Cross-layer Resilience Mechanisms to Protect the Communication Path in Embedded Systems

Tobias Stumpf, Hermann Härtig
Operating Systems Group
TU Dresden, Germany
{tstumpf|haertig}@tudos.org

Eberle A. Rambo, Rolf Ernst
Institute of Computer and Network Engineering
TU Braunschweig, Germany
{rambo|ernst}@ida.ing.tu-bs.de

***Abstract*—With the decreasing feature size of new chip generations, additional protection mechanism are necessary especially to protect safety-critical systems in environments with higher radiation levels. This paper investigates a cross-layer approach combining hardware and software-level techniques into a complementary protection mechanism. This reduces overhead by avoiding overlapping protection without reducing the fault tolerance. The paper starts by introducing one software and one hardware protection mechanism. Then, we discuss the overlap as well as pros and cons of both techniques. Finally, we give an outlook about possible benefits to combine both independent approaches to one cross-layer resilience mechanism.**

I. Introduction

Technology scaling influences the system design and reliability [1]. Because increasing the single thread performance reached its limits, hardware vendors increase the parallelism by scaling up the core count in order to keep increasing performance. This has been enabled by decreasing the feature size, providing several benefits like reduced power consumption and increased transistor density on the one hand. On the other hand, it increases the susceptibility to soft-errors [2], which extends to the whole chip, including the interconnection between the cores, giving rise to the so-called unreliable hardware.

The unreliability can be addressed in software as well as in hardware. In hardware, the effects of such soft-errors are abstracted as bit-flips in registers and memory. In software, it can be abstracted as data corruption or misbehaving execution. Software-based approaches, like replication, checkpoint restart, or encoded processing, can overcome soft-errors or bring the system into a fail-safe state.

A remaining weak point is the inter-core communication. The software layer can add redundancy to detect data corruption, but it is unaware of undelivered messages. Moreover, parts of the message are not visible to software and must be handled at the hardware level. Detecting an error at software level also leads to additional overhead, compared e.g. to error detection and retransmission at the hardware level.

A particle strike can cause single or multiple bit-flips [3]. With decreasing feature size the probability for multi bit-flips, caused by one particle strike, increases. Because ECC can recover only from single bit flips, the messages can be additionally protected in software, which guarantees the message integrity but not its delivery.

In this paper, we investigate how soft-errors, in the form of Single Event Upsets as well as Multi Event Upsets, can impair the whole communication path of a multi-core platform and, more importantly, how to protect it. A cross-layer resilience approach to the problem is discussed, where the protection in software and hardware can maximize the fault-tolerance while avoiding unnecessary accumulated overhead.

II. Related Work

Fault tolerance can be achieved at different abstraction layers. The physical layer ensures the correct transmission between two transmission points, whereas the network or transport layer includes further techniques to ensure that a message is finally delivered to the end point. Because of the wide-spread use of TCP/IP, several fault tolerance approaches exist to extend its reliability. The existing mechanisms focus on different layers. Song et al. [4] developed a fault-tolerant Ethernet protocol using COTS hardware, which is application level transparent. Their approach extends the network stack and adds the fault tolerance functionality between network data link layer and the transport and network protocol layers. Other techniques, like PortLand [5], extend the data link layer. Because higher level protocols like TCP/IP also include fault-tolerance mechanisms, the fault-tolerance overhead is probably higher than necessary.

An overall system approach is made by the Tandem Non Stop technology [6]. It is based on special purpose hardware and the system software was developed with fault-tolerance in mind. Because of the high development effort, the successors of the original Non Stop system goes into the direction of COTS hardware.

Combining hardware and software mechanisms can reduce the fault-tolerance overhead or can improve the overall system performance. Kariniemi and Nurmi [7] designed a NoC where the hardware and software protocols are tightly coupled to increase communication performance. They also considered fault-tolerance in their work.

In the following, we will focus on fault-tolerance to increase the robustness, by decreasing the necessary costs.

III. System Overview & Fault Model

Figure 1 illustrates the system we will harden and which is used for evaluation. On the lowest layer, there are several CPUs which are interconnected. We assume that we have

some CPUs which are more reliable and executing critical software parts, according to the design presented by Engel and Döbel [8]. If we cannot use hardened CPUs, then additional techniques like AN-Encoding [9] are necessary to detect a malfunctioning CPU.

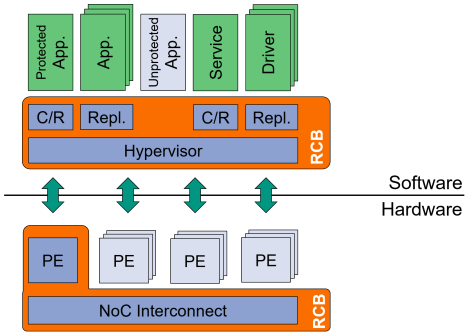


Fig. 1: Many-core system overview.

At software level, we use a microkernel based system. The highlighted part of Figure 1, the RCB, is the critical part which has to be executed on one of the reliable cores. The other software components are protected by existing techniques like Checkpointing/Restart [10] or replication [11].

In this work, we are focusing on transient faults, caused by events such as particle strikes or electromagnetic radiation. Due to its transient nature, after re-executing or rewriting the fault disappears. Moreover, an existing fault, visible at hardware level, may not influence the software at all because it is masked by the hardware or the faulty function unit or memory area is not used at all.

In the sequence, we analyze the failures caused by transient faults at the system level. Next, we discuss the impacts of faults on the different IPC communication steps. In Section VI, we discuss handling the faults in software. In Section VII we extend the analysis to the hardware level and discuss handling faults in hardware.

IV. System Analysis

To examine the failure-vulnerability of our system we performed a full fault analysis for core system functions of the used microkernel. Figure 2 illustrates the results. We grouped the results in four classes: Fault free (OK), system crashes and stops working (CRASH), the system continues work, but the outcome differs, which is called silent data corruption (SDC), and the system does not reach a specific execution point within a specified time frame (TIMEOUT).

System crashes and timeouts can have various reasons. A bit-flip my affect the system state and changes the execution so that an exception is triggered or the execution hangs in an endless loop. Those failures need different memory protection mechanism, which are out of the scope of this paper.

A crash could be the result of a wrongly delivered message and a timeout the result of a non-delivered message. Both cases will be discussed in Section VII.

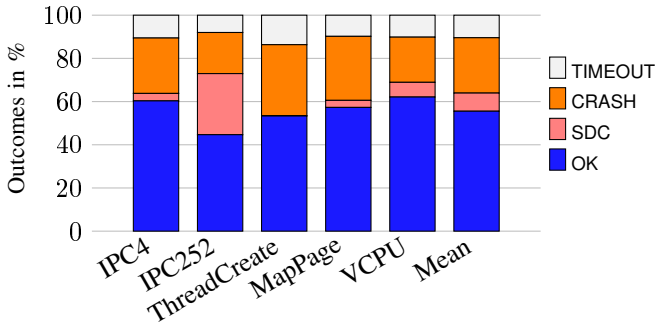


Fig. 2: Fault-injection experiments with L4/Fiasco.OC. Each bar illustrates one experimental setup and covers a basic microkernel functionality.

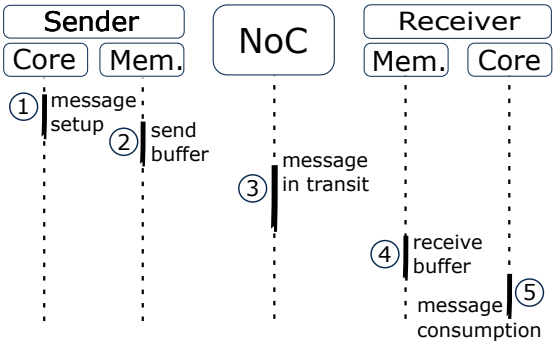


Fig. 3: Steps for delivering a message from client to server.

Further, a crash may also be the result of corrupted message, which leads to a wrong system execution. A detailed analysis of the SDC outcome of the performed test cases indicates that all non-detected failures leading to a wrong system output are the result of corrupted messages.

The next section describes how a message is delivered and explains the different fault cases.

V. IPC communication and fault cases

The inter-process communication in a multi-core system is illustrated in Figure 3. In (1) , a client creates a message and hands over the message to the OS (hypervisor). During these steps, the message is stored in memory (2). In (3), the message is then transferred through the interconnect (the Network-on-Chip) to the receiver, where it is stored in a buffer (4). The OS then hands the message over to the other application (5). Depending on the implementation, the message may be transferred to main memory before being read by the receiver, in which case, the message would go through the NoC and memory once more.

Soft-errors can occur in any of the steps of an inter-process communication and therefore result in different failures. The communication can be protected in hardware (NoC), which guarantees the delivery and integrity of messages. A problem arises when an error corrupts the message before being handed over to the NoC (2), or similarly, after being delivered but before being consumed (4). Even though it is a common

practice to protect the memory with ECC, it is not sufficient [12], especially for future systems with decreased feature size.

VI. Software Mechanism

At software level only some parts of the faults are visible, because some errors are masked by the hardware. Faults manifesting into an error can lead to different results: Malfunctioning software or data corruption. The execution path may be changed because the input values differ, which can also result in a system crash. Other bit-flips in the network packet may influence the outcome of a calculation, which cannot be detected.

For off-chip communication, checksums are state-of-the-art to protect message header and payload. But message transmission within the same die or even across package boundaries of the same system is not appropriately protected [13] or assumed to be fault-free. We expect that the reliability decreases with future systems and makes a message protection within the same die necessary. Therefore, we evaluated a software based approach.

A. Systems Software & Messages

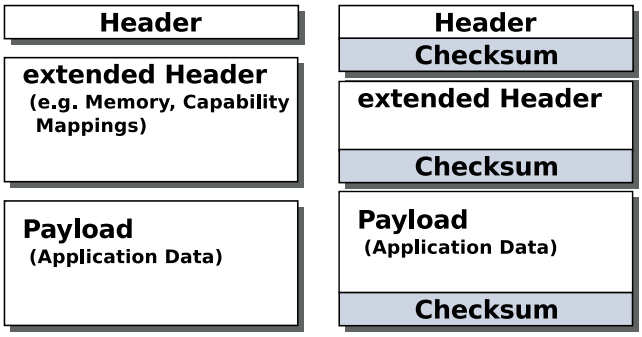
To harden the communication, we looked at a microkernel-based operating system. This is in line with the presented system design in Section I, because the microkernel is part of a small reliable computing base and most software parts can run on top of it, including system software services. In Section V we gave a general overview of the inter process communication. In this section, we focus on the software's point of view.

The simplest form is data exchange. One process creates a message and initiates the transfer. Then, the kernel transfers the data and informs the receiver about the incoming message. For data transfer, a message is composed of two parts, the header (including the receiver, information about the message type, size, etc.) and the payload with the data.

But IPC is not only used to transfer data between sender and receiver. For instance, it is also used to grant access rights. A process A can grant process B access rights to some part of its memory by sending a specific message M_S. The message M_S includes information about the memory area and the rights. Process B has to be ready to receive this information. Therefore, it creates a special message M_R, telling the kernel that it is ready to receive the memory mapping and specifies, at which point its able to receive the mapping.

In addition to communication, IPC messages can be used for OS specific operations, which goes further than simple data transmission. Figure 4a illustrates a simplified version of an IPC message. The extended header is not transferred from the sender to the receiver. Instead, the kernel interprets the information and creates memory mappings or grants access rights.

First of all, we will focus on data transfer only and present our general approach and first measurements. We give an outlook on the advanced part at the end of this section.



(a) unprotected (b) protected

Fig. 4: Comparison of an unprotected and a protected IPC message.

B. Message Protection

A closer look at the results shown in Figure 2 indicates that all SDC corruption is caused by a corrupted message. Using one checksum for the whole IPC package is not feasible, because an IPC packages is modified during transmission, which requires several checks and recalculation during package delivery. Therefore, we added one dedicated checksum for the payload.

To reduce the amount of additional read operations, the checksum is calculated during message creation and finally appended to the message after the last part of the payload is written. We choose two different checksum algorithms: CRC32 and parity byte. In general, CRC32 can deal with more fault cases than a simple parity byte, but the CRC32 checksum is more expensive to calculate.

C. Evaluation

Figure 5 presents the runtime overhead for different implementations. For the measurement, we sent one message with the maximum payload from one thread to another, which gives us an upper bound for the overhead. Our first software implementations of the CRC32 slows down the transmission by a factor of 8 compared to the small overhead of less than 17% percent of a simple parity bit. Therefore, we analyzed two alternatives: A fast software implementation using a table with pre-calculated values. Besides the memory overhead, this solution has the drawback, that the pre-calculated table is again vulnerable to bit-flips. The second alternative uses hardware extensions.

We repeated the experiment with the highest SDC from Figure 2 for the different implementations. For the repeated experiment we see no SDC at all, because the IPC252 experiment sends only data which is now fully protected. The drawback of our experiments is an increased amount of timeouts and crashes.

D. Outlook

To address timeouts and crashes caused by corrupted messages, we currently harden the remaining part of an IPC message. Because the different parts of an IPC message are

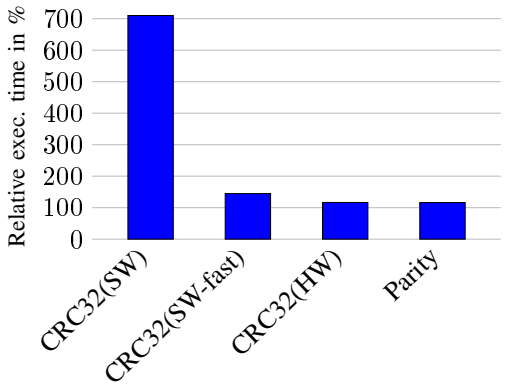
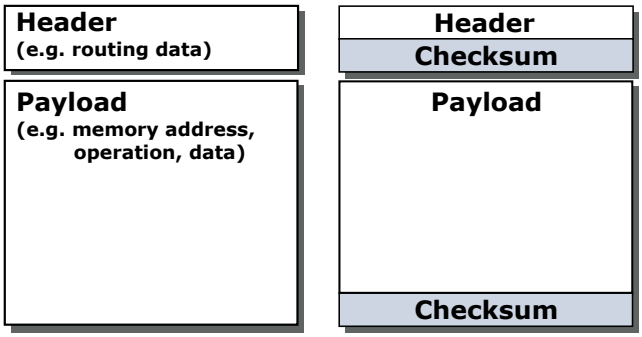


Fig. 5: Runtime overhead for different IPC protection mechanisms.



(a) unprotected (b) protected

Fig. 6: Comparison of unprotected and protected NoC packets.

read or modified at different points, we group the data which is used together. One possible solution is illustrated in 4b, but we will investigate if a more fine grained approach can minimize the protection overhead. If the hardware provides instructions to calculate the CRC32, we will use the hardware-based approach, because it provides the best ratio between overhead and error-detection. For older or cheaper processors without support for CRC32 calculation, we will use a simple parity byte.

The presented software approach addresses failures, which are visible as memory corruption at software level. But it requires that all messages are finally transmitted and the receiver gets notice about the incoming message. If packets, including interrupts, can be lost, further techniques are necessary. In software, we can implement a more complex communication protocol with timeouts and retransmissions. However, we can also design a reliable on-Chip network, to reduce the fault-tolerance overhead caused by additional software. We will discuss such an approach in the next section, followed by a discussion about combining software and hardware techniques.

VII. Hardware Mechanism

The error detection of a software-based approach is limited to delivered packets, including notifying the receiver. If a message is lost while being forwarded by the hardware or delivered to the wrong core, an application may stall forever. Using timeouts in software, also known as watchdogs, solves the problem in some cases while incurring additional delays in the worst-case. In other cases, it does not help. For instance, when forwarding interrupts, which are a special type of message and are related to signals/exception at the software layer. This type of communication is not explicit in software and thus must be handled in hardware.

The NoC protection against transient faults must consider two aspects: the control and the data. The control concerns the network availability, i.e. its capability to restore service after an error affects a router's state machine. The data concerns packet delivery and packet payload integrity. Here, we focus on the latter. We focus on errors affecting data. Therefore, we assume that errors affecting the availability of the network are either handled by an orthogonal approach, such as resilient state machines at design time or error detection and recovery at the component level. We assume that errors affecting control may lead to packet loss, e.g. due to a component reset or irrecoverable routing data. On an end-to-end perspective, transient faults may cause packet loss (i.e. affecting packet delivery) or cause packet corruption (i.e. affecting the payload integrity). These are addressed next.

A. Packet integrity

The impacts of packet corruption depend on whether the packet header or its payload is affected. The unprotected packet is illustrated in Figure 6a.

The packet header contains the routing data, responsible for delivering the packet to the right destination. The payload contains everything else that is not related to routing, e.g. memory address for a memory transaction, access rights, and the data being transferred. The entire IPC message (Figure 4) is here part of the payload of a packet in the NoC. The rest of the payload contains memory address and operation type. Depending on the IPC size, it may be divided and transported as the payload of several packets. The handling of IPCs in hardware and software will be detailed in Section VIII.

We protect the header of the packet and its payload separately. This is shown in Figure 6b. The header's integrity is checked in each router before processing the packet in that router. The payload's integrity is checked in the receiver's network interface. This allows the header to be quickly checked and updated in the routers without requiring the whole packet to be received and processed, as seen in faster wormhole-switched networks [14]. In these networks, the packets are composed of Flow Control Units (flits) and each flit has a header which is encoded such that each flit has a checksum and is checked separately; the payload is distributed among the flits has only one checksum that covers the entire payload (not shown in the Figure).

B. Packet delivery

To provide reliable transmission of data in the NoC without incurring high area overhead in hardware, we implement end-

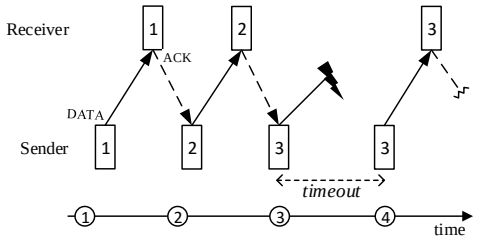


Fig. 7: Illustrative example of Stop-and-Wait ARQ.

to-end transport protocols. These protocols may be selected and configured by software through registers in the network interface. In this work, we consider ARQ retransmission protocols, such as Stop-and-Wait and Go-Back-N [14]. Other transport protocols, such as Multipath Routing (MPR) [15] and other optimized ARQ protocols, can also be used. Disabling the reliable packet delivery is also possible, for instance in the case of sensor readings, whose readings may be occasionally lost (packet integrity however must be ensured).

ARQ-based protocols are widely used to guarantee packet delivery [14]. Its simplest variant is Stop-and-Wait, illustrated in Figure 7. For each packet sent (1), the sender node waits for an acknowledgement from the receiver node before sending the next packet (2) or retransmitting after a timeout (4), in case of error (3). The throughput is improved in Go-Back-N, which allows n packets to be sent (the *send window*) before stopping and waiting for an acknowledgement [14].

C. Evaluation

We evaluated the mechanism with fault injection experiments. The NoC was modeled and simulated in OMNeT++. The traffic was generated by applications from the benchmark MiBench [16] instantiated in a 3x3 2D-mesh NoC [17]: susan and qsort in Double Modular Redundancy (DMR), blowfish, bitcount. Errors were injected randomly in the entire NoC with a varying Bit Error Rate (BER).

First, we evaluate the impact of errors on the NoC without a hardware mechanism to guarantee the packet delivery. Figure 8 shows the packet delivery (considering integer packets) as the BER increases. Because of corruption and errors in routers, more packets are dropped as the BER increases. The BERs utilized in the experiments are artificially high to explore the performance of the approach under stress and lower the simulation time. In practice, soft errors do not occur so often (BER$< 10^{-9}$).

Now, we evaluate the performance of the transport protocol to guarantee packet delivery. Stop-and-Wait ARQ was employed as the transport protocol. Figure 9 shows the variation in latency when increasing the BER, relative to the error free scenario. For each BER, the plot shows the mean value over the variations of latency from all application instances w.r.t. the error free scenario. ARQ is able to deliver all packets as the error rate increases (guaranteed delivery). Since it employs retransmission to do so (which includes timeouts), the impact of increasing error rates is seen in the maximum

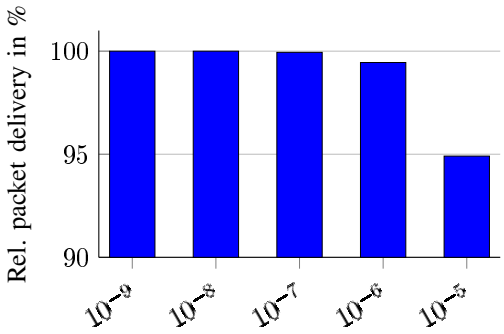


Fig. 8: Impact of errors on MiBench traffic as BER increases. Variation of packet delivery w.r.t. the error free scenario.

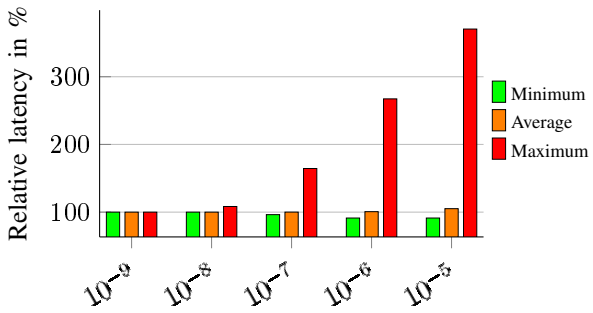


Fig. 9: MiBench traffic latency as BER increases on an Stop-and-Wait ARQ transport protocol. Variation of latency w.r.t. the error free scenario.

latencies. Despite that, the average latency almost does not increase, since errors are not the general case but exceptions. Interestingly, the minimum latencies decrease. This is due to the fact that packets are dropped because an error (cf. Figure 8) causing a temporary decrease of traffic interference for some packets.

D. Outlook

The presented hardware approach is able to handle soft errors occurring in the NoC and affecting any communication. The protection, although transparent to the software execution, is configurable by software, which can select the transport protocol employed. It covers all explicit communication, such as IPCs and shared memory accesses, and also implicit communication, which are usually transparent to software, such as cache line misses, memory coherence protocol messages, and interrupt forwarding.

VIII. Hardware vs. Software

After reading Sections VI and VII, one may notice that the data protecting is overlapping. Both hardware and software protect the transmitted data with checksums. This is illustrated in Figure 10. The IPC message is protected by software and also protected by an additional checksum in hardware. Now a question can be raised: should the protection for IPCs be removed from hardware or from software?

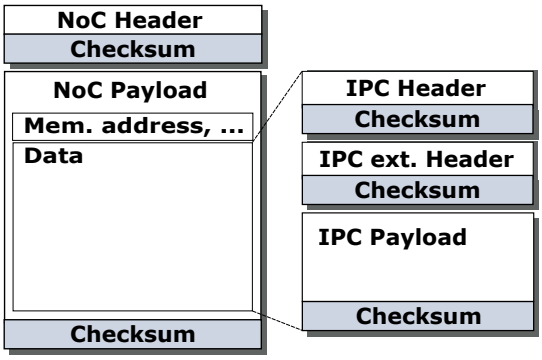


Fig. 10: The relation between the protected NoC packet and the protected IPC message.

Can we remove the software protection? After delivery, the message remains in the local memory and is still prone to errors until the data is consumed.

Can the hardware protection be removed? Some parts of the IPC message is not visible to software, e.g. memory address and other control information, and has to be protected in hardware. This cannot be neglected as it can lead to memory corruption inside the RCB, a case of error propagation. Other parts can be protected in software, e.g. the IPC data itself. Moreover, IPC messages are only part of the NoC traffic. Non-IPC traffic (e.g. interrupt delivery) has to be protected in hardware.

Finally, it is not a question of either-or. It is about how both techniques can cooperate in synergy for the sake of performance and efficiency. We will investigate the possibility of disabling the hardware protection when transmitting messages already protected by software.

A possible solution is to disable the hardware checksum for data when transmitting IPC messages. The integrity check is then only performed in software avoiding double overhead. The solution is illustrated in Figure 11, where the data part of the payload is not covered by the checksum in hardware. The special packet format is configured in the network interface of the sender and applies only to IPC messages (differentiated through the memory address).

Let us reason about the benefits. Considering that a packet in the NoC is able to transport 32 Bytes of data and assuming that it would be protected by a checksum 10 bits long (e.g. CRC-10 [18]), the approach would avoid CRC generation and check for these packets and reduce the amount of transmitted data in 4%. Increasing the packet length to transport 64 Bytes of data and increasing the checksum to 12 bits (e.g. CRC-12 [18]), the reduction of the amount of transmitted data decreases to 2%. Since data corruption will not be detected in the NoC, the retransmission or re-execution of the IPC in case of errors is handled in software.

Another possible approach is to delegate the task of calculating the IPC checksums to hardware. The sender node creates the IPC message in a local memory. After creation, the message is immediately sent through the NoC. The NoC is

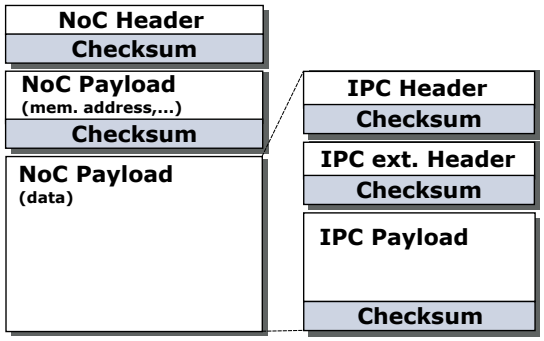


Fig. 11: Optimized protection of a NoC packet for IPC traffic. Checksum is calculated in software.

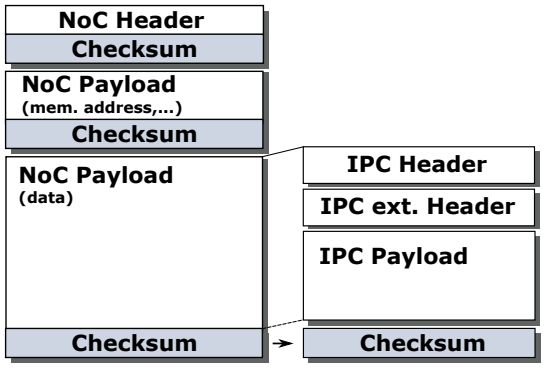


Fig. 12: Optimized protection of a NoC packet for IPC traffic. Checksum is calculated in hardware and checked in software.

then responsible for creating the checksum for the whole IPC message, which takes place together with the transmission. At the destination, the checksum used in the NoC is written to the memory at a specified position in the end of the IPC. The solution is illustrated in Figure 12. The NoC checksum for the IPC is also delivered to the IPC message receiver, which uses it to check the message integrity before consuming it. The check is performed in software.

With this approach, the software overhead can be reduced from 17% to the overhead caused by the hardware implementation, which is one order of magnitude lower. But this benefit comes with the drawback of more complex hardware design, including the hardware implementation of e.g. CRC-32 in each network interface, which increases the chip size and therefore production costs as well as energy consumption.

IX. Conclusion

We have investigated the communication path in an embedded system and how it can be protected to be resilient to soft errors. A software approach can protect data until usage without special hardware. However, it has the drawback of higher overhead in time when compared to a hardware one. Hardware-based protection on the other side increases the production costs because special hardware is needed. A cross-layer approach combining hardware and software techniques can increase the fault-tolerance without prohibitive overheads

in time and area. We have discussed initial ideas towards an efficient cross-layer solution, opening the path for future research.

X. Acknowledgement

This work was partly founded by German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 – spp1500.itec.kit.edu) and the cluster of excellence Centre for Advancing Electronics Dresden (cfAED).

References

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1941487.1941507

[2] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, pp. 10–16, Nov 2005.

[3] R. A. Reed, M. A. Carts, P. W. Marshall, C. J. Marshall, P. J. Musseau, O.and McNulty, D. R. Roth, S. Buchner, J. Melinger, and T. Corbiere, "Heavy ion and proton-induced single event multiple upset," *IEEE Transactions on Nuclear Science*, vol. 44, pp. 2224–2229.

[4] S. Song, J. Huang, P. Kappler, R. Freimark, and T. Kozlik, "Fault-tolerant ethernet middleware for ip-based process control networks," in *Proceedings 27th Conference on Local Computer Networks, Tampa, Florida, USA, 8-10 November, 2000*, 2000, pp. 116–125.

[5] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: A scalable fault-tolerant layer 2 data center network fabric," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 39–50, Aug. 2009.

[6] J. F. Bartlett, "A nonstop kernel," in *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, ser. SOSP '81. New York, NY, USA: ACM, 1981, pp. 22–29. [Online]. Available: http://doi.acm.org/10.1145/800216.806587

[7] H. Kariniemi and J. Nurmi, "Noc interface for fault-tolerant message-passing communication on multiprocessor soc platform," *NORCHIP*, 2009.

[8] M. Engel and B. Döbel, "The reliable computing base - a paradigm for software-based reliability." in *GI-Jahrestagung*, 2012, pp. 480–493.

[9] U. Schiffel, M. Süßkraut, and C. Fetzer, "AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware," in *SAFECOMP '09: Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 283–296.

[10] J. S. Plank, "An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance," University of Tennessee, Tech. Rep. CS-97-372, July 1997.

[11] B. Döbel, "Operating System Support for Redundant Multithreading," Ph.D. dissertation, TU Dresden, 2014.

[12] A. Hwang, I. Stefanovici, and B. Schroeder, "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 111–122. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150989

[13] E. Rambo, A. Tschiene, J. Diemer, L. Ahrendts, and R. Ernst, "Fmea-based analysis of a network-on-chip for mixed-critical systems," in *Networks-on-Chip (NoCS), 2014 Eighth IEEE/ACM International Symposium on*, Sept 2014, pp. 33–40.

[14] A. Tanenbaum and D. Wetherall, *Computer Networks*. Pearson Prentice Hall, 2011.

[15] S. Murali, D. Atienza, L. Benini, and G. De Michel, "A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip," in *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006, pp. 845–848.

[16] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC-4. 2001*, Dec 2001.

[17] B. Motruk, J. Diemer, R. Buchty, R. Ernst, and M. Berekovic, "IDAMC: A Many-Core Platform with Run-Time Monitoring for Mixed-Criticality," in *HASE*, 2012.

[18] P. Koopman and T. Chakravarty, "Cyclic redundancy code (crc) polynomial selection for embedded networks," in *Dependable Systems and Networks, 2004 International Conference on*. IEEE, 2004, pp. 145–154.