Software-Based Failure Detection and Recovery in Programmable Network Interfaces

Yizheng Zhou, Vijay Lakamraju, Israel Koren, *Fellow*, *IEEE*, and C. Mani Krishna, *Senior Member*, *IEEE*

Abstract—Emerging network technologies have complex network interfaces that have renewed concerns about network reliability. In this paper, we present an effective low-overhead fault-tolerant technique to recover from network interface failures. Failure detection is based on a software watchdog timer that detects network processor hangs and a self-testing scheme that detects interface failures other than processor hangs. The proposed self-testing scheme achieves failure detection by periodically directing the control flow to go through only active software modules in order to detect errors that affect instructions in the local memory of the network interface. Our failure recovery is achieved by restoring the state of the network interface using a small backup copy containing just the right amount of information required for complete recovery. The paper shows how this technique can be made to minimize the performance impact to the host system and be completely transparent to the user.

Index Terms—Programmable Network Interface Card (NIC), Single Event Upset (SEU), radiation induced faults, failure detection, failure recovery, self-testing.

1 Introduction

Nowadays, interfaces with a network processor and large local memory are widely used [16], [18], [19], [20], [21], [22], [23]. The complexity of network interfaces has increased tremendously over the past few years. A typical dual-speed Ethernet controller uses around 10,000 gates, whereas a more complex high-speed network processor such as the Intel IXP1200 [24] uses more than 5 million transistors. As transistor counts increase, single bit upsets from transient faults, which arise from energetic particles such as neutrons from cosmic rays and alpha particles from packaging material have become a major reliability concern [1], [2], especially in harsh environments [3], [4] such as deep space. The typical fault rate in deep space for two Myrinet Network Interface Cards (NICs) is 0.35 faults/hour [4]. When a solar flare is in progress, the fault rate in interplanetary space can be as great as 6.87 faults/hour for two Myrinet NICs [4]. These also affect systems on earth, especially far away from the equator [5]. Because this type of fault does not reflect a permanent failure of the device, it is termed soft. Typically, a reset of the device or a rewriting of the memory cell results in normal device behavior thereafter. Soft-error-induced network interface failures can be quite detrimental to the reliability of a distributed system. The failure data analysis reported in [6] indicates that network-related problems contributed to approximately 40 percent of the system failures observed in distributed environments. As we will see in the following sections, soft errors can cause the network interface to completely stop responding, function improperly, or greatly reduce network performance. Quickly detecting and recovering from such failures is therefore crucial for a system requiring high reliability. We need to provide fault tolerance for not only the hardware in the network interface but also its local memory where the network control program (NCP) resides.

In this paper, we present an efficient software-based fault-tolerant technique for network failures. Software-based fault tolerance approaches allow the implementation of dependable systems without incurring the high costs resulting from designing custom hardware or using massive hardware redundancy. However, these approaches impose some overhead in terms of reduced performance and increased code size: it is important to ensure that this overhead have a minimal performance impact.

Our failure detection is based on a software-implemented watchdog timer to detect network processor hangs, and a software-implemented concurrent self-testing technique to detect other failures. The proposed self-testing scheme detects failures by periodically directing the control flow to go through program paths in specific portions of the NCP in order to detect errors that affect instructions or data in the local memory, as well as other parts of the network interface. The key to our technique is that the NCP is partitioned into various logical modules and only active logical modules are tested, where an active logical module is the collection of all basic blocks that participate in providing a service to a running application. When compared with testing the whole NCP, testing only active logical modules can limit significantly the impact on application performance while still achieving good failure

- *Y. Zhou is with the University of Massachusetts, 310 Knowles Engineering Bldg., 151 Holdsworth Way, Amherst, MA 01003-9284. E-mail: yzhou@ecs.umass.edu.*
- *V. Lakamraju is with the United Technologies Research Center, 411 Silver Lane, East Hartford, CT 06108. E-mail: LakamrVR@utrc.utc.com.*
- *I. Koren and C.M. Krishna are with the University of Massachusetts, 309K Knowles Engineering Bldg., 151 Holdsworth Way, Amerhst, MA 01003-9284. E-mail: krishna@ecs.umass.edu.*

Manuscript received 24 Oct. 2005; revised 1 Sept. 2006; accepted 16 Feb. 2007; published online 13 Apr. 2007.
Recommended for acceptance by J. Hou.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0447-1005.
Digital Object Identifier no. 10.1109/TPDS.2007.1093.

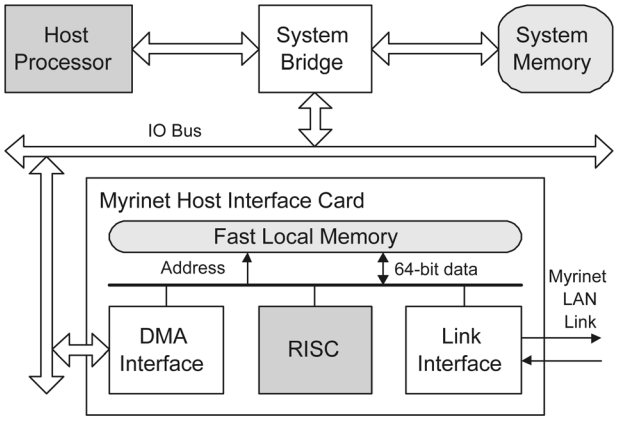

Fig. 1. Simplified block diagram of the Myrinet NIC.

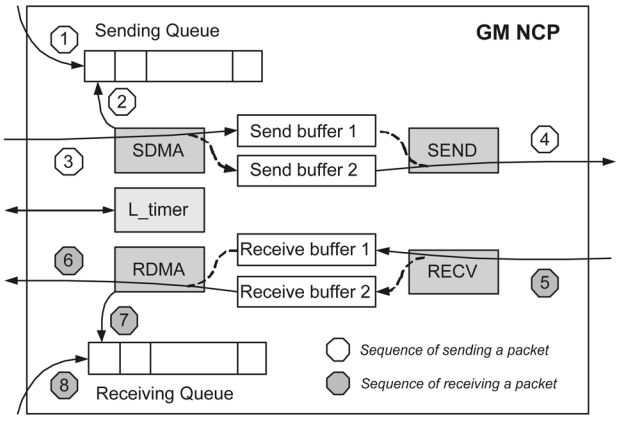

Fig. 2. Simplified view of the GM NCP.

detection coverage. When a failure is detected by the watchdog timer or the self-testing, the host system is interrupted, and a Fault Tolerance Daemon (FTD) is woken up to start a recovery process [7].

The central philosophy behind our failure recovery is to save enough network-related state information in the host so that the state of the network interface can be correctly reestablished in the case of a failure. Clearly, the challenge here is to provide for this "checkpointing" with as little performance degradation as possible. In our technique, the "checkpointing" is a continuous process in which the applications make a copy of the required state information before sending the information to the network interface and update it when the network notifies the application that the state information is no longer required. As the numerical results will show, such a scheme greatly reduces the impact on the normal performance of the system.

In this paper, we show how the proposed failure detection and recovery techniques can be made completely transparent to the user. We demonstrate these techniques in the context of Myrinet, but as we will see, the approaches are generic in nature and are applicable to many modern networking technologies.

2 Myrinet: An Example Programmable Network Interface

Myrinet [16] is a high-bandwidth $(2Gbit/sec)$ and low-latency $(\sim 6.5\mu s)$ local area network technology. A Myrinet network consists of point-to-point full-duplex links that connect Myrinet switches to Myrinet host interfaces and other switches.

Fig. 1 shows the organization and location of the Myrinet NIC in a typical architecture. The card has an instruction-interpreting reduced instruction set computer (RISC) processor, a direct memory access (DMA) interface to/from the host, a link interface to/from the network, and a fast local memory (Static random access memory; SRAM), which is used for storing the Myrinet's NCP and for packet buffering. The Myrinet's NCP is responsible for buffering and transferring messages between the host and the network and providing all network services.

Basic Myrinet-related software is freely available from Myricom [17]. The software, called GM, includes a driver for the host OS, the Myrinet's NCP (GM NCP), a network mapping program, a user library and Application Program Interfaces (APIs). It is the vulnerability to faults in the GM NCP that is the focus of this work, so we now provide a brief description of it.

The GM NCP [17] can be viewed broadly as consisting of four interfaces: Send DMA (SDMA), SEND, Receive (RECV) and Receive DMA (RDMA), as depicted in Fig. 2. The sequence of steps during sending and receiving is illustrated in Fig. 2. When an application wants to send a message, it posts a send token in the sending queue (Step 1) through GM API functions. The SDMA interface polls the sending queue and processes each send token (Step 2) that it finds. It then divides the message into chunks (if required), fetches them via the DMA interface, and puts the data in an available send buffer (Step 3). When the data is ready in a send buffer, the SEND interface sends it out, prepending the correct route at the head of the packet (Step 4). Performance is improved by using two send buffers: while one is being filled through SDMA, the packet interface can send out the contents of the other buffer.

Similarly, two receive buffers are present. One of the receive buffers is made available for receiving an incoming message by the RECV interface (Step 5), whereas the other could be used by RDMA to transfer the contents of a previously received message to the host memory (Step 6). The RDMA then posts a receive token into the receiving queue of the host application (Step 7). A receiving application on the host asynchronously polls its receiving queue and carries out the required action upon the receipt of a message (Step 8).

The GM NCP is implemented as a tight event-driven loop. It consists of around 30 routines. A routine is called when a given set of events occur and a specified set of conditions are satisfied. For example, when a send buffer is ready with data and the packet interface is free, a routine called *send_chunk* is called. It is also worth mentioning here that a timer routine (*L_timer*) is called periodically, when an interval timer present on the interface card expires.

Flow control in GM is managed through a token system. Both sends and receives are regulated by implicit tokens,

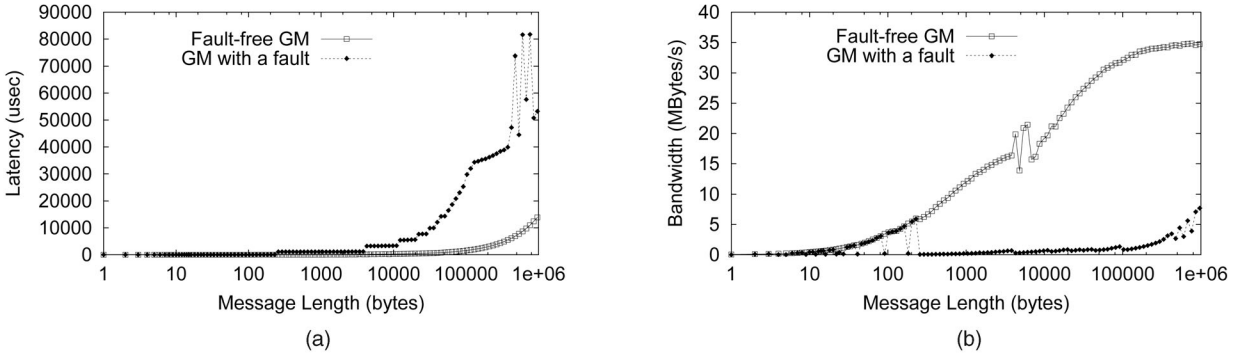

Fig. 3. Examples of fault effects on Myrinet's GM. (a) Unusually long latencies caused by a fault. (b) Bandwidth reduction caused by a fault.

which represent space allocated to the user process in various internal GM queues. A send token consists of information about the location, size, and priority of the send buffer and the intended destination for the message. A receive token contains information about the receive buffer such as its size and the priority of the message that it can accept. A process starts out with a fixed number of send and receive tokens. It relinquishes a send token each time it calls GM to send a message and a receive token with a call to GM to receive a message. A send token is implicitly passed back to the process when a callback function is executed upon the completion of the sending, and a receive token is passed back when a message is received from the receive queue.

3 Failure Detection

In the context of the Myrinet card, soft errors in the form of random bit flips can affect any of the following units: the processor, the interfaces, and more importantly, the local SRAM, containing the instructions and data of the GM NCP. Bit flips may result in any of the following events:

- *Network interface hangs*. The entire network interface stops responding.
- *Send/Receive failures*. Some or all packets cannot be sent out or cannot be received.
- *DMA failures*. Some or all messages cannot be transferred to or/and from host memory.
- *Corrupted control information*—A packet header or a token is corrupted.
- *Corrupted messages.*
- *Unusually long latencies*.

The above list is not comprehensive. For example, a bit flip occurring in the region of the SRAM corresponding to the resending path will cause a message to not be resent when a corresponding acknowledgment was not received. Experiments also reveal that faults can propagate from the network interface and cause the host computer to crash. Such failures are outside the scope of this paper and are the subject of our current ongoing research.

Fig. 3 shows how a bit-flip fault may affect message latency and network bandwidth. The error was caused by a bit flip that was injected into a sending path of the GM NCP. More specifically, one of the two sending paths associated with the two message buffers was impacted, causing the effective bandwidth to be greatly reduced. To achieve reliable in-order delivery of messages, the GM NCP generates more message resends, and this greatly increases the effective latency of messages. Since no error is reported by the GM NCP, all host applications will continue as if nothing happened. This can significantly hurt the performance of applications, and in some situations, deadlines may be missed.

Some of the effects of soft-error-induced bit flips are subtle. For example, although cyclic-redundancy checks (CRC) are computed for the entire packet, including the header, there are still some faults that may cause data corruption. When an application wants to send a message, it builds a send token containing the pointer to the message and copies it to the sending queue. If the pointer is affected by a bit flip before the GM NCP transfers the message from the host, an incorrect message will be sent out. Such errors are difficult to detect and are invisible to normal applications.

Even though the above discussion was related to Myrinet, we believe that such effects are generic and apply to other high-speed network interfaces having similar features, that is, a network processor, a large local memory and an NCP running on the interface card. We detail our approach in Section 3.1.

3.1 Failure Detection Strategy

Our approach to detecting interface hangs is based on a simple watchdog, but one which is implemented in software and uses the low-granularity interval timers present in most interfaces.

Since the code size of the NCP is quite large, it is challenging to efficiently test this software to detect noninterface-hang failures. We exploit the fact that applications generally use only a small portion of the NCP. For instance, the GM NCP is designed to provide various services to applications, including reliable ordered message delivery (*Normal Delivery*), directed reliable ordered message delivery that allows direct remote memory access (*Directed Delivery*), unreliable message delivery (*Datagram Delivery*), setting an alarm, etc. Only a few of the services are concurrently requested by an application. For example, *Directed Delivery* is used for tightly coupled systems, whereas *Normal Delivery* has a somewhat larger communication overhead and is used for general systems; it is rare for an application to use both of them. Typically, an application only requests one transport service out of the seven types of transport services provided by the GM NCP.

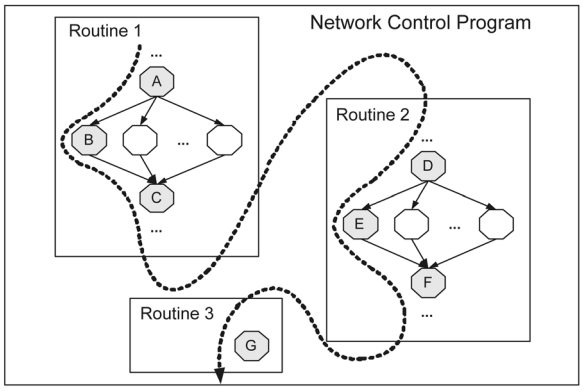

Fig. 4. Logical modules and routines.

Consequently, only about 10 percent to 20 percent of the GM NCP instructions are "active" when serving a specific application. Other programmable NICs such as the IBM PowerNP [18] have similar characteristics.

Based on this observation, we propose to test the functionalities of only that part of the NCP that corresponds to the services currently requested by the application: this can considerably reduce failure detection overhead. Moreover, because a fault affecting an instruction that is not involved in serving requests from an application would not change the final outcome of the execution, our scheme avoids signaling these harmless faults. This reduces significantly the performance impact, compared to other techniques such as those that periodically encode and decode the entire code segment [13].

To implement this failure-detection scheme, we must identify the "active" parts of the NCP for a specific application. To assist the identification process, we partition the NCP into various logical modules based on the type of services they provide. A logical module is the collection of all basic blocks that participate in providing a service. A basic block, or even an entire routine, can be shared among multiple logical modules. Fig. 4 shows a sample NCP that consists of three routines. The dotted arrow represents a possible program path of a logical module, and an octagon represents a basic block. All the shaded blocks on the program path belong to the logical module. In our implementation, we examined the source code of the GM NCP and followed all possible control flows to identify the basic blocks of each logical module. This time-consuming analysis has been done manually, but could be automated by using a code-profiling tool similar to GNU *gprof*.

For each of the logical modules, we must choose and trigger several requests/events to direct the control flow to go through all its basic blocks at least once in each self-testing cycle so that the functionality of the network interface is tested and errors are detected. For example, in Myrinet interfaces, large and small messages would direct the control flow to go through different branches of routines because large messages would be fragmented into small pieces at the sender side and assembled at the receiver side, whereas small messages would be sent and received without the fragmenting and assembling process. We use loopback messages of various sizes to test the sending and receiving paths of the NCP concurrently. During this procedure, the hardware of the network interface involved in serving an application is also tested for errors. The technique can, in addition, be used to test other services provided by network interfaces such as setting an alarm by directing the control flow to go through basic blocks providing these services. Such tests are interleaved with the application's use of the network interface.

To reduce the overhead of self-testing, we implement an Adaptive and Concurrent Self-Testing (ACST) scheme. We insert a piece of code at the beginning of the NCP to identify the requested types of services and start self-testing for the corresponding logical modules. The periodic self-testing of a logical module should start before it serves the first request from the application(s) to detect possible failures; this causes a small delay for the first request. For a low-latency NIC such as Myrinet, this delay would be negligible. Furthermore, we can reduce this delay by letting the application packets follow on the heels of the self-testing packets. If a logical module is idle for a given time period, the NCP would stop self-testing it. A better solution can be achieved by letting the NCP create lists for each application to track the type of services it has requested so that when an application completes and releases network resources, which can be detected by the NCP, the NCP could check the lists and stop the self-testing for the logical modules that provide services only to this completed application.

3.2 Implementation

The software-implemented watchdog timer makes use of a spare interval timer to detect interface hangs. One of them, say IT1, is first initialized to a value just slightly greater than $800\mu s$, which is the maximum time between the *L_timer* routine invocations during normal operation. The *L_timer* routine is modified to reset IT1 whenever it is called. The interrupt mask register provided by the Myrinet NIC is modified to raise an interrupt when IT1 expires. Thus, during normal operation, *L_timer* resets IT1 just in time to avoid an interrupt from being raised. When the NIC crashes/hangs, the *L_timer* routine is not executed, causing IT1 to expire and an interrupt to be raised, signaling to the host that something may be wrong with the network interface. Such a scheme allows the host to detect NIC failures with virtually no overhead.

This detection technique works as long as a network interface hang does not affect the timer or the interrupt logic. This is supported by our experiments: over an extensive period of testing, we did not encounter a single case of a fault that has affected the timer or the interrupt logic. In fact, this simple failure detection mechanism was able to detect all the interface hangs in our experiments. Although it is not impossible that a fault might affect these circuits, our experience has shown this to be extremely unlikely.

In what follows, we demonstrate and evaluate our self-testing scheme for one of the most frequently used logical modules in the GM NCP, the *Normal Delivery* module. Other modules have a similar structure with no essential difference, and the self-testing of an individual logical module is independent of the self-testing of other modules.

To check a logical module providing a communication service, several loopback messages of a specific bit pattern

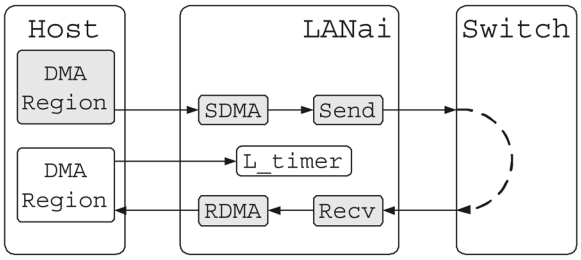

Fig. 5. Data flow of self-testing.

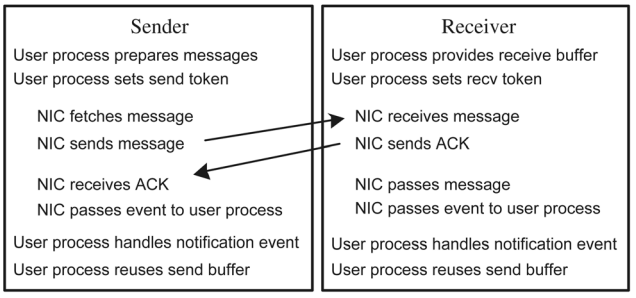

Fig. 6. A typical control flow.

are sent through the DMA and link interfaces and back so that both the sending and receiving paths are checked. Received messages are compared with the original messages, and the latency is measured and compared with normal latencies. If all of the loopback messages are received without errors and without experiencing unusually long latencies, we conclude that the network interface works properly.

We have implemented such a scheme in the GM NCP. We emulate normal sending and receiving behavior in the *Normal Delivery* module. This is done by posting send and receive tokens into the sending and receiving queues, respectively, from within the network interface, rather than from the host. The posting of the tokens causes the execution control to go through basic blocks in the corresponding logical module, so that errors in the control flow path are detected. Similarly, some events such as message loss or software-implemented translation look-aside buffer misses, which might concurrently happen during the sending/receiving process of the *Normal Delivery* module, are also triggered within the NIC to test the corresponding basic blocks. We can emulate different sets of various requests/events to go through most of the basic blocks. To reduce the overhead, we made an attempt to trigger as few requests/events as possible.

Fig. 5 shows the data flow of the self-testing procedure. When the GM driver is loaded, two extra DMA regions are allocated for self-testing purposes. The shaded DMA region is initialized with predefined data. We added some code at the end of the timer routine (*L_timer*) to trigger requests/events for each self-testing cycle. The SDMA interface polls the sending queues, and when some tokens for self-testing are found, the interface starts to fetch the message from the initialized DMA region and passes chunks of data to the SEND interface. For our self-testing, messages are sent out by the SEND interface to the RECV interface at the same node. Then, messages are transferred to the other DMA region. Finally, after a predetermined interval, when an *L_timer* is called, messages are transferred back to the network interface. During this procedure, we can check the number of received messages, messages' contents, and latencies. Such a design insures that both directions of the DMA interface and link interface are tested, as well as the network processor and NCP. Note that such a scheme does not interact with the host processor and hence has minimal overhead. Because the size of the self-testing code is negligible when compared with the size of the GM NCP, the performance impact is minor.

Self-testing can also be implemented using an application running in the host with no modification to the GM NCP. Such an implementation would impose an overhead to the host system that we avoid with our approach. Also, a pure application-level self-testing would be unable to test some basic blocks that would otherwise be tested with our self-testing implemented in the GM NCP, such as the resending path, because of its inability to trigger such a resending event.

Clearly, it is only when the injected faults manifest themselves as errors that this approach can detect them. Faults that are "silent" and simply lurk in the data structures would require a traditional redundancy approach, which is outside the scope of our work.

Since all the modifications are within the GM NCP, the API used by an application is unchanged so that no modification to the application source code is required.

4 Failure Recovery

Recovery from a network interface failure primarily involves restoring the state of the interface to what it was before the failure. However, simply resetting the interface, reloading and restarting the NCP would not be sufficient as it can cause duplicate messages to be received or messages to be lost.

In the context of Myrinet, reliable transmission is achieved through the use of sequence numbers. These numbers are maintained solely by the GM NCP and are therefore transparent to the user. If the NCP is reloaded and restarted upon a failure, the state of the connections and the sequence numbers are lost, and messages cannot be retransmitted reliably. To illustrate this, Fig. 6 shows the schematic of a typical control flow in a GM application, whereas Fig. 7 shows what might happen if a fault occurs and the NCP is reloaded. Suppose that the sending node crashes when an ACK is in transit. After recovering from the failure, since all state information is lost, the sender may try to resend the message with an invalid sequence number. The receiver would reply by sending a negative acknowledgment (NACK) with the expected sequence number. At this point, if the sender resends the message with this sequence number, the receiver would incorrectly accept a duplicate message. This problem arises due to the lack of redundant state information. If information concerning all streams of sequence numbers was stored in some stable storage, the GM NCP could then use this information during recovery to send out messages with the correct

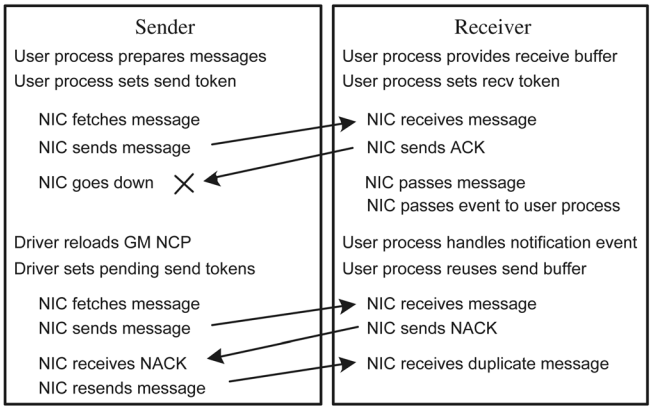

Fig. 7. The case of duplicate messages.

sequence numbers and avoid duplicate messages. The key, however, is to manage such information redundancy so that the performance of the network is not greatly impacted.

Messages could also be lost. Suppose the faulty node is a receiver. Then, there is not much state information that needs to be restored, because the Myrinet programming model is connectionless in that the sender does not explicitly set up a connection with the receiver. The receiver in GM sends out an acknowledgment (ACK) as soon as it receives a valid message. This can lead to a faulty behavior, as shown in Fig. 8. Consider the case when the NIC crashes after the send of the ACK is complete but before the entire message has been transferred to the host memory. This can happen if the DMA interface is not free, and so the DMA operation is delayed. The receiver will never receive that message again because, as far as the sender is concerned, it received the ACK for the message and notified the application that the send was successful. The sender would not resend the message and so, as far as the receiver is concerned, the message is lost forever. This problem arises because of the lack of a proper commit point for a send-receive transaction. The receiver should send out an ACK only when the message has been copied to its final destination.

Even though these problems were observed in the context of Myrinet, similar problems would very likely happen in other programmable network interfaces.

4.1 Recovery Strategy

The above discussion indicates that reloading the NCP alone does not guarantee correct recovery. What is required is to restore the state of the network interface to a point that guarantees the correct handling of future messages, as well as messages in flight at the time of failure.

Since we are considering only network interface failures, a sufficiently safe place for storing the required network interface state is the host's memory. To tolerate interface failures, the host should duplicate all the state information and messages in the host memory with regard to all outstanding sending and receiving events before the next operation in the control flow until the corresponding events finish, irrespective of whether or not the same information has already been present in the network interface. For example, when the network interface on the receiver side

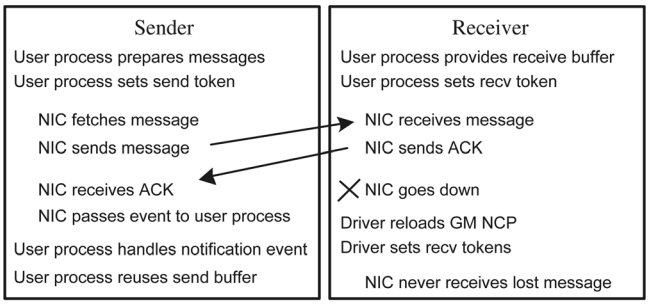

Fig. 8. The case of lost messages.

receives a message, it should first transfer the message to its host and then send an ACK to the sender. This ensures that we can have a copy of the message in the host memory; if a failure happens during the receiving process, we can find a commit point during recovery, that is, both sender and receiver agree on which packet has been delivered. The implementation may vary from interface to interface, but the basic idea mentioned here should be the same for all programmable NICs. The challenge, however, is in recognizing the minimal necessary information to set a recovery point for communication pairs to avoid message duplication and message loss.

Our recovery scheme works for all the failures mentioned in the section on failure detection, except for those causing the corruption of data and control information. This is because of the relatively long failure detection latency in these cases. For such failures, before the recovery process starts, the corrupted data or information may have already been passed to the host application. To account for these failures, we could use our fault-tolerant scheme in conjunction with the checkpointing of the application. When a corruption failure is detected, we could first reset the NIC and then roll back the host application to the last checkpoint. As we will see in the following section, the fraction of corruption failures is small, so we can still achieve fast recovery in most cases. Since the checkpointing of host applications has been widely studied, we will not discuss it here.

4.2 Implementation

Apart from the sequence numbers, it is also important to keep a copy of the send and receive tokens. As discussed earlier, a process implicitly relinquishes a send token (and passes it to the interface) when a call to a GM API function is made to send a message and gets it back when the send is successfully completed. A send token consists of information about the location, size, and priority of the send buffer and the intended destination for the message. It is important to keep an updated copy of all the send tokens that are in possession of the interface so that this information can be used during failure recovery to resend the messages that have yet to be acknowledged. Similar is the case with the receive tokens. Keeping a copy of the forfeited receive tokens allows us to notify the Myrinet interface of all the pinned-down DMA regions that have not yet been filled by the interface.

In our implementation, extra space is allocated by the user process to maintain a copy of the send token queue and

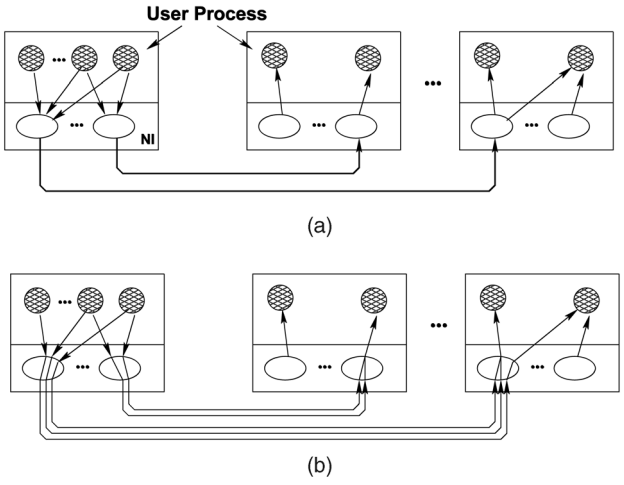

Fig. 9. (a) All Streams are multiplexed into a single connection. (b) Independent streams per connection.

the receive token queue. When a call to any of the GM API functions is made to send a message, a copy of the send token is added to the queue. Since the size of a token is small, the overhead is insignificant. The process also stores a copy of the receive token when it provides receive buffers. The host also needs to have a copy of the sequence numbers used for each connection. This is achieved by having the user process generate the sequence number and pass it through the send token to the Myrinet interface. The GM NCP now uses these sequence numbers rather than generating its own. If messages are to be assigned sequence numbers strictly on a per-connection basis to maintain the original GM protocol, all the processes on a node sending messages to the same remote node need to be synchronized so that a continuous stream of sequence numbers for the connection is obtained. Such a synchronization can, however, introduce unnecessary overhead. A simple solution to this is to generate independent streams of sequence numbers for each remote node on a per-port basis. This generation can be done entirely within a single process, but requires that the receiver now acknowledge on a per-port basis rather than on a per-connection basis. Thus, the receiver now has to keep an ACK number for every connection-port pair. The extra memory requirement is, however, not large since GM allows only eight ports per node. This is the main deviation from the original GM structure, as depicted in Fig. 9.

Another difference from the original GM is with regard to the commit point on the receiver side. In our implementation, we delay the sending of an ACK to after the DMA of the message into the user's receiver buffer is complete. This increases the network overhead of the message but, as the results in Section 5 show, the impact on performance is small. Since the receiver must also keep a copy of the ACK number for every stream, the network processor needs to notify the host of the sequence number of the message that has just been ACKed. This is done by including the sequence number as part of the event posted by the Myrinet into the user process's receive queue. The receiver, at this time, also deletes the corresponding copy of the receive token. Similarly, on the sender side, the copy of the send token is removed just before the callback function for that send token is invoked.

In summary, no substantial modifications to the GM protocol were needed. All the changes were implemented within the GM library functions, thus making them transparent to the user. Based on our experience with Myrinet, we expect that the implementation will not be difficult for other NICs.

5 Experimental Results

Our experimental setup consisted of two Pentium III machines each with 256 Mbytes of memory, a 33MHz PCI bus, and running Redhat Linux 7.2. The Myrinet NICs were LANai9-based PCI64B cards and the Myrinet switch was type M3M-SW8.

5.1 Failure Coverage

We used as our workload a program provided by GM to send and receive messages of random lengths between processes in two machines. To evaluate the coverage of the self-testing of the modified GM, we developed a host program that sends loopback messages of various lengths to test latency and check for data corruption. We call it application-level self-testing to distinguish it from our NCP-level self-testing. This program follows the same approach as the NCP-level self-testing, that is, it attempts to check as many basic blocks as possible for the *Normal Delivery* module. The application-level self-testing program sends and receives messages by issuing GM library calls, in much the same way as normal applications do. We assume that, if such a test application is run in the presence of faults, it will experience the same number of faults that would affect normal applications. Based on this premise, we use the application-level self-testing as baseline and calculate the failure coverage ratio to evaluate our NCP-level self-testing. The failure coverage ratio is defined as the number of failures detected by the NCP-level self-testing divided by the number of failures detected by the application-level self-testing. When calculating the failure coverage ratio, we did not count the failures that are not covered by the proposed technique such as host crashes. To make the baseline application comparable to the NCP-level self-testing, we concurrently trigger exception events within the GM NCP to direct the control flow to cover basic blocks handling exceptions, so that the baseline application can detect all the failures that can be detected by the NCP-level self-testing.

The underlying fault model used in the experiments was primarily motivated by Single Event Upsets (SEUs), which were simulated by flipping bits in the SRAM. Such faults disappear on reset or when a new value is written to the SRAM cell. Since the probability of multiple SEUs is low, we focus on single SEUs in this paper. To emulate a fault that may cause the hardware to stop responding, we injected stuck-at-0 and stuck-at-1 faults into the special registers in the NIC. The time instances at which faults were injected were randomly selected. After each fault injection run, the GM NCP was reloaded to eliminate any interference between two experiments.

TABLE 1
Results of Fault Injection

	Send_chunk			Registers			Entire Code Seg.		
	Failures	% Faults	% Failures	Failures	% Faults	% Failures	Failures	% Faults	% Failures
Host Computer Crash	7	0.7	1.7	46	24.0	35.4	8	0.56	9.09
NCP Hung (By WT)	128	12.1	30.5	10	5.2	7.7	24	1.68	27.27
Send/Recv Failures	151	14.3	36.0	0	0.0	0.0	21	1.47	23.86
DMA Failures	21	2.0	5.0	26	13.5	20.0	12	0.84	13.64
Corrupted Ctrl Info.	0	0.0	0.0	3	1.6	2.3	1	0.07	1.14
Corrupted Message	5	0.5	1.2	45	23.4	34.6	8	0.56	9.09
Unusually Latency	107	10.1	25.5	0	0.0	0.0	14	0.98	15.91
No Impact	637	60.3	–	62	32.3	–	1342	93.85	–
Total	1056	100.0	100.0	192	100.0	100.0	1430	100.00	100.00

To evaluate the effectiveness of our NCP-level loopback without testing exhaustively each bit in the SRAM and registers, we performed the following three experiments:

- Exhaustive fault injection into a single routine (the frequently executed *send_chunk*).
- Injecting faults into the special registers.
- Random fault injection into the entire code segment.

The data structures that can make up a significant fraction of the GM NCP state were not subjected to fault injection because the proposed technique does not provide adequate coverage for them. This kind of faults would need a traditional redundancy approach.

In all the experiments mentioned in this section, only the *Normal Delivery* logical module was active and checked. The workload program and the application-level self-testing program requested service only from this module. If a fault was injected in the *Normal Delivery* module, it would be activated by the workload program; if not, the fault would be harmless and have no impact on the application. The injection of each fault was repeated 10 times and the results averaged.

5.2 Results

The routine *send_chunk* is responsible for initializing the packet interface and setting some special registers to send messages out on the Myrinet link. The entire routine is part of the *Normal Delivery* module.

There are 33 instructions in this routine, totaling 1,056 bits. Faults were sequentially injected at every bit location in this routine. Columns 2 to 4 in Table 1 show a summary of the results reported by NCP-level self-testing for these experiments. Column 2 shows the number of detected failures, column 3 shows the failures as a fraction of the total faults injected, and column 4 shows the failures as a fraction of the total failures observed. About 40 percent of the bit-flip faults caused various types of failures. Out of these, 30.5 percent were network interface hangs, which were detected by our watchdog timer, 1.7 percent of these failures caused a host crash, and the remaining 67.8 percent were detected by our NCP-level self-testing. The failure coverage ratio of the NCP-level self-testing of this routine is 99.3 percent.

For our next set of experiments, we injected faults into the special registers associated with DMA. Columns 5 to 7 in Table 1 show a summary of the results. The GM NCP sets these registers to fetch messages from the host memory to the SRAM via the DMA interface. There are a total of 192 bits in the SDMA registers, containing information about source address, destination address, DMA length, and some flags. We sequentially injected faults at every bit location. From the results, it is clear that the memory-mapped region corresponding to the DMA special registers is very sensitive to faults. In these experiments, faults propagated to the DMA hardware or even the host computer and caused fatal failures. Since the total number of register bits is only several hundred, orders of magnitude smaller than the number of instruction bits, the probability that a fault hits a register bit and causes a host crash is very low. Even though 35.4 percent of the failures from injecting faults in registers resulted in a host crash, they account for a very small fraction of the total number of failures. The failure coverage ratio of this set of experiments is 99.2 percent.

The third set of results (columns 8 to 10 in Table 1) shows how the NCP-level self-testing performs when faults are randomly injected into the entire code segment of the GM NCP. We injected 1,430 faults at random bit locations, but only 88 caused failures. The 27.3 percent of these failures were network interface hangs detected by our watchdog timer, 9.1 percent caused a host crash, and the remaining 63.6 percent of the failures were detected by our NCP-level self-testing. The failure coverage ratio is about 95.6 percent. From the table, we see that a substantial fraction of the faults do not cause any failures and, thus, have no impact on the application. This is because the active logical module, that is, *Normal Delivery*, is only one part of the GM NCP. This reinforces the fact that self-testing for the entire NCP is mostly unnecessary. By focusing on the active logical module(s), our self-testing scheme can considerably reduce the overhead.

Due to uncertainties in the state of the interface when injecting a fault, repeated injections of the same fault are not guaranteed to have the same effect. However, the majority of failures displayed a high degree of repeatability. Such repeatability has also been reported elsewhere [25].

5.3 Recovery Time and Effectiveness

The complete recovery time is the sum of the failure detection time and the time spent in our FTD and the user

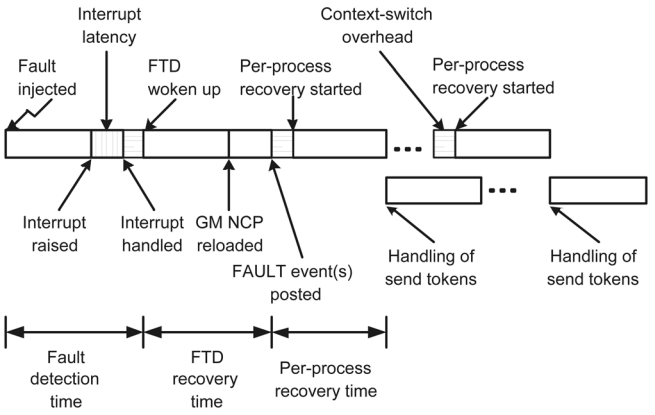

Fig. 10. The timeline of the fault recovery process.

process' failure handler for restoring the state, as shown in Fig. 10. The failure detection time was measured as the time from the fault injection to the time when the FTD is woken up by the driver. It is a function of the maximum time between *L_timer* invocations and the interrupt latency. We will ignore the interrupt latency, because it is negligible $(\sim 13\mu s)$ compared to 800 μs for the watchdog timer interval and the self-testing intervals. The FTD recovery time consists of the time required to reload the GM NCP and restore routing and page hash tables and posting the Fault_Detected event in each open port's receive queue. Averaging over a number of experiments revealed a value of $\sim 765,000\mu s$ for the FTD recovery time with $\sim 500,000\mu s$ being spent in reloading the GM NCP.

The rest of the recovery time depends on the number of open ports at the time of failure. The per-port recovery time is primarily a function of the execution time of the Fault_Detected event handler. Our experimental results show that this value is $\sim 900,000\mu s$. It is arguable whether the time for handling the restored send tokens by the network processor needs to be accounted for in the recovery time. This would however be a function of the number of send tokens that have been restored.

The experiments were repeated using our Fault-Tolerant GM (FTGM). Except for corruption failures and interface hangs, FTGM was able to recover from all other failures. Although all the network interface hangs were correctly detected, there were only five cases out of the 286 hangs that FTGM was not able to properly recover from. We are currently investigating these cases.

5.4 Performance Impact

The performance of a network is usually measured using three principal metrics:

- **Bandwidth** measures the sustained data rate available for large messages.
- **Latency** is usually calculated as the time to transmit from source to destination.
- **Host-CPU utilization** measures the overhead borne by the host-CPU in sending or receiving a message.

GM provides a set of programs that can be used to evaluate these metrics. The workload for our experiments involved both hosts sending and receiving messages at the maximum rate possible. Measurements were performed as bidirectional exchanges of messages of different lengths between processes in two machines.

We first experimented with only the failure detection scheme and evaluated its performance impact, in this section, we will refer to this modified GM software as Failure Detection GM (FDGM). For each message length of the workload, messages were sent repeatedly for at least 10 seconds, and the results were averaged.

Fig. 11a compares the bandwidth obtained with GM and FDGM for different message lengths. The reason for the jagged pattern in the middle of the curve is that GM partitions large messages into packets of at most 4 Kbytes at the sender and reassembles them at the receiver. Fig. 11b compares the point-to-point half-round-trip latency for messages of different lengths. For this experiment, the NCP-level self-testing interval was set to 5 seconds. The figures show that FDGM imposes no appreciable performance degradation with respect to latency and bandwidth.

We also studied the overhead of the NCP-level self-testing when the test interval is reduced from 5 to 0.5 seconds. Experiments were performed for a message length of 2 Kbytes. The latency of the original GM software is 69.39 μs, and its bandwidth is 14.71 Mbytes/s. Fig. 12 shows the bandwidth and latency differences between GM and FDGM. There is no significant performance degradation with respect to latency and bandwidth. For the interval of 0.5 seconds, the bandwidth is reduced by 3.4 percent, and the latency is increased by 1.6 percent when compared with the original GM.

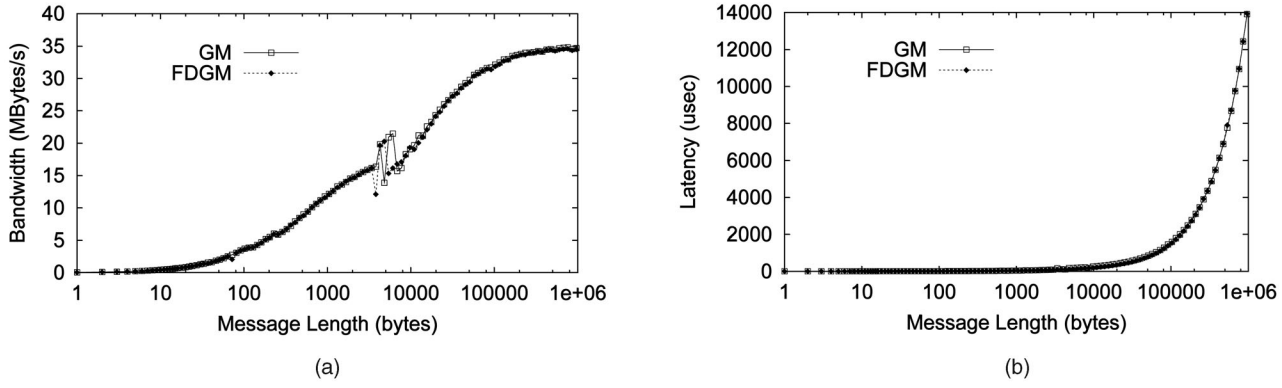

Fig. 11. Comparison of the original GM and FDGM. (a) Bandwidth. (b) Latency.

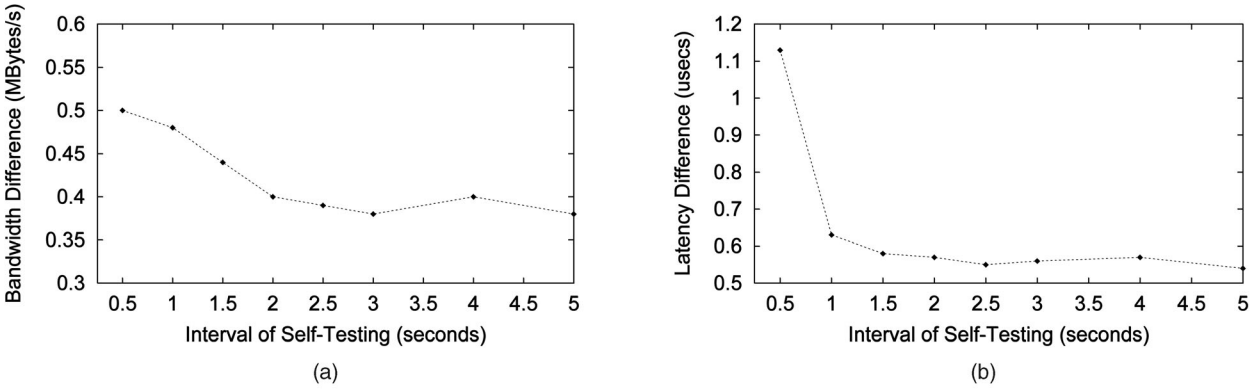

Fig. 12. Performance impact for different self-testing intervals. (a) Bandwidth difference versus interval length. (b) Latency difference versus interval length.

Such results agree with expectations. The total size of our self-testing messages is about 24 Kbytes, which is negligible relative to the high bandwidth of the NIC. Users can determine accordingly the NCP-level self-testing interval, taking into consideration performance and failure detection latency.

We then incorporated both the failure detection and recovery schemes and evaluated the performance impact, and we refer to this version of the modified GM software as FTGM. For each message length of the workload, a large number of messages (we used 1,000 here) were sent repeatedly and results were averaged.

Fig. 13a shows that the sustained bidirectional data rate for GM, as well as FTGM approaches an asymptotic value of $\sim 92 Mbytes/sec$ for long messages. FTGM follows very close on the heels of GM and imposes no appreciable performance degradation with regards to bandwidth.

Fig. 13b compares the point-to-point half-round-trip latency of messages of different lengths. Here, again, the performance of FTGM is not far behind the original GM. The short-message latency, a critical metric for many distributed-computing applications, is about 11.5 μs for GM and 13.0 μs for FTGM, averaged over message lengths ranging from 1 byte to 100 bytes. These latencies consist of a host component and a network interface component. Although the host component is a combination of the host-CPU execution time and the PCI latency, the network

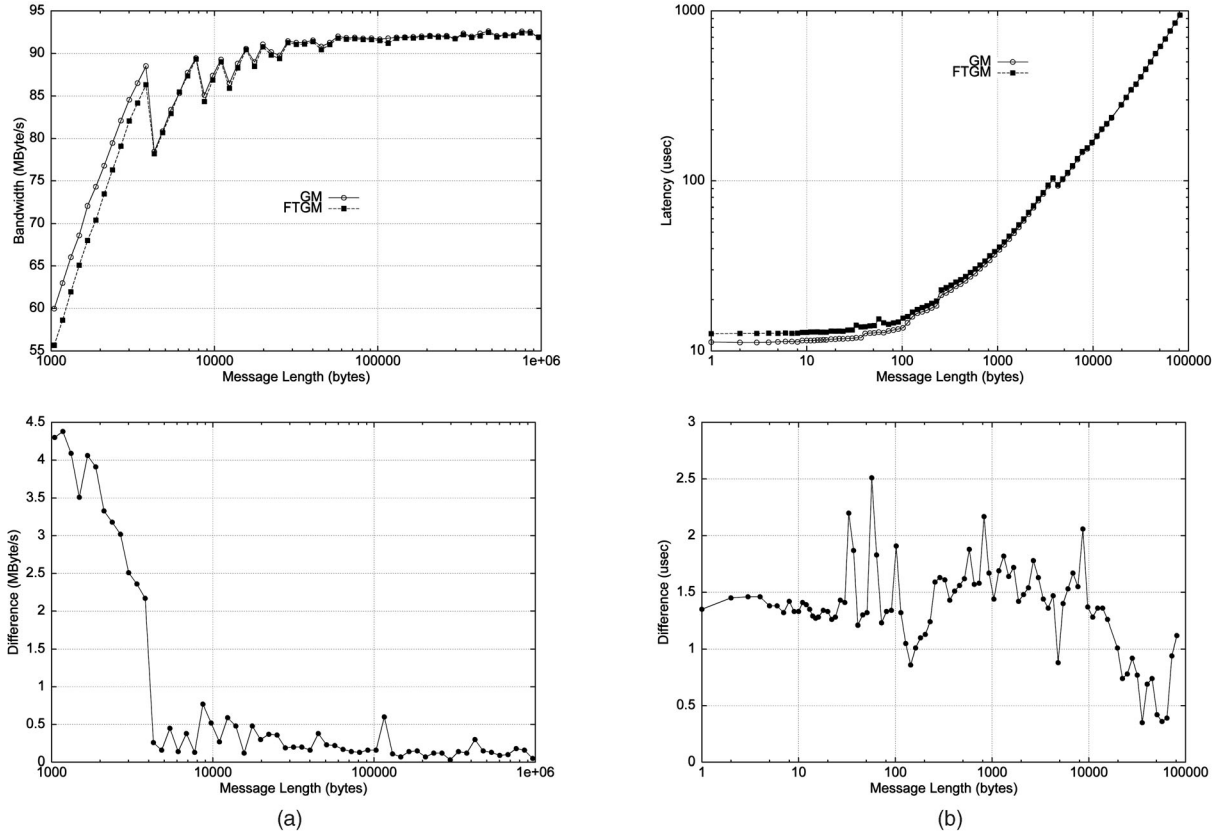

Fig. 13. Comparison of the Original GM and FTGM. (a) Bandwidth. (b) Latency.

TABLE 2
Comparison of Various Performance Metrics between GM and FTGM

	GM	FTGM
Bandwidth	$92.4MB/s$	$92.0MB/s$
Latency	$11.50\mu s$	$13.00\mu s$
Host (send)	$0.30\mu s$	$0.55\mu s$
Host (recv)	$0.75\mu s$	$1.15\mu s$
NCP	$6.00\mu s$	$6.80\mu s$

interface component is a combination of the network processor execution time and the packet interface latency. FTGM was designed to minimize the amount of extra information being transferred through DMA from the host memory to the NIC memory. Moreover, there is absolutely no change in the packet header and no extra information is sent with the packet. Therefore, the effect on the PCI latency and the packet interface latency in the network processor is minimal. The modification in the GM NCP that affects the critical path the most is the delay in sending the ACK after the DMA is complete. Since the ACK needs to be delayed, only when a receive token is returned to the user, a multiple-packet message can be made to take full advantage of the network bandwidth by not waiting for the DMA to be complete, thus allowing several packets of the same message to be in-flight at the same time. For small messages, however, the extra delay comes mainly from the host-CPU utilization. This factor is most evident in protocols employing a host-level credit scheme for flow control such as FM [27].

Minimizing the host-CPU utilization was one of our principal design objectives. Information posted on the Myricom Web site indicates that the measured overhead on the host for sending (receiving) a message is about $0.3\mu s$ $(0.75\mu s)$. In FTGM, the send and receive token housekeeping contributes the most to the increase in delay. It is around $0.25\mu s$ for the send and around $0.4\mu s$ for the receive. The extra overhead for the receive is because the receiver has to update two hash tables for every receive: one containing the receive tokens and the second containing ACK numbers for each stream. Table 2 summarizes the results presented in this section.

6 Related Work

Chillarege [26] proposes the idea of a software probe to help detect failed software components in a running software system by requesting a certain level of service from a set of functions and/or modules and checking the response to the request. This paper however, presents no experimental results to evaluate the efficiency and performance impact. Moreover, since it considers general systems, there is no discussion devoted to minimizing the performance impact and improving the failure coverage as we do in this paper.

Several approaches have been proposed in the past to achieve fault tolerance by modifying only the software. These approaches include Self-Checking Programming [8], Algorithm-Based Fault Tolerance (ABFT) [9], Assertion [10], Control Flow Checking [11], Procedure Duplication [12], Software Implemented Error Detection and Correction (EDAC) code [13], Error Detection by Duplicated Instructions (EDDI) [14], and Error Detection by Code Transformations (EDCT) [15]. Self-Checking Programming uses program redundancy to check its own behavior during execution. It results from either the application of an acceptance test or the comparison of the results of two duplicated runs. Since the message passed to a network interface is completely nondeterministic, an acceptance test is likely to exhibit low sensitivity. ABFT is a very effective approach, but can only be applied to a limited set of problems. Assertions perform consistency checks on software objects and reflect invariant properties for an object or set of objects but effectiveness of assertions strongly depends on how well the invariant properties of an application are defined. Control Flow Checking cannot detect some types of errors such as data corruption, whereas Procedure Duplication only protects the most critical procedures. Software Implemented EDAC code provides protection for code segments by periodically encoding and decoding instructions. Such an approach, however, would involve a substantial overhead for a network processor because the code size of an NCP might be several hundreds of thousands of bytes. Although it can detect all the single bit faults, it is an overkill because many faults are harmless. Moreover, it cannot detect hardware errors. EDDI and EDCT have a high-error coverage, but have substantial execution and memory overheads.

7 Conclusion

This paper describes a software-based low-overhead fault-tolerant scheme for programmable network interfaces. Failure detection is achieved by a watchdog timer that detects network interface hangs and a built-in self-testing that detects noninterface-hang failures. The proposed self-testing directs the control flow to go through the active logical modules; during this procedure, the functionality of the network interface, essentially the hardware and the active logical modules of the software, are tested. The idea behind our failure recovery scheme is to keep a copy of just the right amount of network interface state information in the host so that the state of the network interface can be restored upon failure. The proposed fault-tolerant scheme can be implemented transparently to applications.

The basic idea underlying the presented failure detection and recovery techniques is quite generic and can be applied to other modern high-speed networking technologies that contain a microprocessor and local memory, such as IBM PowerNP [18], Infiniband [19], Gigabit Ethernet [20], [21], QsNet [22], and Asynchronous Transfer Mode (ATM) [23], or even other embedded systems.

Acknowledgments

This work has been supported in part by a grant from a joint US National Science Foundation (NSF) and NASA program on Highly Dependable Computing (NSF Grant CCR-0234363 and NASA Grant NNA04C158A).

References

[1] J.F. Ziegler et al., "IBM Experiments in Soft Fails in Computer Electronics (1978-1994)," *IBM J. Research and Development*, vol. 40, no. 1, pp. 3-18, Jan. 1996.

[2] S.S. Mukherjee, J. Emer, and S.K. Reinhardt, "The Soft Error Problem: An Architectural Perspective," *Proc. 11th Int'l Symp. High-Performance Computer Architecture,* pp. 243-247, Feb. 2005.
[3] Remote Exploration and Experimentation (REE) Project, http://www-ree.jpl.nasa.gov/, year?
[4] A.V. Karapetian, R.R. Some, and J.J. Beahan, "Radiation Fault Modeling and Fault Rate Estimation for a COTS Based Space-Borne Supercomputer," *Proc. IEEE Aerospace Conf.,* vol. 5, pp. 5-2121-5-2131, Mar. 2002.
[5] The Human Impacts of Solar Storms and Space Weather, http://www.solarstorms.org/Scomputers.html, year?
[6] A. Thakur and B.K. Iyer, "Analyze-NOW—An Environment for Collection of Analysis of Failures in a Network of Workstation," *Proc. Seventh Int'l Symp. Software Reliability Eng.,* pp. 14-23, Oct. 1996.
[7] V. Lakamraju, I. Koren, and C.M. Krishna, "Low Overhead Fault Tolerant Networking in Myrinet," *Proc. Dependable Systems and Networks,* pp. 193-202, June 2003.
[8] L.L. Pullum, *Software Fault Tolerance Techniques and Implementation.* Artech House, 2001.
[9] K.-H. Huang and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. Computers,* vol. 33, no. 6, pp. 518-528, Dec. 1984.
[10] D.M. Andrews, "Using Executable Assertions for Testing and Fault Tolerance," *Proc. Ninth Int'l Symp. Fault-Tolerant Computing,* pp. 102-105, June 1979.
[11] S.S. Yau and F.-C. Chen, "An Approach to Concurrent Control Flow Checking," *IEEE Trans. Software Eng.,* vol. 6, no. 2, pp. 126-137, Mar. 1980.
[12] D.K. Pradhan, *Fault-Tolerant Computer System Design.* Prentice Hall, 1996.
[13] P.P. Shirvani, N.R. Saxena, and E.J. McCluskey, "Software-Implemented EDAC Protection against SEUs," *IEEE Trans. Reliability,* vol. 49, no. 3, pp. 273-284, Sept. 2000.
[14] N. Oh, P.P. Shirvani, and E.J. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors," *IEEE Trans. Reliability,* vol. 51, no. 1, pp. 63-75, Mar. 2002.
[15] B. Nicolescu and R. Velazco, "Detecting Soft Errors by a Purely Software Approach: Method, Tools and Experimental Results," *Proc. Design, Automation and Test in Europe Conf. and Exhibition,* pp. 57-62, Mar. 2003.
[16] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.-K. Su, "Myrinet: A Gigabit-per-Second Local-Area Network," *IEEE Micro,* vol. 15, no. 1, pp. 29-36, Feb. 1995.
[17] Myricom, http://www.myri.com/, year?
[18] J.R. Allen, B.M. Bass, C. Basso, and R.H. Boivie et al., "IBM PowerNP Network Processor: Hardware, Software, and Applications," *IBM J. Research and Development,* vol. 47, no. 2/3, pp. 177-193, Mar./May 2003.
[19] Infiniband Trade Assoc., http://www.infinibandta.com/, year?
[20] P. Shivam, P. Wyckoff, and D. Panda, "EMP: Zero-Copy OS-Bypass NIC-Driven Gigabit Ethernet Message Passing," *Proc. ACM/IEEE Supercomputing 2001 Conf.,* p. 49, Nov. 2001.
[21] The Gigabit Ethernet Alliance, http://www.gigabit-ethernet.com/, year?
[22] The QsNet High Performance Interconnect, year? http://www.quadrics.com/.
[23] A.T.M. Forum, *ATM User-Network Interface Specification.* Prentice Hall, 1995.
[24] T. Halfhill, "Intel Network Processor Targets Routers," *Microprocessor Report,* vol. 13, no. 12, Sept. 1999.
[25] D.T. Stott, M.-C. Hsueh, G.L. Ries, and R.K. Iyer, "Dependability Analysis of a Commercial High-Speed Network," *Proc. 27th Int'l Symp. Fault-Tolerant Computing,* pp. 248-257, June 1997.
[26] R. Chillarege, "Self-Testing Software Probe System for Failure Detection and Diagnosis," *Proc. 1994 Conf. Centre for Advanced Studies on Collaborative Research,* vol. 10, 1994.
[27] R.A.F. Bhoedjang, T. Rühl, and H.E. Bal, "User Level Network Interface Protocols," *IEEE Computer,* vol. 31, no. 11, pp. 53-60, Nov. 1998.

Yizheng Zhou received the BS degree in electronics engineering from Tsinghua University, China, in 2000 and the MS degree in computer engineering from North Carolina State University in 2002. He is currently a PhD student in electrical and computer engineering at the University of Massachusetts, Amherst. His research interests include distributed systems, fault tolerance, and embedded systems.

Vijay Lakamraju received the PhD degree from the University of Massachusetts, Amherst, in 2002. He is currently a research scientist at the United Technologies Research Center. From 2002-2005, he was a cofounder of the BlueRISC Inc. and a postdoctoral research associate at the Architecture and Real-Time Systems (ARTS) Lab, University of Massachusetts, Amherst. His research interests include distributed real-time systems, power aware computing, wired and wireless networking, and fault tolerance.

Israel Koren (S'72-M'76-SM'87-F'91) received the BSc, MSc, and DSc degrees from the Technion—Israel Institute of Technology, Haifa. He is currently a professor of electrical and computer engineering at the University of Massachusetts, Amherst. Previously, he was with the Technion—Israel Institute of Technology. He also held visiting positions with the University of California at Berkeley, the University of Southern California, Los Angeles, and the University of California, Santa Barbara. His current research interests include fault-tolerant techniques and architectures, yield and reliability enhancement, and computer arithmetic. He published extensively in several IEEE Transactions and has more than 200 publications in refereed journals and conferences. He currently serves on the editorial board of the *IEEE Transactions on VLSI Systems*, *IEEE Computer Architecture Letters*, and the *VLSI Design Journal*. He was a coguest editor for three special issues of the *IEEE Transactions on Computers* and served on the editorial board of these Transactions between 1992 and 1997. He also served as general chair, program chair, and program committee member for numerous conferences. He is the author of the textbook *Computer Arithmetic Algorithms* (A.K. Peters, Ltd., 2002). He is a coauthor of the textbook *Fault Tolerant Systems*, to be published by Morgan-Kaufman in 2007. He is a fellow of the IEEE and the IEEE Computer Society.

C. Mani Krishna received the PhD degree from the University of Michigan in 1984. Since then, he has been on the faculty of the University of Massachusetts, Amherst. His research interests include real-time systems, distributed computing, and performance evaluation. He has coauthored texts on real-time systems and on fault-tolerant computing. He is a senior member of the IEEE and the IEEE Computer Society.