Strong Partitioning Protocol for a Multiprocessor VME System

Mohamed F. Younis, Jeffrey X. Zhou
AlliedSignal Inc.
Advanced System Technology Group
9140 Old Annapolis Road/MD 108
Columbia, MD 21045, USA
younis | zhou@batc.allied.com

Mohamed Aboutabl
University of Maryland, College Park
Department of Computer Science
College Park, MD 20742, USA
aboutabl@cs.umd.edu

Abstract

The trend in implementing today's embedded applications is toward the use of commercial-off-the-shelf open architecture. Reducing costs and facilitating systems integration are among the motives for that trend. The use of the VME bus becomes very common in many industrial applications. The VME bus attracts developers with its rigorous specifications, multiprocessing support and boards availability through multiple vendors. However, VME bus standard supports multiprocessing through shared memory, which does not impose strong function partitioning and allows fault propagation from one board to another. Such weakness limits the use of the VME bus in highly critical applications such as avionics. This paper presents techniques for strong partitioning of multiprocessor applications that maintains fault containment on the VME bus. The suggested techniques do not require any modification in the standard and the existing boards, and consequently maintains the plug-and-play advantage of the VME bus hardware products. The techniques are equally applicable to other tightly coupled multiprocessor systems. In addition, the paper describes the implementation of these techniques and reports performance results. Finally, the benefits of this technology for a space vehicle and commercial avionics are discussed.

1. Introduction

Nowadays, the use of digital computer systems becomes very common in mission critical applications such as flight control and vehicle management systems. In such applications, it is essential not only to ensure correct semantics but also to provide fault tolerance capabilities. Often proprietary design is used in implementing such applications. Recently both the vendors and consumers have realized that the proprietary approach is very expensive and substantially increases time-to-market. In addition, proprietary design does not provide flexibility to upgrade the system and to adopt new technology. Consequently, the use of open architecture and commercial-of-the-shelf (COTS) components is the trend in developing embedded mission critical applications. COTS-based approach maintains cost advantages and facilitates system upgrade. However, COTS components are usually developed without considering fault-tolerance. The challenge is to develop fault resilient COTS-based design without the need to redesign the components, thus keeping the cost advantage. This paper presents techniques that we used to tackle some of these challenges to build a VME-based fault tolerant architecture for avionics.

This section presents the motivation of our work and why the VME bus has been considered. In addition, an overview of the VME bus operation is provided, highlighting potential fault containment issues.

1.1 Motivation

Advancements in technology have enabled the avionics industry to develop new design concepts, which results in highly integrated software-controlled digital avionics. The new approach, referred to as Integrated Modular Avionics (IMA), introduces methods which can achieve high levels of reusability and cost effectiveness compared to earlier implementations of avionics [1]. The IMA approach encourages partitioning and using standardized modules in building environmental and functional components of avionics. While integration enables resource sharing, some boundaries need to be imposed to maintain system predictability and to prevent bringing down the whole system because of a failure of a single function. An IMA system is called strongly partitioned if the boundaries between the integrated functions are clearly defined so that a faulty function cannot cause a failure in any other function. Strong functional partitioning facilitates integration, validation and FAA certification. Following the IMA guidelines, the cost of both development and maintenance is

expected to decrease because of mass production of the building blocks, lower levels of spares, and reduced certification costs.

The backplane bus is one of the most important components in integrated modular avionics. While many backplane buses have been proposed, only a few are actually used. Selecting a backplane bus is influenced by many design and engineering factors, such as performance, reliability, and fault tolerance. Although such issues are very important to ensure certain level of safety of commercial jet aircraft and high availability of military aircraft, the cost of the bus and associated line replaceable modules is a major concern.

Most of the currently available dependable backplane bus systems, such as ARINC 659 [2], are expensive and supplied by very few vendors. It is clear that there is a need for an affordable bus system that provides the required levels of dependability and complies with the IMA design methodology. The goal of this study is to show how to enhance functionality and overcome inefficiencies in commonly used and widely manufactured low-cost buses to make them suitable for avionics. The VME bus system [3] is a prime candidate because it is both rigorously defined and widely supported. In addition, there is an expanding selection of VME bus boards and vendors that guarantee competitive prices and continuous support. Moreover, the VME bus offers an open architecture that facilitates the integration of multiple vendors' boards. Such a feature makes the VME bus an attractive choice for integrated avionics on high visibility projects such as the Venture Star space launch vehicle, supported by NASA for its 21st century space missions [18].

However, the VME bus architecture does not impose strong functional partitioning and allows fault propagation from one board to another, as we discuss in *Section* 1.4. For example, a faulty board can perform a wild-write, which corrupts the shared memory area and disturbs the computation on other boards [10]. Fault containment is very crucial for the integrated environment to guarantee that a faulty component may not cause other components to fail and risk generating a total system failure. Considering avionics, a failure in the entertainment system must not negatively influence flight critical control systems such as the flight manager. Lacking the capability of fault containment limits the use of the VME bus in mission-critical integrated system. This paper presents techniques for strong partitioning of multiprocessor applications that maintain fault containment on the VME bus. The suggested techniques do not require any modification in the standard and utilize the existing features[1] of the VME boards, and consequently maintain the plug-and-play advantage of the VME bus hardware products. It should be noted that, although the VME bus lacks other capabilities to be fault tolerant [8], this paper focuses on preventing fault propagation over the bus. The next section presents our fault model.

1.2 Fault Model

Strong partitioning implies that any erroneous behavior of a faulty component will not affect other healthy components without detecting the error. This erroneous behavior can be the result of a hardware or software fault regardless the nature (generic, accidental or intentional) and the duration (permanent, transient or intermittent) of that fault. A healthy system component needs to become aware of the faulty component only if they communicate to perform certain function. An attempt by the faulty component to corrupt other system components should not succeed. Other system components, that communicate with the faulty component, need to be aware of the error and perform recovery according to the nature of the application. We assume that there are application-specific semantic checks that verify the validity of the communicated data.

We assume that the system is not liable to Byzantine faults, i.e. all faults manifest themselves into errors, which are detectable in the same way by all the other healthy modules. In this paper, bus failure due to either bus damage, stuck control lines or the failure of bus arbiter are not addressed. Only faults in modules and the data transmission are considered. In addition, this paper focuses only on fault containment, in the sense of [11], without addressing system-level recovery of the faulty component. Fault containment implies that the damage caused to an application by a fault is proportional to the amount of resources needed by the application, not to the total amount of system resources [10].

Before illustrating the fault-tolerance deficiencies of the VME bus, with respect to the fault model presented above, the next section provides an overview of the bus operation.

[1] Assuming that the VME board is equipped with a memory management unit to control read and write access to local memory

1.3 An Overview of The VME bus

The VME bus allows multiprocessing, expandability, adaptability for many different designs and processors. It handles data transfer rates in excess of 40 Mbytes/sec using parallel data transfer. The VME bus is asynchronous and non-multiplexed. Because it is asynchronous no clocks are used to coordinate data transfer. Data is passed between modules (boards) using interlocked handshaking signals where the slowest module participating in the cycle sets cycle speed. Using asynchronous protocol in the VME bus provides reasonable capabilities to integrate products from various vendors.

The VME bus provides support for multiprocessing using shared memory. To avoid inconsistency while updating shared memory, read-modify-write bus cycles are used. The read-modify-write cycle allows updating shared memory as an atomic transaction and prevents race conditions. Master-slave architecture is used in the VME bus. Modules can be designed to act as masters, slaves or both. Before a master can transfer data it must first acquire the bus using a central arbiter.

Although the VME bus does provide reasonable compatibility to integrate products from various vendors, fast parallel data transfer, and a wide support by many manufactures, fault tolerance in VME bus based systems is very limited. The next section discusses error detection mechanisms in the VME bus and elaborates on weaknesses in fault containment.

1.4 Fault Propagation Over the VME bus

The VME bus relies on all connected modules for detecting and reporting faults on a specific failure control line. VME bus modules are expected to have on-board firmware diagnostics to detect faults. The VME bus master (sender) monitors the time for data transfer. If the slave (receiver) does not acknowledge the message, the master times out data transfer and retransmits. The bus provides neither error detection nor correction for the transferred data. There is no redundancy in either the transmission lines or the transferred data on the bus.

Generally, the built-in-test and transmission timing-out provide limited fault coverage. In the absence of message verification, faults can manifest errors that affect healthy modules which is a problem called fault propagation from one module to others. Faults cannot be contained within the faulty module and can jeopardize the behavior of the complete system. In addition, the shared model does not prevent a wild-write by a faulty module, which can corrupt the memory of other healthy module [10].

From the former discussion we can conclude that the VME bus needs enhancements to strengthen its fault tolerance capabilities, specifically in containing faults and recovery from failure. In this paper we focus on fault containment. The following issues need to be addressed in order to improve the fault containment in a VME bus system:

1. Validating the inter-module data transfer to detect transmission errors over the VME bus.
2. Preventing fault propagation from one module to others through the use of shared memory.

The next section provides a discussion of our approach to address these issues.

2. Fault Containment Techniques

Because low cost is an important feature of the VME bus, enhancing the fault containment capabilities should avoid changing the design and the layout of the currently available boards. Changing the design of a VME board will not only require reengineering and revalidation which increases the manufacturing cost, but also will again limit the number of vendors who agree to do the modifications. Thus, the suggested approach should be constrained by preserving the current hardware design of the boards as much as possible.

As illustrated in the previous section, fault containment on the VME bus needs to be added. In the following subsections, our approach is discussed.

2.1 Inter-module Data Transfer

The VME bus features parallel data transfer between modules. There are no error detection or correction bits associated with the transmitted data. Adding such bits will significantly affect the VME board’s design and, therefore, is not an option. As an alternate approach, an error detection code, e.g. cyclic redundancy check, can be appended to the end of the data. A data transmission module within the operating system kernel can generate this error detection code. Although the software-generated error detection code is less efficient than the hardware-based implementation, no board redesign is necessary using the software approach. For higher dependability, an error correction code can be appended. Because that error detection/correction code will reduce the efficiency of the data transfer on the bus and consequently the performance, it may be possible through the kernel to dynamically select either to append error detection or error correction code according to the length of the transmitted data. The receiver module

should validate the data using the error detection/correction code before committing that received data.

Using such information redundancy within the transferred data fits the multiprocessing scheme proposed in the next subsection.

2.2 Strongly Partitioned Multiprocessing

Strong partitioning of modules is one of the most important IMA requirements, which the VME bus lacks. Multiprocessing in the VME bus uses a shared memory mechanism that allows faults in one module to cause errors in other non-faulty modules by writing to their memories. We suggest the use of a message passing mechanism instead. The challenge is to support message-based inter-module communications using the available features provided by the VME boards and still detect errors and prevent fault propagation. Although the shared memory model is convenient for programmers, grouping data is a very common practice to enhance the system throughput. Thus, the use of massage passing will not introduce difficulties that affect the system implementation. In addition, techniques such as [13], [14] and [17] can be used to support the shared memory paradigm if necessary.

To support messages, a buffer is to be declared and dedicated to messages. A message buffer is the only globally accessible memory by other modules. In addition, access to a message buffer is restricted to read-only for modules that do not own that buffer. No board is supposed to write to the memory of other boards. If a master wants to send data to a slave, it writes a message into the master's message buffer. The slave then reads that message from the master memory and reacts to it. Defining a read-only VME global address window is a very common feature on VME boards.

A fault-tolerant message format can be imposed that contains the sender ID, receiver ID, error detection or correction code, and a message unique ID (if necessary). The sender should perform error detection and correction encoding. The receiver will check the contents of the message before reacting to it. The receiver can detect addressing errors in the message by verifying the sender ID and receiver ID. In addition, transmission errors can be detected or recovered using the information redundancy in the form of the error detection or correction code in the message.

Execution synchronization of the application tasks can be achieved either; by polling the message buffer of the sender for the required message, or by notifying the receiver by generating an interrupt[2] as soon as a message is being written by the sender in the designated address. The message ID can be useful to overcome race conditions if the receiver tried to read the message before it is ready which may be possible if the VME bus has a higher priority than the local bus. The message buffer can be partitioned for various boards and receivers can expect a unique location for their messages. The adopted application execution-synchronization mechanism is a designer decision.

Using this technique, errors in the sender can be isolated and prevented from propagating to the receiver because no write permission will be granted for a board to the memory of others. Errors in the message can be either in data, sender ID, receiver ID, Message ID, or message format. The receiver should be able to detect errors in the message body by validating the message format, error detection code, sender ID and the receiver ID. The message ID can be checked to guarantee the right message sequence (It may require knowledge of the application semantics). Any error in the message detected by the receiver will invalidate the entire message and a recovery action will be taken. An addressing fault in the receiver that may get it to read from the wrong board or the wrong address within the right board will invalidate the message format and/or the sender ID. Furthermore, the mapping of message buffers on the boards within the global address space of the VME system can be orchestrated so that an addressing error can not change a valid global address into another valid address. Maintaining a suitable hamming distance can guard the system against permanent or transient stuck failure of one or more address bits. Thus, the system will be functionally partitioned. Faults can be within the faulty module and will not affect other modules.

The following section addresses the implementation issues of such techniques in the currently available VME boards without imposing hardware changes.

2.3 Applicability of the Approach

As we discussed earlier, enhancing the fault containment capabilities and the support for strong functional partitioning on the VME bus should not

[2] Can be implemented by using the address-monitoring feature provided by the VME bus.

impose significant modifications to both the hardware manufacture and the application developer. Imposing significant changes may diminish the cost-effectiveness of the VME bus and may limit the number of vendors who adapt the modifications. This section presents the feasibility of the techniques illustrated in the previous section. After conducting a market survey, we found that the VMEchip2 [4], from Motorola, and the SVC64 [5], from Tundra, are the most commonly used VME bus interface chips on VME boards. We studied the applicability of these techniques in VME boards that use these interface chips.

Considering both interface chips, the strong partitioning protocol can be totally implemented in software. In fact, it is possible to use commercial-off-the-shelf (COTS) operating systems by extending the kernel service to include message handling. In addition, the generation and validation of the error detection code within the message can be included in the message handler. Thus, the applicability of the fault containment techniques depends on the feasibility of partitioning local memory and mapping the message buffer within the VME global address space.

Both the VMEchip2 and SVC64 provide capabilities for software configurable global addressing of the on-board memory of a module. There are software configurable map decoders that can be used to control the global addressing of the VME boards' local memory. Such a feature allows restricting addresses used by the other boards and any wrong addressing will return an error. Furthermore, both chips allow the programmer to restrict access, by other boards in the system, to this local memory to read-only. Thus, a faulty board cannot corrupt the memory of other healthy boards.

To prevent the possibility of other boards reading by mistake from the wrong board, one may use different combinations for the most significant bits for the VME address of each board so that the hamming distance will be more than 1 bit. Thus, we can guarantee that boards cannot read from the wrong board unless there is more than one transmission error. Given that the number of boards is limited (maximum 22 according to the IEEE standard [3]), it is possible to achieve a distance of at least 8 bits among board addresses and to isolate up to 8 simultaneous error. Reading from a non-existing board will be timed out by the VME bus and can be detected. Errors in the least significant 16 bits of the address can be detected by validating the messages. Reading from a different location within the sender message buffer will contain neither the right format nor the correct message semantic.

The strong partitioning protocol has been implemented and tested on boards that use the SVC64 chip. The next section provides details about the implementation and test environment.

3. Implementation and Test Environment

A proof-of-concept prototype has been built using COTS components, *Figure 1*. The prototype includes a VME backplane that hosts six VME-171 PowerPC 80 MHz processor boards from DY4 Systems. VxWorks, a real-time operating system from WindRiver Systems, is installed and integrated with the hardware. The strong partitioning inter-processor communication protocol has been implemented and integrated within VxWorks. Testing and fault injection experiments have been performed. In addition, the performance of multiprocessing communication using the protocol is measured. This section illustrates the implementation of the strong partitioning protocol in this test environment and describes the results of the experiments of fault injection and performance measurements.

Figure 1: A proof-of-concept demonstration prototype

3.1 Protocol Implementation

In a single processor system, tasks may establish communication among each other through the use of message queues. A message queue is an abstraction of a unidirectional channel (The choice of this abstraction is due to the fact that VxWorks applies the same technique), and is typically identified by a "queue ID" or QID. Two communicating tasks use the same QID to exchange messages, *Figure 2*. The delivery of the messages is handled by the Inter-Task Communication (ITC) Service (typically part of the operating system library functions) which may maintain several message

queues depending on how many communication channels are open. The ITC Service takes care of synchronization issues such as the mutually exclusive access to the message queue.

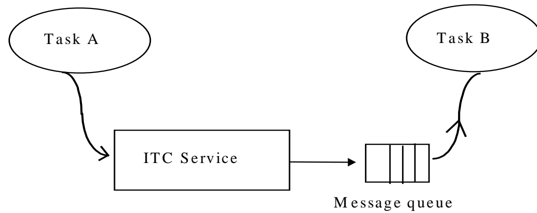

Figure 2: Inter-task communication Service

In order to achieve fault containment within a faulty board, the *inter-processor communication* (IPC) over a VME backplane is to follow the protocol presented in *Section* 2. Across board communication is to be accomplished through message queues that need to be allocated in a globally addressable read-only memory buffer where the sender will leave outgoing messages for other processor boards, or simply processors, to read. Although a separate queue for every pair of communicating tasks can be allocated, managing the message buffer becomes complicated if tasks are created dynamically. The message buffer usually cannot grow dynamically and there will be a potential that the system runs out of free memory space within the message buffer. Instead, queues are allocated to communicating boards. For each (sender, receiver) permutation of processor boards, there is a *processor queue* which is created within the global portion of the sender's main memory to hold any messages sent to that specific receiver by any task running on the sender board. Receivers have read-only privilege on the message queues owned by other processors. *Figure 3* illustrates the implementation of the protocol.

An IPC daemon is in charge of maintaining these queues. The IPC daemon is an independent task created by the system at initialization to handle inter-processor communication. When, say, task A on processor 1 needs to send a message to task B on processor 2, it contacts the local ITC service on processor 1, which recognizes that the target queue belongs to an external processor. Therefore, the message is inserted in the IPC *daemon queue*. The IPC daemon on processor 1 processes the buffered message by appending it to the processor queue associated with processor 2. Processor 2's IPC daemon is then notified that it has a message waiting inside processor 1. This kind of notification takes the form of generating a location monitor interrupt on the recipient board. The location monitor interrupt routine sends a message to the IPC daemon announcing that there is a message ready for delivery at processor board 1. The IPC daemon on processor 2 fetches the message from board 1 and notifies the IPC of processor 1 after successful completion. At that time, the IPC daemon of processor 1 deletes the message from its processor queue. Meanwhile, the IPC on processor 2 delivers the message to the ITC, which finally stores the message in the message queue, connected to Task B (This is a regular message queue maintained by the operating system, VxWorks in our implementation). The notification of successful message delivery takes the form of writing the message ID in a pre-defined location on the receiver VME-mapped memory to acknowledge the receipt of the message. The sender daemon polls for the message acknowledgement until the receiver writes the message ID.

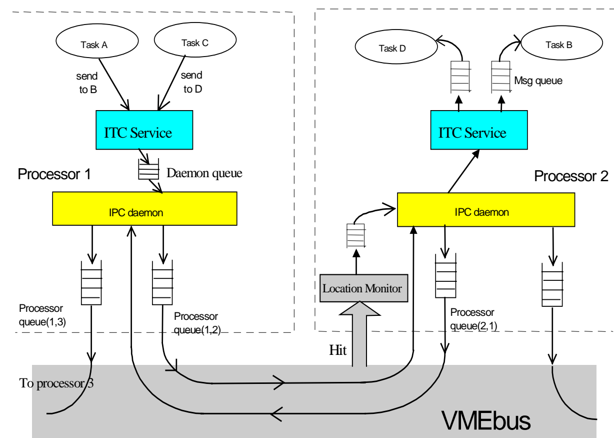

Figure 3: Implementation of the Strong Partitioning Protocol

It is worth noting that the ITC on each processor still handles local inter-task communication. It only contacts the IPC if the target receiver is on a different processor. Consequently, the inter-processor communication service is transparent to the system developer. However, a system configuration needs to be read by each board before starting the IPC daemon to resolve queue addresses and the VME global memory map. We should also note that even though there may be multiple communication sessions established between any two processors, there are only two processor queues carrying the communication; one queue for each direction. The sizes of the daemon queue and the processor queues depend on the amount of available memory and the expected communication traffic density. Moreover the IPC service, in the current implementation, has been

added as a library service and the IPC daemon was defined as a high priority user-task since the source code of VxWorks was not available. However, the protocol can be completely integrated within the operating system by defining the IPC daemon as a system-task and augmenting the library of system calls with the IPC library functions.

3.2 Fault injection Experiment

Several fault injection experiments were conducted to demonstrate the resiliency of the strong partitioning protocol. A test application was developed for a client-server application. The server continuously accepts messages with requests to perform arithmetic and logic operations. The client sends requests with two operands and an operation. The server handles the request and sends a message back with the result. All communication between the server and the client follow the strong partitioning protocol. A third board has been designated as a faulty board whose erroneous behavior interferes with the server and the client. The following three fault injection experiments were run:

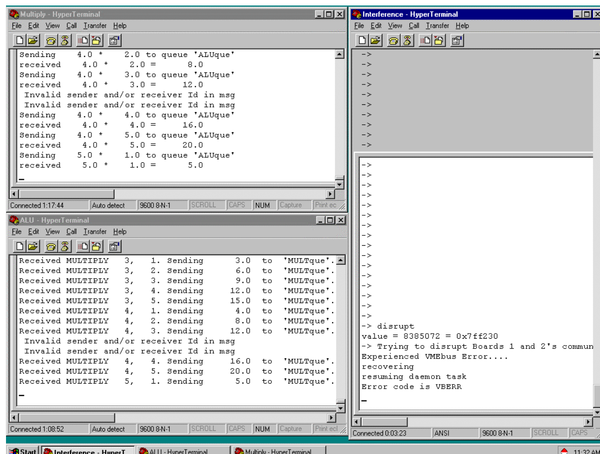

Figure 4: Resiliency to disruption

Experiment 1 – Disruption

While the client and the server are communicating, the faulty board sends bogus messages to both boards attempting to disrupt the communication. Both the client and the server acknowledge receipt of the bogus data by displaying the warning message “Invalid sender and/or receiver ID in msg.”, as shown in *Figure 4*. This experiment demonstrates the protocol ability to detect erroneous messages and to protect healthy boards from bogus traffic on the VME bus. Using shared memory; the faulty board would have easily corrupted the shared memory area without demonstrating odd behavior that other boards can detect as an error.

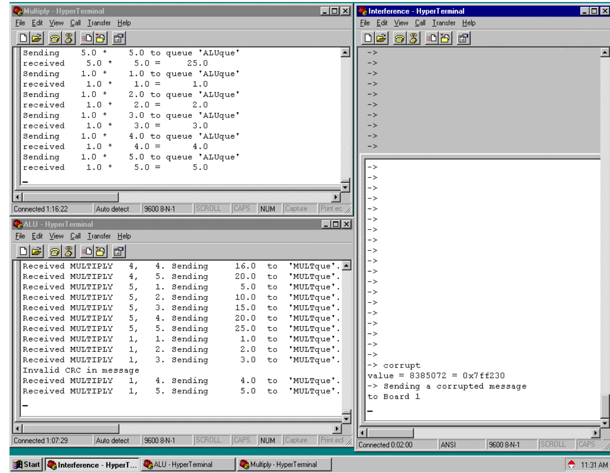

Figure 5: Detection of corrupted messages

Experiment 2 – Corruption

While the client and server are communicating, the faulty board sends a formatted (but corrupted) message to the server. The server detects the receipt of a corrupted message by validating the error detection code (CRC) and displays the error message “Invalid CRC in message”, as shown in *Figure* 5.

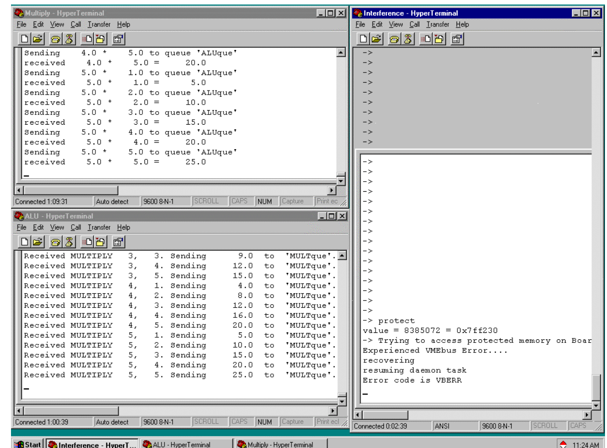

Figure 6: Protected memory experiment

Experiment 3 – Protection

While the client and server are communicating, the faulty board attempts to access protected memory on

both server and client boards. The attempted access of protected memory generates a VME bus error on the offending board, as shown in *Figure* 6. The client and the server continue their operation without interruption.

In addition to the above experiments a VME extender board has been used to allow the emulation of stuck faults in the data and address lines of the VME bus. A series of experiments were performed to test the resiliency of the protocol against stuck faults. The results of the experiments clearly showed the robustness of the protocol and its capabilities in preventing fault propagation.

3.3 Performance Analysis

To measure the performance of the protocol, we used a tool for monitoring embedded application, called "WindView", from WindRiver Systems. Using WindView a set of events can be defined. Programs are instrumented during compilation to time-stamp these events by inserting calls to some library functions. During the execution the marked event as well the time-stamps are stored as a RAM-file. WindView can be executed afterward to visualize the sequence of events and display the time stamps. Using WindView, one can achieve a time resolution of a microsecond with a maximum of 4% intrusion (due to the overhead needed for time stamping). To measure the performance of the protocol, we instrumented the client-server application discussed above. Since the application programs are running on separate boards without clock synchronization, we time-stamped events on each board independently and measured the elapsed time between events. Since both the server and client programs are running on exactly similar boards, we did not need to perform any calibration for the speed of the processors. The end-to-end delivery time for messages of length of 256 bytes has been measured. An information redundancy (sender ID, receiver ID, etc.) of 32 bytes is included in the massage. *Table* 1 shows the time needed to perform each step. It should be noted that the values reported in *Table* 1 are the average of 50 independent readings. The end-to-end message delivery time is 302.3 μ sec (209.9 + 92.4).

We performed two additional experiments to compare the performance of the protocol with other mechanisms. In the first experiment we performed a memory write of 256 bytes from one board to the other using assembly instruction. The time for writing 256 bytes was measure using a logic analyzer and found to be 144 μ sec. This value is consistent with the message transfer time measured by WindView in *Table 1*, noting that the actual size of data transfer is larger by 32 bytes. This experiment provided an upper bound for the bus transfer rate and validated the accuracy of the measurements obtained using WindView. It should be noted that this upper bound could be only achieved using assembly code. Today's embedded applications are so complex that assembly programming can no longer be used in developing these applications. Support for multiprocessing and inter-process communication is always needed from an operating system.

The second experiment considered the operating system overhead to maintain mutual exclusive access to shared data. The client-server application has been rewritten using VxMP, which is a multiprocessor extension of VxWorks. VxMP provides an abstraction of shared memory so that applications can refer to the shared object by name. VxMP only resolves the addresses for the programmer without maintaining mutual exclusion for accessing the shared memory relying on hardware support for that. Obviously VxMP provides only shared memory management without fault containment. *Table 2* shows the performance results for sharing 256 bytes data. It should be noted that VxMP requires the shared memory to be physically located on only one board. Thus transferring data between boards usually requires two bus transmissions; first the sender writes to the shared memory, the second transmission occurs when the receiver reads that data from the shared memory. The end-to-end time for sending 256 byte using VxMP is 433 μ sec (217.6 + 215.4). Comparing the two tables, it is clear that the protocol not only maintains fault containment with reasonable performance penalty but it also out-performs other multiprocessor support tools currently available in the market.

Action Description	Duration in μsec
Receive One Message	
Handle location monitor interrupt	37.6
Fetch message across the VME bus	170.0
Validate and acknowledge message	2.3
Total for receive	**209.9**
Send One Message	
Get message from daemon queue	49.1
Put message in processor queue	42.1
Notify receiver	1.2
Total for Send	**92.4**

Table 1: Performance measurements for the partitioning protocol

It should be noted that the current implementation of the protocol is to prove the concept. There are many opportunities for optimization, which significantly enhance the performance. In addition, we noticed that significant performance improvement could be achieved if multiple messages are sent or received in one cycle of the IPC daemon. Since most of the advanced processors, such as the PowerPC, have instructions cache, the daemon code is usually fetched to this cache upon activation of the daemon. Thus, the first send or receive operation always suffers a cache miss. We observed about 35% improvement in the message handling, not including the time for data transfer over the VME bus, after the first sending or receiving activity. Moreover, the strong partitioning protocol proved to be very useful in debugging and system integration since it defines clear boundaries that facilitate detection of programming errors.

Action Description	Duration in μsec
Receive One Message	
Handle location monitor interrupt	30.4
Read message across the VME bus	187.2
Total for receive	**217.6**
Send One Message	
Put message into shared memory	213.4
Notify receiver	2.0
Total for Send	**215.4**

Table 2: Performance measurements for VxMP

4. Related Work

This paper proposes a mechanism, which is fully implementable in software and integrable with the operating system, to support message passing on shared memory multiprocessor architecture. On the contrary, there is plenty of work on supporting shared memory on a loosely coupled system, which is referred to as *distributed shared virtual memory* [7]. Examples of this include the TreadMarks system [13] and Mirage [17]. A survey for different approaches can be found in [14]. To support recovery from a failure on distributed shared virtual memory system; a checkpointing mechanism is often introduced [15][16]. During normal fault-free operation, snapshots of the system state (registers and memory contents) are stored in a repository to assist the applications recover from a failure. The reader is referred [6] to for a survey of various recovery techniques and strategies.

Achieving fault containment in shared memory multiprocessor systems has been the subject of intense study at the Computer System laboratory at Stanford for the FLASH multiprocessor architecture [9],[10] and [11]. The approach taken is to partition the multiprocessors into cells. Every cell controls a portion of the hardware and shared resources. A cell may include one or multiple processors. Each cell runs an independent multiprocessor operating system. A hardware solution has been suggested to protect a cell from a wild-write by any faulty cell. The operating system continuously runs across-cell checks for failure detection and a distributed recovery routine needs to be executed upon the detection of a fault. Our approach is different in that we rely on the memory management unit to protect boards from wild-write since we cannot introduce any hardware change. We prevent across-board writing and consequently restrict inter-processor communication only to massage passing and do not support memory sharing. However, we do not see that as a drawback since virtual memory is usually not recommended in mission critical real-time applications due to its negative impact on system predictability [12]. In addition, our fault detection is quite simple. We rely on timing out messages, a feature, which is built in the VME bus interface logic. Furthermore, the strong partitioning protocol, presented above, is easy to implement and integrate with COTS single processor operating systems.

5. Conclusion

In this paper, it has been shown that the VME bus can be strongly partitioned with fault isolation capabilities that prevent fault propagation to healthy boards. A multiprocessor message passing communication protocol has been presented. The message passing protocol provides fault containment over the bus. Messages can be verified with respect to sender and receiver IDs as well as the message ID. In addition, information redundancy in the form of error detection and correction code can be provided within the message to verify data transmission over the bus.

The strong partitioning protocol has been implemented in software without imposing any hardware modifications. A proof-of-concept prototype has been built, in which the strong partitioning protocol has been integrated with COTS real-time operating system. The protocol has been added as an operating system service, which is transparent to programmers. The fault injection

experiments demonstrated the efficiency of the approach in fault containment.

Although the increased data transfer due to message overhead and information redundancy will affect the performance of the VME bus, it has been shown that the burden on the bus bandwidth is limited. Clearly the longer the data size within the message, the more efficient the strong partitioning protocol. In the future, we would like to extend our approach to provide support for programs that use the shared memory paradigm in order to facilitate the integration of legacy code.

The technology presented in this paper is crucial for multiple space and avionics programs within AlliedSignal, Inc. The vehicle management computer (VMC) for an unmanned space shuttle is currently under development. The VMC integrates multiple mission critical control modules such as mission manager, flight control and vehicle subsystems manager with other less critical modules within a VME cabinet. It is extremely important to contain any fault within the faulty module and protect other modules in the system. The integration of utility control system is another project that benefits from this technology. The goal is to integrate multiple utility control units, which are currently federated on the aircraft, into one cabinet. Utility controls such as electric power system, cabinet pressure and bleed air system are regarded as medium-criticality control functions on the aircraft. However, integrating multiple utility control modules makes the integrated unit highly critical since it may affect the safety of the flight. The strong partitioning protocol is to be used for handling inter-module communication to prevent fault propagation and protect the system from a complete failure when one module fails.

Acknowledgement

Thanks are due to Billy He, Eric Burton, Tim Steele and Chris Dailey for their help in the implementation. The authors are also indebted to Dr. Yang-Hang Lee, Dr. Dar-Tzeng Peng and members of the reliable systems group at AlliedSignal for their constructive critiques.

References

[1] "Design Guide for Integrated Modular Avionics", ARINC report 651, Published by Aeronautical Radio Inc., Annapolis, MD, November 1991.

[2] "Backplane Data Bus", ARINC Specification 659, Published by Aeronautical Radio Inc., Annapolis, MD, December 1993.

[3] "IEEE Standard for a Versatile Backplane Bus: VME bus", std 1014-1987, Published by The Institute of Electrical and Electronics Engineers, New York, NY, March 1988.

[4] "Motorola MVME162LX Embedded Controller Programmer's Reference Guide", Motorola, Inc. Computer Group, Tempe, Arizona.

[5] "Tundra Universe SCV64 Trooper II VME bus Interface Component Manual", Tundra, Kanata, Ontario, Canada.

[6] C. Morin and L. Puaut, "A Survey of Recoverable Distribute Shared Virtual Memory Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No 9, pp. 959-969, September 1997.

[7] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems", *ACM Transactions on computer Systems*, Vol. 7, pp. 321-357, November 1989.

[8] M. Younis and J. Zhou, "A VME-based Fault Tolerant Architecture", *Technical Report*, AlliedSignal, Inc., 1997.

[9] D. Teodosiu, et al., "Hardware fault Containment in Scalable Shared-Memory Multiprocessors", in the *Proceedings of the 24th ACM International Symposium on Computer Architecture (ISCA-24)*, pp. 73-84, June 1997.

[10] J. Chapin, et al., "Hive: Fault Containment for Shared-Memory Multiprocessors", in the *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, pp. 15-25, December 1995.

[11] M. Rosenblum, et al., "Implementing efficient fault containment for multiprocessors: confining faults in a shared-memory multiprocessor environment", *Communications of the ACM*, 39(9), pp. 52-61, September 1996.

[12] W. Halang and A. Stoyenko, *Constructing Predictable Real-Time Systems*. Boston-Dordrecht-London: Kluwer Academic Publishers, 1991.

[13] C. Amza, et al. "TreadMarks: Shared Memory Computing on Networks of Workstations", *IEEE Computer*, Vol. 29, No 2, pp. 18-28, February 1996.

[14] M. Stumm and S. Zhou, "Algorithms Implementing Distributed Shared Memory", *IEEE Computer*, Vol. 23, No. 5, pp. 54-64, May 1990.

[15] B. Fleisch, "Reliable Distributed Shared memory", in the *Proceedings of the 2nd Workshop on Experimental Distributed Systems*, pp. 102-105, 1990.

[16] M. Stumm and S. Zhou, "Fault Tolerant Distributed Shared Memory Algorithms", in the *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pp. 719-724, December 1990.

[17] B. Fleisch and G. Popek, "Mirage: A Coherent Distributed Shared Memory Design", in the Proceedings of the 12th ACM Symposium on Operating Systems Principles, published in *Operating Systems Review* 23(5) Special Issue, pp. 211-223, December 1989.

[18] J. Keller, "Avionics innovation marks new space shuttle", *Military & Aerospace Electronics*, pp. 1-7, Vol. 8, No. 4, April 1997.