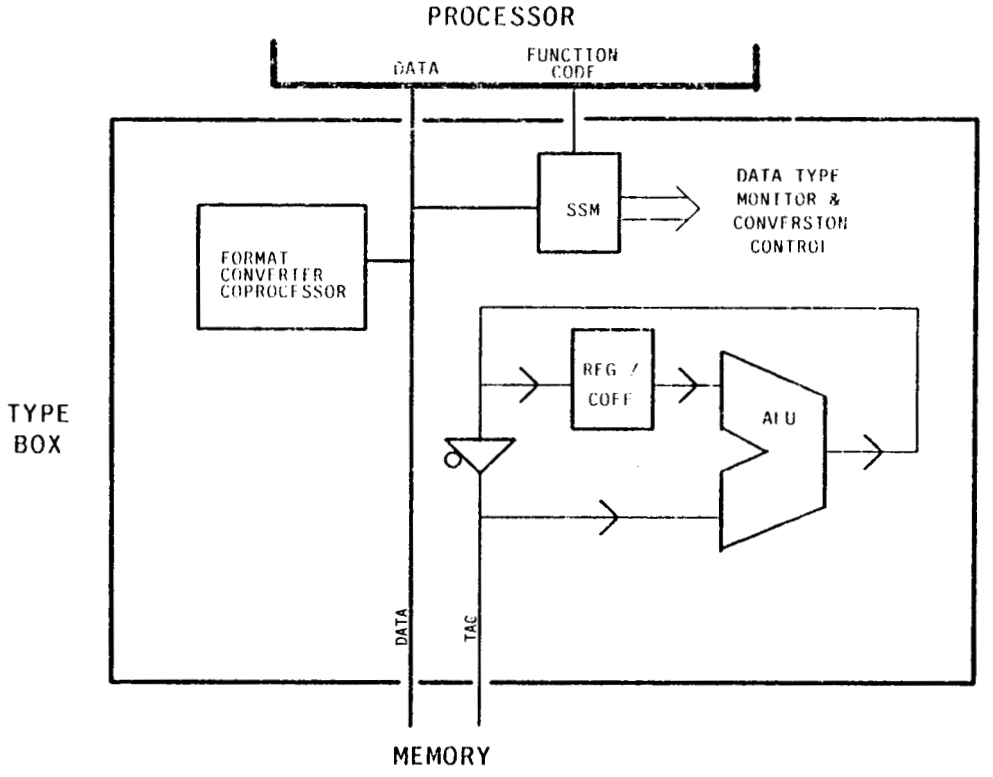

Figure 4-10: The type box is used to retrofit a variety of existing processors to an object based RAPIDbus II.

4.6.4. Garbage Collection

Finally, *OIL* supports incremental garbage collection assisted by explicit release of shared objects by coprocessor instructions. Each pointer to a dynamically allocated object is tagged for ready identification. Starting a garbage collection cycle, each task access entry for each object has a marker bit, which is initially set. As each task is temporarily deactivated, *OIL* on the node upon which execution has just completed proceeds through the list of accessible objects, signaling the object owner to clear the marker bit associated with the task. Access grants conveyed to tasks during the garbage collection cycle also clear the bit. At the completion of the garbage collection cycle all objects with no clear bits for any task access entries are returned to the heap for reallocation.

4.6.5. Summary

The structure of a possible object based enhancement to RAPIDbus is described here to support the extensibility of the basic design. By taking advantage of very high bus bandwidth, local copies of shared data can be kept coherent across many processor nodes without any bus monitor logic. Removing the need to monitor bus traffic to maintain coherency, as is done in many current multiprocessors with local data caches, [32] opens the possibility of many new interconnect structures, such as the switching plane described in the next chapter. The *OIL* interface helps to identify at run-time programmer assertions on both the scope and typing of shared information. Reliability is enhanced by the ability to migrate either tasks or groups of objects from one node to another transparently. *OIL* improves the quality, reliability, and performance of the programming environment, building on the foundation RAPIDbus II architecture.

Chapter 5
RAPIDbus II: Implementation & Realization

5.1. The Implementation

The RAPIDbus II implementation is intended to provide efficient protocol and functional structure to allow a high performance, practical realization of the architecture specification. Performance is primarily dependent on the use of existing host processor nodes executing one to twenty million operations per second and having frequent need for low latency communication with shared data structures. Practicality demands that the realization reduce performance gradually in response to as many single point failures as possible.[36]

The RAPIDbus II implementation draws heavily from Zoccoli's original RAPIDbus, and the RAPIDbus I implementation [81, 77]. Going beyond the insights and lessons that were suitable for use with commercial hosts and off-the-shelf realization technology, the implementation section will conclude with a discussion of an enhanced interconnect implementation.

---

[36]The need for graceful degradation with RAPIDbus should not be confused with designs where the primary requirement is reliability at the expense of considerable duplication of hardware and machine resources. Examples of multiprocessors where primary emphasis is placed on uptime include C.VMP, Tandem's Non-Stop series, or NASA's FTMP [65].

5.1.1. Packet Switching Structure

The RAPIDbus II switching structure has evolved from a time-multiplexed, circuit-switched bus into a packet-switched interconnect structure. Although RAPIDbus II breaks communication into several, multiple stage, discontinuous bus cycles, it differs from common packet interconnect definitions in two important ways. Sequential bus cycles implementing a single packet are not sent on adjacent bus slots.[37] Secondarily, each host is capable of handling bus cycles from only one packet transaction at any one time in the proof-of-concept implementation.

Both departures optimize support for existing commercial hosts.[38] Virtually all candidate hosts were limited to producing 42 or fewer lines of new information during any processor clock cycle.[39]By transferring information as it was generated by the host, routing complexity was increased, but the width of a given information path is reduced. The second departure came from the inability of existing circuit-switched host bus protocols to initiate or begin service on a second interchange transfer while an earlier request was still outstanding.

Packets form the fundamental unit of communication between any two host nodes across the RAPIDbus II interchange network. Each packet in turn is composed of two or more transfer cycles. If both the master and slave associated with a transfer cycle are in the same cage, only one bus cycle is required to implement the transfer. If the master and slaves are situated in different cages,[40] one bus cycle is required across each intervening cage to complete a single transfer cycle. Bus cycles are implemented as short temporal windows during which one set of drivers on each bus[41] within a cage are gated onto the backplane. At

---

[37] The 56 bit width of each bus allows substantial concurrent information transfer on each bus cycle however.

[38] During the design process, the impact of integrating a variety of board and chip level hosts was considered. Versabus hosts from IBM Instruments [CS-9000], SKY Computer, BioResearch, and Motorola were considered. Compatibility with Multibus I and II specifications from Intel, the IEEE P896 Advanced Bus standard and the Analogic AP-500 generalized host port and auxiliary I/O ports provided longitudinal tests of host extensibility. At the chip level, consideration was also given to interfacing native hosts based on the Motorola's 68000 and 68020, Intel's 80286, and National's 16032 and 32032.

[39] These are generally some form of 32 address lines, three function code, three data type [externally added], two size, write, and read-modify-write lines.

[40] A cage of processors is the physical implementation of an architecture society

[41] The bus grantee

the conclusion of this bus cycle, the interface to which a bus cycle is being routed,[42] latches in all lines of the accessing bus.

A packet transfer begins by connecting the master, or originating node, with the slave, or destination node using an address transfer cycle. Once a memory block is assigned to a packet, no further address transfer cycles will be accepted until the transfer protocol is completed or aborted. The address cycle conveys the physical address being referenced, a function code detailing the nature of the transfer, and control information designating the transfer as a read, write, or read-modify-write and the bus transfer width. Following data transfer cycles within the packet may only be sent after acceptance of the address transfer cycle by the destination, and then only in agreement with the type of transfer indicated by the function code of the initial address transfer cycle.

Following acceptance of the address cycle, up to four bytes (a quadlet) of data, an address acknowledge from a remote cage, an abort cycle, or an interrupt cycle may be transmitted using a single data transfer cycle.[43] Packets designated read-modify-write by the initial address cycle require two data cycles, the first from the slave to the master, the second, a write, from master to slave. Multiword data packets require an overhead data transfer from master to slave (the word count), followed by the number of data transfer cycles indicated by the packet word count parameter.

Each cage supports one backplane with two redundant 56 line buses, the ABUS and DBUS. Within either bus, 32 lines are used for address or data information, eight lines to communicate the transfer function code, eight for routing information,[44] four bits for control with address cycles, and four bits of parity covering the first 52 lines. The function code field is used to specify the class of access request for an address transfer cycle. During a data cycle, the function code can either describe the data, indicate an abort, or a remote acknowledge. Function code assignments are shown in figures 5-1 and 5-2.

Within each of the buses, the control fields are used only for address cycles. Bit fields are provided to identify *read*, *write*, and *read-modify-write* atomic cycles. A pair of bits within the

---

[42] The bus accessee

[43] Additional packet types are required to support an *OIL*-like enhanced bus.

[44] During an address cycle, the routing byte indicates the interface address of the originating host node. Data transfers use the routing byte to indicate the destination of the data.

Cycle Type Markers:

Lines 7654321	
XXXX X000	Control cycle
XXXX X001	Supervisor code request cycle
XXXX X010	Supervisor data request cycle
XXXX X011	Data cycle (see below)
XXXX X100	Reserved
XXXX X101	User code request cycle
XXXX X110	User data request cycle
XXXX X111	Reserved

Figure 5-1: The least significant three bits of the function code field indicate the transfer class.

control field indicate the number of bytes following the given (byte) address for which transfer is requested.

Bus cycles on the ABUS or DBUS are confirmed by signals on the respective three line acknowledge busses. Not to be confused with a remote acknowledge cycle on the ABUS or DBUS, the sole function of these buses is to confirm the integrity of a single bus cycle two bus windows after the primary (ABUS or DBUS) grant. Acknowledge bus codings are described in figure 5-3.

Timing for both 56 bit backplane busses within a cage is tightly controlled by unbussed, point to point links between each slot in a cage and the cage arbiter, as shown in figure 5-4. Each arbiter cable allows the interface to request one or both buses at a time, to specify a request for an bus as an address or data cycle, and to prohibit address accesses to an interface while a packet transfer is in progress. A four line slot address indicates the immediate destination of the requested bus cycle within the cage. Returning from the arbiter to each interface slot are bus grant signals for each of the buses, bus access signals, a cage timebase, and a cage reset line.

Cycle Sub Types:

Lines 7654321	
XXXX 0XXX	Route to master
XXXX 1XXX	Route to slave
0000 XXXX	Byte data (data cycle)
0001 XXXX	Doublet (data cycle)
0010 XXXX	Integer Quadlet (data cycle)
0011 XXXX	Packed BCD (data cycle)
0100 XXXX	IEEE Single floating point (data cycle)
0101 XXXX	IEEE Double floating point (data cycle)
0110 XXXX	IEEE Extended floating point (data cycle)
0111 XXXX	Unpacked BCD (data cycle)
1000 XXXX	Untype data [wild type] (data cycle)
1001 XXXX	Code type (data cycle)
1010 XXXX	Pointer (data cycle)
1011 XXXX	Multiword parameter (data cycle)
0000 1XXX	Single word access [32 bit master] (request cycle)
1111 1XXX	Single word access [16 bit master] (request cycle)
0110 1XXX	Multiword access [32 bit master] (request cycle)
0111 1XXX	Multiword access [16 bit master] (request cycle)
1101 1000	Interrupt access (control)
1011 X000	Abort access (control)
1000 X000	Remote acknowledge (control)

Figure 5-2: The most significant five bits of the function code elaborate on the class of the transfer.

5.1.1.1. Packet Routing

Host nodes are logically identified by a unique home address used by software to name the executing processor node, and by hardware to coordinate sets of bus cycles within a packet transfer operation. RAPIDbus II uses the most significant four bits of the home address to designate the society number, and the least four bits to designate host slots zero through fifteen within a society. Upgrades could add bits to either field within the home address.

Lines	210
000	Cycle received, but repeated to another RAPIDbus backplane
001	Sixteen bit value received at final destination
010	Eight bit value received at final destination
011	Thirty-two bit value received at final destination
1xx	Error, repeat bus cycle

Figure 5-3: Each primary bus has a paired acknowledge bus to confirm each bus cycle.

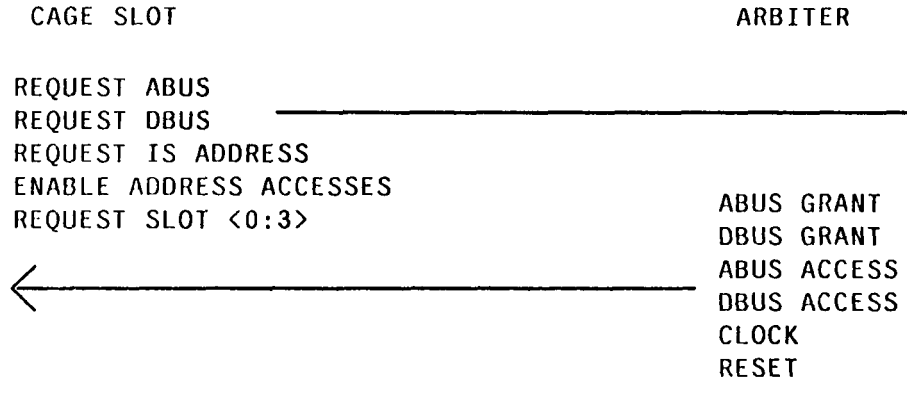

Figure 5-4: Timing for the high speed buses is done by one arbiter module global to each cage.

The home address is readable on each RAPIDbus host interface at the address location written to set the address extension. This allows software running on each processor to choose unique portions of system data structures, and to report diagnostic information referenced to the physical position of the host node.

Interface hardware uses the home address to create virtual links between master and slave processors across the interchange network. The home address of a host node accessing memory on another node is carried within the routing field address transfer cycle so that the slave host will know where to respond.[45] Each data cycle following the address cycle uses

[45]The address cycle is routed to the appropriate slave based on the concatenation of address lines A31:A28 and A23:A20 within the bus cycle.

the transfer routing field to designate the destination of the particular bus cycle. Bus cycles implementing a data write transfer will convey the slave home address. Respectively, a data read transfer will contain the master home address.

Upgrade paths could increase the number of bits in the home address to provide for multiple memory blocks accessible through one host node, or processors capable of several simultaneous outstanding packets. Provision for multiple concurrent memory accesses within a single host node increases the parallelism of the bus memory server, potentially decreasing memory contention. Depending on the processor node architecture, multiple outstanding packets can arise from either a write-through cache, or a processor running several interleaved instruction streams. Each unit of processor or memory parallelism visible to RAPIDbus must be assigned a unique home address.

The home address, or subset of the address value in the case of an address transfer, indicates the destination to which the cycle is to be directed, but does not specify the *path* to be taken when the reference is in a remote cage. This mapping from destination to path takes place in stages. A transfer cycle between communicating hosts encounters a new routing table on entering each cage through which it must pass. The table selects the slot within the cage to which each transfer must be route; either to another repeater link, or to the intended destination.

It is useful to consider making such routing tables reconfigurable in software, permitting paths to adapt to changing system conditions. Many arrangements of society links provide potentially redundant paths between distant hosts (for instance the ring, shown in figure 4-4 or the 3-cube, shown in figure 4-5.). Use of writable routing tables would allow either new paths to circumvent failed nodes, or migration of physical address space segments from primary to secondary storage areas.

The modularity of bus repeaters allows rapid reconfiguration of processor cages, or inter-society links to optimize connectivity for changing applications. With a writable routing table, proper code in each host node would allow automatic resource identification and table creation during system boot. Although RAPIDbus II was designed for proof-of-concept using fixed (ROM) routing tables, making these tables routable is a clear upgrade path toward a more flexible interchange system.

5.1.1.2. Bus Justification

A variety of different schemes are used for transferring data of lower width than the bus, depending on the type of transfer for which the system is to be optimized [42]. In order to optimize bus bandwidth for 32 bit transfer operations, RAPIDbus uses an unjustified 32 bit bus. Each byte-wide lane of the bus is logically paired with separate byte-wide memory sections within a bus. For instance, bytes with the *A0* and *A1* set to one are always transferred on data lines *D31-D24*.

When narrow hosts are attached to the bus, crossover buffers are required to allow access to all byte locations. A sixteen bit host requires two octal bidirectional buffers. An eight bit host requires three. Integration of a sixteen and 32 bit host is shown in figure 5-5.

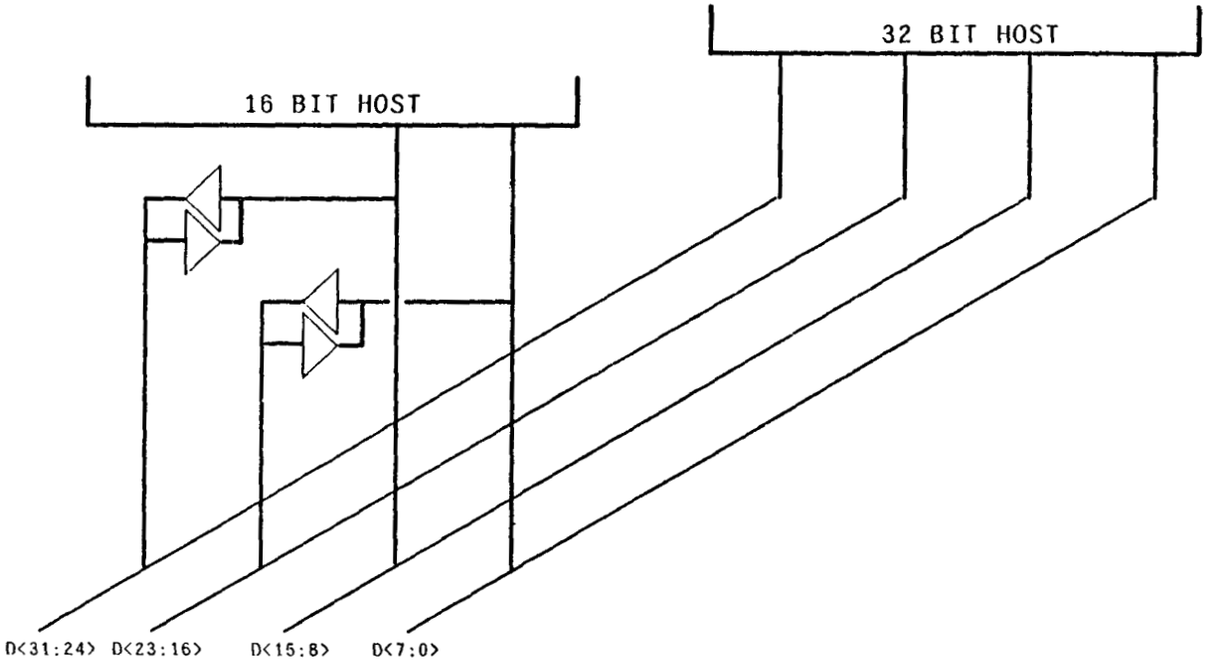

Figure 5-5: Use of a sixteen bit host on a thirty-two bit unjustified bus requires a crossover to allow access to all bytes in memory along low data lines.

In order to hide implementation details of one host from another, it is important that one host need not know the width of another's data path. Thus the bus buffers are also used to allow the RAPIDbus interface serving a narrow data path host to run multiple transfer cycles locally so as to transparently fulfill a transfer request.

5.1.1.3. Bus Allocation

Performance maximization based on available interconnect bandwidth is critically dependent on an effective allocation scheme. Practical considerations almost always lead designers to some form of fixed priority allocation scheme with ranking determined in hardware [29, 70, 46]. Simulation of different allocation schemes for the RAPIDbus II backplane busses suggested that even within a fixed allocation framework, significant differences could be made in the load level which resulted in bandwidth starvation of some nodes depending on the arbiter chosen. With a dual port 68000 processor implementation integrated directly onto RAPIDbus, running a code stream with 60% of the references off board, starvation occurred with as few as nine processors per cage using a simple arbitration scheme. The approach finally chosen showed no starvation under the same conditions for up to fifteen processors per cage (the implementation limit).

Two separate arbiters control ABUS and DBUS allocation. Normally a request is made to both arbiters by an interface desiring a bus cycle. The ABUS arbiter grants cycles with highest priority to cage node *fifteen*, and lowest to node *zero*. Conversely, the DBUS arbiter gives node *zero* highest priority, and *fifteen* the lowest priority. A cycle will only be allocated to the same interface on both busses simultaneously if only one bus request is active.[46] This approach still allows hardware to recognize one of the two busses in each cage as unreliable, removing requests from the suspect bus until repair can be made. Transfers continue at a reduced throughput.

5.1.2. Data Transfer

Data transfer packets are initiated by an address transfer cycle from an originating master to the requested slave. Only after the slave accepts the transfer request can one or more data or control transfer cycles be exchanged. If the master and slave are in different cages, one or more repeaters must be used to echo bus cycles between cages so as to complete the required transfer cycle.

In order to understand this implementation of the RAPIDbus II bus protocol in moderate detail, it is useful to consider the state behavior of host nodes involved in each kind of transfer operation. In order to avoid details of the host implementation, only those state changes involving RAPIDbus interface ports will be considered.

---

[46] Arbiter hardware arbitrarily selects the ABUS to run the cycle if both busses are granted simultaneously to a single requestee.

Considerable performance can be achieved through concurrent operations within each host node, depending on the host design. The following state descriptions mask such concurrency in the interest of clarity. The states and state transitions shown here are similar to, but do not represent a one-to-one mapping with the microcode interface realization. Unless otherwise noted, states are executed sequentially.

0. In the quiescent, or reset state, the host has no transfers active, and the RAPIDbus port is unassigned to any transfer. The sequencer loops in this state until a data, interrupt, or remote acknowledge transfer is initiated by the host or received from RAPIDbus. For a data or interrupt initiation, execution continues with the appropriate section below depending on the source. A remote acknowledge from the bus causes the sequence to branch based on the suspended processor control lines, which are waiting for the slave in a remote cage to accept a transfer request.

5.1.2.1. Single Cycle Read Request

1. A single cycle read request is initiated by the host processor on the requesting node. The address, function code, control, and parity information is sent to the outgoing latches of the RAPIDbus interface ports (ABUS or DBUS). Bits are abstracted from the address field to provide input to the routing table. A request is placed for an address transfer cycle on either the ABUS or DBUS with the cage arbiter. If the home address of the originating node is greater than the address of the node being accessed, the interface is assigned to this transfer.[47] Once an interface is assigned to a transfer, a simple RAPIDbus II host cannot accept incoming address packets until the assigned transfer is completed or aborted.

2. When a cycle is granted, the arbiter activates bus drivers on the originating interface, directing the address cycle stored earlier onto one of the two interchange busses within the cage. At the conclusion of the cycle, the arbiter will activate the latches in the slot being accessed, as designated by the output of the routing table on the requesting interface.

3. During the following bus window, the arbiter blocks requests from the previously granted interface while the grantee removes the bus request.

4. On the third bus following the bus grant, the originating interface latches in three acknowledge lines corresponding to the bus on which the original address cycle was sent. These lines can indicate that the cycle was refused, that it was repeated to another cage, or that the cycle was accepted by the final destination. If the cycle was refused, the bus cycle must be repeated, starting with state *one*. If the cycle was echoed to another cage, the interface must wait for a remote

---

[47] This conditional allocation of the interface is to prevent deadly embrace in which two dual port host nodes are allocated simultaneously to outgoing transfers which involve the other's memory. Since many host port conventions support only one transfer at a time, a deadlock would result under which neither transfer could be completed.

acknowledge bus cycle before exchanging data. In response to such a deferred acknowledge, an unassigned interface must return to state *zero* above. Once assigned to a transfer, a wait loop must be executed in state *five* below. If the cycle was accepted, then data transfer can proceed with state *six*, with the outgoing host port assigned to this packet transfer.

5. When a remote acknowledge is received, as denoted by the remote acknowledge function code, parity is checked by the waiting host node to confirm integrity. If the parity checks, the remote acknowledge cycle is accepted from the immediate source, and a data transfer cycle is initiated with the remote host, entering state *six*. If the parity check fails, the acknowledge sent back to the interface sending the remote acknowledge cycle indicates a refusal, and the interface returns to the previous waiting loop, either state *zero* or state *five*. If the remote acknowledge cycle is not received within some generous timeout interval, the transfer is aborted using an abort cycle directed at the unanswering node. The abort cycle is initiated from state *eight*.

6. A successfully acknowledged request for a single cycle read causes the host port to enter another waiting state for a return from the memory location being read. Looping in state *six*, expiration of a generous timeout interval will again lead to an abort through state *eight*. Incoming data accesses or abort cycles are checked for parity. If the parity is incorrect, the cycle is refused on the second cycle following receipt. If either cycle is received with correct parity, the cycle is accepted from the immediate source. A validated data access advances to state *seven*, an abort to state *eight*.

7. In the event of a successful return of read data from another host, the data held in the last incoming ABUS or DBUS latch is returned to the processor along with the type coding returned by the function code lines from the remote node. The RAPIDbus port is deassigned, and address accesses enabled again by returning to reset state *zero*.

8. If an abort is received from the remote node, or generated locally from a timeout, the transfer packet is terminated with a bus error signal to the host processor. The interface is deassigned, and on a simple interface, again able to receive incoming address packets.[48]

9. After the processor acknowledges the bus error, the sequencer returns to quiescent state *zero*.

---

[48] A flag is set in a diagnostic register, along with information pointing to the source of the bus error. This register is visible either to the host processor within the local control page, or via a diagnostic path to a front panel processor.

5.1.2.2. Single Cycle Write Request

The single write cycle request begins identically to states *one* through *five*. The write cycle continues from state *four* or *five* as appropriate, branching to state *ten* below.[49]

10. A successfully acknowledged request for a single write cycle causes the originating RAPIDbus port to load the data to be written into the outgoing RAPIDbus latches. The routing field is derived from the upper address lines of the address to which the write is taking place. The function code fields indicate the type of the data being written. A bus cycle is requested from the central arbiter, with an immediate slot destination based on the earlier output of the routing table.

11. As before, when the arbiter responds with a bus grant, the output buffer lines on the originating interface are driven, and latched into the respective bus port as designated by the routing instruction to the arbiter.

12. The incoming data cycle is sent to the parity check logic, and on the second window following transmission, the accessed interface accepts or requests a repeat on the bus cycle. If a repeat is required because of a parity error or a busy bus repeater, execution proceeds with state *ten*, perhaps using the alternate bus. If the accessed interface accepts the cycle, it is assumed to repeat the data transfer as required to the destination, or to execute the write if it is the actual destination. An accepted bus write returns the interface to state *zero*.

5.1.2.3. Multiple Cycle Read Request

The multiword read cycle is designed to facilitate the transfer of many sequential words of memory. The transfer begins as in the single word read, with states *one* through *five* above. Unlike the single word read, the function code indicates to the slave serving the transfer that a multiword transfer has been requested. The address field within the outgoing address cycle indicates the base address with which the transfer is to begin. State *four* or *five* is then followed by a data transfer cycle from master to slave indicating the number of words to be transferred, starting in state *thirteen*.

13. After the slave accepts the multiword packet, the originating interface loads the outgoing latches of the RAPIDbus ports with the number of words to be transferred. The function code indicates that this is multiword parameter transfer, and the routing indicates the home address of the slave node from the map table. A bus cycle is requested from the cage arbiter.

14. Following transmission of the multiword parameter data transfer cycle, parity is checked at the receiving end, and the cycle is repeated if needed beginning with state *thirteen*. The parameter is repeated, or accepted by the immediate destination as directed by the routing field.

---

[49] In state one, the outgoing packet indicated that the transfer was a write in this case, not a read.

15. The originating interface now sits in a loop at state *fifteen* waiting for data to begin returning from the slave interface. Receipt of a data word for which parity checks advances the sequencer to state *sixteen*. Receipt of a valid abort directs the sequencer to state *eight*, prematurely terminating the transfer cycle. Lack of any response from the slave during a generous timeout period results in a abort cycle generated by the originating host, followed by execution at state *eight*.

16. For each word of data received, the local packet address generator can supply a storage address and does decrement the local word count. If the word count is non-zero after storage of a given word, the sequencer reverts to state *fifteen*. Expiration of the word count normally terminates the multiword packet and reverts the interface back to the quiescent state *zero*. Note that data coming in faster than the local write operations can be executed will not be lost since the master will refuse to acknowledge receipt. If needed, this will back up all the way to the slave node, slowing the pipeline down without loss of data.

5.1.2.4. Multiple Cycle Write Request

The multi-word write cycle, like the multi-word read cycle, is designed to facilitate the transfer of large quadlet blocks located at successive addresses. This request begins with states *one* through *five* above, indicating both a write operation in the control field, and a multi-word access in the function code. The parameter is sent between master and slave as in state *thirteen* and *fourteen*.

17. With a multi-word write, the parameter from state *fourteen* is followed by successive data cycles. In each cycle, the information field contains the data to be written, the function code contains the data type, and the routing field contains the home address of the slave. After each quadlet is accepted by the immediate destination, be it slave or repeater, the sequencer advances to state *eighteen*.

18. After each quadlet is sent, the local packet address generator increments the address and decrements the word count. A non-zero word count continues execution at state *seventeen*. If the originating host needs to terminate early, an abort cycle can be sent to the slave. Alternately, if the receiving slave needs to terminate prematurely, an abort cycle may be received. Either abort is sent through state *eight*.

19. Expiration of the word count terminates the transfer normally, deassigning the host node, and resuming the quiescent state *zero*

5.1.2.5. Read-Modify-Write Request

Read-modify-write packets are run as atomic transactions between a particular task and memory block as intended by host architectures that support instructions such as test and set. RAPIDbus II flags such transfers during the initial address cycle through a *RMW** line in the control field of each bus. Two data cycles are then required before the allocated block of memory is released. With the exception of the *RMW** control line, such cycles begin as in states *one* through *seven*. However in state *seven*, instead of terminating the cycle with the incoming read, both master and slave expect the processor to issue a write cycle.

The second half of the cycle is then run as in states *ten* through *twelve*, releasing both master and slave resources at the conclusion of the write operation to state *zero*. If the second data cycle is not forthcoming within a generous timeout interval, an abort cycle can be sent by either side, terminating execution through state *eight*.

5.1.2.6. Single Cycle Read Service

Previous discussions in this section have centered on the role of the master within data transfer operations. The slave plays a crucial support role in each of the above cycles. Note that designations of master and slave can only be made with respect to roles in a single packet transaction. Any node can function as either master or slave if supported by the host. Depending on the complexity of the RAPIDbus port at a particular node, a host could simultaneously act as master and slave for two different transactions.[50]

21. A node is activated as a slave by an address bus cycle delivered by the cage arbiter to the incoming latches of either the ABUS or DBUS. If the local node bus is available, the address cycle is sent to the parity logic to confirm the validity of the cycle. The information field indicates the location of the operand, two size bits within the control field indicate the width, and the routing field indicates the home address of the master.

22. The address cycle is acknowledged differently depending on the proximity of the master making the transfer request. If the master is within the same cage, only a code on the three line acknowledge bus paired to the main bus that the cycle came in on is required. If the master is in a remote cage, as indicated by the cage address bits of the routing field, a remote acknowledge packet must be sent along the ABUS or DBUS to confirm acceptance of the address cycle, in addition to the acknowledge cycle to the immediate bus cycle source. During such a remote acknowledge cycle, the routing field indicates the home address of the master, the function code indicates a remote acknowledge cycle, and the information field is left high.

---

[50] A bit within the function code field of all transfer cycles is required to differentiate between data returning from a read to the processor, and data being written to the host slave.

23. A valid request to available memory results in an acknowledge cycle on the three line acknowledge bus corresponding to the ABUS or DBUS that the cycle arrived on, two bus cycles after the incoming address cycle. If the address packet is accepted, the memory block is assigned to the packet, and other incoming address cycles are refused through the arbiter. If the parity was invalid, or the node unable to handle the request, a refusal is sent on the acknowledge lines, returning the interface to state. *zero*.[51]

24. After accepting the read address cycle, the memory block must either provide the requested memory value, or return an abort cycle to the master. If an abort control cycle is received from the master, the memory access is terminated and the sequencer returns to state *zero*. During normal operation, the memory obtains the desired operand, calculates parity over the outgoing cycle, and requests a bus data cycle from the cage arbiter. The home address of the master is supplied to the routing table to select the immediate destination of the returning data packet. If the master is in the same cage as the slave, this destination will be the master slot. Otherwise it will be the slot occupied by the chosen repeater path.

25. On receiving bus grant, the slave drives the data cycle onto the backplane lines of the chosen bus. The immediate destination, designated by the output of the routing table, receives the data cycle as an access. Two bus windows later the accessed node either accepts or rejects the data cycle. A rejected cycle is resent starting with state *twenty-two*. An accepted cycle causes the slave to return to state *zero*.

5.1.2.7. Single Cycle Write Service

Service of a single cycle write request proceeds similarly to the single cycle read directly above, except the data flows from master to slave following acceptance of the address cycle. States *twenty*, *twenty-one*, and *twenty-two* are identical. From state *twenty-two*, a write service proceeds to state *twenty-five* below.

25. After accepting the write request, the memory block goes into a wait loop at state *twenty-five*. The loop exits on receipt of a data or abort cycle for which parity checks, or expiration of a generous timeout interval. Successful receipt of data advances to state *twenty-six*. Receiving an abort cycle, the service terminates, enabling incoming address cycles, and returns to state *zero*. Expiration of a timeout causes an abort cycle to be sent to the master, followed by an enabling of incoming address cycles and a return to state *zero*.

26. On the successful receipt of a data cycle, the node conveys a positive acknowledge to the immediate source of the bus cycle, and stores the latched data in the previously assigned memory location. On termination of the storage cycle, the node deassigns the memory block and returns to state *zero*.

---

[51] If the sequencer was unable to process the request, the default is to refuse address cycles, even without the intervention of the sequencer.

5.1.2.8. Multiple Cycle Read Service

Service of a multiword read request begins with the acceptance of the address cycle, as in state *twenty* through *twenty-two* with a multiword packet function code. The host node then continues with state *twenty-seven*.

27. After accepting and acknowledging the address cycle, the interface loads the address cycle information field into the base address register of the slave packet address generator, and enters a wait loop at state *twenty-eight*.

28. The slave must then wait for the receipt of a parameter cycle indicating the number of memory words to be transferred during the following data cycles. If the parameter is not received within a generous timeout interval, an abort cycle is sent to the master, the memory block deassigned, and incoming address cycles enabled. Receipt of the parameter cycle advances the sequencer to state *twenty-nine* after parity has been validated and the immediate bus cycle acknowledged.

29. The packet address generator now supplies an address to the local memory block for a quadlet read access. On successful read, the sequencer advances to state *thirty*.

30. The data produced by the memory is sent via a data bus cycle to the master node. The data type is supplied by the memory block, and is sent back to the master within the function code field. The routing field contains the home address of the master. The packet address generator (address) is incremented, and the word count decremented. If the word count is zero, the memory block is deassigned, incoming address cycles are enabled, and the interface returns to state *zero*. Otherwise another quadlet is accessed and sent by looping to state *twenty-nine*.

5.1.2.9. Multiple Cycle Write Service

Multiple word write cycle service begins with receipt of an address cycle with a multi-word function code, and activated write control line. It is acknowledged either directly by the three line acknowledge bus directly within the cage, or through a remote acknowledge cycle in the case of foreign server. The base address register of the packet address generator is loaded with the information field from the address cycle, and the parameter cycle is loaded into the word count register as in states *twenty-seven* and *twenty-eight* above. Execution then continues with state *thirty-one*.

31. The slave loops in state *thirty-one*, waiting for either a data or an abort packet for which parity checks. The slave responds with a receipt refusal for all incoming cycles with bad parity. Receipt of a valid abort cycle deassigns the slave to the transfer. On receipt of a valid data cycle, the immediate source is acknowledged, and execution proceeds with state *thirty-two*.

32. A valid data cycle is stored in the memory location selected by the address register of the slave packet address generator. The word count is decremented, branching on a non-zero value back to state *thirty-one*. When the word count reaches zero, the slave terminates normally by deassigning the memory block and accepting new incoming address cycles.

5.1.2.10. Read-Modify-Write Service

Service of read-modify-write packets occurs in two stages. An incoming address cycle is accepted with the *RMW* line asserted, and execution proceeds like a read operation in states *twenty-one* through *twenty-five*. Instead of terminating in state *twenty-five*, the slave loops, waiting for the write portion of the cycle. If the write is not forthcoming within the timeout, an abort cycle is sent to the master, and the slave deassigned. Normally the write section will proceed as in state *twenty-six*, terminating normally by deassigning.

5.1.2.11. Repeater Forward Service

Links between cages are implemented using pairs of special nodes, called repeaters. One repeater is inserted in each of the two cages that are to be linked to create a point-to-point link. The repeater forwarding service is initiated by the receipt of any transfer cycle at the RAPIDbus port of the repeater. Execution of the service routine begins with state *thirty-three*.

33. Any bus cycle received by the RAPIDbus port of the repeater is checked for correct parity. Any parity error results in a refusal to the immediate source along the acknowledge bus paired to the bus on which the cycle was received. If the cycle is valid, and the repeater is not already handling a request, it accepts the bus cycle, continuing with state *thirty-four*.

34. Upon receiving a valid bus cycle, the repeater requests a forwarding by the repeater on the other side of the point-to-point link. The servicing repeater loops in state *thirty-four*, waiting for the second half of the link to acknowledge the request. Since the second repeater is running on a different time base, two synchronization steps are required for the link grant to be returned, and later another to confirm the cycle. When the link grant is received, execution proceeds to state *thirty-five*. Timeout causes the link request to be removed and an abort cycle sent back to the source of the cycle, deassigning the repeater.

35. Receipt of a link grant causes the incoming information, function code, control, routing, and parity fields, augmented with a validity strobe, to be passed over the link to the second cage, where parity is again checked. If parity is confirmed, a transfer acknowledge will be received back, confirming the transfer and deassigning the first repeater node. If the acknowledge is not received within a timeout interval, an error signal is asserted, and an abort sent back to the source of the cycle. Following the abort, the repeater sets bits in it's diagnostic register and deassigns itself.

5.1.2.12. Repeater Forward Request

Repeater nodes can act as either servers or requesters at any instant, either taking a cycle from the bus for relay across the link, or taking a cycle on the link for transmission across the backplane of the cage in which the repeater is installed. A service request is initiated by the remote pair of the link asserting link request. Execution begins in state *thirty-six*.

36. An incoming link request must be synchronized to the time base of the second cage. If the repeater is not previously assigned, it will respond with a link grant. The repeater then waits until the validity strobe is asserted before proceeding to state *thirty-seven*. If timeout occurs before the strobe is asserted, link error will be asserted by the granting repeater, and the connection terminated.

37. Assertion of the validity strobe on the link results in a comparison of link parity against the information on the link. If parity is correct, an acknowledge is sent to the first repeater, terminating the intercage link. The receiving repeater then calculates the address of the next node based on the function code, routing, and upper address lines. An appropriate cycle is requested on the backplane busses, and the cycle forwarded. Refusals result in a retry, or eventually in an abort cycle directed back to the original source of the bus through the first repeater. Successful acknowledge of the bus cycle deassigns the repeater from the transfer.

5.1.2.13. Interrupt Generation

Interrupt packets begin with a node functioning as an interrupter. An interrupter functions identically to the single word write request above in states *one* through *five* and *ten* through *twelve* above except for the contents of the transfer cycles. The information field contains the address of the handler, as determined by an interrupt routing table, in the high order byte. The lower information lines are unused. The routing field has the home address of the interrupter. The control fields indicate a quadlet write. The function code lines indicate a interrupt cycle.

During the data cycle, the upper doublet of the information lines encodes the level of the interrupt in the least significant three bits, and a doublet vector in the lower doublet. The doublet vector is a previously agreed upon descriptor of the required service.

5.1.2.14. Interrupt Reception

Interrupt reception functions analogously to a single cycle write service, translating the incoming interrupt packet into the proper interrupt format for the interrupt handler. In the instance of a 68000 handler, the encoded interrupt level is prioritized and sent to the processor interrupt lines. Meanwhile, the interrupt parameter is queued in a FIFO. When the processor runs a RAPIDbus priority interrupt acknowledge cycle, the lowest byte of the parameter is provided to vector the interrupt handler. The second byte of the parameter is discarded in the proof-of-concept implementation.

5.1.3. System Reliability

Successful achievement of RAPIDbus II design goals requires that the resulting design effectively support algorithm research. The stress inherent in a low cycle time realization with numerous packages dictates that the resulting design must be tolerant of subsystem failures and support rapid localization of faults. Both the interchange redundancy and the diagnostic assistance built into the design are examples of the system reliability considerations which are intended to make RAPIDbus II a practical laboratory tool.

5.1.3.1. Interchange Redundancy

The choice of many, multiply interconnected bus segments contributes not only to performance by allowing localized allocation of bus bandwidth, but also to the reliability of the entire processor ensemble. Perhaps the best support for this strategy comes from the telephone switching industry, with a long history of successfully fielding large interchange systems.

Bell's experience with the first electronic switching systems [*ESS*] put solid experience behind the need to divide a complex system into many small, redundant fault modules [44]. The reliability of any large system, such as that shown in figure 5-6 can be represented as the product of the reliability of all the subcomponents which must work reliably for proper operation. As additional parts are added with a finite failure rate, the mean-time-to-failure drops.

The *ESS's* divide the required structure into a multitude of small, replaceable modules which directly spare each other, as shown in figure 5-7. If the sparing mechanism does not form a single point of failure, then several failures must occur in order to interrupt operation. As the

Figure 5-6: A single monolithic structure presents a multitude of independent sources of failure, any one of which can fail the system.

number of components subject to failure decreases in a module, and the parallelism increases, reliability can be made almost arbitrarily high. This parallelism translates into increased performance with RAPIDbus since the spared units are all doing useful work until a failure is detected. Since the sparing mechanism (repeater links) used by RAPIDbus introduces a performance penalty as the fault module (cage size) becomes smaller, the reliability of each RAPIDbus interface to a backplane bus interacts with the performance cost of increased repeater cycles to set the cage size.

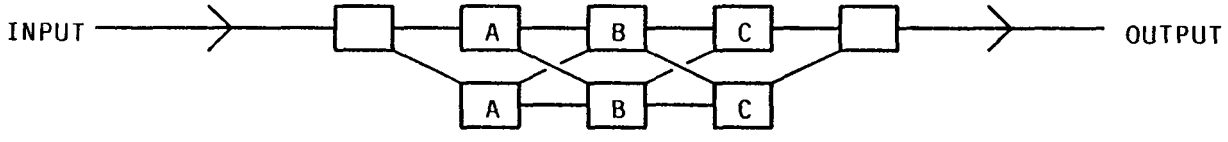

Figure 5-7: Dividing a system into many, spared modules can increase fault tolerance.

5.1.3.2. Diagnostic Assistance

Dividing the system interconnect into many small pieces decreased the probability of a given subsystem failure stopping the system. Yet if failures are allowed to accumulate, any spared system will fail. Thus the reliable design and liberal sparing of small modules must be assisted by a rapid fault localization process to the board level. In the design of a fault localization system, it is useful to divide subsystem failures into four classes; interconnect, power, and logic faults. Each fault class is best handled by a different localization system.

In a debugged system, interconnect failures are probably the most common failure mechanism. As the average module size decreases, the number of interconnects generally increase for a given system complexity. Failure modes include not being fully inserted, contact oxidation, and broken connectors. If each module can have a minimal functionality test initiated and monitored through two or more interconnect ports, localization of any single failed interconnect is simplified. Since each processor node can have ROMed diagnostics run from either a diagnostic serial line or through the RAPIDbus, processor-interchange connections can be verified if testable memory locations, such as the control page, are visible

from both the bus and processor. Nodes populated solely with memory cannot run such a test, and thus present a potential fault ambiguity.

In an age of dynamic storage systems, where a system cannot do operate at less than tens of thousands of cycles per second, localized power failures provide a practical excuse for a visual indicator on each module. In combination with current monitoring on power busses, voltage monitoring lights help spot both power distribution and interconnect problems.

The RAPIDbus II design provides for the isolation of both intermittent and hard faults. Each interface is equipped with an eight bit fault diagnosis register, visible either to the host processor or through an extension of the arbiter-interface cable to a cage level monitor. The monitor could either be a socket for a logic analyzer, or an input port for a simple eight bit cage diagnostic processor.

Both hardware fault isolation, and kernal level debugging are assisted by one or more test headers to trace processor and node state changes. In combination with ROMed diagnostics initiated via a test switch, such test points can rapidly confirm a hard failure diagnosis at the subsystem level.

5.1.4. Upward Compatibility

The basic RAPIDbus II implementation described above was designed to support a simple proof-of-concept demonstration of both the RAPIDbus II architecture, and the underlying application hypothesis. The implementation was built around the limitations of existing performance microprocessors, and standard logic parts. If these requirements are relaxed, it is useful to consider how performance might be practically extended significantly beyond the current implementation while maintaining high level language compatibility and increasing cost-effectiveness.

In order to maximize the effectiveness of enhancements, comparable changes must occur simultaneously in the interchange and data storage components of the system. A design is proposed for a switching plane which reimplements the redundant buses within a RAPIDbus cage using a parallel switching plane.

5.1.4.1. Parallel Switching Plane

In order to motive the topological transformation from a time-multiplexed common bus implementation to a parallel switching plane, it is useful to consider the efficiency with which the RAPIDbus II interface hardware is used, both within a cage and across links between cages.

Along each of the primary busses, the ABUS or DBUS, only one pair out of potentially fifteen drivers and latch can be active during any bus window. Although the backplane can achieve near 100% productive information efficiency,[52] bus interface hardware at any one node, on average, is used less than 15% of the time. Considering drivers and receivers separately, less than 7% efficiency is achieved. Driving a physically disperse (17 inch) backplane also places a lower limit on the minimum window cycle time.[53] A packet switch bus system, *MARTINUS*, provides a topological link between the packet switched common bus implementation and the switching plane proposed for an enhanced RAPIDbus implementation. Designed by the Norwegian Defense Research Establishment (NDRE), the *MARTINUS* multiprocessor is based on a pair of custom NMOS chips [69]. Conceptually, the same bit position from sixteen different host ports is collected together on a single chip per bit position, queued, and then switched on a very high speed bus internal to the die as a packet. This reduces the bus loading from a large backplane to the drivers and receivers on a single chip. Links to and from the hosts are point to point, running in parallel to and from the switch matrix for all hosts.

Although there is a vast literature on switching plane networks, this design provided a topological link between the bandwidth limitations of a common bus and a compatible switching plane structure. Once the lines were collected together on one die, it was then useful to consider ways of increasing parallelism so as to increase the average duty cycle of each port beyond one part in fifteen.

Evaluation of new bipolar gate array technology suggested that sixteen, sixteen input multiplexers, output latches, and logic to retain routings for each multiplexer could be placed in a single package, effectively a bit slice of a cross-bar. Unfortunately, each multiplexer required four bits to steer the output, or sixty-four lines for routing if all were brought to the

---

[52] Refused address cycles and unused control field cycles prevent absolutely efficient bus usage.

[53] Estimates suggest that use of single ended ECL would permit a 25 nanosecond bus window, or differential ECL a 12.5 nanosecond cycle. Even with packets traveling at the speed of light, little more than a factor of fifty increase in throughput is available with a common bus in this geometry over the current implementation.

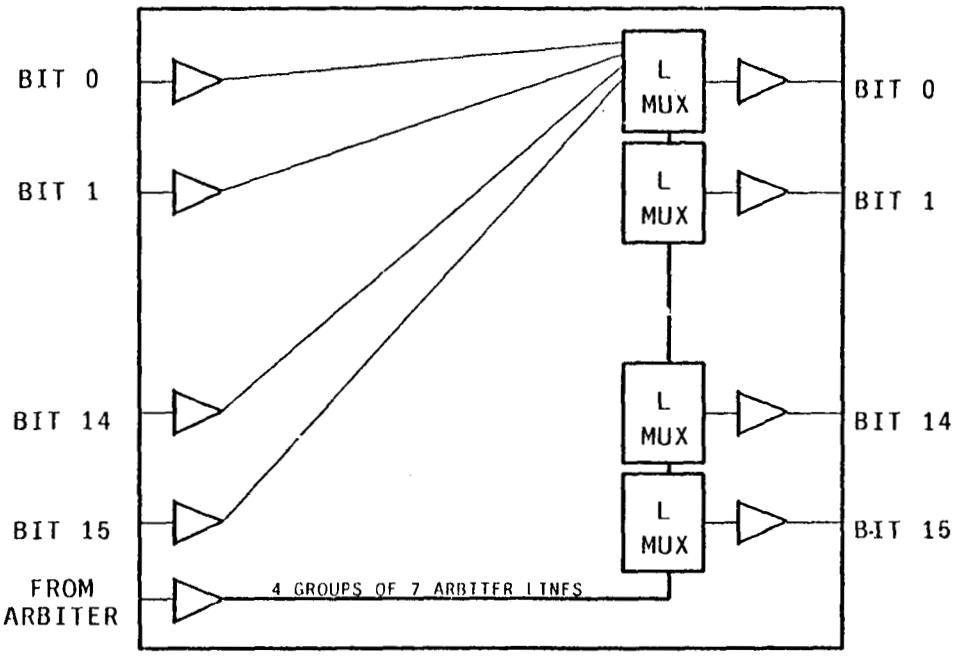

Figure 5-8: Bit slice crosspoint switch permits changing one routing per cycle in each of four groups.

outside of the chip. This routing problem has commonly driven switching plane packages to multiple stage bit planes with a smaller fan-in and fan-out, or to the incorporation of routing information into the streams being switched.[54]

Using a switching plane strategy, the cycle time of the interconnect is not limited by the time to charge and later latch a physically distributed backplane, rather all connections are either point-to-point (host/switch) or one driver to many receivers (arbitration routing for all chips). Since both can be pipelined to almost arbitrary throughput, the switching plane suggested the possibility of reducing the fifty-six lines sent on one long window into four cycles of sixteen bits sent on faster windows. A potential packing scheme is illustrated in figure 5-9, where the auxiliary fields suggest the ease with which additional information can be incorporated.

Since once a connection was made between input and output, it was retained for at least four cycles, routing information to the multiplexers could be multiplexed as well with little loss of throughput. For each group of four multiplexers, two lines select the routing latch, one enables it, and four lines provide the routing information. Thus routing can be accomplished

---

[54] Adding depth to the switching plane increases the number of chip I/O buffers traversed, increasing latency. Adding routing information into the bit stream can cause competition for data transfer bandwidth.

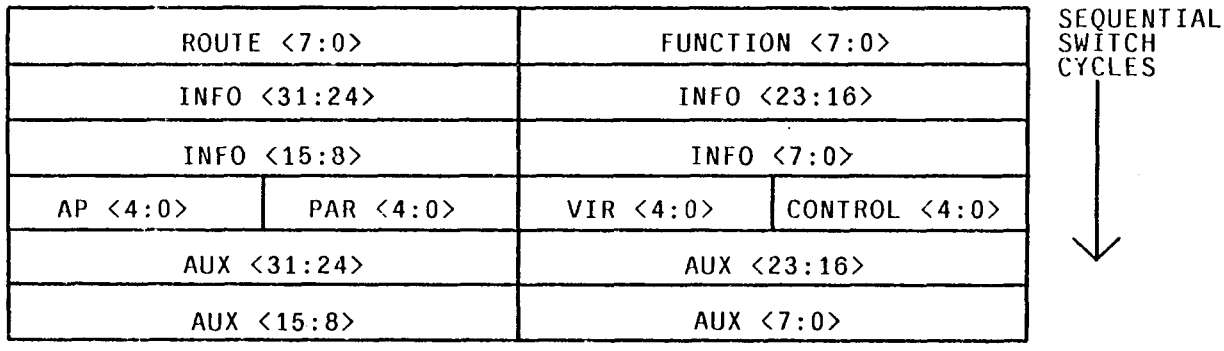

Figure 5-9: Many of the same fields carried in parallel with the common bus implementation are doublet serialized with the crosspoint switch, decreasing data path width.

by twenty-eight pins, accommodated along with thirty-two data lines and a clock on a large contemporary bipolar gate array as shown in figure 5-8. Preliminary indications suggest that the regularity of the switch will not prohibit routing of the chip.

Each RAPIDbus II society is then capable of being implemented by an array of eighteen such chips as in figure 5-10. Connections to and from the RAPIDbus port on each host reduce to thirty-six unidirectional lines, replacing the fifty-six bidirectional lines used previously. The *live* bit shown in figure 5-10 indicates that valid information is passing to the destination (in contrast to null words sent while waiting for arbiter routing to change). The *ack* bit fulfills some of the functions of the acknowledge bus used on the common bus implementation. The additional lines to and from the host ports in addition to the sixteen switched lines control clocking and port reset. Links to other processor societies would be implemented between switch ports in extension of the current link scheme.

As a suggestion for an enhanced RAPIDbus II implementation, the 16 x 16 cross-point module appears to both reduce the number of chips required to implement a society of processors, and to increase the interconnect bandwidth. Conversion of the host/switch link to a narrower, unidirectional data path supports use of fiber optic or other high-bandwidth, unidirectional media.

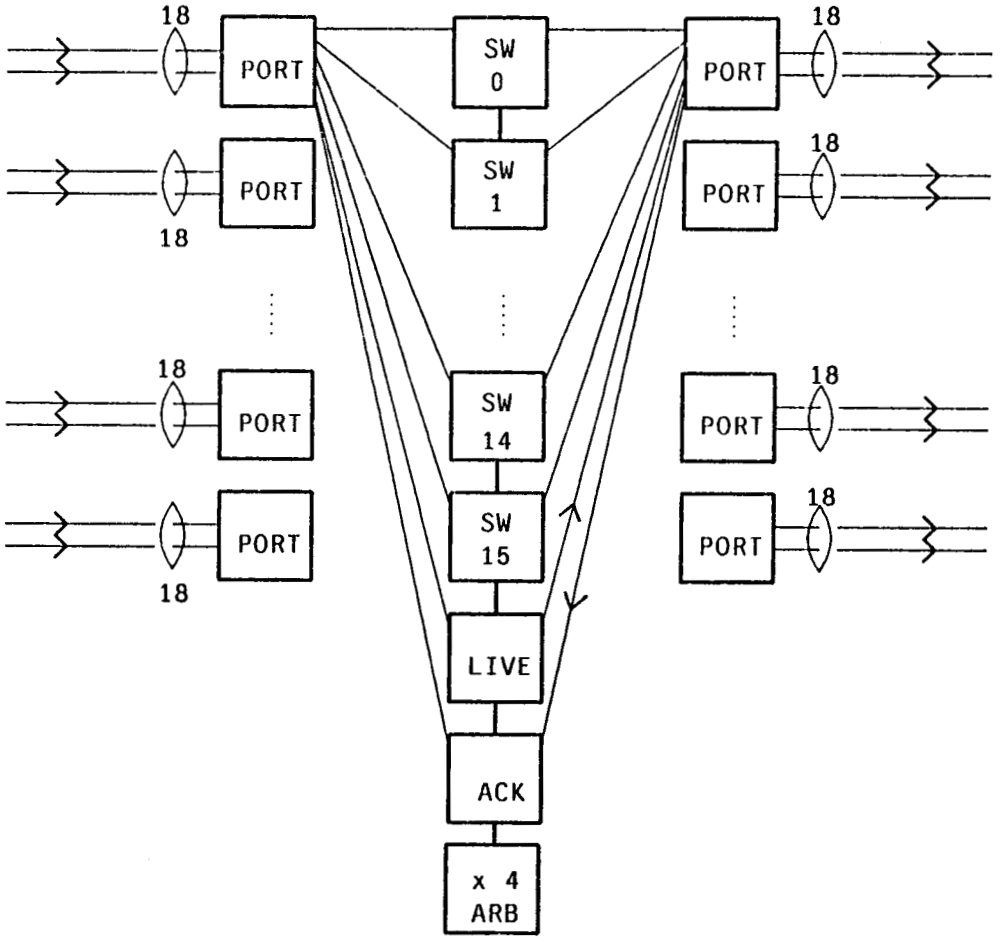

Figure 5-10: Eighteen bit slice crosspoint chips interconnect a society of RAPIDbus II processor nodes.

5.2. Realization

Building on the experience with high speed hardware design gained with RAPIDbus I, the RAPIDbus II architecture and implementation are being realized in a minimal proof-of-concept system. Technological advances with registered proms and bidirectional registered tranceivers contribute performance and flexibility enhancements beyond those optimizations possible at higher levels of abstraction.

5.2.1. Physical Structure

A RAPIDbus II proof-of-concept realization has been designed based on a combination of Advanced Schottky TTL and 10K ECL logic within an extended Versabus packaging. Existing hosts are integrated into the system using a microcoded RAPIDbus interface card which doubles as a cage repeater as shown in figure 5-11.

The ABUS and DBUS are realized on existing Versabus backplane lines, while the arbiter, clock, reset, diagnostic port, and serial line connections are made via the I/O pins on the P2 connector. The cage arbiter is suspended under the Versabus cage, with a cable to the P2 connector of each occupied host slot.

The microcoded RAPIDbus interface fits in a standard Versabus enclosure with an additional inch at the top to support a host P1 and P2 connector and four fifty-pin ribbon cables. The P1 and P2 edge connectors allow a modified Versabus card to be inserted in the top of the interface card to create a processor node a little more than twice the height of a standard Versabus cage. Alternately, the four, fifty pin ribbon cables can be used to tie two interface card together to create a repeater pair between cages, or to tie into an existing circuit-switched backplane. The flexibility of the microcode interface allows a remote host adapter to be populated with as few as ten chips.

5.2.2. Microcoded Host Interface

The micro-coded RAPIDbus II interface was designed to support rapid integration of a multitude of existing hosts into a proof-of-concept system. Shown in block diagram form in figure 5-12, the RAPIDbus ABUS, DBUS, and arbiter interface is shown at the bottom of the drawing. The generalized host interface, or HBUS, is shown at the top of figure 5-12.

Under microcode control, information is exchanged among ABUS, DBUS, and HBUS ports

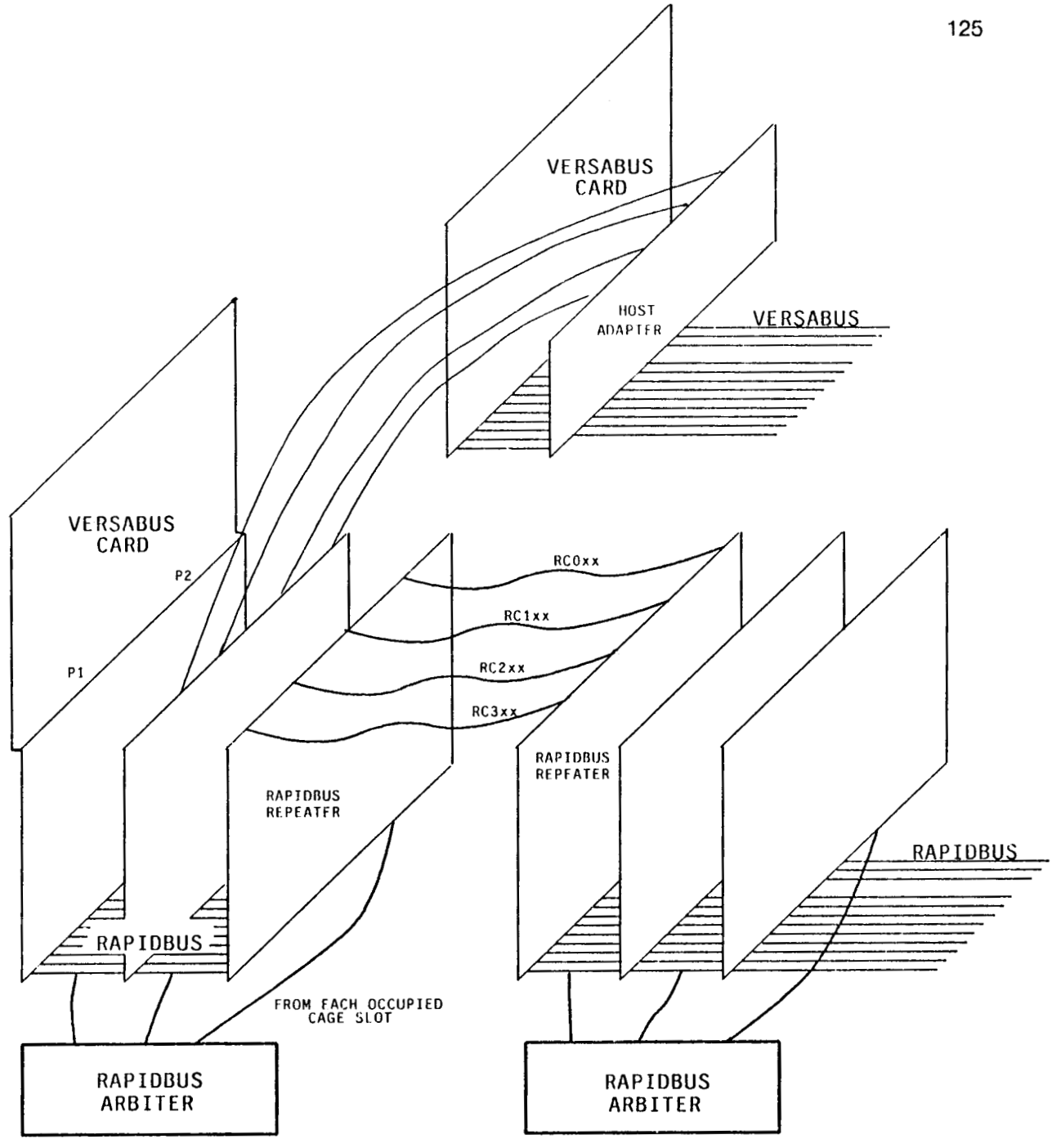

Figure 5-11: RAPIDbus II proof-of-concept realization.

along one of three buses internal to the interface card, the IBUS, the AMBUS, or the RBUS. The IBUS is composed of 32 address/data lines and six control lines; *AS**, *WRITE**, *RMW**, *SIZE.0**, *SIZE.1**, *and DS**.[55] The AMBUS is used for address modifier codes, and the RBUS for interface routing codes, both along byte-wide data paths.

[55] Address strobe and data strobe are not carried on the bus, but are regenerated on each RAPIDbus card.

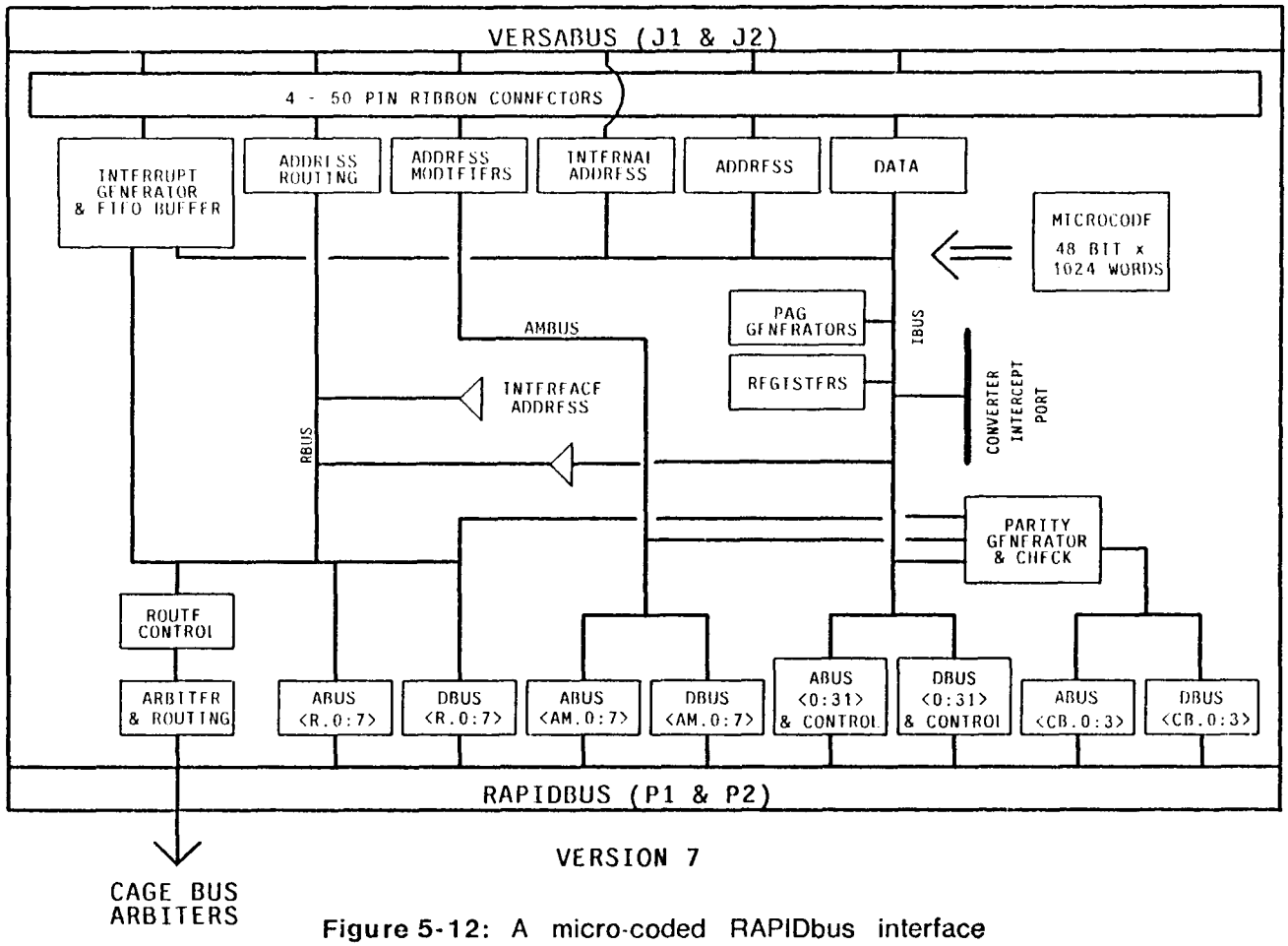

Figure 5-12: A micro-coded RAPIDbus interface simplifies the integration of existing processor nodes.

The IBUS supports several devices which serve to source, sink, or transform externally supplied IBUS information. Octal bidirectional transceivers port to the RAPIDbus's ABUS and DBUS transfer lines. Address and data ports within the HBUS port demultiplex the IBUS for presentation to the Versabus host. Internal address buffers allow the interface to monitor address and control lines internal to the attached host to determine where an address reference will go prior to granting bus access to the host.

In order to support a mixture of sixteen and thirty-two bit hosts within a system, interface microcode must perform dynamic bus sizing using an IBUS crossover. This crossover allows information from the lowest sixteen IBUS lines to be routed to the upper sixteen, or the upper sixteen to be routed to the lower sixteen IBUS lines. For instance a thirty-two bit host can initiate a thirty-two bit write operation which is resolved as two independent sixteen bit write operations at the destination RAPIDbus interface.

The packet address generator, another IBUS attachment, generates sequential addresses and decrements word count to support multiword data packets. Since no current Versabus hosts support the sequential access modifier code, the interface traps on a local control page address, replacing the data transfer modifier code and initiating a microcode sequence which traps each successive read or write operation to the sequential address trigger location until the word count in the transmitting interface is exhausted.

In order to provide support for hosts which use data formats incompatible with the majority of other hosts in a RAPIDbus system, an edge connector is provided with connections to IBUS and required control lines. Physically located in the center of the interface card, this allows connection of a converter intercept board personalized to the host processor data types. For instance, one converter board could be used to convert user data transfer cycles from ASCII to EBCDIC and EBCDIC to ASCII on the fly. Another converter intercept board allows conversion between a DEC floating point format array processor and the IEEE floating point format used by RAPIDbus. Most conversions require two to three additional interface clock cycles.

The IBUS is used by the interrupt buffer/generator in order to unify interrupts with other data transfer operations on the bus. In response to a host interrupt, the interface runs an interrupt acknowledge cycle during which the interrupt level is encoded on *IBUS.01**, *IBUS.02**, *and IBUS.03**. The interrupting host is then expected to return an interrupt vector byte on the lowest eight data lines. The interrupt generator combines this byte with an encoded interrupt level on *IBUS.16**, *and IBUS.17** to generate a RAPIDbus interrupt packet which includes the interrupt vector.

An interface receiving an interrupt packet queues the vector and interrupt level in a FIFO. If the top level of the FIFO is occupied, an interrupt is sent to the Versabus host. In response to an interrupt acknowledge cycle, the vector stored in the FIFO is supplied to the host, completing the interrupt cycle. Note that although four interrupt levels are supported,[56] a host configured to service more than one host interrupt level will only see the oldest queued interrupt, not the highest priority interrupt, in the queue.

Two explicitly addressable registers are provided on the interface, a control register, and an address extension register. The control register bits zero through four are used to bank

---

[56] Versabus levels one, two, three, and four.

switch Versabus interrupts into one of sixty-four different groups. Bits five, six, and seven of the control register are used to direct conversion of user data by the converter intercept board. The address extension register is used by hosts with a 24 bit address space to enable bank switching into the full 32 bit RAPIDbus II addressing range. Both registers are only accessible to a processor running in supervisor mode.

The AMBUS, or address modifier bus, is driven by four sources, the address modifier field of the ABUS or DBUS, the address modifier field of the HBUS, or the interface address modifier generation buffer. The generation buffer is used to replace the address modifier codes during a multiword packet transfer, or while obtaining an interrupt vector from an interrupting host. The ABUS, DBUS, and HBUS buffers also monitor AMBUS lines, latching them according to microcode instructions.

The RBUS, or routing bus, is used to control the coupling of the originating and destination interface RAPIDbus host ports. When an address packet is sent, host address lines are used by the routing control circuit to select the destination host slot within the originating cage. If the destination is not in the local cage, the routing control selects the required repeater. When the originating interface is granted a bus cycle, the address of the originating interface goes along with the destination address. In this way the slave interface will know where to respond with data from a read request. During a data transfer, the routing lines of either ABUS or DBUS are used to designate the destination host slot. If a memory page is accessed outside of the originating cage, the home address of the originating interface is returned by the slave interface to indicate a successfully received address packet.[57]

Parity is checked on the high speed bus for groups of eight bits on an interleaved basis. Thus a parity error on either a single line, or on several physically adjacent lines will be detected, forcing repeat of the transfer operation from the immediate sending node. A fifty-nanosecond bus window is intended to make parity errors a very infrequent event.

On the ABUS and DBUS, a three line packet transfer acknowledge scheme is used. On the third bus cycle following a grant on either bus, the corresponding three acknowledge lines are driven. The most significant bit of the three is driven high to indicate a receipt error, requiring a retransmission. The remaining two lines indicate the width of data accepted by the local destination, or indicate that the cycle was repeated.

[57]Data packets are only acknowledged on a link by link basis, never reverting all the way back to the originating cage when the reference traverses repeaters. In accepting the original address packet, the slave host is confirming that it can process the required transfer request.

5.2.3. Timing Analysis

In order to provide a basis for performance evaluation, it is useful to estimate the service latencies for both local and interchange references on a RAPIDbus II processor node comparable to the Versabus VM02 monoboard computer used in RAPIDbus I. A version of the microcoded host interface described above is considered with a dedicated 68000, a single bank of dual ported memory, no error detection, and a state machine synchronized to the RAPIDbus cage timebase. The processor is assumed to run at half the clock frequency of the board state machine. Below, the microcode state changes are enumerated for both a local and cage level access. Write operations can skip some stages, as noted by the * prefacing the line.

5.2.3.1. Local Memory Access

1. Processor *AS** to local resource decode
2. Processor address to memory block on Ibus
3. Initiate memory access (row address latched)
   (Processor to memory block data transfer on Ibus)
4. *(Column address latched)
5. *(Ram access time)
6. *(Ram data available, four more clock cycles to complete ram cycle)
7. *Memory to processor data transfer on Ibus

Thus the processor can access local memory in 3.5 processor clock cycles for a read, or 1.5 for a write if there is no contention from either refresh or the RAPIDbus.

5.2.3.2. RAPIDbus Access

1. Processor *AS** to local resource decoded
2. Processor address to RAPIDbus interface port
3. Routing table lookup
4. RAPIDbus cage arbiter delay (contention may add arbitration cycles depending on the bus priority of the host and momentary bus load.)
5. RAPIDbus address cycle on ABUS or DBUS

6. Slave routes address to local parity tree

7. Slave accepts reference and routes to memory block

8. Slave initiate memory access (row address latched), address acknowledge is returned to originating processor.
   Write: Processor sends data via IBUS to RAPIDbus interface.

9. Slave continues access (Column address latched)
   (9) Write: Originating RAPIDbus interface requests ABUS or DBUS cycle.

10. Slave continues memory block operation.
    Write: Arbiter considers bus cycle request (may add cycles from contention).

11. Read: Slave memory block produces data and requests a cycle on the ABUS or DBUS.
    Write: Master sends data via the RAPIDbus.

12. Read: Data flows from memory block to slave RAPIDbus interface.
    Write: Data flows from the slave RAPIDbus interface to the local parity tree and the appropriate memory block.

13. Read: Slave sends ABUS or DBUS cycle
    Write: Memory block completes cycle, parity accepts bus cycle

14. Read: Master host sends data from RAPIDbus to processor, parity tree.
    Write: Slave runs data cycle acknowledge to master.

15. Read: Processor receives data acknowledge upon parity confirmation.
    Write: Processor receives acknowledge upon receiving acknowledge from slave.

Thus, at least seven and a half processor clock cycles are required by this implementation to read or write within the immediate processor cage. This translates to six wait cycles for the 68000.[58] Faster memory times would improve the read cycle, but the write is limited by the time for the slave to accept the access request.

5.2.4. Evaluation Methodology

Based on the estimated performance above, and timing data taken from a Motorola VM02 Versabus processor card, it is useful to compare the relative performance of the two bus structures under conditions of high interprocess communication. A queuing simulation was constructed, driven by a synthetic workload, and the relative throughputs of the two systems measured for varying numbers of processors.

---

[58] This could be reduced to four wait cycles if a bus interface was used that allowed parity to be checked without gaining IBUS mastership. Such a capability was provided by our custom bus interface, but was not available with the stock bus interface parts used here.

The workload was based on the assumption that 25% of the memory access were write operations, and that an "average" 68000 instruction had 2.25 clock cycle between assertions of address strobe.[59] For the workload, a poisson distribution was assumed around 2.25 instructions to account for the variety of instructions in a "typical" mix.

On the X axis, the number of processors was varied from one to fifteen. The Y axis shows the system efficiency metric, an index of how the throughput of N processors compares to a single VM02 processor running the same code. Since most problems require additional instructions when decomposed across multiple processors, this speedup index probably represents an upper bound on what can be expected.

In order to simulate a variety of different algorithms with differing degrees of coupling, simulation runs were done with fourty, sixty, and eighty percent of the references to local memory. Note that this differs from the definition of r used earlier by a factor of two. Such high degrees of interprocessor communication were intended to stress the interconnect and are not typical of some applications.

5.2.5. Extended Versabus Simulation

Figure 5-13 shows the results of running the simulated workload in a Versabus cage with a collection of one to fifteen processors. Although the application with fewer references across the common bus ran more efficiently due to the lower contention, all three leveled out at no more than eight times the throughput of a single VM02.

Since the Versabus is only allocated to a single master for the duration of an entire memory cycle, the contention seen here as additional processors contend for the bus is completely understandable. Even if bus window repeaters were installed to allow more than fifteen processor slots, the same bandwidth would be available, shared by a larger number of processors.

Versabus is not an effective way to tie together large numbers of tightly coupled processors. For an application where 40% or more of the references are mapped through Versabus, no more than four processors result in useful performance increases.

---

[59]The estimate of 2.25 clock cycles was derived from monitor logic traces, and PDP-11 instruction frequencies mapped into the 68000 instruction set [47].

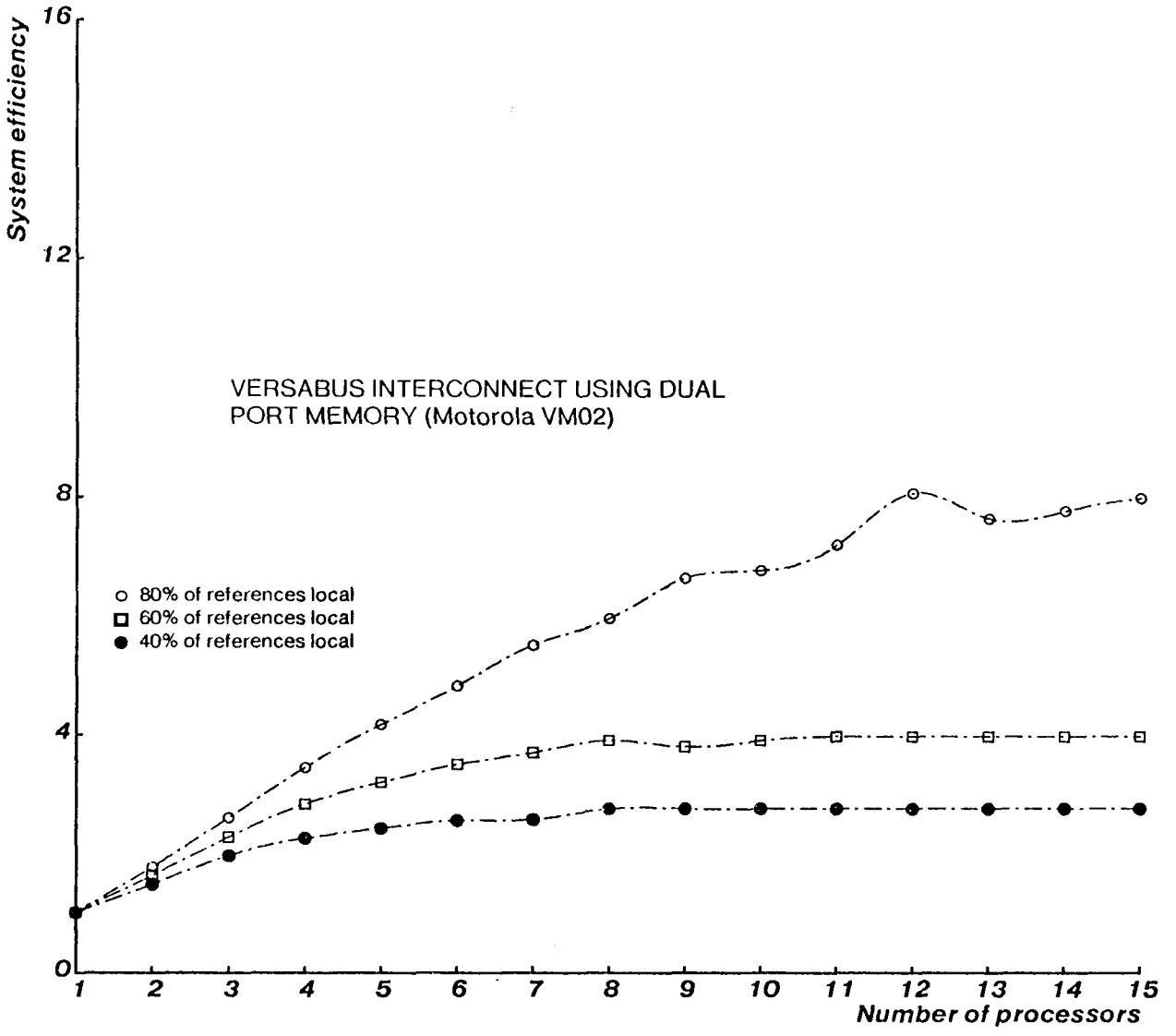

Figure 5-13: Versabus systems quickly saturate in a tightly coupled system such that increasing the number of processors does not improve the throughput.

5.2.6. RAPIDbus Society Simulation

The result of building a RAPIDbus based system is shown in figure 5-14 for the same workload, and high inter-processor load. Resulting from the lower contention for backplane bandwidth, and reduced contention for each host port, performance continues to increase up to the implementation limit, roughly linearly, although *not* with unity slope.

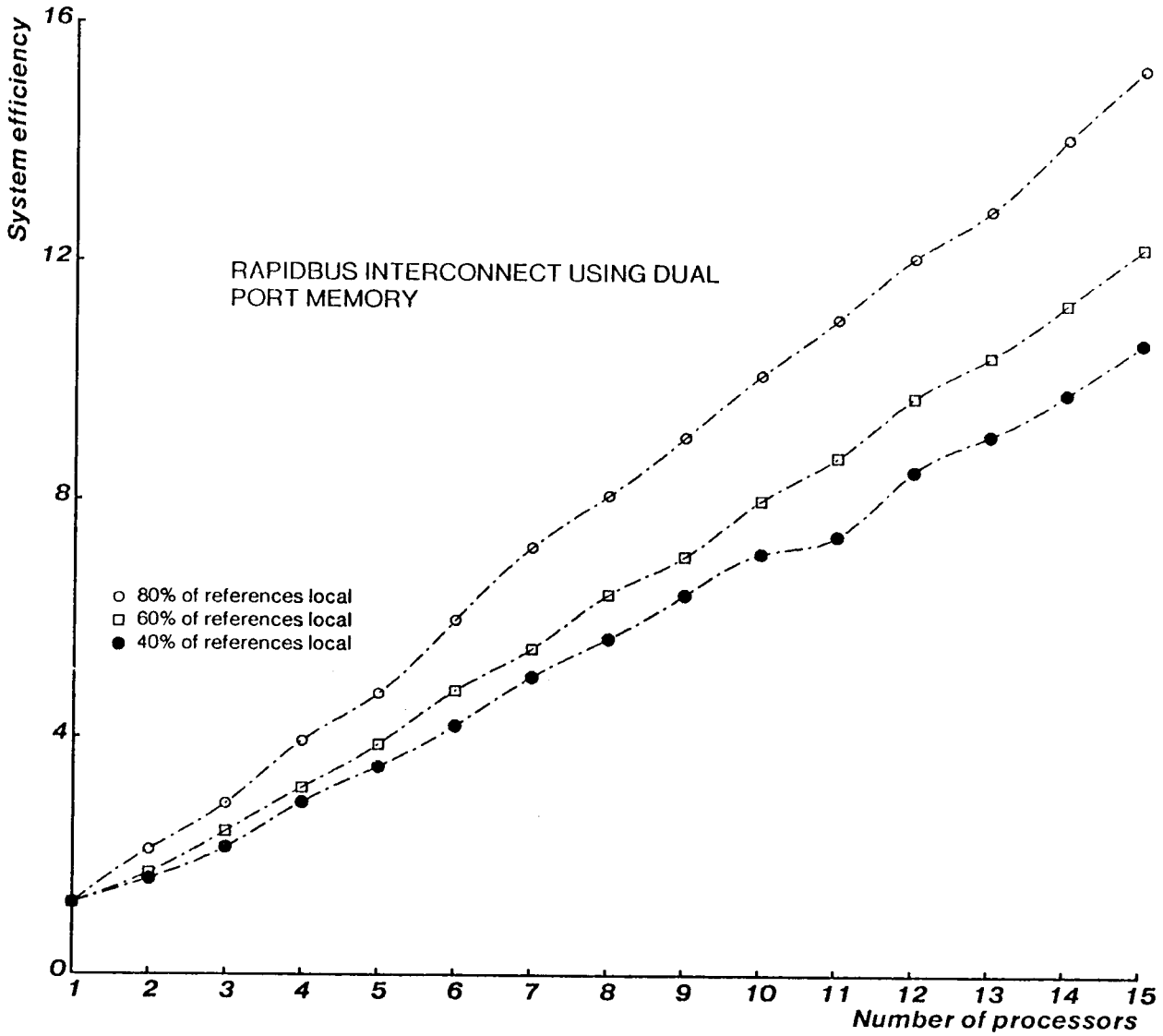

Figure 5-14: Use of a RAPIDbus II interchange network significantly reduces bus contention in a tightly coupled application with respect to a similar Versabus system.

In contrast to a circuit-switched protocol such as Versabus, RAPIDbus II allows additional cages to be added to the system with independently allocated bandwidth. Thus as additional societies are added to a system, the total bandwidth increases linearly, short that required for references crossing a cage link.

Based on this simulation, RAPIDbus II shows potential for the high bandwidth

interconnection of many host nodes grouped into societies. The radius at which performance limits the number of processors is both configuration and code dependent, and has not been accurately estimated.

5.2.7. The Design Space

Although numerous attempts have been made to categorize multiprocessors [36, 26, 65], current literature suggests that we still do not understand enough about them to create a comfortable metric describing a design space [37]. Thus it is useful to locate RAPIDbus II within the design space by briefly describing several other multiprocessor designs. For a detailed understanding of each, the reader is referred to the cited references.

5.2.7.1. RAPIDbus I

RAPIDbus I supported four virtual Versabuses using a time-multiplexed common bus implementation [77]. Designed by the machine vision group at Carnegie-Mellon's Robotics Institute, a two processor system was fabricated using Motorola VM02 processors. A 1.7 microsecond transfer latency was observed with the VM02, although the theoretical system bandwidth over a 32 bit data path was sixteen megabytes per second. As a project in hardware design, no system software was written for the machine.

5.2.7.2. RAPIDbus II

RAPIDbus II, described in this report, builds on the lessons learned from RAPIDbus I. Using societies of fifteen dual port processor-memory nodes internally linked by two redundant packet switched busses, bus repeaters support a variety of hierarchical structures. Still in the fabrication stage as of this writing, eighty megabytes per second of random access bandwidth is expected per society, augmented by a multiword packet mode which asymptotically provides 160 megabytes per second as packet lengths increase. Although the proof-of-concept implementation provides no caching or hardware support for objects, an upgrade path has been proposed for handling access protection, typing, and cache coherency.

5.2.7.3. VAX - SBI

The VAX Synchronous Backplane Interconnect implemented in the DEC VAX-780 [23] is not intended to support a true multiprocessor, although its protcol closely resembles that used by either of the intra-society RAPIDbus II busses. During 200 nanosecond windows either a word of data or an address is transmitted. When possible, one address cycle initiates an eight

byte memory access packaged as two bus data cycles. RAPIDbus has generalized such octal word accesses into a multiword packet. Proposed RAPIDbus enhancements diverge at the implementation level from the SBI.[60]

The VAX SBI bus arbitration mechanism uses a separate request line for each potential bus master. At the beginning of the next bus cycle, the bus is granted to the interface driving the highest priority line. Routing information carried on the bus window selects the destination.

RAPIDbus II modified this protocol to support a fifty nanosecond bus window. A central arbiter, implemented in higher speed logic than the data path with minimal skew accepts the request lines for a bus, generating grant and access signals respectively to link origin and destination during the following bus cycle. Since the routing decision is made by the arbiter, the each interface need not decode routing information during the bus window or hold all bus cycles for consideration as RAPIDbus I did.

5.2.7.4. University College, London

Researchers at University College, London have designed a multimicroprocessor computer based on time-sliced synchronous buses [22]. Sharing features of both RAPIDbus I and RAPIDbus II, fifty nanosecond windows were allocated on a rotary basis connecting processors and separate bus memory servers. Each processor used a commercial memory management unit for segment level address space translation and protection of global resources, local storage, and local I/O devices.

Memory requests were queued by busy memory units receiving a memory request, minimizing bus bandwidth and retry latency. According to their simulation, FIFO queueing reduces the variance in processor access times, and preserves the sequentiality of access requests. Although there is no indication that such an approach was implemented, dual porting of memory to multiple buses was suggested by the authors in order to create a hierarchical structure.

---

[60] In an enhanced RAPIDbus which was less concerned about optimizing for Versabus style hosts, the SBI practice of sending data as part of a write operation address cycle would be adapted, as shown in figure 5-9.

5.2.7.5. CA2 - Hamburg

Researchers at the Deutsches Elektronen-Synchrotron DESY and Institut fur Mathematik und Datenverarbeitung in der Medizin designed both the CA1 and CA2 multiprocessors for use in image processing. Two asynchronous busses are used, one with a 100 nanosecond window for data, the other with a one microsecond window for instructions. A heterogeneous processor mix was used, supporting both general purpose and dedicated processors.

Interleaved memory was used for image store, accessible in either single word or variable length blocks as required, providing up to 40 megabytes per second across a 66 line bus with separate address and data lines.[61] Asynchronous bus allocation protocols were used to make integration of hosts with different time bases easier.

5.2.7.6. Synapse N + 1

The Synapse N + 1 [32] is a commercial, 68000 based multiprocessor intended for on-line transaction processing. Dual 32 bit "expansion busses" connect up to 28 processors and 16 memory blocks with separate address and data windows. Each processor node can have a cache controller, which monitors both expansion buses to maintain cache coherence.

The overhead required for efficient cache operation led the designers to a sixteen byte transfer block, with a 32 bit address or data window run on each bus every 100 nanoseconds. After accounting for address overhead, this provides a 64 megabyte per second throughput.

RAPIDbus II differs from the Synapse design in several important ways. Caching is supported in the enhanced mode using cache coherence packets which do not depend on each node have access to every transfer cycle. Thus although the RAPIDbus II design is easily extensible to networks having segments with separately allocated bandwidth, additional bandwidth is required to support cache coherence packets. The fixed sixteen byte block size is an implementation tradeoff which may not fit well in applications such as robotics.

---

[61] Up to 80 megabytes per second are supported by a 64 bit data path.

5.2.7.7. APTEC DPS

The APTEC DPS [49] is designed to network up to eight commercial array processors using interleaved shared memory blocks. An internal, synchronous bus transfers up to 24 megabytes per second between microprogrammed DMA controllers.

5.2.7.8. C.MMP

C.MMP, designed by Carnegie-Mellon University in the early seventies around a hardware crossbar switch and sixteen PDP-11 processors, represents a standard multiprocessor data point [64]. Almost four thousand MSI parts were used to create a switch system with a one microsecond access and 27 megabytes per second of throughput.

5.2.7.9. CM*

CM* built upon many of the lessons learned with C.MMP and inherited the Hydra operating system [41]. Composed of a hierarchical tree structure, five clusters of ten LSI-11 microprocessors each were supported. Two intercluster busses were implemented to link each of the five cluster or map busses. CM* made major contributions to multiprocessor literature through the microcode support of an object environment, and the StarOS operating system.

RAPIDbus II differs from CM* in many important ways. Using a wide mix of processor types within a society, RAPIDbus is optimized for research in real-time, numerically intensive application work. Taking advantage of technology improvements, throughput is significantly increased. Problem decomposition is directly under the control of the programmer in contrast to efforts at automatic problem decomposition studied on CM*.

Chapter 6
Conclusions

At the beginning of this project report, a set of architectural goals were set for the RAPIDbus multiprocessor structure. Attaining these goals require contributions at three levels of abstraction; the architecture, the implementation, and the realization. Two machine designs were carried to the gate level in order to provide verification of high level concepts.

6.1. Architecture

The architecture sought to support "a multitude of diverse, heterogeneous tasks grouped into packages with a tight locality of reference". The resultant architecture structure directly responded with a network of processor societies using configuration dependent inter-society links.

The large pool of potentially concurrent tasks coupled with the hypothesized locality of communication led to a two level hierarchy of reference. At the local level, up to fifteen processor nodes are grouped together in a society to collectively execute one or more software packages. At the intersociety level, random society-to-society links can be set up to reflect application data flow patterns. Use of a single, large physical address space distributed throughout the system provided a foundation for such interprocess communication mechanisms as mailboxes and semaphores.

The variety of different code requirements led to the concept of a society of processors complementing each other's capabilities instead of duplicating them. Effective support for such a heterogeneous mix of processors led to strong data typing to allow differing host data representations and the associated conversion logic. Differing data object sizes required a variety of transfer mechanisms ranging from a single byte to an atomic transaction with hundreds or thousands of words accessible using two basic mechanisms (single and multiword transfer).

Effective support for broadly multitasking code led to several different access protection mechanisms. Each implementation supported protection at the segment level. A suggested enhancement of the RAPIDbus II design, *OIL*, created an object based interchange coupled to an operand based processor. *OIL* combined efficient support for large objects required in a signal processing machine with support for objects down to the byte level of granularity. Under the enhanced interchange design, each object required a capability assigned to the task before the object was made accessible. Shared address space relocation was also supported at the same level of granularity as the protection mechanisms in each design.

Garbage collection of dynamically allocated storage was recognized as an important function for hardware to assume, but requiring the additional typing and support of the *OIL* enhancement. Making use of typed pointers and information fields associated with each object, garbage collection was suggested as an upgrade path.

The architecture specification presented a challenge at the implementation level to provide the required functionality and still achieve significant performance increases over existing alternatives. Since the resulting design would only be successful if it was robust enough to be used as a laboratory tool, reliability considerations put additional constraints on the implementation.

6.2. Implementation

The resulting implementation placed each society in a separate cage with fifteen host slots interconnected with two, redundant interchange busses, the ABUS and the DBUS. Repeater hosts were designed to work in pairs, occupying one node in each linked cage, repeating transfer cycles as needed.

By separating the functionality of a data transfer into several different components, the

usage efficiency of each backplane was shown to increase dramatically over existing circuit-switched interconnects such as Versabus. A request-grant paradigm for bus allocation coupled with a fast arbiter minimized the number of unused cycles while a transfer was ready to take place.

By pipelining the routing decision within the bus arbitration mechanism, each temporal window needed only be long enough to allow a line to be driven on the backplane and reliably latched. No address recognition delays were required during the bus window, nor was more than one level of bus latching.

Use of a writable routing table on each host was suggested as a means for rapid reconfiguration of the interchange paths in software in response to a failed repeater or host node. The dual, redundant busses, coupled with potentially redundant inter-society links supported graceful degradation in response to single point bus failures.

A parallel switching plane was suggested as an upward compatibility path for increasing interchange bandwidth while decreasing parts count, and simplifying host/switch cabling. In a system with many host nodes, reliability and cost requirements dictate that techniques such as the parallel switching plane be used to minimize the number of components per node.

The implementation is then critically dependent on the proper functioning of the realization layer. Although little original work was done at the lowest level of abstraction, it required a considerable expenditure of time in order to gain reasonable master of the required electro-magnetic interactions.

6.3. Realization

Experience with the RAPIDbus I realization supported the need to avoid shortcuts when building high speed logic systems. Through proper synchronization of incoming asynchronous signals, generous tolerances for propagation delays, and an adequate power environment, we gained confidence in the design of high speed bipolar logic.

RAPIDbus II made use of the power, speed, density tradeoffs between ECL and advanced TTL to maximize bus performance. The bus timing was controlled by an ECL arbiter with well characterized skew and edges, but low density. The bus made use of dense Advanced TTL transceiver chips to create a fast, cost effective interface.

Comparison with a benchmark Versabus multiprocessor system running a very tightly coupled instruction stream suggested that the RAPIDbus II interconnect structure significantly increased both the bandwidth and the number processors that could be incorporated in a system.

6.4. Trial by Fire

As a project in applied computer engineering, the true test of the RAPIDbus II design lies in its ability to support the intended robotics research environment. As this report is filed, two prototypes are being designed and built. Final conclusions must ultimately be based on the performance of these machines, and their ability to teach us about practical multiprocessor systems for robotics.

References

[1] *NS16032 High Performance Microprocessors*
Preliminary edition, National Semiconductor, 1982.

[2] Intel.
High Performance Microprocessor with Memory Management and Protection
Preliminary edition, 1982.

[3] Dharma P. Agrawal and Ramesh Jain.
A Multiprocessor System for Dynamic Scene Analysis.
In K. S. Fu (editor), *Computer Architecture for Pattern Analysis and Image Database Management*, pages 96-103. IEEE, November, 1981.

[4] Andrew A. Allison.
Status Report on the P896 Backplane Bus.
IEEE Micro 1(1):67-82, February, 1981.

[5] ANSI.
American National Standard Reference Manual for the Ada Programming Language.
ANSI, 1430 Broadway, New York, New York 10018, 1983.

[6] D. H. Ballard and C. M. Brown.
Computer Vision.
Prentice-Hall, 1982.

[7] Thomas Balph.
Implementing High Speed Logic on Printed Circuit Boards.
In Stephen E. Grossman (editor), *Transporting High-Speed Digital Signals Point-to-Point on Circuit Board Assemblies*, pages 18/1/1-18/1/7. Wescon/81, September, 1981.

[8] W. P. Birmingham.
MICON: A Knowledge-Based Single Board Computer Designer.
Master's thesis, Carnegie-Mellon University, June, 1982.

[9] Gerrit A. Blaauw and Frederick A. Brooks.
Computer Architecture.
(in draft), 1981.

[10] William R. Blood Jr.
MECL System Design Handbook.
Motorola Semiconductor Products, 1980.

[11] Paul L. Borrill.
Microproccessor Bus Structures and Standards.
IEEE Micro 1(1):84-95, February, 1981.

[12] B. A. Bowen and R. J. A. Buhr.
The Logical Design of Multiple Microprocessor Systems.
Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

[13] Rafael Bracho and Arthur C. Sanderson.
Design of RIP1: An Image Processor for Robotics.
Technical Report CMU-RI-TR-82-3, Carnegie-Mellon University Robotics Institute, November, 1982.

[14] D. Casasent and D. Psaltis.
Hybrid Processor to Compute Invarient Moments for Pattern Recognition.
Optics Letters 5(9):395-399, September, 1980.

[15] W. Thomas Cathey.
Pure and Applied Optics: Optical Information Processing and Holography.
Wiley InterScience, 1974.

[16] Thomas J. Chaney.
Measuring Flip-Flop Responses to Marginal Triggering.
IEEE Transactions on Computers 32(12):1207-1209, December, 1983.

[17] *CP-32 Processor Manual*
Charles River Data Systems, Natick, Mass., 1983.

[18] J. Christ and A. C. Sanderson.
A Prototype Tactile Sensor Array.
Technical Report CMU-RI-TR-82-14, Carnegie-Mellon Robotics Institute, 1982.

[19] Danny Cohen.
On Holy Wars and a Plea for Peace.
Computer 14(10):48-54, October, 1981.

[20] George R. Couranz and Donald F. Wann.
Theoretical and Experimental Behavior of Synchronizers Operating in the Metastable Region.
IEEE Transactions on Computers C-24(6):604-616, June, 1975.

[21] J. L. Crowley and A. C. Sanderson.
Multiple Resolution and Probabilistic Matching of 2-D Grey-Scale Shape.
In *Proceedings 2nd IEEE Computer Society Workshop on Computer Vision, Representation, and Control*, pages 95-105. IEEE, May, 1984.

[22] O. J. Davies and A. B. Kovaleski.
Feasibility Study of a Modular Multimicroprocessor based Computer Architecture with Time-sliced Synchronous Busses.
In *Microprocessors in Automation and Communications*, pages 19-27. Institution of Electronic and Radio Engineers, January, 1981.

[23] DEC.
VAX Hardware Handbook.
Digital Equipment Corporation, Maynard, Mass., 1980.

[24] J.-D Dessimoz, J. Birk, R. Kelley, J. Hall.
A Vision System with Splitting Bus.
In K. S. Fu (editor), *Computer Architecture for Pattern Analysis and Image Database Management*, pages 62-66. IEEE, November, 1981.

[25] Thad Dreher.
Cabling fast pulses? Don't trip on the steps.
The Electronic Engineer :71-75, August, 1969.

[26] Philip H. Enslow.
Multiprocessor Organization - A Survey.
Computing Survey 9(1):103-129, March, 1977.

[27] John A. Falco.
Reflection and crosstalk in logic interconnections.
IEEE Spectrun :44-50, July, 1970.

[28] *FAST Logic Data Book*
2nd edition, Fairchild Digital Products, South Portland, Maine, 1982.

[29] U.S. NIM Committee.
FASTBUS: Modular High Speed Data Acquisition System
First edition, 1982.

[30] A. Fisher, H. T. Kung, K. Oflazer, M. K. Ravishankar, S. Yu.
Architecture of the Universal Host: Preliminary Progress Report.
1982.
Department of Computer Science, Carnegie-Mellon, Internal Report.

[31] Werner Fleischammer and Osman Dortok.
The Anomalous Behavior of Flip-Flops in Synchronizer Circuits.
IEEE Transactions on Computers C-28(3):273-276, March, 1979.

[32] Steven J. Frank.
Tightly Coupled Multiprocessor System Speeds Memory-Access Times.
Electronics 57(1):164-169, January, 1984.

[33] Wolfgang K. Giloi et al.
Fachbereich 20 - Informatik: Realizing Innovative Multicomputer Architectures With Off-The-Shelf VLSI Components.
Technische Universitat Berlin, 1000 Berlin 12, Strafe des 17 Juni 135, 1982.

[34] D. Giuse, D. P. Siewiorek, and W. P. Birmingham.
DEMETER: A Design Methodology and Environment.
Proceedings of the IEEE International Conference on Computer Design/VLSI in Computers , 1983.

[35] Stan Groves.
The Inter-Relationship Between Access Time and Clock Rate in an MC68000 System.
Engineering Bulletin EB-83, Motorola Semiconductor Products Inc., March, 1980.

[36] Jean-Loup Baer (editor).
The Impact of Classification Schemes on Computer Architecture.
IEEE Computer Society, 1977.

[37] Segall et al.
Position Papers - Workshop on Multiprocessors for High Performance Parallel Processing.
1983.
Seven Springs, Penn. June 27-29.

[38] IEEE P754 Committee.
A Proposed Standard for Binary Floating Point Arithmatic.
IEEE Draft.

[39] Intel Corporation.
Intel 432 System Summary: Manager's Perspective.
Intel Corporation, 1981.

[40] Caulfield (editor).
Tenth International Optical Computing Conference.
IEEE Computer Society, 1983.

[41] A. Jones, E. Gehringer.
The CM Multiprocessor Project: A Research Review.*
Technical Report CMU-CS-80-131, Carnegie-Mellon University Computer Science Department, 1980.

[42] Hubert Kirrmann.
Data Format and Bus Compatibility in Microprocessors.
IEEE Micro 3(4):32-47, August, 1983.

[43] James S. Kolodzey.
Cray-1 Computer Technology.
IEEE Transactions on Components, Hybrids, and Manufacturing Technology CHMT-4(2):181-186, June, 1981.

[44] George D. Kraft and Wing N. Toy.
Microprogrammed Control and Reliable Design of Small Computers.
Prentice-Hall, 1981.

[45] H. T. Kung.
Why Systolic Architectures ?
Computer :37-46, January, 1982.

[46] *Versabus Specification Manual*
Third edition, Motorola MicroSystems, Phoenix, Arizona, 1981.

[47] Madhav V. Marathe.
Performance Evaluation at the Hardware Architecture Level and the Operating System Kernal Design Level.
PhD thesis, Computer Science Department, Carnegie-Mellon, December, 1977.

[48] *MC68000 16-Bit Microprocessor User's Manual*
Third edition, Motorola Semiconductor Products, 1982.

[49] Gary L. McAlpine.
Dimensional Processing System: A Controller for Multi-Processor Architectures.
In *Peripheral Array Processor Conference*, pages 140-156. IEEE, October, 1982.

[50] David M. McKeown and John McDermott.
Toward Expert Systems for Photo Interpretation.
In *IEEE Proceedings of Trends and Applications*, pages 33-39. IEEE Computer Society, 1983.

[51] *MTOS-68K System Kernal Manual*
Second edition, Industrial Programming Incorporated, Jericho, New York, 1982.

[52] Jerre D. Noe and Gary J. Nutt.
Macro Enets for Representation of Parallel Systems.
IEEE Transactions on Computers C-22(8):718-727, August, 1973.

[53] Graham R. Nudd.
Image Understanding Architectures.
AFIPS :377-390, 1980.

[54] G. J. Nutt.
Evaluation Nets for Computer System Performance Analysis.
AFIPS Fall Conference Proceedings 42(1):279-286, December, 1972.

[55] Elliott Organick.
The Multics System.
MIT Press, Boston, Mass., 1980.

[56] L. Pichon and J. P. Huignard.
Dynamic Joint-Fourier-Transform Correlator by Bragg Diffraction in Photoreflective $Bi_{12}SiO_{20}$.
Optical Communications 36(4):277-280, February, 1981.

[57] A. V. Pohm and O. P. Agrawal.
High-Speed Memory Systems.
Reston Publishing, Reston, Virginia, 1983.

[58] Lawrence R. Rabiner.
Theory and Application of Digital Signal Processing.
Prentice-Hall, 1975.

[59] Chuck Rieger.
ZMOB: Doing it in Parallel!
In K. S. Fu (editor), *Computer Architecture for Pattern Analysis and Image Database Management*, pages 119-124. IEEE, Computer Society, November, 1981.

[60] A. C. Sanderson, and L. E. Weiss.
Adaptive Visual Servo Control of Robots.
In *Proc. 26th SPIE International Symposium*. SPIE, 1982.

[61] A. C. Sanderson and G. Perry.
Sensor-Based Robotic Assembly Systems: Research and Applications in Electronic Manufacturing.
Proceedings of IEE 71(7):856-871, July, 1983.

[62] A. C. Sanderson.
Robot Vision and Industrial Automation.
Industrial Applications of Image Analysis.
D.E.B. Publishers, Pijnacker, The Netherlands, 1983, pages 31-49.

[63] Thomas McWilliams, Lawrence Widdoes, and Lowell Wood.
Advanced Digital Processor Technology Base Development for Navy Applications: The S-1 Project (SCALD Section)
Lawrence Livermore Laboratory, Livermore, California, 1978.

[64] Daniel Siewiorek, C. Gordon Bell, Allen Newell.
Computer Science: Computer Structures, Principles and Examples.
McGraw Hill, 1982.

[65] Daniel P. Siewiorek and Robert P. Swarz.
The Theory and Practice of Reliable System Design.
Digital Press, 1982.

[66] D. P. Siewiorek, D. Giuse, and W. P. Birmingham.
Proposal for Research on Demeter - A Design Methodology and Environment.
Technical Report, Carnegie-Mellon University, January, 1983.

[67] Kevin Smith.
Wideband Local Net Allocates Bandwidth.
Electronics :76-80, February, 1984.

[68] Alan Snyder.
A Machine Architecture to Support an Object-Oriented Language.
Technical Report 209, Computer Science Laboratory- MIT, 1979.

[69] Bjorn Solberg, Oddvar Sorasen, Steinar Forsmo.
A Very Fast Packet-Switched Bus SYstem Based on Two Custom NMOS Chips.
VLSI '83 :295-304, 1983.

[70]
MULTIBUS II Bus Architecture Will Carry Designers to the 1990's.
Solutions Magazine.

[71] Robert K. Southard.
Interconnection System Approaches for Minimizing Data Transmission Problems.
Computer Design :107-116, March, 1981.

[72] Peter A. Stoll.
How to Avoid Synchronization Problems.
VLSI Design :56-58, November/December, 1982.

[73] Richard J. Swan.
The Switching Structure and Addressing of an Extensible Multiprocessor: Cm.*
PhD thesis, Carnegie-Mellon, 1978.

[74] J. M. Tenenbaum, H. G. Barrow, R. C. Bolles, M. A. Fischler, H. C. Wolf.
Map-Guided Interpretation of Remotely-Sensed Imagery.
AFIPS :391-408, 1980.

[75] Harry J. M. Veendrick.
The Behavior of Flip-Flops Used as Synchronizers and Prediction of their Failure Rate.
IEEE Journal of Solid-State Circuits SC-15(2):169-176, April, 1980.

[76] Mike Williams and Stuart Miller.
Series 54ALS/74ALS Schottky TTL Applications
B215 edition, Texas Instruments, 1982.

[77] John C. Willis and Arthur C. Sanderson.
RAPIDbus: Architecture and Realization.
Technical Report 82-13, Carnegie-Mellon University Robotics Institute, 1982.

[78] Patrick H. Winston and Richard H. Brown.
Artificial Intelligence: An MIT Perspective.
MIT Press, 1982.

[79] R. Winter.
An Evaluation Net Model for the Performance Evaluation of a Computer Network.
In H. Beilner and E. Gelenbe (editor), *Measuring, Modelling, and Evaluating Computer Systems*, pages 95-113. North-Holland Publishing Company, 1977.

[80] William A. Wulf and Samual A. Harbison.
Reflections in a Pool of Processors.
Technical Report CMU-CS-78-103, Carnegie-Mellon Computer Science Department, February, 1978.

[81] Mario P. Zoccoli and Arthur C. Sanderson.
Rapid bus Multiprocessor System.
Computer Design :189-200, November, 1981.