

Software-Based Adaptive and Concurrent Self-Testing in Programmable Network Interfaces*

Yizheng Zhou, Vijay Lakamraju, Israel Koren, C.M. Krishna
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003
E-mail: {yzhou, vlakamra, koren, krishna}@ecs.umass.edu

Abstract

Emerging network technologies have complex network interfaces that have renewed concerns about network reliability. In this paper, we present an effective low-overhead failure detection technique, which 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 adaptive and concurrent 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. The paper shows how this technique can be made to minimize the performance impact on the host system and be completely transparent to the user.

1. Introduction

Interfaces with a network processor and large local memory are widely used [14, 16, 17, 18, 19, 20, 21]. The complexity of network interfaces has increased tremendously over the past few years. This is evident from the amount of silicon used in the core of network interface hardware. A typical dual-speed Ethernet controller uses around 10K gates whereas a more complex high-speed network processor such as the Intel IXP1200 [22] uses over 5 million transistors. This trend is being driven by the demand for greater network performance, and so communication-related processing is increasingly being offloaded to the network interface. As transistor counts increase dramatically, 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]. In most cases, the failure

is soft, i.e., it does not reflect a permanent failure of the device, and typically a reset of the device or a rewriting of the memory cell returns the device to normal functioning. As we will see in the following sections, soft errors can cause the network interface to completely stop responding, function improperly, or even cause the host computer to crash/hang. Quickly detecting and recovering from such network interface 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 the local memory of the network interface where the network control program (NCP) resides.

In this paper, we present an efficient software-based failure detection technique for programmable network interfaces. Software-based fault tolerance approaches are attractive, since they allow the implementation of dependable systems without incurring the high costs of using custom hardware or massive hardware redundancy. On the other hand, software fault tolerance approaches impose overhead in terms of reduced performance and increased code size. Since performance is critical for high-speed network interfaces, fault tolerance techniques applied to them must have a minimal performance impact.

Our failure detection scheme is based on a software-implemented watchdog timer to detect network processor hangs, and a software-implemented adaptive and concurrent self-testing technique to detect non-interface-hang failures, such as data corruption and bandwidth reduction. The proposed scheme achieves failure detection 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 the functionalities of active logical modules are tested (a logical module is defined as the collection of all basic blocks that participate in providing a service, and an active logical module is the one providing a service to a running application). When compared

*This work has been supported in part by a grant from a joint NSF and NASA program on Highly Dependable Computing (NSF grant CCR-0234363, NASA grant NNA04C158A).

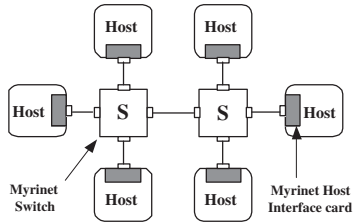


Figure 1. Example Myrinet Network

with testing the whole NCP, testing only active logical modules can significantly reduce the impact of these tests on application performance while achieving good failure detection coverage. When a failure is detected, the host system is interrupted and informed about the failure. Then, a fault-tolerance daemon is woken up to start a recovery process [5].

In this paper, we show how the proposed failure detection technique can be made completely transparent to the user. We demonstrate this technique in the context of Myrinet, but our approach is generic in nature, and is applicable to many other core modern networking devices that have a microprocessor core and local memory.

The remainder of this paper is organized as follows. A brief overview of Myrinet is given in Section 2. We keep the description sufficiently general so as to highlight the more generic applicability of our work. Section 3 then details our failure detection technique. In Section 4, we discuss the results and performance impact of our failure detection scheme. Section 5 discusses related work. Finally, we present our conclusions in Section 6.

2. Myrinet: An Example Programmable Network Interface

Myrinet [14] is a cost-effective, high bandwidth (2Gb/s) and low latency ($\sim 6.5\mu s$) local area network (LAN) technology. A Myrinet network consists of point-to-point, full-duplex links that connect Myrinet switches to Myrinet host interfaces and other Myrinet switches. Figure 1 shows the components in an example Myrinet network.

2.1. Myrinet NIC

Figure 2 shows the organization and location of the Myrinet Network Interface Card (NIC) in a typical architecture.

The card has an instruction-interpreting RISC processor, a DMA interface to/from the host, a link interface to/from the network and a fast local memory (SRAM) which is used for storing the Myrinet Control Program (MCP) and for packet buffering. The MCP is responsible for buffering and transferring messages between the host and the network and providing all network services.

2.2. Myrinet Software

Basic Myrinet-related software is freely available from Myricom [15]. The software, called GM, includes a driver for the host OS, the MCP, a network mapping program, a user library and APIs. It is the vulnerability to faults in the MCP that is the focus of this work, so we now provide a brief description of the MCP.

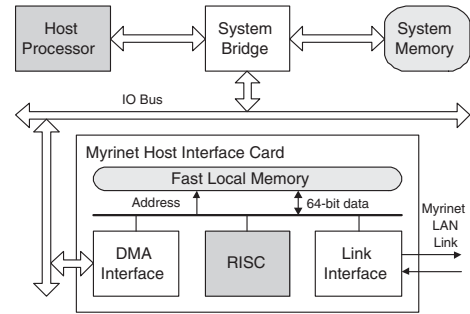


Figure 2. Simplified block diagram of the Myrinet NIC

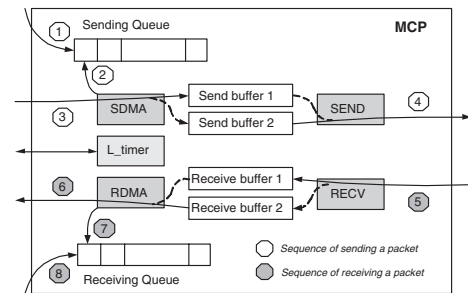


Figure 3. Simplified View of the Myrinet Control Program (MCP)

The MCP [15] can be viewed broadly as consisting of 4 interfaces: Send DMA (SDMA), SEND, Receive (RECV) and Receive DMA (RDMA), as depicted in Figure 3.

The sequence of steps during sending and receiving is illustrated in Figure 3. 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 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 send buffer, over the Myrinet link.

Similarly, two receive buffers are present. One of these buffers is made available for receiving an incoming

message, by the RECV interface (step 5), while the other can 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 MCP 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.

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 MCP. 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. The packet header and token contain significant information about the message, such as its length, its type, and from where an application can access it.
- *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 research.

Figure 4 shows how a bit-flip fault may affect message latency and network bandwidth. The error was caused by a bit-flip that occurred in a sending path of the MCP. 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 MCP generates more message resends, and this greatly increases the effective latency of messages. Since no error is reported by the MCP, 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.

Other subtle fault effects can also occur. 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 MCP transfers the contents of the message through the DMA interface from the host, an incorrect message will be sent out. Such errors are difficult to detect and are invisible to the application.

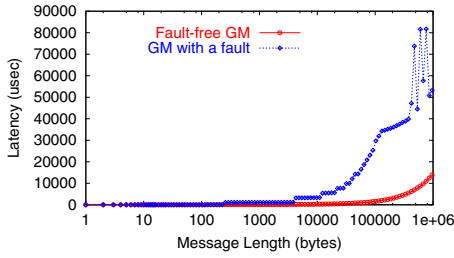
Even though the above discussion was related to Myrinet, we believe that such effects are generic and apply to other high-speed network interfaces having similar features, i.e., a network processor, a large local memory and a NCP running on the interface card. We detail our approach in the next subsection.

3.1. Failure Detection Strategy

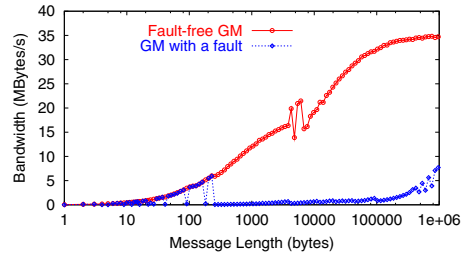
Since communication bandwidth and latency are critical to network interfaces, the failure detection scheme should minimize the performance impact to the network interface card while still achieving good failure coverage. Another consideration of our design is to minimize the intervention to the host processor.

We can use a watchdog timer to detect interface hangs [5], but because the code size of the NCP is quite large, it is more challenging to efficiently test the software of a programmable network interface to detect non-interface-hang failures. This can become even worse as the size of the NCPs increases. A useful observation regarding the behavior of the NCP helps us to achieve our design goal: applications generally use only a small portion of the NCP. For instance, the MCP is designed to provide various services to applications, including reliable ordered message delivery (*Normal Delivery*), directed reliable ordered message delivery which 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, while *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, and thus only a subset of the NCP is involved in serving requests from a specific application. Some other programmable network interfaces, such as the IBM PowerNP [16], have similar characteristics.

Based on this observation, we propose to test the functionalities of only that part of the NCP which corresponds to the services currently requested by the application at hand, instead of the entire NCP. Since the



(a) Unusually long latencies caused by a fault



(b) Bandwidth reduction caused by a fault

Figure 4. Examples of Fault Effects on Myrinet’s GM

NCP can be very large, such a scheme can considerably diminish failure detection overhead. Moreover, because a fault affecting an instruction which is not involved in serving requests from an application would not change the final outcome of the execution, our scheme avoids signaling these benign faults and significantly reduce the performance impact, compared to other techniques, such as those that periodically encode and decode the entire code segment [11].

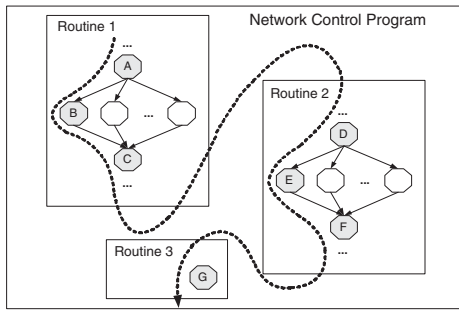


Figure 5. Logical Modules and Routines

To implement this failure detection scheme, we must identify the “active” parts of the network control program for a specific application. To assist the identification process, we logically 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. Figure 5 shows a sample NCP which 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 MCP and followed all the possible control flows to identify the basic blocks of the corresponding logical module.

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 functionalities of a network interface are tested and errors on the path 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, while small messages would be sent and received without the fragmenting and assembling process. Therefore, to test all the basic blocks of a logical module, we have to send at least a small message and a large message of the corresponding type of service. To test the sending path of the NCP, we can simply trigger sending requests, but to test the receiving path, we need to set up receiving requests and cause packets to arrive. We therefore use loopback messages of various sizes to test the sending and receiving paths of the network control program 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 services provided by network interfaces that are not directly used for sending/receiving messages, such as setting an alarm, by directing the control flow to go through basic blocks in the logical modules providing these services. Such tests are interleaved with the application’s use of the network interface.

To reduce the overhead of self-testing, we can implement an Adaptive and Concurrent Self-Testing (ACST) scheme. We insert a piece of code at the beginning of the NCP to detect 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, which would cause a small delay for the first request. For a low-latency NIC such as Myrinet, this delay would be negligible. Furthermore, we can reduce the delay by letting the application packets follow on the heels of the self-testing packets in the control flow. 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

In what follows, we demonstrate and evaluate our self-testing scheme for one of the most frequently used logical modules in the MCP, 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 are sent through the DMA and link interfaces and back again 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 latency, we conclude that the network interface works properly.

We have implemented such a scheme in the MCP. We emulate normal sending and receiving behavior in the *Normal Delivery* logical module in the MCP, as if there were a sending and a receiving client running in the host. 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 appropriate tokens causes the execution control to go through basic blocks in the corresponding logical module, so that errors on the control flow path are detected. Similarly, some events such as message loss or software-implemented translation lookaside buffer misses, which might concurrently happen during the sending/receiving process of the *Normal Delivery* logical module are also triggered within the network interface for each self-testing cycle to test the corresponding basic blocks. To reduce the overhead, we emulate a minimum number of requests/events within the network interface to go through all the basic blocks of the *Normal Delivery* logical module.

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

Since all the modifications are within the MCP, the API used by an application is unchanged so that no modification

to the application source code is required.

In the following sections, we refer to the modified GM software as Failure Detection GM (FDGM).

4. Experimental Results

Our experimental setup consisted of two Pentium III machines each with 256MB 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.

We used a program provided by GM to send and receive messages of random lengths between processes in the two machine as our workload. To evaluate the coverage of the self-testing of the modified GM, we developed a host program which 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 MCP-level self-testing. This program follows the same approach as the MCP-level self-testing, that is, it attempts to check as many basic blocks as possible for the *Normal Delivery* logical 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 error coverage ratio to evaluate our MCP-level self-testing. The error coverage ratio is defined as the number of errors detected by the MCP-level self-testing divided by the number of errors detected by the application-level self-testing. To make the results comparable, we modified the MCP to make the application-level self-testing capable of testing the basic blocks that cannot be tested by general applications in normal cases, such as the resending path.

The underlying fault model used in the experiments was primarily motivated by Single Event Upsets (SEUs). An SEU fault is simulated by flipping a bit 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 network interface card. The time instances at which faults were injected were randomly selected. After each fault injection run, the MCP was reloaded to eliminate any interference between two experiments.

To evaluate the effectiveness of our MCP-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 errors into the special registers.
- Random fault injection into the entire code segment.

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* logical 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 in the experiments was repeated 10 times and the results averaged.

	Send_chunk		Registers		Entire code	
	Errors	%	Errors	%	Errors	%
Host Crash	7	0.7	46	24.0	8	0.56
MCP Hang	128	12.1	10	5.2	24	1.68
Send/Recv Failures	151	14.3	0	0.0	21	1.47
DMA Failures	21	2.0	26	13.5	12	0.84
Corrupted Ctrl Info.	0	0.0	3	1.6	1	0.07
Corrupted Message	5	0.5	45	23.4	8	0.56
Unusually Latency	107	10.1	0	0.0	14	0.98
No Impact	637	60.3	62	32.3	1342	93.85
Total	1056	100.0	192	100.0	1430	100.00

Table 1. Results of fault injection

4.1. Failure Coverage

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* logical module.

There are 33 instructions in this routine, totaling 1056 bits. Faults were sequentially injected at every bit location in this routine. Columns 2 and 3 of Table 1 show a summary of the results reported by MCP-level self-testing for these experiments. Column 2 shows the number of detected errors and column 3 the errors as a fraction of the total faults injected. About 40% of the bit-flip faults caused various types of errors. Out of these, 30.5% were network interface hangs, which were detected by our watchdog timer, 1.7% of these errors caused a host crash, and the remaining 67.8% were detected by our MCP-level self-testing. The error coverage ratio of the MCP-level self-testing of this routine is 99.3%.

For our next set of experiments, we injected faults into the special registers associated with DMA. Columns 4 and 5 of Table 1 show a summary of the results. The MCP 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 errors 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. The error coverage ratio is 99.2%.

The third set of results (columns 6 and 7 of Table 1) shows how the MCP-level self-testing performs when faults

are randomly injected into the entire code segment of the MCP. We injected 1430 faults at random bit locations, but only 88 caused errors. 27.3% of these errors were network interface hangs detected by our watchdog timer, 9.1% caused a host crash, and the remaining 63.6% of the errors were detected by our MCP-level self-testing. The error coverage ratio is about 95.6%. From the table we see that a substantial fraction of the faults do not cause any errors and thus have no impact on the application. This is because the active logical module, i.e., the *Normal Delivery* logical module, is only one part of the MCP. 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 errors displayed a high degree of repeatability. Such repeatability has also been reported elsewhere [23].

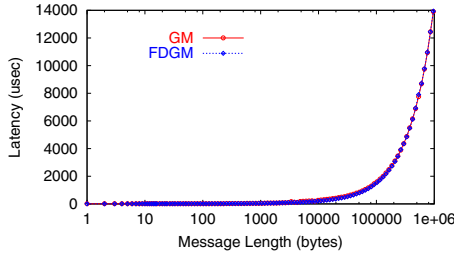
4.2. Performance Impact

We measure the network performance using two metrics. One is latency, which is usually calculated as the time to transmit small messages from source to destination, the other is bandwidth, which is the sustained data rate available for large messages. Measurements were performed as bi-directional exchanges of messages of different length between processes in the two machines. For each message length of the workload, messages were sent repeatedly for at least 10 seconds and the results averaged.

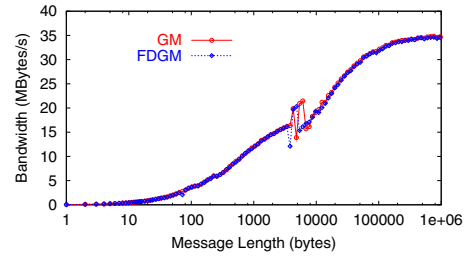
Figure 6(a) compares the point-to-point half-round-trip latency obtained with GM and FDGM for different message lengths. Figure 6(b) compares the bandwidth for messages of different lengths. For this experiment, the MCP-level self-testing interval is set to 5 seconds. The reason for the jagged pattern in the middle of the curve is that GM partitions large messages into packets of at most 4KB at the sender and reassembles them at the receiver. The figures show that FDGM imposes no appreciable performance degradation with respect to latency and bandwidth.

We also studied the overhead of the MCP-level self-testing when the test interval is reduced from 5 seconds to 0.5 seconds. Experiments were performed for a message length of 2K bytes. The latency of the original GM software is 69.39 μ s, and the bandwidth of the original GM software is 14.71 MB/s. Figure 7 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%, and the latency is increased by 1.6%, when compared with the original GM.

Such results agree with expectations. The total size of our self-testing messages is about 24K bytes which is negligible relative to the high bandwidth of the network

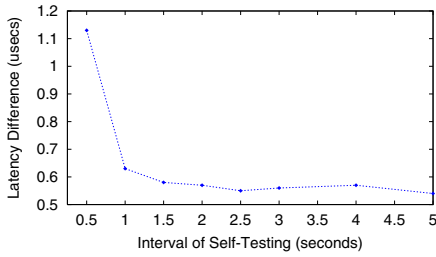


(a) Latency comparison of GM and FDGM

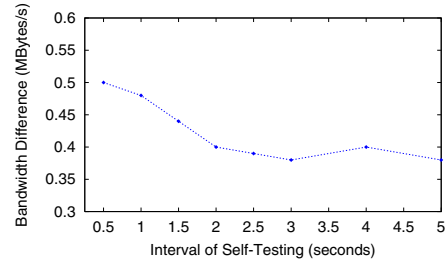


(b) Bandwidth comparison of GM and FDGM

Figure 6. Performance Impact of the Fault Detection Scheme



(a) Latency vs Interval



(b) Bandwidth vs Interval

Figure 7. Performance Impact for Different self-testing Intervals

interface card. Users can determine accordingly the MCP-level self-testing interval taking into consideration performance and failure detection latency.

5. Related Work

Chillarege [24] proposes the idea of a software probe to help detect failed software components in a running software system by requesting service, or a certain level of service, from a set of functions, modules and/or subsystems and checking the response to the request. This paper however, presents no experimental results to evaluate its efficiency and performance impact. Moreover, since the author considers general systems or large operating systems, there is no discussion devoted to minimizing the performance impact and improving the failure coverage as we did 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 [6], Algorithm Based Fault Tolerance (AFBT) [7], Assertion [8], Control Flow Checking [9], Procedure Duplication [10], Software Implemented Error Detection and Correction (EDAC) code [11], Error Detection by Duplicated Instructions (EDDI) [12], and Error Detection by Code Transformations (EDCT) [13]. Self-Checking Programming uses program redundancy to check its own behavior during execution. It results from either the application of an acceptance test or from the application of

a comparator to 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, while 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 overkill because many faults are harmless. Moreover, it cannot detect hardware unit errors. EDDI and EDCT have a high error coverage, but have substantial execution and memory overheads.

6. Conclusion

This paper describes a software-based low-overhead failure detection scheme for programmable network interfaces. The failure detection is achieved by a watchdog timer that detects network interface hangs, and a built-in ACST scheme that detects non-interface-hang failures.

The proposed ACST scheme directs the control flow to go through all the basic blocks in active logical modules; during this procedure, the functionalities of the network interface, essentially the hardware and the active logical modules of the software, are tested. Our experimental results are very promising. In the local memory of the Myrinet interface card, over 95% of the bit-flip errors that may affect applications can be detected by our self-testing scheme in conjunction with a software watchdog timer. The proposed ACST scheme can be implemented transparently to applications. To the best of our knowledge, this is the first paper that applies self-testing to a programmable network interface to detect bit flips in memory cells.

The basic idea underlying the presented failure detection scheme is quite generic and can be applied to other modern high-speed programmable networking devices, such as IBM PowerNP [16], Infiniband [17], Gigabit Ethernet [18, 19], QsNet [20] and ATM [21], or even other embedded systems. Such a failure detection scheme can take advantage of the high bandwidth available in these systems, thereby achieving its failure detection goals with very little overhead.

References

- [1] J. F. Ziegler, et al., "IBM experiments in soft fails in computer electronics (1978 - 1994)," *IBM Journal of Research and Development*, vol. 40, No. 1, pp. 3 - 18, Jan. 1996.
- [2] S. S. Mukherjee, J. Emer, S. K. Reinhardt, "The Soft Error Problem: An Architectural Perspective," *Proceedings of the Eleventh International Symposium on High- Performance Computer Architecture*, pp. 243-247, Feb. 2005.
- [3] Remote Exploration and Experimentation (REE) Project. <http://www-ree.jpl.nasa.gov/>.
- [4] A. V. Karapetian, R. R. Some, J. J. and Beahan, "Radiation Fault Modeling and Fault Rate Estimation for a COTS Based Space-borne Supercomputer," *IEEE Aerospace Conference Proceedings*, vol. 5, pp. 9-16, Mar. 2002.
- [5] V. Lakamraju, I. Koren and C. M. Krishna, "Low Overhead Fault Tolerant Networking in Myrinet," *Proceedings of the Dependable Computing and Communication Symposium*, pp. 193- 202, Jun. 2003.
- [6] L. L. Pullum, *Software Fault Tolerance Techniques and Implementation*, Artech House, 2001
- [7] K. H. Huang, J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. 33, pp. 518-528, Dec. 1984.
- [8] D. Andrews, "Using executable assertions for testing and fault tolerance," *Proceedings of the Ninth International Symposium on Fault-Tolerant Computing*, Jun. 1979.
- [9] S. Yau, F. Chen, "An Approach to Concurrent Control Flow Checking," *IEEE Transactions on Software Engineering*, vol. 6, No. 2, pp. 126-137, Mar. 1980.
- [10] D.K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice Hall PTR, 1996
- [11] P.P. Shirvani, N.R. Saxena, E.J. McCluskey, "Software-implemented EDAC protection against SEUs," *IEEE Transactions on Reliability*, vol. 49, No. 3, pp. 273-284, Sep. 2000.
- [12] N. Oh, P.P. Shirvani, E.J. McCluskey, "Error Detection by Duplicated Instructions in Super-scalar Processors," *IEEE Transactions on Reliability*, vol. 51, No. 1, pp. 63-75, Mar. 2002.
- [13] B. Nicolescu, R. Velazco, "Detecting Soft Errors by a Purely Software Approach: Method, Tools and Experimental Results," *Design, Automation and Test in Europe Conference and Exhibition (DATE'03 Designers' Forum)*, pp. 57-62, Mar. 2003.
- [14] 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.
- [15] Myricom Inc. <http://www.myri.com/>.
- [16] J. R. Allen, Jr., B. M. Bass, C. Basso, R. H. Boivie, Et al, "IBM PowerNP network processor: Hardware, software, and applications," *IBM Journal of Research and Development*, vol. 47, No. 2/3, pp. 177-194, Mar./May 2003.
- [17] Infiniband Trade Association. <http://www.infinibandta.com/>.
- [18] P. Shivam, P. Wyckoff, and D. Panda, "EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing," *Proceedings of ACM/IEEE Supercomputing 2001 (SC2001)*, pp. 57- 57, Nov. 2001.
- [19] The Gigabit Ethernet Alliance. <http://www.gigabit-ethernet.com/>.
- [20] The QsNet High Performance Interconnect. <http://www.quadrics.com/>.
- [21] A.T.M. Forum. *ATM User-Network Interface Specification*. Prentice Hall, 1998.
- [22] T. Halfhill. "Intel network processor targets routers," *Microprocessor Report*, vol. 13, No. 12, Sep. 1999.
- [23] D. T. Stott, M.-C. Hsueh, G. L. Ries, and R. K. Iyer, "Dependability Analysis of a Commercial Highspeed Network," *Proceedings of the Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing*, pp. 248-257, Jun. 1997.
- [24] R. Chillarege, "Self-testing Software Probe System for Failure Detection and Diagnosis," *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative Research*, pp. 10, 1994.