Chapter 23

Pluribus—An Operational Fault-Tolerant Multiprocessor[1]

David Katsuki / Eric S. Elsam / William F. Mann
Eric S. Roberts / John G. Robinson
F. Stanley Skowronski / Eric W. Wolf

Summary The authors describe the Pluribus multiprocessor system, outline several techniques used to achieve fault-tolerance, describe their field experience to date, and mention some potential applications. The Pluribus system places the major responsibility for recovery from failures on the software. Failing hardware modules are removed from the system, spare modules are substituted where available, and appropriate initialization is performed. In applications where the goal is maximum availability rather than totally fault-free operation, this approach represents a considerable savings in complexity and cost over traditional implementations. The software-based reliability approach has been extended to provide error-handling and recovery mechanisms for the system software structures as well. A number of Pluribus systems have been built and are currently in operation. Experience with these systems has given us confidence in their performance and mantainability, and leads us to suggest other applications that might benefit from this approach.

I. Introduction

The multiprocessor discussed in this paper had its beginnings in 1972 when the need for a second-generation interface message processor (IMP) [Heart et al., 1970] for the ARPA network (ARPANET) [Roberts and Wessler, 1970; Heart, 1975; Wolf, 1973] became apparent. At that time, the IMP's Bolt Beranek and Newman (BBN) had already installed at more than thirty-five ARPANET sites were Honeywell 316 and 516 minicomputers. The network was growing rapidly in several dimensions: number of nodes, hosts, and terminals; volume of traffic; and geographic coverage (including plans, now realized, for satellite extensions to Europe and Hawaii). A goal was established to design a modular machine which, at its lower end, would be smaller and less expensive than the 316's and 516's while being expandable in capacity to provide ten times the bandwidth of, and capable of servicing five times as many input–output (I/O) devices as, the 516 [Heart et al., 1973]. Related goals included greater memory addressing capability and increased reliability.

We decided on a multiprocessor approach because of its promising potential for modularity, for cost per performance advantages, for reliability, and because the IMP algorithm was clearly suitable for parallel processing by independent processors.

The IMP's communicate with host computers and with asynchronous terminals (IMP's with terminals attached are called TIP's [Ornstein et al., 1972]). Hosts use the network of IMP's and lines to communicate data messages of up to about 8000 bits; the IMP's divide these messages into packets up to about 1000 bits long. The functions performed by the IMP are those of a communications processor; they include storing and forwarding packets, generating headers, routing, retransmission, error checking, packet and message acknowledgment, message assembly and sequencing, flow control, line error detection, host and line status monitoring, and related housekeeping functions. The IMP's also send status and performance data to a network control center (NCC) which monitors and controls network operations [McKenzie et al., 1972; Ornstein and Walden, 1975]. The ARPANET IMP's operate 24 hours a day, often in unattended locations.

In applications of this sort, reliability requirements differ from those commonly found in other real-time systems. The IMP network forms only a part of a larger system; even a perfectly operating network is not sufficient to guarantee perfect overall system performance. Failures in the host, or in the interface between the host and IMP, may still introduce errors. What this means is that some sort of host-process to host-process error control is required for critical applications; the best that the IMP network can provide is a good environment for host-level error recovery processes. These processes need a network which rarely makes errors and which, when such errors do occur, can effectively process host-to-host retransmissions. In other words, occasional dropped messages and brief outages are acceptable; outages of more than a few minutes are undesirable even if scheduled in advance.

Once we realized that what was needed was not so much reliability as the ability to recover gracefully from failures, we began to see ways to provide a much more robust network by coding this type of fault-tolerance into our operating system and application algorithms, and by including special modular hardware designs. The machine that emerged [Heart et al., 1973; Ornstein and Walden, 1975; Bressler, Kraley, and Michel, 1975; Ornstein et al., 1975; Heart et al., 1976] we call the Pluribus (Fig. 1 shows a typical Pluribus installation). It provides simple checking procedures such as parity, amputation features which allow failing equipment to be isolated and, optionally, redundant components. The software uses these features to detect, report, and isolate hardware failures. Since the symptoms of many subtle software failures are similar to those of intermittent hardware errors, fault-tolerant procedures which adequately recover from one can also recover from the other.

There is a spectrum of fault-tolerant approaches which are appropriate in various applications [Avizienis, 1976; Avizienis, 1975]; our approach opts for a relatively inexpensive system which

[1]*Proc. IEEE*, vol. 66, no. 10, October 1978, pp. 1,146–1,159.

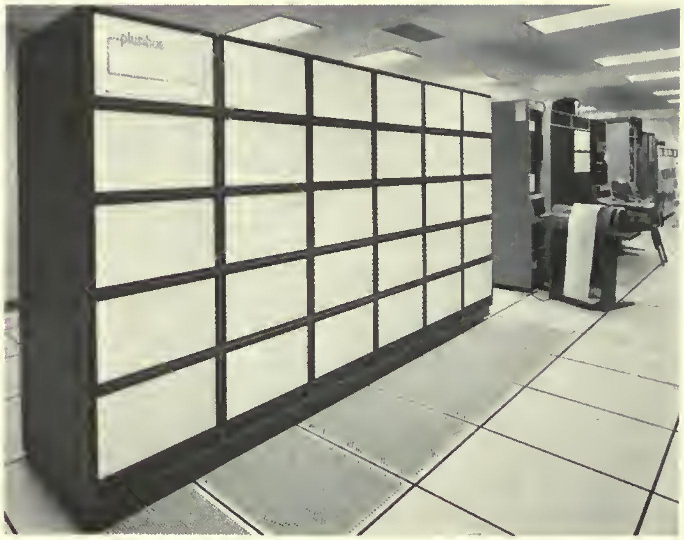

Fig. 1. The Pluribus front-end processor at Bolt Beranek and Newman's Research Computer Center.

can quickly reinitialize itself, omitting troublesome components. This approach is especially suitable for applications in which brief outages can be tolerated and where overall correctness can be ensured by other techniques.

II. Pluribus System Architecture

The Pluribus may be characterized as a symmetric, tighlty coupled multiprocessor, designed to be flexible and highly modular. Modules are physically isolated to protect against common failures, and a form of distributed switch is employed for intermodule communications. In this section, we discuss these characteristics and describe the hardware architecture of the Pluribus.

A. Major Design Decisions

In order to make the basic operation of the Pluribus clearer, it is useful to examine some of the major design decisions that have directed its development, and to consider those decisions in the context of other options for multiprocessor system design. We have identified three areas which we believe are key aspects of the Pluribus approach to multiprocessing, each of which is considered in greater detail below.

Processor Symmetry One dimension of multiprocessing involves the degree of inter-processor symmetry within the system [Enslow, 1974, p. 83]. In this dimension, one extreme might be a typical general purpose computer system, including a central processor, a front-end processor, and perhaps one or more channel processors. Such an asymmetric system is relatively inflexible in power since increasing its central processing capacity requires the introduction of a more powerful central processor. Building redundancy into an asymmetric system can be expensive, since replication of all critical resources involves duplicating virtually the whole machine.

At the other extreme are systems like the Pluribus in which all processors are identical. In such systems, the advantages of redundancy and flexibility are much easier to achieve since they include only one type of processing unit. Even without explicit redundancy, a symmetric system can provide graceful degradation of throughput when a processing element fails. Pluribus systems which are sized for fully redundant operation include just one extra processing module; thus the degradation which results from failure of any processing module consists only of a loss of excess throughput capacity.

Processor Coupling Another multiprocessing dimension is the level at which processors cooperate to accomplish overall system requirements. At one extreme the processors might run totally separate programs under the direction of a supervisor program, communicating only at arm's length. Such processors may be described as "loosely coupled" [Enslow, 1974, p. 15]. At the other extreme, which is characterized by array processors such as ILLIAC IV [Barnes et al., 1968], the processors run in lockstep, with a single program operating simultaneously on a number of data streams. The Pluribus lies between these extremes. Its processors are tightly coupled in the sense that all processors can access all system resources and perform all parts of the operational program; they operate independently except for necessary software interlocks on specific I/O devices and data structures.

Flexibility Although one of the goals in the creation of the Pluribus was to develop a machine with high throughput, this goal was complemented by the need for a smaller, cheaper machine with relatively low throughput. Similarly, although the Pluribus was conceived as having at least two of every resource to permit recovery after failures, it was also clear that not all applications required or could afford a fully redundant system. Thus it was desirable for the architecture to be flexible in at least two ways: The size-flexibility goal was to smooth large incremental steps in the cost-performance curve by utilizing a highly modular design, which could provide processing capacity well beyond our anticipated needs. Flexibility in the area of fault-tolerance and fault-recovery was a related goal, since the need for fault-tolerance involves primarily economic considerations and we wanted to allow our customers to select fault-tolerance features independent of their throughput requirements. Also implied in each of these goals was the requirement for easy expansion to meet changing requirements.

B. System Overview

A central requirement in any multiprocessor is that processing elements be able to communicate both among themselves and with shared resources such as memories and I/O equipment. Ease of communication is always desirable and is vital in tightly coupled systems, since any delays or unwieldiness would immediately impact system operation and reduce programmability. These considerations, together with a natural desire for symmetry and simplicity, led us to adopt a unified addressing structure in which all common memory and I/O devices share the same address space. The Pluribus development was strongly influenced by previous unified-bus architectures in which processing, memory, and I/O units share not only a common address structure but also a single, time-multiplexed bus (the DEC PDP-11 is perhaps the most familiar example of this). Although multiprocessors based on the unified bus are both easily extensible and conceptually simple structures, they are vulnerable to single failures anywhere along the bus. In addition, the maximum throughput of such multiprocessors is limited both by the design bandwidth of the bus as well as by contention for common resources. To avoid these problems we used a unified bus to create the functional modules which make up the system, but not to form the main connection structure. We defined three basic functional modules which share a common address space but have separate intermodule communications paths: processor *buses*, memory *buses*, and I/O *buses*. A simplified system diagram is shown in Fig. 2.

(In the following sections we will often use the term *bus* to mean a logical and physical module, as in "processor *bus*," rather than just an interconnection system. All such usages will be italicized for clarity.)

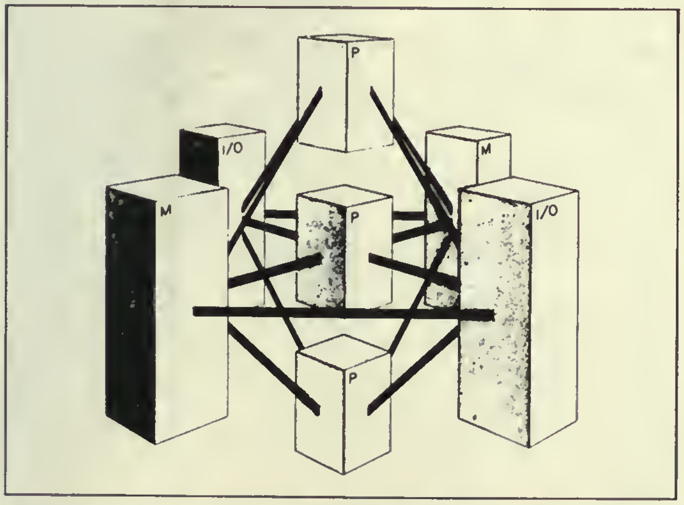

Fig. 2. A simplified view of the functional modules in a typical Pluribus system showing their interconnectivity. No physical relationships are implied.

The system for interconnecting these modules had several major requirements. It had to be easily extensible to support as many as eight memory or I/O *buses* (common *buses*) and eight or more processor *buses*. It had to permit the operating software to remove malfunctioning modules from the system and incorporate newly acquired or repaired modules. In addition, it had to impose minimal cost penalties for smaller systems, while scaling up smoothly to produce large systems. Finally, it had to have no common point of failure which could lead to total system failure.

The approach we finally adopted is similar in function to a central crossbar switch although it differs greatly in implementation. The crossbar switch approach allows an extremely high-bandwidth interconnection scheme and has been used to advantage in several multiprocessors [Wulf and Bell, 1972]. However, the usual implementation techniques are vulnerable to single-point failures. To avoid these problems, we distributed the components of the switch among the various system modules in such a way that no single failure points remain. Switch elements are called bus couplers and consist of two circuit boards connected by a cable.

The bus couplers function by recognizing a range of addresses on processor or I/O *buses*, and initiating an access request on the appropriate common *bus* as a result. Since memory and I/O *buses* share a 20-bit address space, bus couplers must map 16-bit processor addresses into 20-bit system addresses under program control (see Fig. 3). In addition to handling inter-*bus* communications, bus couplers perform several other functions which will be described later.

Modularity Since the basic Pluribus was modular at several levels, an unusual degree of flexibility was available when we set out to define standard structures within the system. The three basic system modules described above have clear logical functions within the system, but their actual implementation depended on various tradeoffs between cost, throughput, and available physical components.

It was decided early that the goals of flexibility and symmetry could be achieved by segmenting the operational tasks into strips of code (task distribution routines, task-oriented application routines, timers, etc.) which could be run by any available processor. The concept was that the code should be both reentrant and accessible to all processors at all times. The primary function of the common memory modules is to provide space for data buffers, program work areas, and inter-processor communications areas. Code storage is divided into two parts: lightly used code is stored on common memory *buses* and is shared between processors; heavily used code is replicated in local memory on each processor *bus*. This strategy minimizes contention for access to common memory while holding down costs, especially since, in

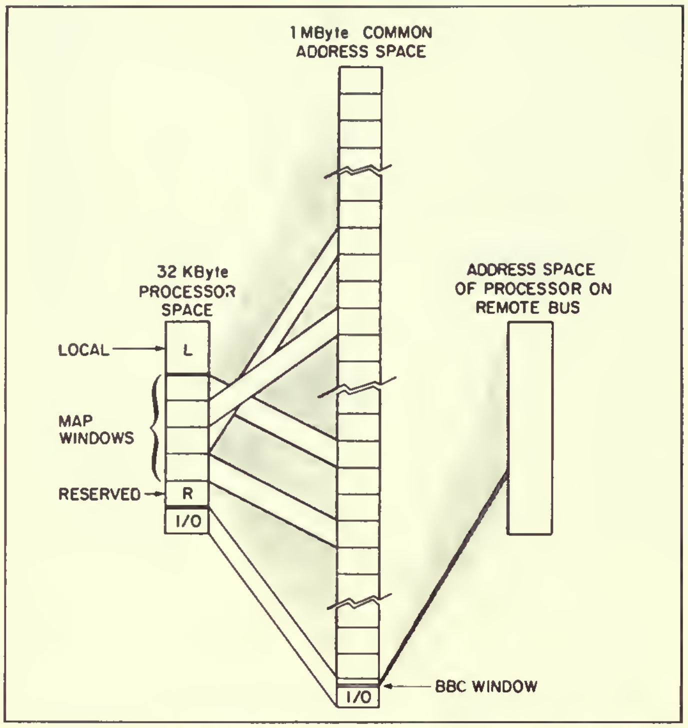
1 MByte COMMON ADDRESS SPACE
32 KByte PROCESSOR SPACE
ADDRESS SPACE OF PROCESSOR ON REMOTE BUS
LOCAL
L
MAP WINDOWS
RESERVED
R
I/O
I/O
BBC WINDOW

Fig. 3. Pluribus system address space, showing the mapping of processor "local" address space into the system space. "Backwards bus coupling" path from one processor bus through an I/O *bus* to another *processor bus* is shown on the right.

most applications, only a small part of the code is heavily used. The I/O modules were intended to support both polled low-speed I/O devices and high-speed interfaces capable of direct memory transfers. Couplers provide direct paths both from processor *buses* to I/O *buses* for control and polling, and from I/O *buses* to memory *buses* for direct memory transfers.

All normal processor-to-processor communication occurs through locations in common memory. However, to initialize the system, it must be possible for one processor to access the local memory and control registers of a processor on a different *bus*. To allow this, the bus couplers provide a limited reverse path through any common I/O *bus*.

In the following sections, we describe the physical implementation of these system modules and detail several support functions required by the architecture.

C. Physical System Structure

As mentioned in previous papers [Heart et al., 1973; Ornstein and Walden, 1975], we chose the Lockheed SUE minicomputer as the point of departure for our system. It is a 16-bit machine, generally similar to the DEC PDP-11, which incorporates a unified address structure and an asynchronous, time-multiplexed bus. It also permits the attachment of a flexible combination of processors, memory, and I/O units. In contrast to the PDP-11, the SUE has its bus arbitration logic physically separated from the processor. This feature permits a bus to have one or several processors, or none at all. The Pluribus uses the bus, arbitration logic, processors, memories, and several minor I/O units of the SUE.

The basic Pluribus building block is the *bus* module. This module contains a modified SUE bus and card cage for up to twenty-four cards, together with completely self-contained cooling fans and power supply. Two *bus* modules can be connected to form an extended *bus*. A Pluribus system rack contains up to five *bus* modules, and each rack is typically supplied with a separate source of ac power. Systems sized to be fully redundant allow any *bus* module or any rack to be powered down for maintenance without affecting system availability (see Fig. 4).

Bus Structure (See Fig. 5) A processor *bus* contains one or two processors and their associated local memory, a bus arbiter, and one bus coupler per logical path. Our current applications require 8 to 12K words of local memory for each processor. The flexibility

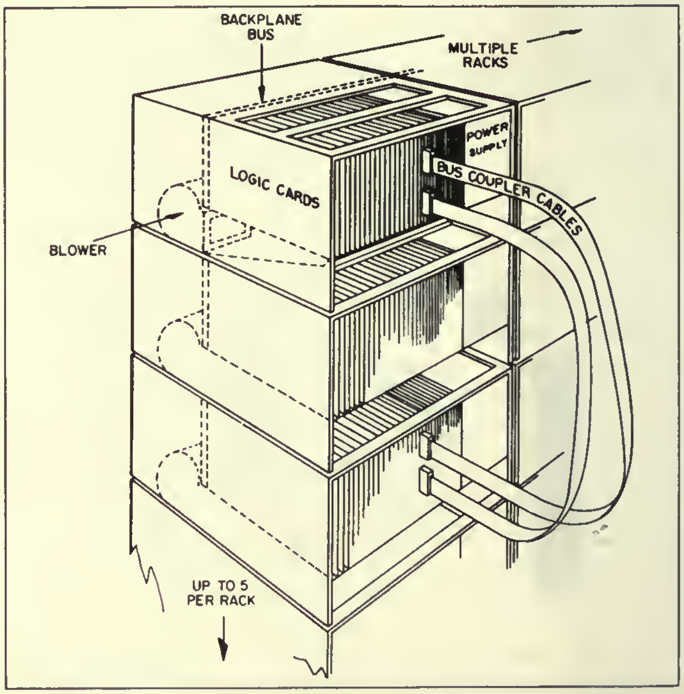

Fig. 4. Physical organization of bus modules. Modules are independently supplied with power and cooling.

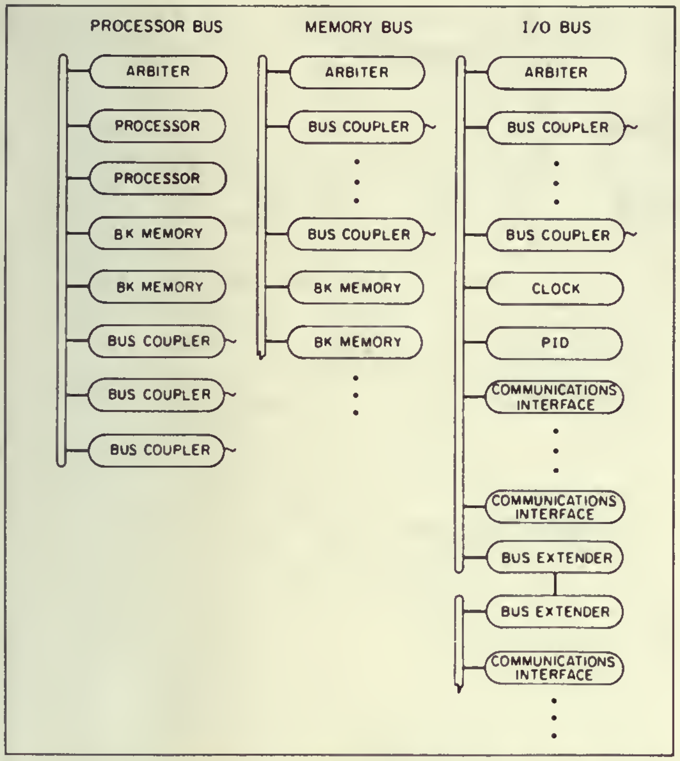

Fig. 5. Local bussing structure and contents of the three kinds of bus modules.

of the processor *bus* allows us to easily vary this parameter as memory prices or the requirements of the application change.

The common memory *bus* contains an arbiter, bus coupler cards for all the connected paths, and enough memory modules to support the application. Up to 512K words of common memory can be supported in a system, although that amount of memory would probably not be concentrated on one memory *bus*. Typical Pluribus systems have from 32K to 80K words of memory on each *bus*, depending on the application.

In addition to the bus arbiter and bus coupler cards, an I/O *bus* also contains cards for each of the various types of I/O interfaces that are required, including interfaces for modems, terminals, host computers, etc., as well as interfaces for standard peripherals. The I/O *bus* also houses a number of special units including (1) a real-time clock (RTC) which is used by the system for timing processes and communications links (2) a special hardware task disbursing unit known as the pseudo-interrupt device (PID) discussed further below and (3) a reload card which monitors up to eight communication lines, watching for (and processing) specially formatted reload messages from the outside world.

Inter-Bus Connection System Since all processors in our system must be able to perform any system task, *buses* are connected so that all processors can access all shared memory and control the operation and sense the status of any I/O unit (see Figs. 2 and 6).

To connect processors and common memory, one card of a bus coupler is installed on a common memory *bus*, and the other on a processor *bus*. Similar connections are made from every processor *bus* to every common I/O *bus*. Coupler cards are connected by cables which may be up to 30 ft long, although most systems require a maximum of 10 ft.

The memory or I/O end of a bus coupler contains address-recognition circuitry and may be strapped to recognize and pass on to the memories or I/O devices any desired address range. When a processor makes a reference to common memory or I/O *buses*, the bus coupler cards on the processor *bus* all map the 16-bit address on the processor *bus* into a 20-bit system address and pass it to bus couplers at the other ends of the connecting cables. If the address is within the recognition range of a memory or I/O end bus coupler, it will request a service cycle on its *bus*. Data from the selected memory cell or device register are then passed back along the coupler path to the processor. This feature differentiates the system address space so that requests for memory or I/O *bus* access only cause service cycles on appropriate *buses*, thereby avoiding unnecessary contention.

Given a bus coupler connecting each processor *bus* to each common memory *bus*, all processors can access all common memory; I/O devices which do direct memory transfers must also access the common memories. These I/O devices are attached to as many I/O *buses* as are required to physically accommodate the number of devices and allow redundancy if necessary. Couplers connect each I/O *bus* to each memory *bus*. This coupler path is much like the processor-to-memory coupler path except that no address mapping needs to be done. I/O devices must respond to processor requests for action or information and in this respect the I/O devices act like memories. Bus couplers are also used to connect each processor *bus* to each I/O *bus*. Here also, a mapping must be done between the 16-bit processor address space and the 20-bit system space (see Fig. 3).

Processor *buses* need to access each other in order to start and stop each other and reload local memories. We provide this low bandwidth interconnection by allowing a processor to access another processor *bus* via its processor-to-I/O bus coupler. The coupler provides a small (4-word) mapping window from I/O space to each processor's space. A processor accesses another processor on a different *bus* by setting up and referencing this "backwards bus-coupling" window in system I/O space.

The coupler paths that connect processor *buses* into memory and I/O *buses* have program-settable enabling switches at their far (memory and I/O) ends, thus permitting processors to be cut into and out of ("amputated" from) the system. The reverse paths in the processor-to-I/O couplers also have enabling switches; nor-

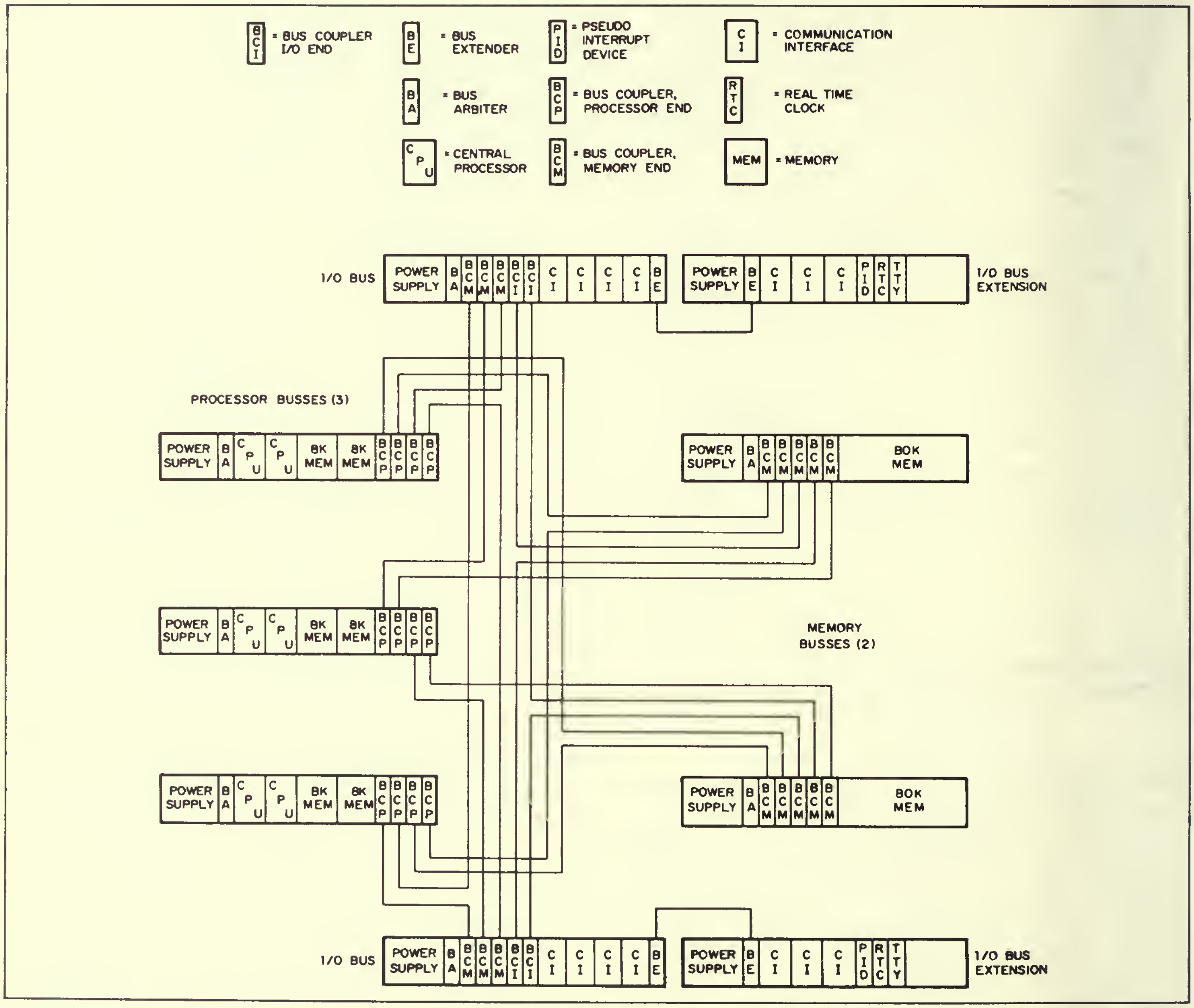

Fig. 6. Logical organization of a typical Pluribus system, showing interconnections of the distributed switch (bus coupler) structure.

mally the forward paths are turned on and the backwards paths are shut off. Since these paths represent a hazard whereby a "sick" processor or device could damage the system, we have arranged that only by storing a password at the proper address can a switch be changed. A processor can neither enable or disable its own access paths but one processor, deciding that another is sick and should be eliminated from the system, can amputate the *bus* of the offending processor. Reinstatement of an amputated *bus* happens in a similar manner.

Parity To aid in detecting faulty bus couplers or defective memory, we compute and check parity across all bus coupler paths using a parity computation based on both data and address [U.S. Pat., 1977]. The scheme detects both "all zeros" and "all ones" failures. For writes to common memory, parity is computed at the processor or I/O end of the bus coupler and stored in the memory cell with the data. When the memory cell is read, the stored parity is checked at the processor or I/O end of the bus coupler. For accesses from processors to units on the I/O *buses* we use "feedback" parity; for writes to I/O the parity is computed by a special card on the I/O *bus*. The parity is then sent back up the coupler to the processor *bus* where it is compared with parity computed on that *bus*. For reads from I/O the special I/O parity card computes parity and compares it with recomputed parity on the processor *bus*.

Pseudo-Interrupt Device Real-time systems or, more generally, systems requiring fast response, employ priority interrupt mechanisms to direct the attention of the processor to the most urgent tasks. Reliability and load sharing requirements make it desirable that any processor be able to service any I/O device, but also raise such questions as which processor to interrupt for servicing. We have opted for a simple yet flexible method: each "interrupt event" (DMA completion, RTC tick, software events, etc.), instead of actually interrupting a processor, writes a value associated with its priority to a hardware queuing device called the PID. The software is designed to allow each processor to put aside the context of its present computation periodically and check the PID. The PID, upon being read, will produce the highest value that has been stored in it and simultaneously delete that value from its internal queue. The processor can then use that value as an index to a table of tasks to be performed. The software uses the PID in a similar manner: each time a "strip" of code completes, it writes the number of the next strip in that task to the PID. When that becomes the highest number in the PID, the next available processor will execute the associated strip.

Our system does have two traditional interrupts, however. One is a 60-Hz clock interrupt. Each bus has its own 60-Hz clock, but conceptually this is an interrupt going to all processors; its main function is to time out locked data structures. The other classical interrupt is the power-fail/power-restore interrupt; each processor handles a power-fail interrupt from its own *bus* in the traditional way. Furthermore, bus couplers connected to processor *buses* will pass on any power-fail interrupt detected at their memory or I/O ends. A restoration of power causes first a *bus* master-reset and then a processor interrupt. We have adapted this interrupt mechanism to serve also as a *bus* activity watchdog timer. If any *bus* fails to show access activity for one second, a hardware timer fires, causing an artificial power-restore reset and interrupt. This provides recovery from some illegal hardware and software states.

D. Redundancy

To assure that a particular machine has enough redundant resources to allow survival in the face of component failures, we include at least one extra *bus* of each type so that a failure of any one resource, or the *bus* holding that resource, will not result in system failure. This approach also permits the system to survive many combinations of multiple failures. Thus if a system requires four processors to function at minimum acceptable throughput, six processors would be provided for reliability since the failure of any processor *bus* would disable two processors. Similarly, if a machine required at least 60K of memory to function, we would provide two *buses* each containing 60K of memory, or three *buses* each containing 30K of memory. It is important to note that redundant resources configured into a given machine are not idly standing by since they are used by the running machine to produce performance greater than the acceptable minimum.

I/O ports pose a special problem, since the devices and lines to which they are connected are frequently not doubled. For reliability, I/O interfaces can be doubled on separate I/O *buses*, but both interfaces must usually drive a single cable leaving the machine. We allow this by constructing all of our I/O port drivers with circuits that present a high impedance while unpowered. In addition, each I/O interface has a watchdog timer which, if not held off by repeated processor accesses, will disconnect the driver circuits within a second. Thus the likelihood that malfunctioning or unpowered I/O interfaces will interfere with the signals put on the external cable by the backup I/O interface is kept to a minimum.

III. The Pluribus Operating System[1]

Unlike most conventional systems, the principal responsibility for maintaining reliability in the Pluribus is placed on the system software rather than in the hardware structure. The Pluribus hardware was designed to provide an appropriate vehicle for software reliability mechanisms. Besides normal error checking and reporting in the hardware itself, programmed tests, using known data patterns are run at intervals. When hardware errors are detected, system software exploits the redundancy of the hardware by forming a new logical system configuration which excludes the failing resource, using redundant counterparts in its place.

Pluribus systems also check the validity of their software structures. Redundant information is intentionally introduced into the data structures at various points and checked by processes operating upon those structures. An example of this technique applied to buffer structures is described in Sec. IV. In addition, periodic background processes are used to recompute certain

[1]Portions of Secs. IV, V, and VII of this paper have appeared in "Software Fault-Tolerance in the Pluribus," J. G. Robinson and E. S. Roberts, *AFIPS Conference Proceedings*, vol. 47, copyright AFIPS Press, Montvale, NJ. Reproduced with permission.

variables which are maintained by the operational system. If the recomputation uncovers a discrepancy, the variables are fixed directly or a more drastic recovery procedure is initiated.

In many cases, a failure is not detected at the exact time of occurrence but later when the software encounters some failure-induced discrepancy. By this time, the effects of the failure may be more widespread and the actual cause of the failure may be difficult to determine. In such cases, the system is not able to perform instantaneous recovery and seeks instead to restore normal operation as quickly as possible.

The remainder of this section discusses the organization of the Pluribus operating system and some of the techniques used for achieving coordination of multiple processors. These techniques are further explored below where two examples of Pluribus fault-tolerant software strategies are presented. One of these examines the Pluribus IMP buffer system in detail, and the other covers strategies for understanding failures when they occur and effecting necessary repairs.

A. General Responsibility of the Operating System

The software reliability mechanisms for a Pluribus system are coordinated by a small operating system (called STAGE) which performs the management of the system configuration and the recovery functions. The overall goal of the operating system is to maintain a reliable, current map of the available hardware and software resources. The map must include accurate information not only about the hardware structure of the machine, but also about variables and data structures associated with the processes that use that hardware. Moreover, the operating system must function correctly even after parts of the system hardware have ceased to be operational. New resources, as they are discovered (e.g., because hardware has been added or repaired), should be incorporated as part of the ongoing operation of the application system.

Since any component of the system may fail at any time, the operating system must monitor its own behavior as well as that of the application system. It may not assume that any element of hardware or software is working properly—each must be tested before it is used and retested periodically to ensure that it continues to function correctly. The operating system must be skeptical of its current picture of the system configuration and continually check to see if the environment has changed.

Based on these considerations, the Pluribus operating system builds the map of its environment step by step. Each step tests and certifies the proper operation of some aspect of the environment, relying on those resources certified by previous steps as primitives. Early steps examine the operation of the local processor and its associated private resources. Subsequent steps look outward and begin to discover and test more global resources of the system, giving the checking process a layered appearance. In the Pluribus operating system, each processor begins by checking its own operation and by finding a clock for use as a time base. Once these resources have been verified, the processor can begin to coordinate with the other active processors to develop an accurate picture of the system.

At the same time, the system must balance the need for reliable primitives with the need to accomplish normal operation efficiently. When all the environment has been certified, the system should spend most of its processing power on advancing the operational algorithms and return only occasionally to the task of reverifying its primitives. When failures of the environment have been detected, however, the power of the system must be brought to bear on the task of reconfiguring to isolate the failure.

B. Hierarchical Structure of the STAGE System

The Pluribus operating system is organized as a sequence of stages which are polled by a central dispatcher. A processor starts with only the first stage enabled. As each stage succeeds in establishing a proper map of its segment of the system state, it enables the next stage to run. Each stage may use information guaranteed by earlier stages and thus may run only if the previous stage has successfully completed its checks. Once enabled, a stage will be polled periodically to verify that the conditions for successful completion of that stage continue to apply. The system applies most of its processing power to the last stage that is enabled but returns periodically to poll each earlier stage. The application system is the final stage in the sequence and may run only after the earlier stages have verified all the configuration information of the application and the validity of the data structures.

Table 1 lists each stage of the Pluribus operating system, together with the aspects of the environment it guarantees. Many of the functions listed will not be discussed further but are provided to illustrate the layering of stages.

Since processors continue to perform each of the stages periodically, changes in the environment will eventually be noted. Any stage detecting a discrepancy in the configuration map will disable all later stages until the discrepancy is repaired. Then, all the later stages, which might depend on data verified by the disabling stage, will be forced to run all their checks, guaranteeing that they will make any further modifications to the configuration map necessitated by the first change. A serious failure, such as a nonexistent-memory interrupt, disables all but the first stage. In these cases, some reconfiguration might be needed, and all stages should perform all their checks before the application system is resumed.

C. Establishing Communication

So far, we have described the progress of one processor through the staged checking procedures of the operating system. All processors in the Pluribus perform the same checks, since it is

Table 1 Pluribus Operating System Stages

Stage	*Function*
0	Checksum local memory code (for stages 0, 1, 2). Initialize local interrupt vectors, and enable interrupts. Discover Processor bus I/O. Find some real-time clock for system timing.
1	Discover all usable common memory pages. Establish page for communication between processors.
2	Find and checksum common memory code (for stages 3, 4, 5). Checksum whole page ("reliability page").
3	Discover all common busses, PIDS, and real-time clocks.
4	Discover all processor bus couplers and processors.
5	Verify checksum (from stage 2) of reliability page code (for rest of stages plus perhaps some application routines). External reloading of missing code pages is possible once this stage is running.
6	Checksum all of local code.
7	Checksum common memory code. Maintain page allocation map.
8	Discover common I/O interfaces.
9	Poll application-dependent reliability and initialization routines. Periodically trigger restarts of halted processors.
10	Application system.

important that they agree about the state of the system resources. Coordination of multiple processors with potentially different views of the hardware configuration requires two mechanisms: the processors must agree on an area of common memory in which to record the machine configuration map, and they must cooperate in their decisions to modify the map.

The first step in coordinating the multiple processors of a Pluribus is to agree on a page of memory through which to communicate. The procedure for initially establishing the page for communication is clearly delicate. Prior to establishing the page, the processors have no way to communicate about where it will be. The procedure must operate correctly in the face of failures which might leave some of the processors seeing a different set of common memory pages from the rest. Processors which are unable to see the communication area will attempt to use another memory page and must be prevented from interfering with the unaffected processors.

Any processor that is first starting up (or restarting after some massive failure) can assume nothing about the location of the communication page. Any page may be used, and therefore a small area for communication control variables is reserved on each page of common memory. Part of this area is used for a brief memory test, which must succeed before the page may be used at all. Every processor attempts to establish the lowest numbered (lowest address in memory space) page that it sees as the page through which to communicate. To be valid, any page must have a pointer to the current communication page, and the communication page must point to itself.

Each processor looks at the pointer on the lowest numbered page it can see. There are three possible states for the pointer. First, if it points to the page itself, the processor has found the communication page and may now proceed to interact with other processors about the common environment. If it points to a higher numbered page, the processor may just fix the pointer, as the requirement that the communication page be lowest makes this case inconsistent. If it points to a lower numbered page, the processor must attempt to check if the indicated communication page is active. It must assume that the data might simply be old or invalid and must time it out using a dedicated entry in a special array of timers which is allocated on each page. The processor increments the timer and, if it ever reaches a certain threshold, unilaterally fixes the communication pointer and starts to use this page for communication. The processor is prevented from doing this by any other processor which is successfully using the lower numbered communication page; all such processors periodically zero all the timers on all memory pages in the system.

Consider what happens during various possible hardware failures. If the memory *bus* containing the communication page is lost, all processors will attempt to establish a new communication page on the other *bus*. Using their timers on the new lowest page (which initially points to the old one after the failure), they await the threshold. No one is holding the timers to zero, so the new page becomes the communication page when some processor's timer first runs out.

A processor blinded to the communication page by a *bus* or coupler failure will try to establish a higher numbered page for communication. From the point of view of the failing processor, this case is indistinguishable from the previous case, where the common *bus* failed. Since the rest of the processors are satisfied with the communication pointer, they will hold all timers to zero, and the failed processor will never be able to change the communication page pointer. If the processor sees a set of pages disjoint from the rest of the system, it behaves as if no other processors are running, but there is no memory where it may interfere and now we have two systems operating independently. In this case it is likely that the two systems will interfere over other resources; since multiple failures are required for this situation to occur in a Pluribus, we choose not to attempt recovery here.

D. The Consensus Mechanism

When configuration data must be updated, it is crucial to coordinate the Pluribus processors before making the modification. The mechanism to accomplish this goal we call consensus. Each stage has a consensus which is maintained as part of its environment. The first step in forming a consensus is to determine the set of processors that is executing the corresponding stage. This set has certified the primitives necessary to maintain successfully this stage's portion of the configuration map. In order for the system to respond to failures, the consensus must be kept current—new processors must be able to join it rapidly and processors that may have halted or ceased to run the stage must be erased from the set.

Each processor, based on its hardware address in the Pluribus, is assigned a bit in three consensus arrays, called "next," "smoothed," and "fix-it." As part of running the corresponding stage, every processor periodically sets its bit in the next consensus array to show that it wishes to participate in the consensus. After enough time has elasped for each properly running processor to set its bit, this array is copied into the smoothed consensus and cleared. The set of processors in the smoothed array will then be used as a basis for decisions to reconfigure some portion of the resource map.

Any processor which wishes to modify some configuration information sets its bit in the appropriate fix-it array. Processors that agree with the configuration map clear their bits, and bits corresponding to processors not in the smoothed array are also cleared.

In effect, the bits in the fix-it array represent the votes of the individual processors in favor of a potential modification. In most cases, it is desirable that all processors agree before making the change. All processors wait until the fix-it array matches the smoothed array before implementing the fix. Other modifications might require only majority or two-thirds agreement. The choice of policy often depends on some tradeoff between resources (e.g., should we use more memory or more processors?). The Pluribus approach allows us to make this choice independently at each stage.

Since each processor in the Pluribus performs each stage of the checking code, the consensus mechanism provides the coordination needed to change the configuration map gracefully. When a stage detects a failure, the processor sets the appropriate fix-it bit and disables the following stages. When enough processors detect the failure they implement the fix to the configuration map. Now these processors can complete the later stages, devoting their attention to any further changes required by the failure. A processor which sees a different picture of the resources and cannot reach agreement with the rest of the system hangs forever at the point of detecting the discrepancy. This technique effectively prevents the processor from damaging the system.

E. Application-Dependent Checking

In general, it is desirable for the application system to perform its own checks before initiating or resuming normal operation. The last stage provides a mechanism which polls application-oriented processes to perform consensus-driven checks and repairs of their own data structures. This stage uses the results of the hardware (application-independent) discovery stages to certify its own data structures. For example, it could allocate or deallocate device parameter blocks as the I/O devices are discovered or disappear and initialize spare memory pages for use as data buffers as they become available. User-written reliability checks can be performed on any of the application data structures, and the appropriate reinitialization invoked to remedy failures.

Occasionally, it is possible for a processor checking application data structures to implement minor repairs to the data structures unilaterally. For major reconfigurations of the data structures, such as complete application system reinitialization, the checking routines must signal to the stage dispatcher that consensus is needed. The last concurring processor is then permitted to perform the reinitialization routine. Just as the early stages guarantee the hardware map, the application-dependent routines have the consensus mechanism at their disposal to validate the system data structures before entering the system. In addition, the application system data structures are rechecked periodically during normal system operation.

IV. An Example of Application Reliability

We use two general techniques to ensure the validity of data structures in the Pluribus. First, redundant information, where it exists, is checked for discrepancies, and appropriate action taken if they exist. Second, since detailed examination of all data for inconsistency is deemed impossible for any system of nontrivial complexity, we use watchdog timers to ensure the correct operation of the application system at various levels. As an example, we will discuss the buffer management strategy for the Pluribus IMP system.

Buffers in the Pluribus IMP circulate through the system from queue to queue; in some cases, they may be shared between two or more processes. Since a compromised queue structure may, in general, rapidly degrade the performance of the system, elaborate checking methods are built into the IMP program at various levels. In particular, we must be able to detect queues that are crossed or looped and buffers that have been lost (are on no queue at all).

Associated with each buffer in the system is a set of use bits corresponding to various processes that consume buffers. Any process that enqueues a buffer for some other process first sets the use bit for that process. When a process dequeues a buffer, the appropriate use bit must be on or the buffer will not be processed. As a special case, buffers on the system free list must have all their bits turned off. The buffer-freeing routine only returns a buffer to the free list if the last remaining use bit is that of the freeing process.

This technique intentionally generates redundant information and continually validates it as a buffer circulates through the system. In other words, the existence of a buffer on a queue informs the system that some processing is desired for that buffer. In principle, the use bit signals the same thing. Each buffer-processing routine could scan all the buffers in the system for those with its use bit set, but such a strategy would clearly be inefficient. The redundancy check gives preference to neither the queue nor the use bit as an indication of need for service, but rather requires agreement between the two indicators. When they disagree, the system assumes that a failure has indeed occurred and attempts to correct it by forcing the queue to be empty, so that the effects of the failure can be contained as much as possible.

The use bits allow the prompt detection of looped and crossed queues. In addition, an improper buffer pointer will often lead to a failure of the use bit check.

We must also consider the case of a buffer which has been lost from all queues. This condition could arise due to a program bug or as a result of a queue being emptied after a use bit failure. We could employ a classical garbage-collection scheme for this purpose; unfortunately, the demand for buffers is often great in a high-speed communication system, and the requisite locking of the buffer resources during such a garbage collection would likely result in lost inputs.

The recovery scheme we have chosen is a watchdog timer mechanism. Each buffer has associated with it a flag set by normal activity of the buffer which, in this case, is defined to be the periodic appearance of that buffer on the free list. Whenever a buffer is freed, its flag is set. In addition, flags for all the buffers on the free list are set periodically. In the high-speed communications environment, where data passes through a network node very rapidly, each buffer must appear on the free list at least once every two minutes. Therefore, each buffer flag is checked every two minutes to be sure it is set, and then cleared. A zero flag indicates that the buffer has dropped out of normal activity, and the buffer is unilaterally freed and its use bits cleared. In this way, any lost buffer is detected within at most four minutes and returned to normal usage.

V. Advantages of the Pluribus Approach to Fault-Tolerance

Two factors help to make our approach a cost-effective one. First, fault-tolerance is implemented primarily in software. This not only allows us to use unspecialized off-the-shelf hardware for much of our system, but also gives us considerable flexibility by allowing us to try new ideas as the product develops. When the time comes to upgrade machines in the field, a new software release is infinitely preferable to hardware modification. Implementing most fault detection in software also allows more complete error reporting than is characteristic of static-redundancy approaches.

The second factor is the modular nature of the Pluribus. Initially, the modular approach was chosen to permit easy expansion of the capabilities of a system to fit an application without being hampered by system-size boundaries. Our system expands by adding the same hardware modules as those which are duplicated to create a dynamic fault-tolerant system. Thus any system with more than the minimum number of processors for a given application both performs well and is fault-tolerant. A processor failure in such a system merely causes it to run a little slower. Since individual processors are relatively inexpensive, the percentage increase in system cost for processor redundancy is usually small, especially in large systems.

Sometimes the system requirements justify only limited fault-tolerance. An example is the large front-end processor which services the BBN Research Computer Center [Mann, Ornstein, and Kraley, 1976]. Here the bulk of the machine is fully redundant, but several of the host interfaces are used only occasionally for experimental systems, and their users can tolerate an occasional outage. Therefore, these interfaces are not duplicated, with a resultant savings in cost.

An additional factor contributing to cost-effectiveness is the relatively low percentage of processing power spent in explicit error detection (about 1 percent for current systems). We depend to a large extent on checks embedded in the operating program (such as code checksums) to detect errors, since the program is able to recover from failures whose effects are detected well after the fact. It is common practice for large software systems to include checks for some "impossible" software states and bad data structures. We have expanded these checks to be comprehensive, including checks which catch many types of hardware errors as well as lingering software problems.

One interesting effect of our approach is to make even a minimal, nonredundant machine significantly more resilient to transient failures caused by either hardware or software. All of the fault-tolerant mechanisms which run in the large systems run also in the small ones, and there are many transient failures which cause only momentary confusion which is usually solved by some level of reset or reinitialization. Obviously, a solid failure of some critical component or destruction of the program cannot be resolved without redundant resources, but these are by no means the only possible failures.

One result of our modular approach is that in contrast to the usual state of affairs, we expect larger systems to be more reliable than smaller ones, since more resources are available to be redistributed in case of trouble.

VI. Recent Field Experience

During the past year, we have had the opportunity to observe eight Pluribus IMP systems both under general operational conditions and in controlled field tests; the availability of these machines has been above 99.7 percent (by availability we mean uptime divided by scheduled uptime, excluding power and air-conditioning failures). Almost all the downtime was caused by program bugs which have been corrected since. Most recently, availability has been above 99.9 percent and we expect it to improve further as the machines reach maturity.

In evaluating this experience in terms of fault-tolerant performance, we feel that it is important to go beyond overall availability numbers and discuss the kinds of faults that the Pluribus system can report, the kinds we observed in the field, and the effects these faults had on system behavior.

The concepts of availability and fault-tolerance are complex when applied to a Pluribus since failure of a component generally results in a reduction in, rather than a complete loss of, performance. In many applications this is an advantage since extra capacity is useful during periods of peak load and reduced service is tolerable while repairing faults. For example, if an I/O interface or an entire I/O *bus* fails, the machine automatically substitutes a spare element with only a momentary (often unnoticeable) interruption of service and with no loss in performance. In the case of processors and memory, however, all resources are normally in use (none are in a standby mode) and the loss of any one (or several) of them forces a reduction in performance, but does not keep the system from running.

When used as an IMP, the principal measure of Pluribus performance is throughput. In the tests described below, the presence of program bugs (since corrected) resulted in somewhat lower availability than we had expected, but the three machines easily exceeded their contractual requirements and were able to deliver better than 92 percent of their rated throughput capacity 99.76 percent of the time and better than 50 percent of capacity 99.83 percent of the time.

Under normal operating conditions, it is possible to observe an IMP only by means of its reports to the NCC or by the reports of its neighbors in the network. Since IMP's often operate unattended, emphasis has been placed on the ability of each Pluribus to evaluate and report its internal hardware and software health. Three varieties of trouble-report messages are sent to the NCC.

Since the Pluribus continually evaluates the state of its hardware (see the discussion of the STAGE system), one type reports trouble in the hardware area. Examples of this are I/O errors, memory parity errors, power failures, and changes in configuration. The second type reflects the results of numerous interlocks and consistency checks which are made regarding tables, queues, variables, and other software entities. The third category concerns the Pluribus' role as part of the network. These reports monitor normal throughput statistics and temporary discontinuities in IMP-IMP message handling protocols, and are normally not directly pertinent to the fault-tolerance of the Pluribus itself. In a few cases the reports are received some time after a fault has been detected and dealt with by the Pluribus, but most fault messages appear within a few seconds.

In the normal course of building and operating Pluribus systems during the past year, we observed a number of unexpected hardware and software faults, but to verify our ideas and procedures we also wanted to observe a number of failure modes which would be expected to occur infrequently under normal operating conditions. To this end, we conducted an extensive series of tests over a three-month period using three four-processor Pluribus IMP's with redundant I/O interfaces, interconnected by high-speed terrestrial and satellite links. These tests demonstrated how the Pluribus handles many of the possible faults that might be encountered during the life of the equipment. We believe that the combination of the unexpected and planned faults we experienced constitutes a valid sample of the wide variety of intermittent failures in either hardware or software which such systems are likely to encounter. Examples of the types of fault recovery which were provoked or observed during these tests are discussed in the following.

1 *Failures on the processor bus.* We powered off various combinations of processor *buses* to demonstrate that the system would continue with traffic processing. We also tried placing bad instructions in various processors' local memories. In power failure situations, the remaining processors continued to operate without reinitialization. Data handled by the failed processor(s) was recovered by network protocols and a number of trouble-reports indicated this fact. Data structures which were "locked" by the failed processors were "unlocked" by a software watchdog timer. When power was restored, the processors were smoothly readmitted to the system. Processors with bad local memory either halted or looped, and were quickly reloaded by other processors and brought back into operation automatically.

2 *Errors in or loss of common memory.* We created situations whereby the system suddenly saw common memory disappear. In some cases we powered off the memory *bus*; in others we "removed" memory from usability tables. We also observed some spontaneous parity errors. Since common memory pages are assigned specific roles at initialization time, loss of one or more pages caused a variety of reactions, depending on the role of the lost memory and the amount remaining. At one extreme, loss of all common memory prevented the system from continuing. At the other, loss of one of several pages of message buffers caused only a brief adjustment of memory assignments by the Stage program. Most Pluribus systems are organized for fully redundant operation and have spare

code and variable pages. Loss of a primary code or variables area caused a short transient in operations while the spare was initialized. As an example, loss of one-half of physical common memory (several pages of code, variables, and buffers) caused a reconfiguration lasting 15 s or less. During this period, all processors agreed on the reallocation of the remaining memory and reevaluated its usability. As a further test, we destroyed the integrity of various pages of common memory by storing random data in the checksummed areas. The system reacted by restoring the contents of the affected page from the backup copy. This process required about 10–12 s. We also created test conditions in which the system found that all copies of critical programs in common memory were unusable (their checksum was bad). At this time the system automatically requested that it be reloaded (from another of the Pluribus IMP's or the NCC). It should also be emphasized that the integrity of message buffers is also protected by software checksums; data harmed in any way is reported to the NCC, and the originator is notified so that retransmission can take place.

3 *Loss of I/O device.* We both created and observed several situations wherein I/O devices were either removed or experienced errors. In these cases, the I/O device was eliminated from usability tables by all processors and a backup device substituted. The system continued to operate, although in some cases, depending on the configuration being used, reinitialization was required. Loss of an entire I/O *bus* was handled in much the same way.

4 *Loss of critical hardware.* We observed that redundantly configured Pluribus systems would survive the loss of the RTC and the PID by swapping to the backup. Very little time was lost before the system continued. Errors in PID and RTC operation also are checked for and reported.

5 *Internal software errors.* As previously mentioned, the STAGE system and the IMP code are designed to check on the internal consistency of various software structures. In addition, the system ensures that none of the asynchronous processes is allowed to remain in a waiting state or in a loop. On a very infrequent basis, we observed that a Pluribus will report that such a condition was detected and corrected. We also forced many of these situations to occur by destroying key data structures or by causing queues to be looped or crossed. The system detected these, reported the problem, and continued normally, reinitializing if necessary.

6 *Artificial pathological conditions.* We did not attempt to cause pathological behavior of Pluribus hardware components which would, for example, write zeros to portions of memory or amputate *buses* at random, although we simulated these conditions with the software. Our observations of pathological behavior in the field, although infrequent, convince us that many of these cases can be withstood by the fault-tolerant software. For example, during field tests we observed that some extraneous data appear occasionally in certain critical tables causing the Pluribus to reinitialize quickly or to suspend activity on a communications link briefly. The problem was traced to a special reloading device which was being improperly activated. This situation was eliminated by a minor program change.

We have now gained enough experience with the Pluribus fault-tolerant mechanisms to have confidence in their ability to detect and cope with failures. In the field, spontaneous failures have been of a relatively minor nature and have been successfully dealt with. Under test conditions, all the major and minor failures which occurred or which we created were well tolerated and the systems continued to function within their rated capacities.

VII. Pluribus System Maintainability

Most fault-tolerant systems are designed to be repaired, sooner or later, by humans. Maintainability thus becomes a significant factor in long-term system performance. Since many systems are designed to recover from any single failure, but not from all multiple failures, the mean time to repair (MTTR) directly influences on-line spares requirements and hence the system cost for any given performance goal. To minimize MTTR, the system must provide accurate and unambiguous information about the nature of the detected fault and the automatic recovery process initiated. The environment in which the system operates is also important since the maintaining authority must be notified and must initiate the repair process as soon as possible.

The actual repair process may be carried out at several levels depending on the accuracy of the diagnostics and the obscurity of the failure symptoms. At the lowest level, the repair is accurately defined by the diagnostic and involves only the replacement of a faulty component. At the highest level, the failure may be caused by a design bug in either hardware or software. For the latter, the system must provide sufficient tools to permit overriding the operational recovery procedures. They must permit the repair personnel to reconfigure the system and run any required diagnostic procedures. The more powerful repair tools must be guarded to avoid operator-induced errors. Ideally this "fool-tolerance" [Goldberg, 1975, p. 32] should extend into all phases of repair. In practice we use only a two-level protection scheme that relies on experienced personnel not to make catastrophic errors.

Although we tend to think of hardware malfunctions as separate from software malfunctions, the symptoms of failure and the recovery procedures are frequently similar. In the Pluribus, the first detection of a fault is usually through failure of an embedded check in the main program, and frequently that is all that is required to initiate a correct recovery procedure. When the diagnostic value of an embedded check is insufficient to define a recovery procedure, various modular diagnostics may be run on the system. Thus in the case of a memory whose checksum is discovered to be wrong, the recovery action is to run a brief memory diagnostic and, if the memory appears usable, to restore the code from a spare copy.

Including a spare copy of some resource helps system recovery only if that spare resource works. Although it is traditional to run modular diagnostics on spare resources, our strategy has been to force the system to rotate use of resources from time to time. In some cases we use manual procedures, but the tendency has been to include automatic rotation procedures in the operational system software. This technique is clearly more appropriate to our application than it would be to a more traditional fault-tolerant requirement, since rotating faulty hardware into the operational system could cause a transient malfunction. On the other hand, it provides a better test of the hardware than modular diagnostics would provide.

One advantage of our reliance on embedded checks for failure detection is that we can detect that class of failure which is rarely caught by diagnostics. It is axiomatic that the operational program is the best program for certifying the hardware, but our operational program has also become the most comprehensive diagnostic for the hardware. In our experience, some of the most subtle hardware failures occur during operation of the application system, even though hardware diagnostic programs detect no errors. By augmenting the operational system with diagnostic capabilities, we have often been able to isolate even obscure or intermittent failures without interrupting normal operation.

A. Reporting Facilities

In the Pluribus IMP, the mechanism for reporting errors, recovery operations, and change-of-status information is the system trap (i.e., a supervisor call). Traps are reported locally on the system terminal and are also sent via trouble-reports to the network log at the NCC, where they serve a variety of diagnostic purposes. Understanding the nature of a failure in the running system requires fairly accurate knowledge of the state of the machine at the instant of the failure. The initial implementation of the trap mechanism recorded only the code number of the trap, which set of processors had encountered it, and a total occurrence count. This proved inadequate for accurate diagnosis and we have augmented the original trap mechanism to allow for saving a large snapshot of the instantaneous state of the processor, including such information as the contents of general registers, the global system time, map register settings, the last value read from the PID, and other important local data. These snapshots allow us to examine diagnostic information about the failure after the recovery code has taken effect and normal operation of the system has resumed. In an operational IMP, the snapshot information is sent to a data collection program at the NCC, where it is both stored for future reference and printed out on a log terminal. The snapshot facility is usually only enabled for that set of traps which indicate system malfunctions of some kind, since there are many normal traps which indicate such things as network topology changes. The same data collection program also keeps track of the current configuration of each machine and reports any changes on the log terminal. Thus the reconfiguration resulting from some module failure is immediately apparent. Correlating a reconfiguration with preceding snapshot error messages is usually sufficient to isolate solid failures.

B. Remote Diagnosis and Repair

Where the failure is intermittent, or error indications are ambiguous, we can make further diagnosis from the NCC using the remote connection capabilities of the network. This allows personnel at the NCC to interact with a system at a remote site exactly as if they were using the system control terminal at the site. We have provided a command structure in the system which allows us to make either "soft" or "firm" overrides of the configuration control structure, loop communications links, and run a variety of special diagnostics, monitors, and traffic generators. This enables us to diagnose many problems from the NCC even before dispatching repair personnel to the site (this can be especially appropriate for diagnosing program bugs). The current software is best at diagnosing the solid failures typical of mature hardware and treats most long-term intermittents as unrelated transients. Although we plan to implement heuristics which can deal with this type of problem, the diagnosis of long-term intermittents currently requires human intervention. Fully redundant Pluribus systems may be thought of as networks of paths and *buses*, so by causing the system not to use a particular path or *bus* and watching the trap log, we are usually able to localize the source of a hardware intermittent. Partitioning the *bus* and using some subset of the modules on the *bus* further localizes an intermittent traced to a particular *bus*, and repairs can then proceed. The same tools for reconfiguration are, of course, also available to maintenance personnel on site through the system control terminal, and trap reports sent to the NCC are duplicated also.

C. Partitioning

In extreme cases, when all normal diagnostic approaches have been exhausted, it is also possible to partition a fully redundant machine into two separate machines and run the operational system in one half while running stand-alone diagnostics or another copy of the system in the other half. We originally expected to use this approach quite frequently, but experience has shown the technique to be less useful than we expected.

Splitting a system is a combination of many "firm" overrides of the configuration control which are not currently protected against operator error (i.e., deleting the last copy of a resource from the use tables, or overlapping system resources across the partition). There is also the problem of identifying fault-free components to include in the operational system half. In general, being able to identify a faulty module which is to be excluded from the operational system implies that we can fix the fault by replacing the module, which usually obviates the need for partitioning into two machines. And finally, once a machine has been split, any new failures are likely to cause fatal problems that the machine might have been able to cope with had it not been split. Our current feeling is that the risks of splitting an operational system usually outweigh the advantages.

D. Reloading and Down-Line Loading

An important facility provided by the Pluribus hardware allows us to load and start the machine with no onsite personnel. This is accomplished by special-format messages which trigger a simple reload device when received over the network. This device is used to load a software package capable of dumping or reloading the operating system and application code. The source of reload code may be either some other Pluribus IMP on the network, or a disk file at the network control center. These reloading facilities are also used for distributing software updates to the machines in the field. A Pluribus IMP which discovers all copies of some application code page to be compromised will attempt to get a down-line reload from a neighbor IMP. This request is reported to the NCC where an operator then sets up the reload source for the transfer. Its use enables an IMP without duplicated resources to recover quickly from transient failures caused by hardware or software.

E. Maintenance Experience

The prototype Pluribus systems performed their error recovery functions well in many cases. Minor problems were often bypassed so effectively that the users and maintenance personnel were never aware of the problem. Even following drastic failures, such as the loss of a common memory bus, normal system operation was restored within seconds. From our experience with these early systems, however, certain deficiencies in our original strategies have become clear.

In some failure cases, one repair would lead to another, until eventually a fairly major reinitialization would be performed, with obvious effects on the users of the system. Unfortunately, the massive recovery often destroyed evidence of the original failure, or masked evidence necessary for effective diagnosis. While the goal of restoring the system to normal operation was achieved, we were left without any idea of why the reinitialization was required. This was particularly frustrating when the frequency of occurrence was on the order of hours or days.

In other cases, normal operation seemed to continue while some hardware failure occurred undetected. Either the failure was covered by effective recovery at a fairly low level in the system or it occurred in a redundant portion of the hardware which was not being exercised. A second failure in conjunction with the first would remove the last copy of some critical resource, causing the system to fail.

These initial experiences led through several intermediate steps to the current set of maintenance tools and diagnostics. In the prototype systems, we were forced to remove the system software and run stand-alone diagnostics when trouble arose. Development of the original recovery algorithms into early versions of the current STAGE system allowed diagnosis and repair while running the operational system; however, system programmers were required to interpret the traps and wrestle the system into different configurations during repair. The usual repair team during this period included a system programmer (usually at the NCC) watching and interpreting the traps, with a maintenance technician on site replacing components.

At present, the tools and diagnostics are well enough defined and documented so that usually only maintenance personnel are required for a repair. Hardware and software staff at the NCC may offer suggestions when maintenance personnel are dispatched to a site and may still direct occasional repair efforts if a difficult problem or inexperienced personnel require it, but this is the exception rather than the rule.

VIII. Other Applications and Extensions

Since the Pluribus has evolved from a communications application where overall system availability rather than total fault-coverage is the goal, our approach is most obviously suitable for similar applications. We have opted for an approach which depends heavily upon reconfiguration and reinitialization when faults are detected, and which requires very little special hardware beyond that needed to implement our multiprocessor architecture. Our approach would not be suitable for applications where absolutely no downtime can be tolerated, where total computational context must be preserved over failures, or where overall correctness must be ensured. In these cases, traditional approaches involving some form of static redundancy or execution redundancy are indicated [Avizienis, 1975; Avizienis, 1976]. Techniques somewhat similar to ours, but for a redundant uniprocessor, are in use in the Bell System's latest Electronic Switching System [Myers et al., 1977]. Although we have not closely investigated applications outside thee communications area, we believe our approach is suitable for many other tasks, and we discuss several of these briefly below.

A. Message Systems

We have made an extensive study of the possibility of using the Pluribus computer as the basis for a message system. By message system we mean not only traditional message-switching such as is done in the Telex system, but also a system of mailboxes and files by which users can exchange and file messages without recourse to the U.S. Postal System, secretaries, or filing cabinets, and which will permit complicated searches and sorts of message files. Such a system must have high availability but could easily tolerate brief outages after a failure.

B. Real-Time Signal Processing

We have already built one system which is the front-end and control processor for a seismic data collection network, and which performs some preprocessing of seismic data [Gudz, 1977]. We believe this application can be extended to other areas of real-time signal processing with requirements for high overall system availability. Since many signal processing tasks can be broken into parallel components, the multiprocessor architecture would be especially appropriate.

C. General-Purpose Timesharing Systems

It seems to us that explicit use of fault-tolerant techniques could benefit general purpose timesharing systems and large operating systems. These systems operate continuously and are subject to minor hardware errors and subtle software bugs, but do not require totally uninterrupted operation. Although most large systems include some self-checking in the software, software fault-tolerance, to be truly effective, must be well integrated into the overall system design, and into the special hardware features which are usually required.

One of the primary purposes of most large operating systems is to provide disk and tape handling features. In this context, reinitialization in response to faults is a much more serious problem than, for example, in the IMP. Various checkpointing procedures may be required to restore the overall system state to a point where restart is possible [Yourden, 1972, pp. 340–353]. Large operating systems often support a variety of checkpointing services since the best techniques to use under these circumstances depend in part on the applications being serviced; in cases involving on-line database updates, the application programs themselves must be designed around their fault-tolerance requirements.

D. Reservations Systems

Airline, hotel, and car rental reservation systems provide good examples of on-line database systems which could benefit from well-designed software fault-tolerance systems. Once a reservation has been accepted, it must not be lost. Backup techniques such as dual updating of two copies of the database, perhaps located in different cities with independent central processors and telecommunications systems, may be worthwhile. On the other hand, minor problems (hardware or software) may be tolerated, especially if the problems can be resolved by reentering on-line transactions which were affected by the fault. Even with dual machines in remote locations, using a machine like the Pluribus would increase the reliability of each site separately, and provide substantial computing power in an expandable package. Further research will be required to understand fully the implications to the Pluribus of database integrity requirements for reservation systems.

E. Process Control

Our approach is clearly more appropriate to some areas of process control than to others. We envision a typical application in the area of overall supervisory systems coordinating a number of subsidiary systems or controllers, and incorporating tasks such as inventory control and job scheduling. Processes that could afford to stop momentarily would be controlled directly. End-to-end error correction and fault-masking hardware would be used in the machine interface for applications needing overall fault-tolerance. As with the previous applications, some form of checkpointing would be built in to preserve context over restarts.

References

Avizienis [1975]; Avizienis [1976]; Barnes et al. [1968]; Bressler, Kraley, and Michel [1975]; Enslow [1974]; Goldberg [1975]; Gudz [1977]; Heart [1975b]; Heart, Kahn, Ornstein, Crowther, and Walden [1970]; Heart, Ornstein, Crowther, and Barker [1973]; Heart, Ornstein, Crowther, Barker, Kraley, Bressler, and Michel [1976]; Mann, Ornstein, and Kraley [1976]; McKenzie, Cosell, McQuillan, and Thrope [1972]; Myers et al. [1977]; Ornstein, Crowther, Kraley, Bressler, Michel, and Heart [1975]; Ornstein, Heart, Crowther, Rusell, Rising and Michel [1972]; Ornstein and Walden [1975]; Roberts and Wessler [1970]; U.S. Pat. 4,035,766 [1977]; Wolf [1973]; Wulf and Bell [1972]; Yourden [1972].