New Voter Design Enabling Hot Redundancy for Asynchronous Network Nodes

Felix Siegle and Tanya Vladimirova
University of Leicester
Leicester, LE1 7RH, United Kingdom

Jørgen Ilstad
European Space Agency / ESTEC
2200 AG Noordwijk, The Netherlands

Omar Emam
Airbus Defence and Space
Stevenage SG1 2AS, United Kingdom

***Abstract*—In this paper, a novel voter design is presented which allows the voting of asynchronous network streams in flow-controlled networks. The voter synchronises incoming data streams automatically and is able to handle failure modes that typically occur in streaming applications. The voter degrades to a comparator if one of the redundant channels has failed and reintegrates the channels once they are functional again. While the voter is mainly intended to be connected to a routing switch of the network, it also comprises a broadcast mechanism that enables a stand-alone operation. The design has been successfully implemented in hardware and evaluated by means of fault injection experiments.**

I. Introduction

Modern payload data processing approaches on board spacecraft demand increased processing capabilities. In best case, data can be processed in real time while being streamed from a data source to a data sink, e.g. from a camera to a mass memory device. On its way, the data is possibly processed by several processor nodes in series.

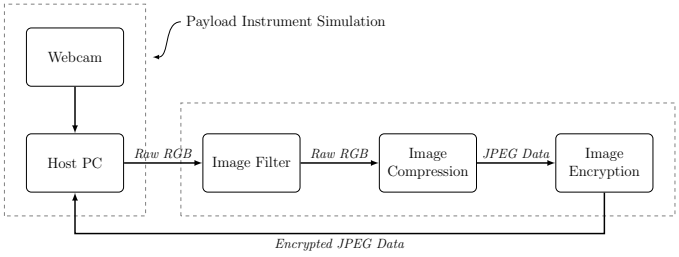

Fig. 1. Example for an image processing pipeline.

An example would be an image processing pipeline as outlined in Figure 1 in which video data is first filtered, then compressed and finally encrypted. To make such a processing pipeline adaptable in terms of functionality and reliability, the different processing steps can be implemented on reconfigurable Field Programmable Gate Arrays (FPGAs). Since fast hardware implementations of the processing steps can be rather resource-demanding, techniques are necessary to also exploit Multi-FPGA systems. An example for such a system is the Dynamically Reconfigurable Processing Module (DRPM) developed by University of Braunschweig and Airbus Defence and Space, UK [1]. This hardware development platform comprises a scalable number of payload data processing units with two reconfigurable SRAM-based Virtex-4 FPGAs and one LEON3 microprocessor per unit. The development platform is available for our own research on an adaptive Fault Detection, Isolation and Recovery (FDIR) methodology for such systems. As the methodology is based on hot redundancy which can be applied to the different processing steps on demand, a new voting mechanism is necessary that integrates well into such a Multi-FPGA system.

In the proposed FDIR framework, the aforementioned processing steps are executed by dedicated stream processors which can process incoming data streams independently.

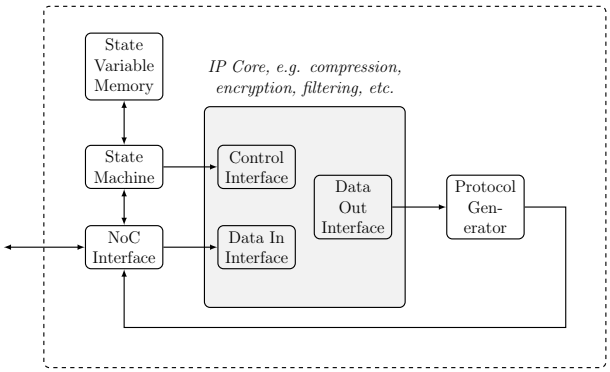

Fig. 2. Stream Processor Architecture.

A typical architecture of such a stream processor is shown in Figure 2. An IP core of the desired functionality is embedded into a wrapper. This wrapper comprises a Network on Chip (NoC) interface for the data exchange, some state machine logic and a memory for state variables. Input control words are interpreted by the state machine while input data words are directly fed into the IP core. An additional memory holds all variables necessary to configure the IP core. If the processing pipeline uses a specific protocol, a protocol parser and/or protocol generator may be added to the input and output of the core. Partitions which can host such a stream processor are implemented on SRAM-based FPGAs. They are connected to a packet-switched, flow-controlled NoC and can be reconfigured during operation by means of dynamic partial reconfiguration.

Without any loss of generality, the here presented work is based on a NoC implementation called SoCWire [2]. SoCWire is a minimal version of SpaceWire, a popular point-to-point network architecture standardised by the European Space Agency [3]. The flow-control between two network nodes is shown in Figure 3. Each network packet may start with a logical address (that is typically used for routing within switches) and is terminated by an End of Packet (EOP)

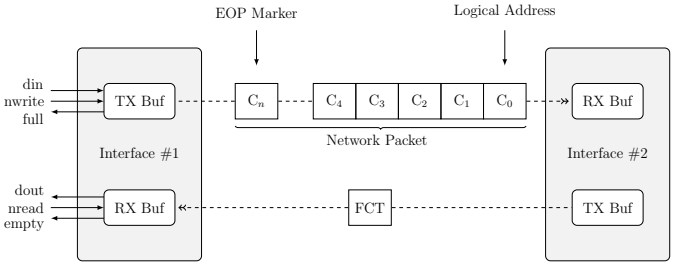

Fig. 3. Flow-control mechanism between two network nodes.

marker. Every time the receive buffer has space for eight more characters, the receiving node sends out a Flow Control Token (FCT). Therefore, the receiving node can easily apply back-pressure to the communication channel, i.e. it can force the source node to freeze by ceasing the transmission of further FCTs.

The paper is structured as follows. In Section II, the motivation for the development of the new voter design is explained. In Section III, a brief overview of related work is given. In Section IV, different failure modes are analysed which must be anticipated to occur in the output network streams of processors. Based on this analysis, a robust voter design is proposed in Section V which is able to cope with all failure modes found. Finally, in Section VI, a hardware implementation is evaluated by means of fault injection experiments.

II. Problem Definition

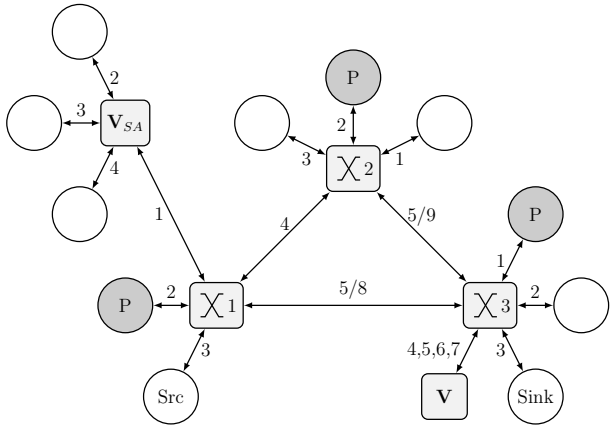

Fig. 4. Distributed Failure Detection: Example network topology.

Our *Distributed Failure Detection* methodology, first outlined in [4], makes failure detectors part of the network. This novel approach allows the free distribution of redundant processors throughout the network because the output of each processor can be routed to any failure detector, independent of its location in the network. As some processors can be quite resource-demanding, the possibility to place redundant processor instances even on different FPGAs can be a great benefit for some applications.

An example network topology is shown in Figure 4. Several partitions (circles) are interconnected via routing switches. A processor has been triplicated and the resulting instances (gray circles) are placed on some of these partitions. Say, data is sent from a source node *Src* to the processor and the processor sends the resulting data to sink node *Sink*. As the processor is triplicated, the data must be first broadcast which is typically done by the routing switches. For instance, routing switch 1 broadcasts the packets to output port 2, 4 and 5. In switch 2 and 3, the packets are then routed to the other two redundant instances. After processing, the resulting packets are routed to the failure detector which in this case is the voter module **V**, connected to routing switch 3. Finally, the output of the voter is routed to the sink node. It is quite likely that the packets do not arrive simultaneously at the redundant processors. The latency between each redundant processor and the voter may differ too. In addition, the partitions are possibly implemented in different clock domains. As a result, the voter module must be able to deal with asynchronous network streams.

Due to an embedded broadcast mechanism, the voter module can also be used as a stand-alone device. An example is shown in Figure 4 where three partitions are directly connected to the stand-alone voter module $\mathbf{V}_{SA}$. Network packets arriving via link 1 are broadcast to all three partitions. Then, the data returning from the three partitions is voted and the result leaves the voter module again via link 1.

In this paper, implementation details of the proposed voter module $\mathbf{V}_{SA}$ are discussed. As the voter module is part of the network, it must be able to deal with asynchronous network streams because (i) the redundant stream processors might be in different clock domains and (ii) the network packets arriving from these redundant processors can be misaligned due to different latencies within the network. Aside from simply voting the content of the network packets, the voter module must also be able to detect failure modes which are typical for networks. For instance, a faulty network node may cease the transmission of packets or it may start to transmit packets with random content and size.

III. Related Work

The concept of using majority voters as failure detection and masking mechanisms for the improvement of the reliability of computer systems goes back to the early 60s. For instance, Brown et al. presented the concept of Triple Modular Redundancy (TMR) in [5] and Fleck his redundancy techniques for reliable flight-control computers in [6]. Every now and then the topic emerges with new technologies again. One example from the heydays of the microcontroller is the fault-tolerant microcomputer system proposed by Yang and Smith in 1986 [7]. In recent years, reconfigurable SRAM-based FPGAs for space applications established a popular research field. As these devices are prone to Single Event Effects triggered by radiation, a revival of work on Triple Modular Redundancy could be observed. Two typical mitigation approaches can be found in literature. The first approach applies TMR to the netlist of the circuit. Single bit voters are then placed at appropriate positions of the netlist. The second approach is rather course-grain and shows great similarity with the classic approaches in fault-tolerant computing. Here, a whole processing module is triplicated and only the output is voted. This approach has some advantages because the TMR partitions are clearly separated which allows a quick recovery of a faulty module. There are, however, also some drawbacks like the classic problem of data re-synchronisation after repair. Rather

early work in this field has been presented for instance by Paulsson et al. [8], [9] and Azambuja et al. [10], [11].

To the best of our knowledge, voter modules proposed in literature are always hardwired to the instances they should observe. In contrast, the novel voter design proposed in this paper can be integrated into a flow-controlled network. The design offers some distinct advantages like an increased reusability, automatic data stream synchronisation and the option of a spatial separation between the redundant instances and the failure detector.

IV. Failure Modes and their Effects

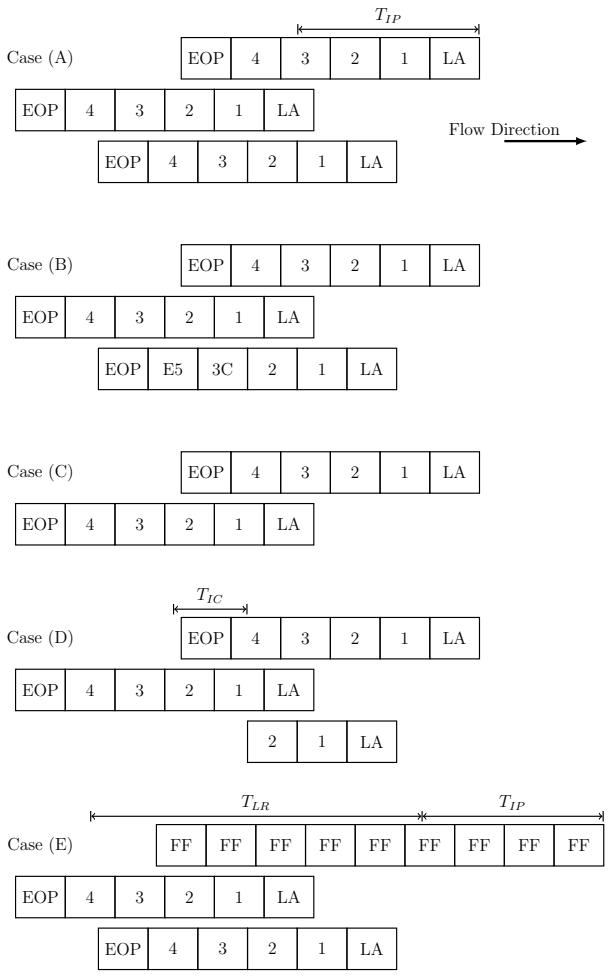

Fig. 5. Different failure modes that can be observed in network streams.

We found in a recent Failure Mode and Effects Analysis (FMEA) that two types of failure modes must be expected, those which affect the payload of network packets (application data) and those which affect the network traffic itself. A short summary of the analysis is shown in Figure 5:

- Case (A): Typical operation. The network packets are identical but may arrive at different points in time due to the aforementioned asynchronicity.
- Case (B): The network packets have an identical structure but their payload differs due to a failure in the application.
- Case (C): One of the network packets does not arrive at all. It seems that the corresponding processor became faulty and stopped the transmission for some reason.
- Case (D): The transmission of one of the network packets suddenly stops before the EOP marker is reached.
- Case (E): One processor becomes a *babbling idiot* and is transmitting confusing data with unpredictable timing (including infinite streams).

In most applications, case (B) will be the most observed one because typically, the probability of a failure in the application is much larger than the probability of a failure in the network interface. This failure mode can be detected by a voter mechanism that compares the (synchronised) network streams character by character. Case (C) can be handled by defining a timeout value T_{IP}, hereafter also referred to as *Inter-Packet Timeout*, which starts once the first packet arrives in one of the *slots* (i.e. a receive buffer assigned to a particular processor) of the voter. It is assumed that all redundant streams arrive within this timeout period during normal operation. If a network packet is missing, however, the corresponding slot (and with that the associated processor) is marked as faulty. Case (D) can be handled by a second timeout value T_{IC}, hereafter also referred to as *Inter-Character Timeout*, which always starts when a new data character arrives in one slot while other slots are still empty. If the timeout expires, it must be assumed that the processor associated with the still empty slot suddenly stopped the transmission and thus the slot is marked as faulty. Dealing with case (E) can be tricky if the data from the babbling idiot arrives much earlier than the data from two other healthy processor instances. Then, the Inter-Packet Timeout would expire first and the two healthy slots would be spuriously marked as faulty (actually all slots would be marked as faulty because further voting is not possible). As it is rather unlikely that two processors fail at the same time, we handle this case by assuming that the early packet is wrong, i.e. the corresponding slot is temporarily marked as faulty. Then, a second timeout value T_{LR}, hereafter also referred to as *Last Resort Timeout*, is started. If the packets from the healthy processors arrive within this timeout period, no further action is required. If they do not arrive, however, all slots must be marked as faulty.

Aside from the aforementioned failure modes, another mode must be considered when broadcasting data. If a processor becomes faulty, it may start to block incoming traffic. This case can be handled by using a non-blocking broadcast mechanism that comprises a Broadcast Timeout. If one of the processors blocks incoming data throughout the timeout period, it is excluded from the broadcast until the end of the current packet transmission.

It is important that the voter degrades to a comparator once one of the slots is marked as faulty. Furthermore, it must be feasible to signal the health status of the slots to an external device, e.g. a microcontroller which can initiate a failure recovery approach. After recovery, the external device should be able to reset the health status register within the module. Then, the voter module will integrate the freshly

TABLE I. VARIABLE DEFINITIONS: BROADCAST

Variable	Description
`min1ActiveIsFull`	Transmit buffer of at least one active NoC Port is full
`allActiveFull`	Transmit buffers of all active NoC Ports are full

repaired processor into the running voting process as soon as possible.

V. VOTER DESIGN

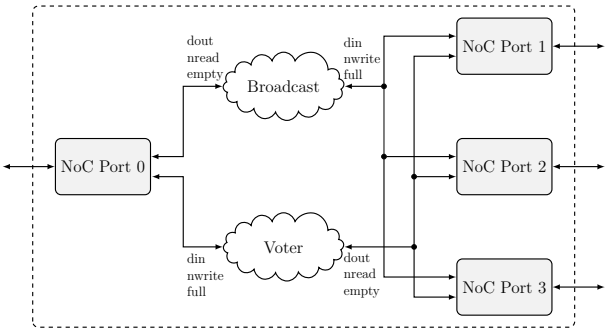

Fig. 6. Block diagram of the voter module.

A block diagram of the voter module with embedded broadcast mechanism is shown in Figure 6. From the perspective of the broadcast sub-module, input data is provided by NoC Port 0 and is broadcast to NoC Port 1-3. From the perspective of the voter sub-module, input data is provided by NoC Port 1-3 and the voted, respectively compared data is sent out via NoC Port 0. The interface between the application layer and each port is rather simple. To write data to the transmit buffer, the input word must be valid at `din` while `nwrite` is pulsed low. The fullness of the transmit buffer is flagged by signal `full`. To retrieve the next data character from the receive buffer (available at `dout`), `nread` must be pulsed low. The emptiness of the receive buffer is flagged by signal `empty`. A signal not shown in Figure 6 is called `active` which is asserted high when the corresponding NoC interface is up and running.

A. Broadcast

The broadcast mechanism comprises a small state machine with two states `S1` and `S2`. The machine remains in the first state until the beginning of a new packet is detected, i.e. when the receive buffer of NoC Port 0 is not empty anymore. Then, the timer is pre-loaded with the Broadcast Timeout value and the `broadcast_en` register with the `active` signal:

```
if not empty(0) then
    timer             := Broadcast Timeout Value
    broadcast_en(3:1) := active(3:1)
    state             := S2
end if
```

Once in state `S2`, the packet is broadcast to all enabled ports. In principle, data is only written to the transmit buffers of the enabled output ports when data is available in the receive buffer of NoC Port 0. On the other hand, data is read from NoC Port 0 only if there is some space in the transmit buffers of the enabled output ports. If one of the redundant processors becomes faulty and starts to block incoming traffic, the broadcast mechanism must ensure that the remaining processors still receive the input data stream. For this reason, the timeout counter is enabled if the `full` flags of the enabled output ports disagree because once a processor blocks incoming traffic, all buffers in the network path will fill up, including the transmit buffer connected to the broadcast mechanism. It is, however, absolutely valid that all processors block incoming traffic during normal operation. Thus, the timeout counter is always reloaded when either all transmit buffers of the active output ports are full or empty (see Table I for variable definitions):

TABLE II. VARIABLE DEFINITIONS: VOTER

Variable	Description
`data_available(3:1)`	not `empty(3:1)`
`min2SlotsOk`	At least two slots are marked as healthy
`min1OkSlotHasData`	At least one healthy slot has data available
`min2OkSlotHaveData`	At least two healthy slots have data available
`all0kSlotsHaveData`	All healthy slots have data available
`all0kSlotsHaveEOP`	An EOP character arrived in all healthy slots
`all0kSlotsVotedOk`	Data in all healthy slots is identical

```
timerReload := not min1ActiveIsFull or allActiveFull
```

If the timeout expires anyway, one of the enabled output ports seems to block incoming traffic. Then, the `broadcast_en` register is updated with the negated full flag of the corresponding port, i.e. those ports with full transmit buffers are marked as faulty and are therefore disabled. State `S2` is left once an EOP character has been detected or if all output ports has been disabled:

```
nread(0)    := min1ActiveIsFull
nwrite(1:3) := empty(0) or min1ActiveIsFull

if timerReload then
    timer := Broadcast Timeout Value
else
    if timer = 0 then
        broadcast_en(3:1) := not full(3:1)
    else
        timer := timer - 1
    end if
end if

if dout(0) = EOP or broadcast_en(3:1) = "000" then
    broadcast_en(3:1) := "000"
    state             := S1
end if
```

A diagram of the circuitry that is active while the machine is in state `S2` can be seen in Figure 7. To simplify matters, only the control signals are shown.

B. Voter

Compared to the broadcast mechanism, the design of the voter circuit is more complex as it needs to take both data and timing failures into account. The voter comprises a state machine with five states `S1 .. S5` as can be seen in the state diagram depicted in Figure 8.

First, some variables are defined which simplifies the following description of the state machine, see Table II. Aside from these variables, `voting_ok(3:1)` is a bus provided

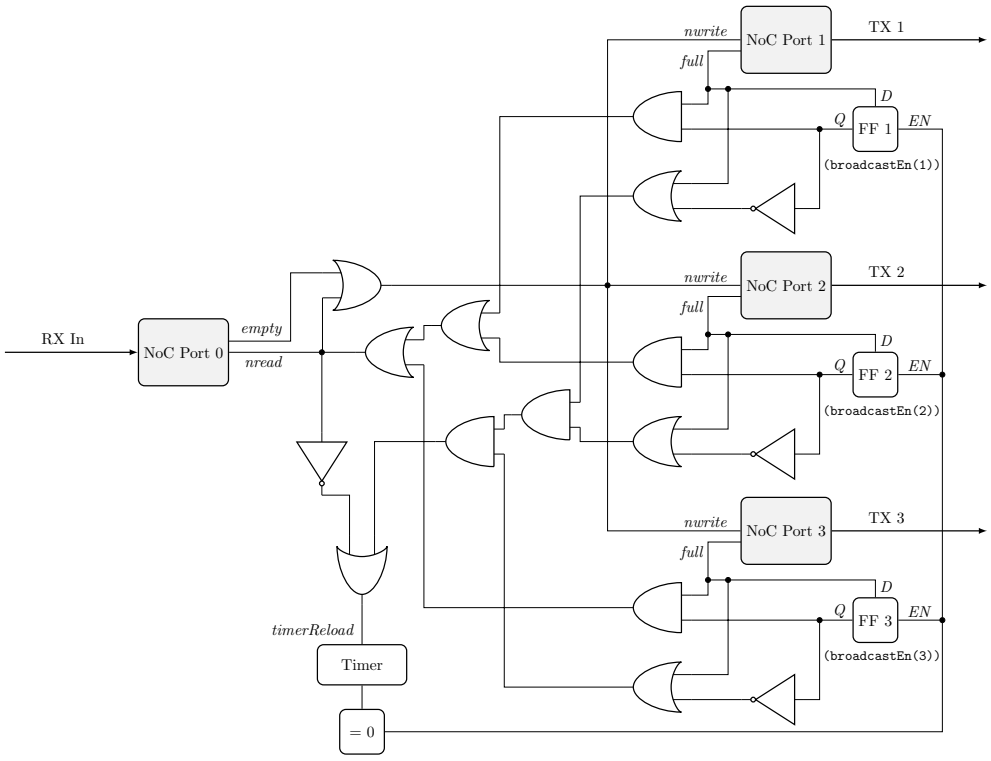

Fig. 7. Circuit of the non-blocking broadcast mechanism.

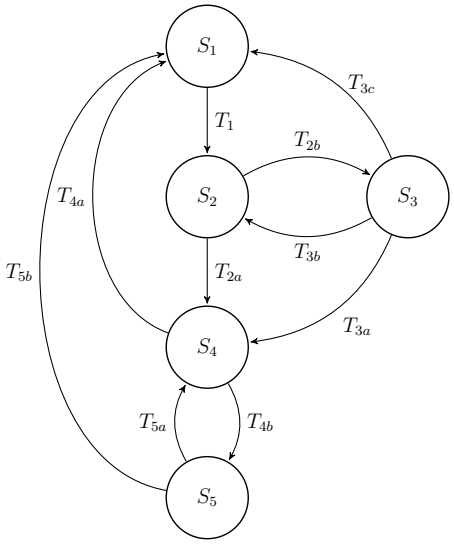

Fig. 8. State diagram of the voter sub-module.

by the embedded *Word-Voter* [12] which is implemented as follows:

```
voting_ok(3:1) := "111" when dout(1)=dout(2)=dout(3)
                  "011" when dout(1)=dout(2)
                  "101" when dout(1)=dout(3)
                  "110" when dout(1)=dout(2)

din(0) := dout(1) when voting_ok(3:1)
                       = "111" or "011" or "101"
          dout(2) when voting_ok(3:1)
                       = "110"
```

The signal `slot_status(3:1)` is a register which stores the health status of each slot and which can also be updated by an external instance. The registration of the new status is done in several steps. If a write enable signal is active, the new status is temporarily stored and a pending flag is set for each updated slot.

Then, the pending flag register is moved to a temporary register once data is flowing in all pending slots again. Afterwards, a status update flag `reg_status_update` is set for each slot in which an EOP character has been detected. This flag is used by the state machine to finally reactivate the slot in its idle state.

Waiting for the data to flow again is necessary since there might be some delay between the registration of the new status and the arrival of the first valid data. Waiting for the EOP character is necessary due to the integrated *spilling* mechanism: If a slot is marked as faulty, all incoming data in this slot is deleted (or spilled) to prevent other parts of the network from being blocked. After the slot is marked as healthy again, the next arriving packet in this slot will be valid (detectable by an EOP of the current packet). Thus, the spilling must be stopped once a status update has been registered successfully. This is achieved by setting the following default value for the `nread` signals:

```
for i = 1:3 loop
    nread(i) :=
        slot_status(i) or reg_status_update(i)
end loop
```

1) State `S1`: This is the idle state in which the state machine remains until a new packet arrives in at least one healthy slot under the premise that at least two slots are healthy.

Furthermore, if the slot status register has been updated from an external instance in the meanwhile, the new status is registered now.

```
for i = 1:3 loop
    if register_status_update(i) then
        slot_status(i) := register_status(i)
    end if
end loop

-- Transition T1
if min2SlotsOk and min1OkSlotHasData then
    timer := Inter-Packet Timeout Value
    state := S2
end if
```

2) State S2: The role of this state is the synchronisation of the incoming data streams. Since data is not taken out of the receive buffers yet, back pressure is applied to the slots where data arrives earlier. While the state machine remains in this state, the Inter-Packet Timeout counter is active. The state is left under two possible conditions: Either data has arrived in all healthy slots or the timeout expired. In the first case, the state machine proceeds to the normal voting operation in state S4. In the latter case, the next action is determined in state S3.

```
timer := timer - 1

-- Transition T2a
if allOKSlotsHaveData then
    timer := Inter-Character Timeout Value
    state := S4
-- Transition T2b
elsif timer = 0 then
    state := S3
end if
```

3) State S3: This state is entered once the timeout has expired in state S2, i.e. not every healthy slot received a packet. The state is left under three possible conditions. If only two slots were healthy before, all slots must be marked as faulty and the state machine moves to idle state S1. If all slots were healthy and two packets arrived, the slot without packet is marked as faulty and the state machine moves on to state S4. If three slots were healthy and only one packet arrived, the slot in which this packet arrived is marked as faulty and the Last Resort Timeout value is loaded into the timer. Then, the state machine moves back to state S2.

```
if slot_status(3:1) = "111" then
    -- Transition T3a
    if min2OKSlotsHaveData then
        for i = 1:3 loop
            slot_status(i) := slot_status(i) and
                              data_available(i)
        end loop
        state := S4
    -- Transition T3b
    else
        for i = 1:3 loop
            slot_status(i) := not(
                              slot_status(i) and
                              data_available(i))
        end loop
        timer := Last Resort Timeout Value
        state := S2
    end if
-- Transition T3c
else
    slot_status(3:1) := "000"
    state            := S1
end if
```

4) State S4: During the voting, respectively comparison of the incoming data, the state machine remains in this state. Data is only read from each healthy slot if the transmit buffer of the output port is not full. On the other hand, data is only written to the transmit buffer if at least two healthy slots have data available. As long as some healthy slots have data while some others are empty, the Inter-Character Timeout counter is active. Once all slots have data available, however, the counter is reloaded. This state is left under two conditions: If the current packet was transferred successfully, the state machine moves back to idle state S1. If the Word-Voter flags a mismatch or the timeout expired, the state machine moves to state S5.

```
for i = 1:3 loop
    if slot_status(i) = '1' and full(3) = '0' then
        nread(i) := '0'
    end if
end loop

if min2OKSlotsHaveData then
    nwrite(3) := '0'
end if

if allOKSlotsHaveData then
    timer := Inter-Character Timeout Value
else
    timer := timer - 1
end if

-- Transition T4a
if allOkSlotsHaveEOP then
    state := S1
-- Transition T4b
elsif timer = 0 or (allOKSlotsHaveData and not
                    allOkSlotsVotedOk) then
    last_char_voting_ok(3:1) := voting_ok(3:1)
    state := S5
end if
```

5) State S5: This state is entered if either the Inter-Character Timeout has expired or a voting mismatch has been detected. If only two slots were healthy so far, all slots are marked as faulty and the state machine moves to idle state S1. Otherwise, it is checked if the timeout expired or a voter mismatch occurred. In the first case, the empty slot is marked as faulty. In the latter case, the slot that disagreed with the other slots during voting is marked as faulty. Then, the state machine moves back to state S4.

```
-- Transition T5a
if slot_status(3:1) = "111" then
    if timeout expired then
        for i = 1:3 loop
            slot_status(i) := slot_status(i) and
                              data_available(i)
        end loop
    else
        for i = 1:3 loop
            slot_status(i) := slot_status(i) and
                              last_char_voting_ok(i)
        end loop
    end if
    state := S4
```

```
-- Transition T5b
else
    slot_status(3:1) = "000"
    state := S1
end if
```

VI. Evaluation

The voter module has been successfully implemented on a Xilinx Virtex-4 SX55 FPGA as part of the DRPM hardware system, see Section I. If the network data width is chosen to be 16 bit wide, the design occupies 1,237 slices (5%) and the achievable clock frequency (after place & route and without further timing constraints) is ca. 107 MHz, i.e. the design can easily handle data streams with throughput rates of 1.6 and more Gbit/s. The utilisation of logic resources is ruled by the four NoC interfaces.

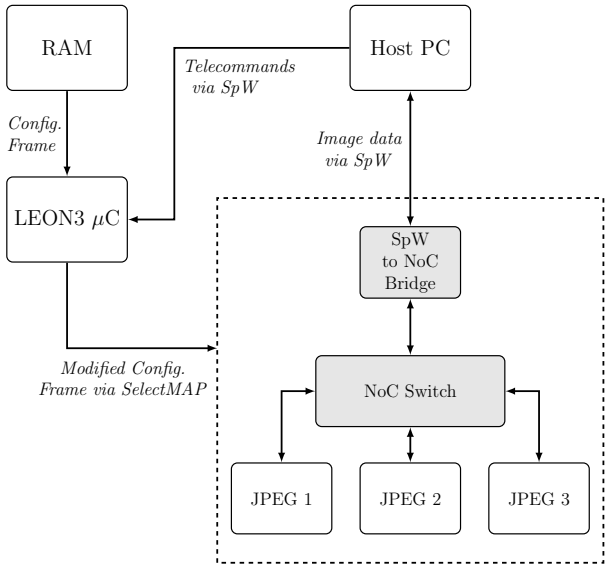

Fig. 9. Fault injection system.

To evaluate the functionality of the voter module, a fault injection system has been set up as outlined in Figure 9. Within the FPGA fabric (dashed box), three reconfigurable partitions are connected to a SoCWire routing switch. In addition, the switch is connected to a SpaceWire-to-SoCWire bridge which allows the communication between the Host PC and a processor installed on one of these partitions. An image compression stream processor has been implemented and redundant instances are placed on each partition. The Host PC retrieves a video stream from a webcam and transmits the raw image data via SpaceWire (SpW) to one of the processors which replies with compressed JPEG images. The Host PC interacts with a LEON3 microcontroller which can inject a fault into the configuration memory of the FPGA on demand. Therefore, the Host PC can initiate a fault injection and check the functionality of the stream processor afterwards by sending a raw image to the processor and analysing its response. In the course of a fault injection experiment with 150,000 random faults per partition, a SQL database has been created which eventually contained more than 20,000 sensitive configuration bits for each partition that are guaranteed to lead to measurable failures. The database contains the information necessary to address the sensitive configuration bits, i.e. the Frame Address Register (FAR), the word position within the frame and the bit position within the word. Furthermore, the failures are classified during the fault injection campaign and the results are stored in the database too. For instance, the failures are classified by their occurrence (application data or network protocol) and by the way the system can be recovered from the failure (scrubbing, scrubbing with subsequent reset or partial reconfiguration).

After the fault injection campaign, the SoCWire routing switch has been replaced by a stand-alone voting module and its slot status signals have been connected to a General Purpose Input/Output (GPIO) Port of the LEON3 microprocessor. If a slot fails, the falling edge of its status signal triggers an interrupt which is caught by a software routine. Then, the software routine initiates a partial reconfiguration of the corresponding stream processor and updates the health status register in the voter module.

For debugging purposes a second bus is connected to the LEON3 which helps us to understand why a slot has been detected as being faulty. With the aid of the SQL database, it is now easy to precisely test the voting module in hardware. A few hundred sensitive bits have been tested with a 100% success rate. The voter always detected the failure and the failure recovery mechanism as well as the reintegration of repaired partitions worked flawlessly. Although no detailed statistics have been collected, the assumption was correct that most failures are detected due to a voter mismatch, although Inter-Packet and Last Resort Timeouts can be observed too. The only detection mechanism that could not be observed yet is the Inter Character Timeout. It can be assumed that this failure mode is rather rare because the failure must emerge during a transmission. The transmission time is, however, very short for the image compression processor as it sends out packets in small 1 kB sized chunks.

VII. Conclusions

In this paper, the design of a novel voter module has been presented. In fact, the module is much more than just a majority voter: First, it integrates into a network architecture which allows the voting of processor instances that can be freely distributed over the network. Secondly, it is able to synchronise streams and can cope with several failure modes which typically emerge in such streams. Thirdly, it can signal the status of each slot to an external supervisor and can automatically reintegrate repaired processors. Finally, it also includes a broadcast mechanism and can therefore be used as a stand-alone device. Furthermore, the evaluation of a real hardware implementation revealed that the amount of needed logic resources is reasonable, especially considering the high data rates the module is able to deal with. In addition, fault injection experiments confirm that the voter offers an excellent failure detection coverage.

Acknowledgments

Sponsorship from ESA under the NPI Programme, Airbus Defence and Space, UK and the University of Leicester is gratefully acknowledged.

REFERENCES

[1] F. Bubenhagen, B. Fiethe, J. Ilstad, H. Michalik, P. Norridge, B. Osterloh, W. Sullivan, and C. Topping, "Enhanced dynamic reconfigurable processing module for future space applications," in *International SpaceWire Conference*. International Space Wire Conference, June 2010, pp. 475–482.

[2] B. Osterloh, H. Michalik, B. Fiethe, and K. Kotarowski, "SoCWire: a network-on-chip approach for reconfigurable system-on-chip designs in space applications," in *NASA/ESA Conference on Adaptive Hardware and Systems, 2008. AHS '08*, Jun. 2008, pp. 51 –56.

[3] ECSS, "Spacewire - links, nodes, routers and networks. ECSS-E-ST-50-12C," ESA/ESTEC, Tech. Rep., 2008.

[4] F. Siegle, T. Vladimirova, O. Emar, and J. Ilstad, "Adaptive FDIR Framework for Payload Data Processing Systems using Reconfigurable FPGAs," in *Proc. of 8th NASA/ESA Conference on Adaptive Hardware and Systems*, June 2013.

[5] W. G. Brown, J. Tierney, and R. Wasserman, "Improvement of electronic-computer reliability through the use of redundancy," *Electronic Computers, IRE Transactions on*, vol. EC-10, no. 3, pp. 407–416, Sept 1961.

[6] J. J. Fleck, "Redundancy techniques (or reliable flight-control computers," *American Institute of Electrical Engineers, Part I: Communication and Electronics, Transactions of the*, vol. 82, no. 4, pp. 535–546, Sept 1963.

[7] T. Yang and K. Smith, "A proposed fault-tolerant microcomputer system," *Electrical Engineering Journal, Canadian*, vol. 11, no. 3, pp. 138–144, July 1986.

[8] K. Paulsson, M. Hubner, M. Jung, and J. Becker, "Methods for run-time failure recognition and recovery in dynamic and partial reconfigurable systems based on Xilinx Virtex-II Pro FPGAs," in *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures, 2006*, vol. 00, Mar. 2006, p. 6 pp.

[9] K. Paulsson, M. Hubner, and J. Becker, "Strategies to on-line failure recovery in self-adaptive systems based on dynamic and partial reconfiguration," in *First NASA/ESA Conference on Adaptive Hardware and Systems, 2006. AHS 2006*, Jun. 2006, pp. 288 –291.

[10] J. Azambuja, C. Pilotto, and F. Kastensmidt, "Mitigating soft errors in SRAM-based FPGAs by using large grain TMR with selective partial reconfiguration," in *2008 European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, Sep. 2008, pp. 288 –293.

[11] J. Azambuja, F. Sousa, L. Rosa, and F. Kastensmidt, "Evaluating large grain TMR and selective partial reconfiguration for soft error mitigation in SRAM-based FPGAs," in *On-Line Testing Symposium, 2009. IOLTS 2009. 15th IEEE International*, Jun. 2009, pp. 101 –106.

[12] S. Mitra and E. J. Mccluskey, "Word-voter: A new voter design for triple modular redundant systems," in *Proc. VLSI Test symposium*, 2000, pp. 465–470.