Software fault-tolerance in the Pluribus

by JOHN G. ROBINSON and ERIC S. ROBERTS
Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

INTRODUCTION

Over the past decade, the decreasing cost of minicomputer components has encouraged the use of multiprocessor techniques in the design of high-speed, cost-effective computer systems. Multiprocessor architectures have two principal advantages over conventional single-processor designs. First, a multiprocessor system can achieve greater computational speed through parallelism in its task structure. A second advantage is that multiprocessors realize greater reliability through redundancy of processing elements. This paper discusses several aspects of multiprocessor reliability as it applies to the Bolt Beranek and Newman (BBN) Pluribus system.

Recent advances in processor technology, including the development of inexpensive LSI-based microprocessors, have further increased the cost advantages of multiprocessor systems and have led to the consideration of multiprocessor architectures involving very large numbers of processors. Early examples of multiprocessor systems designed around minicomputers include the C.mmp system at Carnegie-Mellon and the BBN Pluribus. Each of these early systems has prompted the development of larger microprocessor-based systems, such as the Carnegie-Mellon Cm* project and the BBN "Butterfly" machine.

THE BBN PLURIBUS

The BBN Pluribus system has been described in detail in a number of previous papers. The Pluribus was originally designed as a reliable, modular, high-speed Interface Message Processor (IMP) for use with the ARPANET. To date, eight such systems have been delivered. The Pluribus has also been used to support a real-time processing and display system for seismic data and a terminal handler connecting a large number of terminals to several host computers. Future applications include a high-speed version of the Satellite IMP. On the whole, our experience with building Pluribus systems has been favorable, and we are currently trying to attract a more general market.

The basic organization of a typical Pluribus system is illustrated in Figure 1. Each processor bus contains a number of processing units (usually two) and associated local memory for each processor. The processors are SUE minicomputers produced by the Lockheed Electronics Corporation. Common memory (that which is accessible to all processors) is distributed among the system memory busses. The I/O bus units provide an interface to a variety of input/output devices, such as communication lines and modems in conventional Pluribus applications. In addition, each I/O bus includes a real-time clock used to coordinate processor activity and a special hardware device called a PID (Pseudo Interrupt Device) used for process scheduling. The individual busses are connected to each other via bus couplers which are indicated by the solid lines in Figure 1.

The hardware organization of the Pluribus makes it possible to achieve system reliability through the redundancy of hardware components and interconnection paths. For example, if a processor bus becomes unusable through hardware failure, the remaining processor bus(es) will generally be able to provide sufficient computational power to run the system. Similarly, if the connection between a processor and common memory bus is lost, the system can proceed normally by removing either of the two busses from the operational system.

RELIABILITY IN THE PLURIBUS

Fault-tolerant computer techniques originally arose in response to the unreliability of early digital hardware. Historically, the principal concern of fault-tolerant system design has been with completely error-free system operation. The classical technique used to provide reliability at this level is to include three or more instances of each system component; each component performs each calculation simultaneously. Additional hardware is then used to check the individual results for correctness, relying on a majority decision in the case of a discrepancy.

The notion of relaxed reliability

Most of the past research in reliable computer system architecture has been directed toward applications in which completely fault-free operation has been necessary. In designing the Pluribus, we have been concerned with applications whose reliability requirements are much less stringent. We have found that these applications suggest a
relatively new set of techniques and that many of the classical approaches to fault-free reliability do not directly apply.

In the Pluribus, we have not been as concerned with continual error-free operation as with providing maximum system availability. In the environment of a communications network, for example, it is desirable to keep each node functioning even if data is lost during a failure-induced transient. A communications network is characterized by a relatively steady-state operation which is largely independent of all but the most recent network activity. If a node in the network is forced to reinitialize, some active message traffic will generally be lost, but the node will very quickly be able to handle new messages in a normal fashion. This property of the communications network makes a relaxed reliability mechanism very attractive in that the continuity of system operation can be insured without incurring the cost of a more elaborate hardware reliability scheme.

We feel that the relaxed reliability model is widely applicable in other areas. For example, by enhancing a general-purpose operating system with checkpointing facilities, it should be possible to create an environment where such approaches to fault-tolerance may be used to advantage. The techniques used in the Pluribus should permit such a system to reconfigure following a failure and automatically resume normal operation from the previous checkpoint. In the network environment, checkpointing is not necessary since there is no long-term context to be saved. A system with checkpointing approximates the network environment by providing a clean state (the previous checkpoint) at which operation can resume.

Relaxed reliability in the Pluribus

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 the software reliability mechanism. When hardware errors are detected, the software exploits the redundancy of the hardware by constructing a new logical system configuration which excludes the failing resource, using redundant counterparts in its place.

Pluribus systems make use of redundancy in their software structures for much the same reason. 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 later. In addition, periodic background processes are used to recompute certain variables which are maintained by the operational system. If the recomputation uncovers a discrepancy, the variables are fixed or a more drastic recovery is attempted.

In many cases, a failure is not detected at the exact time of occurrence but at the time that 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 detect. In such cases, the system is not able to perform instantaneous recovery and seeks instead to restore normal operation as quickly as possible.
The value of software reliability

Our experience with the Pluribus has convinced us that fault-tolerance need not be expensive, provided complete fault-free operation is not the goal. Pluribus hardware can be configured for redundancy by including one extra copy of each critical system resource (i.e., one extra processor bus, one extra memory bus, and so forth). In general, software reliability mechanisms are highly flexible and are largely independent of the application routines.

As a result of placing the responsibility for system recovery on the software, Pluribus systems are capable of surviving (or isolating the effects of) a wide variety of intermittent failures and design errors in either hardware or software. Diagnosis of the failures can often be accomplished while normal operations continue.

THE PLURIBUS OPERATING SYSTEM

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 in later sections, which provide two examples of Pluribus fault-tolerant software strategies. 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.

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.

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.

Hierarchical structure of the stage system

The Pluribus operating system is organized as a sequence of stages which are pooled 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 config-

<table>
<thead>
<tr>
<th>STAGE</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>1</td>
<td>Discover all usable common memory pages. Establish page for communication between processors.</td>
</tr>
<tr>
<td>2</td>
<td>Find and checksum common memory code (for stages 3, 4, 5). Checksum whole page (&quot;reliability page&quot;).</td>
</tr>
<tr>
<td>3</td>
<td>Discover all common busses, PIDs, and real-time clocks.</td>
</tr>
<tr>
<td>4</td>
<td>Discover all processor bus couplers and processors.</td>
</tr>
<tr>
<td>5</td>
<td>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.</td>
</tr>
<tr>
<td>6</td>
<td>Checksum all of local code.</td>
</tr>
<tr>
<td>7</td>
<td>Checksum common memory code. Maintain page allocation map.</td>
</tr>
<tr>
<td>8</td>
<td>Discover common I/O interfaces.</td>
</tr>
<tr>
<td>9</td>
<td>Poll application-dependent reliability and initialization routines. Periodically trigger restarts of halted processors.</td>
</tr>
<tr>
<td>10</td>
<td>Application system.</td>
</tr>
</tbody>
</table>
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 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; such a processor will periodically zero all the timers in the array for each memory page in the system before the threshold is reached.

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.

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 elapsed 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 trade-off 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. As one of its stages 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.

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 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 reconstructions 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 concurrent 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.

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 non-trivial complexity, we use watch-dog 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 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 on buffers is often large 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 watch-dog timer mechanism. Each buffer has associated with it a flag which is 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.

**FAILURE DIAGNOSIS IN THE OPERATIONAL SYSTEM**

The Pluribus design permits the continuation, after perhaps a brief outage, of normal system operation following the failure of any single component. The component might be a single bit of memory or an entire common memory bus. Much of the responsibility for ensuring recovery from failures lies in the software; very little hardware beyond that necessary to construct a multiprocessor should be required.

The initial Pluribus systems performed their recovery quite well in many cases. Minor problems were often repaired so effectively that the maintainers and users were never aware of the repair being carried out. Even following drastic failures, such as the loss of a common memory bus, normal system operation was restored within seconds. As we gained experience with the operational systems, however, certain deficiencies in our strategies became 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 the evidence of the original failure, or occurred too late to permit effective diagnosis. While the stated goal of restoring the system to normal operation was achieved, we were left without any idea of why the reinitialization was necessary. This was particularly frustrating when the frequency of the reinitialization 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.

Such experiences have led us to an improved design for the recovery software. Any recovery operation is reported by the system using a system trap (i.e., a supervisor call). Even in those cases where an error might occur occasionally in normal operation, we prefer to report the failure and filter out the non-important occurrences of the report. In this way, an incipient hardware failure is often detected early enough that it can be remedied well before the inevitable second failure brings the system down.

We have also adopted strategies which exercise all the hardware in the machine periodically. Including a spare copy of some resource helps the system recover only if that spare resource is known to be good. Therefore, we force the system to exercise all of its 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.

One beneficial side-effect of this policy is that the operational program has become the best diagnostic for the hardware. Traditionally, some of the most subtle hardware failures occur during operation of the application system, though the hardware diagnostic program never detects any errors. By augmenting the operational system with diagnostic capabilities, we are able to isolate hard-to-find or intermittent failures, sometimes even without interrupting normal operation. The system-integration personnel routinely use the operational program as the last step in constructing a new Pluribus machine.

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 processor or set of processors had encountered it, and a total occurrence count. More recently, we have augmented the trap mechanism to allow for saving a larger snapshot of the instantaneous state of the processor, including such information as the contents of the 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 critical information about the failure after the fact, while permitting the recovery code to take effect and normal operation of the system to proceed.

To aid in the development of new software, we have also included a password-protected facility whereby a processor which has detected a serious malfunction may signal all the other processors to stop before obtaining their next task from the PID. This capability allows examination of the failure in the global environment, which might otherwise be repaired by recovery code or cause some other more dramatic failure. These mechanisms combine to provide the programmer with a reasonably tractable debugging environment. In particular, such facilities have greatly aided software development for new Pluribus applications.
SUMMARY

The structure of the Pluribus operating system has been described above, together with some techniques for software reliability and fault diagnosis. We have built Pluribus systems for several applications, and our experience has been favorable on the whole. While the emphasis of our applications has been upon communications, we expect our techniques could be applied to many different uses. Recoverable faults arise in most computer applications, and we feel that non-redundant computer systems would benefit from the incorporation of recovery-oriented operating systems such as that of the Pluribus.

ACKNOWLEDGMENTS

Much of the initial development of the Pluribus computer was supported by the Information Processing Techniques Office of the U.S. Defense Advanced Research Projects Agency, under Contract Numbers DAHC15-69-C-0179, FO8606-73-C-0027, and FO8606-75-C-0032, and by the Defense Communication Agency under contract DCA200-C-616. Additionally, a number of the application systems were developed under contracts from various branches of the U.S. Government.

Many people have contributed to the Pluribus project; Frank Heart has led the effort since its inception. Will Crowther, Mike Kraley, and Bill Mann in particular have contributed many of the ideas incorporated into the Pluribus operating system. Jane Barnett and Bob Brooks provided encouragement and help with the actual writing of the manuscript.

REFERENCES
