Fault-tolerant satellite computing with modern semiconductors

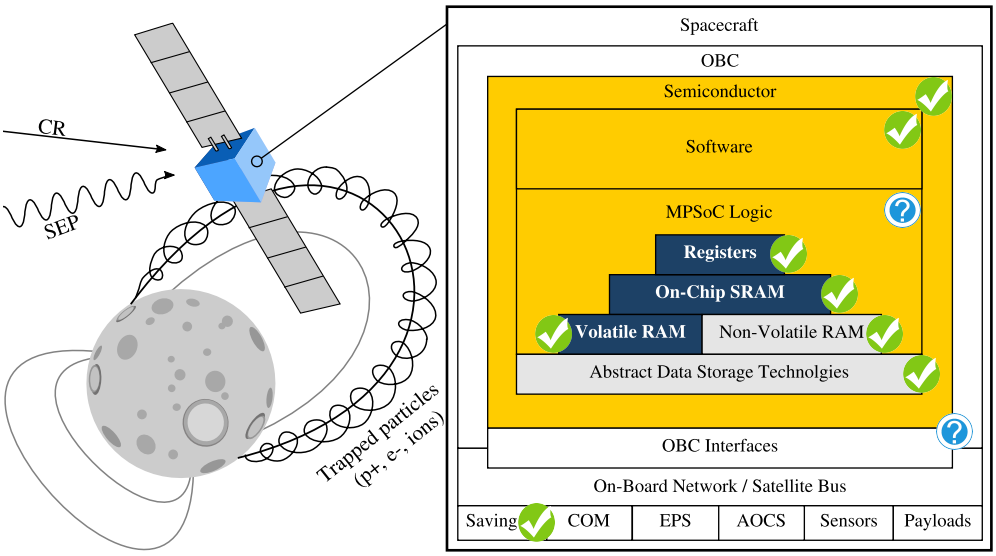
Fuchs, C.M.

Chapter 9

Combining Hardware and Software Fault Tolerance

High-Level System Design

In this chapter, we describe in detail the topology of our multiprocessor System-on-Chip (MPSoC) to address RQ6 by providing an ideal platform architecture for the lockstep described in Chapter 4. We show how it can be assembled in its entirely from well tested COTS components using commodity processor cores and library IP. The resulting MPSoC is the result of a true hardware-software co-design process, and utilizes the concepts presented in the previous chapters. It is designed as ideal platform for our architecture, where each design decision was taken to reinforce the fault tolerance properties of the system as a whole. This chapter therefore servers the final step in developing our fault-tolerant system architecture. In Chapter 10, we then present practical implementation results of this MPSoC.



9.1 Introduction

Satellite miniaturization has enabled a broad variety of scientific and commercial space missions, which previously were technically infeasible, impractical or simply uneconomical. However, due to their low reliability, nanosatellites, as well as light microsatellites, are typically not considered suitable for critical and complex multi-phased missions and high-priority science. The on-board computer (OBC) and related electronics constitute a large part of such spacecraft, and were shown to be responsible for a significant share of post-deployment failure [2]. Indeed, these components often lack even basic fault tolerance (FT) capabilities.

Due to budget, energy, mass, and volume restrictions, existing FT solutions originally developed for larger spacecraft can not be adopted. In this chapter we describe an multiprocessor System-on-Chip (MPSoC) that utilizes conventional hardware, providing FT for miniaturized satellites. The MPSoC is assembled from well tested COTS components, library logic (IP), and powerful embedded and mobile-market processor cores, yielding a non-proprietary, open architecture. Our key contribution is a fault-tolerant OBC architecture for CubeSat use that consists only of extensively validated standard parts, and can be reproduced with minimal manpower and financial resources.

9.2 Background & Related Work

Aboard nanosatellites, subsystems are controlled by just one command & data handling system, whereas aboard a larger satellite these tasks are distributed across multiple dedicated payload and subsystem computers. This implies a varying OBC workload throughout a nanosatellites mission, which traditional FT solutions only handle through over-provisioning. The MPSoC design presented in this chapter can efficiently handle faults through thread migration and partial reconfiguration. Major parts of our approach are implemented in software, allowing the OBC to deliver the desired combination of performance, robustness, functionality, or to meet a specific power budget. To enable strong FT with low-cost commodity hardware, we combine fault detection, isolation and recovery in software, FPGA configuration scrubbing with other fault detection, isolation and recovery (FDIR) measures across the embedded stack.

Nanosatellites today utilize almost exclusively COTS microcontrollers and application processors-SoCs, FPGAs, and combinations thereof [40, 237]. Due to manufacturing in fine technology nodes, and the use of extensively optimized standard IP, they offer superior efficiency and performance as compared to space-grade OBC designs. The energy threshold above which highly charged particles can induce faults (SEE – single event effects) in such components decreases, while the ratio of events inducing multi-bit upsets (MBU), and the likelihood of permanent faults, increase. To adapt such hardware-FT based concepts additional FT-circuitry is required, inflating logic size and producing diminishing returns, resulting in limited scalability and low clock frequencies [188, 190, 192]. We can observe that traditional FT-concepts applied to modern COTS hardware yield no nanosatellite compatible architectures.

While more sensitive to transient faults than ASICs [142, 143], FPGA-based Soft-SoCs have been shown to offer excellent FDIR potential for miniaturized satellites [238]. Transients in critical parts of the FPGA fabric can be scrubbed [242], while permanent faults may be compensated through reconfiguration with differently routed

configuration variants [105]. Fine-grained, non-invasive fault detection in FPGA fabric, however, is challenging, and subject of ongoing research [239, 240]. Relevant FT-concepts thus rely on error scrubbing, which has scalability limitations and cover only parts of the fabric [239, 242]. We overcome these limitations by implementing fault-detection in software through thread-replication and coarse-grain lockstep within an MPSoC using weakly coupled cores.

Tiled architectures [246,328] are often used for well paralellizable applications with many low-performance processor cores. Among others, [329] and [328] showed that this topology can also be exploited to achieve FT for image processing applications with a very specific structure. We combine a compartmentalized topology with a coarse-grained lockstep described in Chapter 4, enabling FDIR without constraining the application type or system architecture. Thus, the architecture presented in this chapter is well suited for platform control and can be used as a template, allowing a high level of OBC design freedom, and enabling a considerable amount of testing to be inherited from COTS components and logic.

Thread migration has been shown to be a powerful tool for assuring FT, but prior research ignores fault detection, and imposed tight constraints on an application's type and structure (e.g., video streaming and image processing [241]). Thread-level coarse-grain lockstep of weakly coupled cores instead supports general purpose computing, and in the past, has already been used for high availability, non-stop service, and error resilience concepts. However, in prior research, faults are usually assumed to be isolated, side effect free, and local to an individual application thread [208] or transient [199, 205], entailing high performance [209] or resource overhead [210, 211]. More advanced proof-of-concepts [198, 199], however, attempt to address these limitations, and even show a modest performance overhead between 3% and 25%, but utilize checkpoint & rollback or restart mechanisms [199], which make them unsuitable for spacecraft command & control applications.

Many of these limitations and obstacles ultimately can be attributed to low maturity, as a majority of software-FT concepts are published as a concept TRL1 but remain unvalidated. Hence, they could be uncovered, and in many cases, can be potentially resolved through implementation and practical validation [198], increasing maturity to TRL2 or TRL3. However, development of a testable proof-of-concept is a time consuming and costly undertaking [300], as outlined among others by Sangchoolie et al. [301] with limited immediate yield for academic publication. Fault injection for entire OS instances is especially non-trivial [302], as thorough preparation and careful tool-selection is necessary to obtain representative results from a fault injection experiment [303]. Therefore, a broad variety of TRL1 software-FT concepts exist today at a theoretical level [212–214], for which validation was only conducted statistically using modeling with different fault distributions or not a all. In this chapter, we therefore conduct validation of our coarse-grain lockstep approach using systematic fault-injection. Thereby we verify the effectiveness of our coarse-grain lockstep FDIR mechanisms under stress using a RTOS-based proof-of-concept implementation, increasing maturity to TRL3.

9.3 A Hybrid Fault Tolerance Approach

Conventional FT architectures require proprietary logic in hardware to facilitate fault detection and coverage. In contrast, the architecture described in this chapter can

offer strong FT using just COTS components and proven standard library logic. This is made possible through the use of the FT approach we presented in Chapter 4. The high-level functionality of this approach is depicted in Figure 62, and consists of three interlinked fault mitigation stages implemented across the embedded stack:

Stage 1 implements forward error correction and utilizes coarse-grain lockstep of weakly coupled cores to generate a distributed majority decision across compartments. Fault detection is facilitated through application callback functions, without requiring deep modifications to an application or knowledge about intrinsics.

Stage 2 recovers failed compartments through reconfiguration and self-testing. It assures the integrity of programmed logic and deploys configuration scrubbing, as well as Xilinx Soft-Error-Mitigation (SEM), to correct transients in FPGA fabric. Its objective is to assure and recover the integrity of processor cores and their immediate peripheral IP through FPGA reconfiguration and the use of differently routed and placed alternative configuration variants, thereby counteracting resource exhaustion.

Stage 3 engages when too few healthy compartments are available, and re-allocates processing time to maintain reliability. To do so, thread-level mixed criticality is exploited, assuring sufficient compute resources are available to high-criticality applications by sacrificing performance or availability of lower-criticality threads.

Further details including benchmark results are available in Chapter 4. The main target in our project is the ARM Cortex-A53 application processor, which is today widely used in embedded and mobile-market devices. However, this research is processor and ISA independent. In this chapter, we describe an MPSoC design and

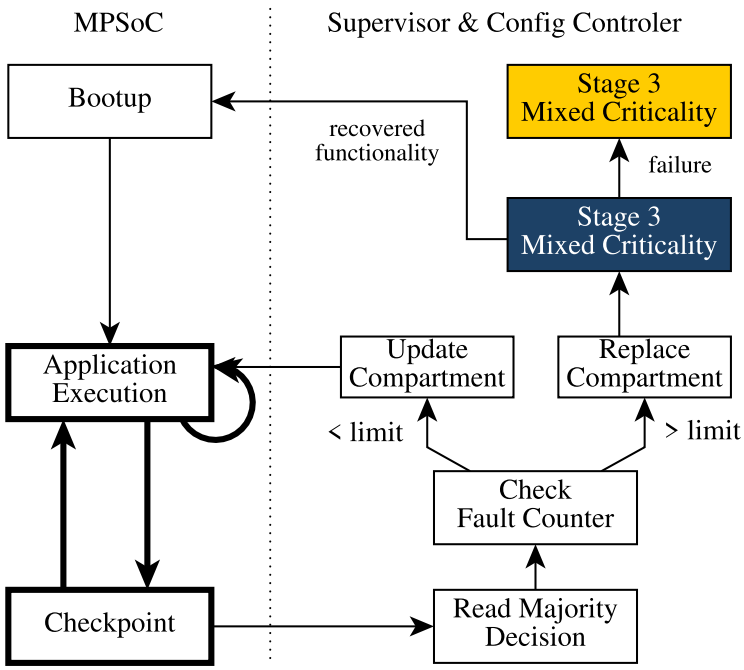


Figure 62: Stage 1 (white) assures fault detection (bold) and fault coverage. Stages 2 (blue) and 3 (yellow) counter resource exhaustion and adapt the on-board computer application schedule to reduced system resources.

architecture template, which is enabled by this approach and can be reproduced in Xilinx Vivado 2017.1 and later.

9.4 The MPSoC Architecture

We developed our software-FT architecture for use on top of an MPSoC consisting only of COTS technology. The main target in our project is the ARM Cortex-A53 application processor. For many size-optimized space applications, smaller cores such as the Cortex-A32, A35 and A5 may also offer a better balance between performance, universal platform support, and logic utilization. The Cortex-A53 core was chosen as it is today widely used in a variety of industrial and mobile-market devices, though our architecture is processor and instruction set architecture (ISA) independent.

In this section, we describe a publicly reproducible MPSoC design variant implementing our architecture, which can be designed in full using Xilinx library IP and Microblaze processor cores. The architecture minimizes shared logic, compartmentalizes compartments, and offers a clearly defined access channel between compartments and the supervisor, and is depicted in Figure 63.

9.4.1 Supervision & Reconfiguration

Stage 1 can be implemented on a single chip, but we utilize an off-chip supervisor to facilitate FPGA reconfiguration and transient fault scrubbing in the running configuration. The outlined multi-stage FT approach puts only minimal load on the supervisor, and it can thus be again implemented using a traditional radiation hardened or tolerant microcontroller. The FeRAM-based TI-MSP430FR family would be a solid somewhat radiation-tolerant but non-FT substitute, which is today widely used aboard a broad variety of CubeSats and low-performance COTS products designed for nanosatellite use. The level of performance offered by such microcontrollers is usually sufficient only for educational CubeSats and federated systems. However, a supervisor

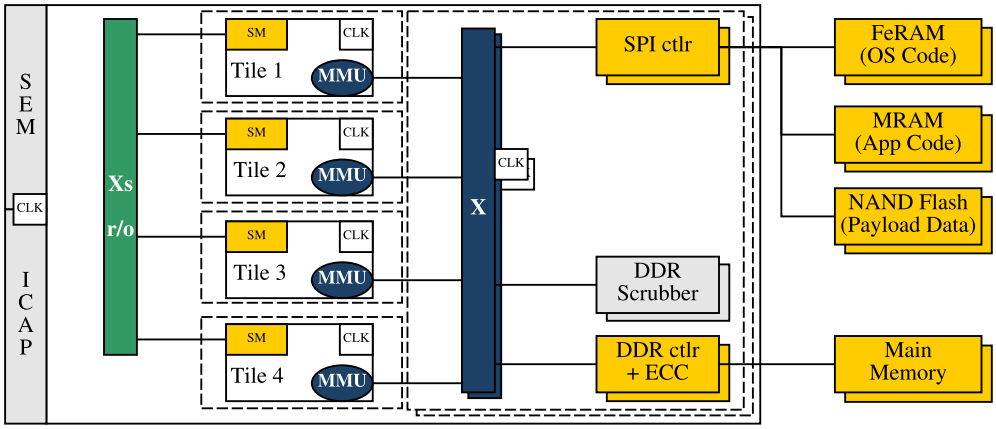


Figure 63: The topology of our compartmentd MPSoC design. Each compartment exists in its own reconfiguration partition and therefore also clock domain, simplifying routing and logic placement. Reconfiguration partitions are indicated with dashed lines.

in our architecture only receives the majority voting results from the coarse grain lockstep, controls the FPGA, and facilitates reconfiguration through an ICAP controller in static logic. Hence, the low level of performance of an MSP430FR, for example, is sufficient, and allows an ultra-low-cost implementation of our approach for academic CubeSat projects and scientific instrumentation.

We deployed configuration error mitigation through Xilinx SEM in combination with supervisor-side scrubbing to safeguard logic integrity. However, SEM and scrubbing only detect faults in specific components of the FPGA fabric (e.g., not in BRAM), leaving significant parts of the design unprotected unless logic-side ECC is used.

These measures alone do not provide sufficient protection for fine-feature size FPGAs. Thus, our software-FT functionality can locate faults in the partition of a specific compartment, allowing the supervisor to resolve them using reconfiguration. We place compartments in separate configuration partitions to enable partial reconfiguration of individual compartments, without affecting the rest of the system.

As depicted in Figure 62, the supervisor only reacts to disagreement between compartments, otherwise remaining passive. It maintains a fault-counter for each compartment and acts as a watchdog. When resolving transient faults within a compartment, it increments the fault-counter and induces a state update through a low-level debug interface. After repeated faults, the supervisor will replace the compartment by adjusting the thread-mapping of a spare compartment, activating it, and rebooting the faulty compartment. In case a system developer indicated threshold is exceeded, the disagreeing compartment is assumed permanently defunct and not re-used as a spare.

To allow supervisor access to a compartment and its address space, each compartment is equipped with an AXI debug-bridge (Figure 64). The supervisor can trigger execution of self-test functionality within a compartment to detect faults in peripherals. It can also trigger an adjustment of a compartment's thread allocation as part of Stages 1 and 3, making the MPSoC's computational performance, robustness and energy consumption adjustable at runtime.

Majority voting between compartments can be implemented as distributed majority decision [330], then requiring no direct intervention of the supervisor during regular operation. If this is not desired, or lockstep through interrupt triggered checkpoints is implemented, then the supervisor should also take care of receiving the voting results generated on each compartment. In that case, the supervisor can access each compartment's thread mapping via each compartment's debug interface, and if necessary induce a reset or otherwise manipulate a compartment without requiring its cooperation.

9.4.2 Tile Architecture

Our MPSoC design implements multiple isolated SoC-compartments accessing shared main memory and OS code. Even though the purpose and function of these compartments is different, the topology resembles a compartmentalized architecture instead of a conventional MPSoC design, in which cores share infrastructure and peripherals. This topology increases Stage 1's fault coverage capacity and allows task mapping for general-purpose software. Each such compartment contains a processor core, local interconnect, and peripheral IP-cores and interfaces as depicted in Figure 64, resides in its own clock domain, and can be reset independently. Allocating a clock domain to each compartment improves timing, and reduces logic-overlap and interdependence

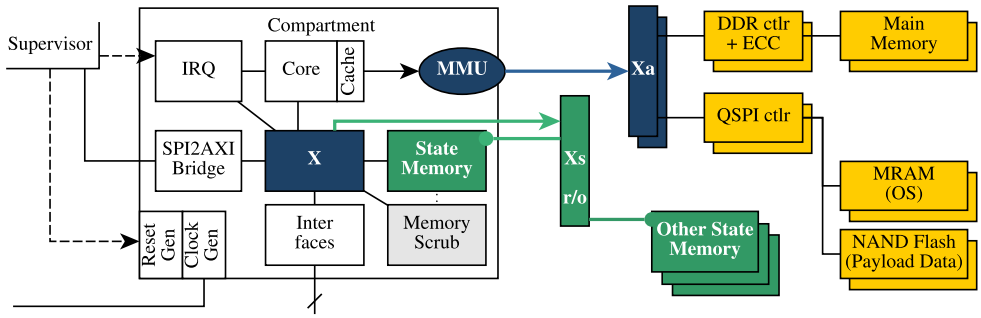


Figure 64: The logic-side architecture of a compartment. Access to local IP bypasses the cache, while access to global memory passes is cached for performance reasons.

between compartments. Furthermore, we can then also utilize partial reconfiguration and frequency scaling for each compartment, as well as clock gating.

A compartment executes a set of thread replicas, and its loss can be compensated by the rest of the system. To assure a failed compartment can not cause performance degradation in the rest of the system (e.g., by continuously accessing DDR or program memory), it can be disconnected off from the global interconnect by the supervisor. Non-masked faults (due to radiation, aging, and wear) disrupt the data or control flow of the software running on a compartment. Stage 1 builds upon this capability at the thread-level, as state differences can be detected by other compartments and often even by the malfunctioning compartment itself as described in Chapter 8.

All compartments are equipped with an identical set of peripheral interfaces, with controllers being mapped to identical locations and address ranges. The compartment address space layout is uniform across the system and compartments are indistinguishable for software. Hence, application code and data structures are portable between compartments, simplifying thread migration drastically. This allows us to reduce the computational cost and complexity of software-lockstepping.

Thread allocation and information relevant to the coarse-grain lockstep is stored in a dedicated dual-ported on-chip BRAM on each compartment. We refer to component is as state memory, and indicate it as SM in the figures. One port is accessible to the compartment's processor core, while the other is read-only accessible to the system. This allowing low-latency information exchange between compartments without requiring inter-compartment cache-coherence or main memory access. The state memory architecture is depicted in Figure 65. The supervisor can access and modify each compartment's state memory through its debug interface on each compartment.

9.4.3 Interconnect Topology and Shared Memory

Figure 63 depicts the MPSoC's high-level topology. Our MPSoC design utilizes an AXI interconnect in crossbar mode to allow compartments access to shared main and non-volatile memory controllers, though we are currently reworking our MPSoC to instead use a NoC [329].

Main memory is shared between compartments, as SD- and DDR memory controllers are too large and require too much I/O to instantiate for each compartment. Each compartment has full access to a segment of main memory, which is mapped to the same address range on all compartments (the MMU component in the figures).

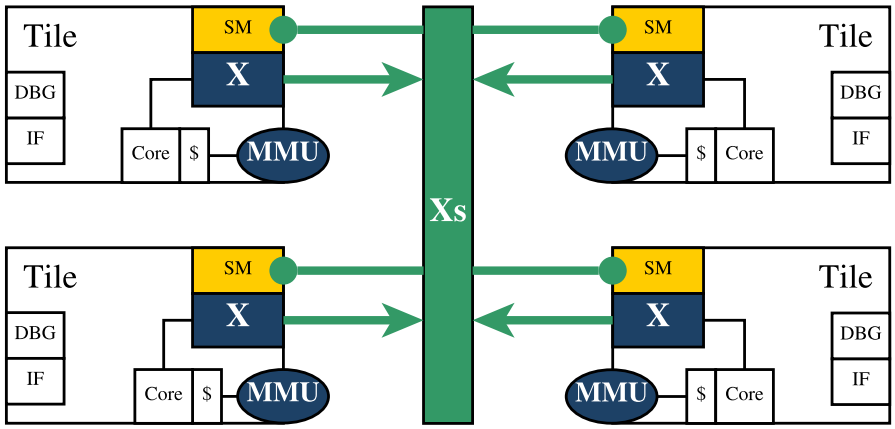


Figure 65: A compartment's state memory is accessible to all other compartments in the system. It provides a write protected, high-speed on-chip possibility to expose state-relevant data to the MPSoC as a while.

All compartments can access main memory read-only to simplify state synchronization and IPC. The supervisor can access each set of main memory controllers directly.

For nanosatellite missions to LEO, often only SECDED ECC support is required and readily available in library IP already [331], while basic error scrubbing can be facilitated in software. For critical, deep-space, and long-term missions, block coding should be used instead to compensate for the increased impact of SEEs and higher likelihood of MBUs in high-density SDRAM. Reed-Solomon ECC as well as error scrubbers are available commercially, or can be assembled from open-source IP. The main memory scrubbers are controlled by the supervisor to avoid potential interference by malfunctioning compartments. ARM Cortex-A53 as well as Microblaze caches and several local memories and buffers offer ECC support as basic functionality [331].

To safeguard main memory, FeRAM [332], MRAM [150], and mass memory from SEFIs, as well as permanent failure, these memories are implemented redundantly to enable failover. To allow non-stop operation during FPGA reconfiguration, we also implement their controllers, and the AXI interconnects they are attached to redundantly. This also enables further protective measures which we described in Chapter 7, and allows load distribution for timing critical main memory through segment interleaving. Thereby the available DDR memory bandwidth is increased and the overall latency for memory access can be reduced. This also enables us to recover an instance of a memory controller on short notice without requiring the full system to be halted[1].

Tiles compete for DDR memory access. As our architecture is implemented on FPGA, the clock frequency of each compartment's processor core is lower as on ASIC implemented MPSoCs. In consequence, the global interconnect as well as DDR memory controllers offer abundant throughput at drastically higher clock frequencies. Each processor core caches access to shared memory, drastically reducing the strain on the memory subsystem. Access to a compartment's state memory still bypasses the cache, but this is implemented directly in high-speed, low-latency on-chip BRAM. Hence,

[1]Note that depending on the used OS, a reboot of a compartment may be required. Linux supports modifications to the memory layout and relocation, while simpler OS, such as RTEMS, do not currently know such functionality.

while in principle competing for memory bandwidth, even an 8-compartment system can not saturate the two available DDR4 channels in our current MPSoC design. Ideally however, our architecture should be implemented using a NoC instead of a global AXI-interconnect crossbar, which would offer drastically better scalability, more effective caching and buffering, and also a degree of FT.

9.5 Subsystem Connectivity and Peripheral I/O

A fault resolved in Stage 1 may cause incorrect data to be emitted through I/O interfaces. This is an inherent limitation of coarse-grain lockstep concepts, and can only be slightly alleviated through additional application-intrusive work-around as described, for example, in [199]. Instead, this limitation is better solved at the logic level through interface-level voting, which is possible with minimal extra logic. For most CubeSats, most nanosatellites, and less critical microsatellite missions, however, this is usually foregone.

Larger spacecraft already utilize interface replication or even voting to assure full hardware TMR, usually requiring considerable effort in hardware or logic to facilitate this replication. Our MPSoC architecture inherently provides interface replications by design, requiring no extra measures to be taken, as the individual compartment-interfaces can be directly used for TMRed architecture. Further safeguards are necessary for very small CubeSats where interface replication is undesirable, for example, due to PCB-space constraints.

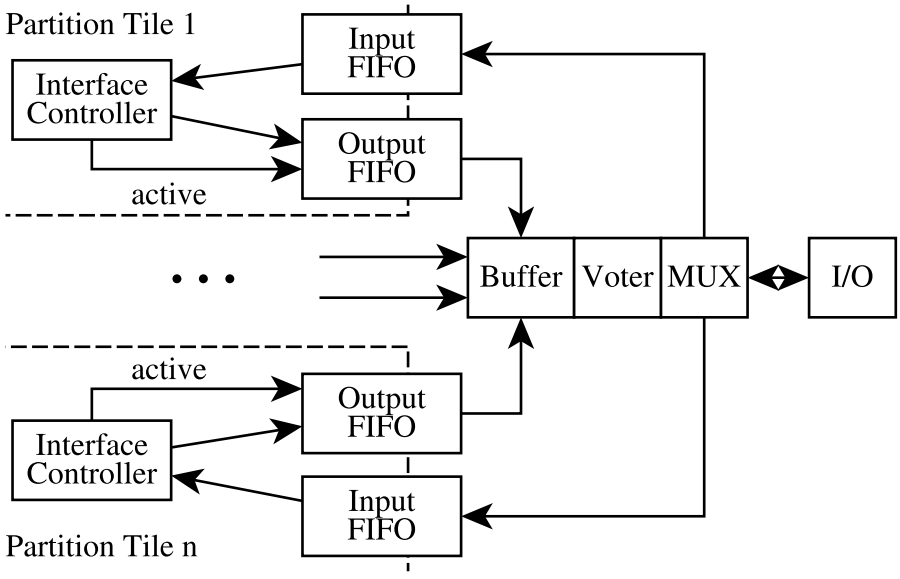


Figure 66: An activation-driven, buffered output voter with input de-multiplexer can be constructed for low-pin-count CubeSat interfaces. Note that an additional re-sampling step would be required in case of different thread scheduling on lock-stepped compartments.

9.5.1 Electrical- and Logic-level Interface Voting

For simple embedded interfaces like I2C and SPI connected to "dumb" sensors or actuators with no user configurable firmware, a simple majority decision per I/O line is possible. While hardware voting is challenging for large arrays of voters running synchronized at very high frequencies, the CubeSat-relevant interfaces are electrically simple, have a very low pin count, and run at relatively low clock frequencies. Hence, voting for these interfaces can efficiently be implemented on-chip through simple voters assuming compartments signals interface activity.

Our coarse grain lockstep mechanisms allow software to be executed with slight timing variations. These may be caused by clock-domain interactions, competition of compartments for global interconnect DDR4 and QSPI access, as well as differences in compartment partition routing and or I/O pin placement. In general, these variations will be limited to few clock cycle duration. I/O on these interfaces must be buffered, which can be done within the FPGA as discussed further also by Li et al. in [333]. For simplicity, compartments should also indicate that an interface is active, and we can double-use the chip-select pins present in almost all I2C and SPI implementations. The voter can use activity on these pins as indication that the interfaces is active, and delay voting for a given amount of clock cycles using a set of FIFO buffers. The depth of these FIFOs thereby determines the maximum delay compensated by the voter [334]. In our design we can utilize a combination of re-sampling majority voter and MUX as depicted in Figure 66.

Note that larger MPSoC variants with 6 or more compartments can host multiple independent lockstep sets as described in Chapter 6. In this case, simple buffered voting is insufficient, as compartments could then also run mixed lockstep groups where threads may be scheduled with much larger time differentials. This differential will always be shorter than the duration of a lockstep cycle or the frame time, but in LEO these may extend to up to several seconds. It would be uneconomical and, depending on the application, even technically infeasible to buffer I/O for long duration. However, we consider the design-combination of a low-end CubeSats that can not afford subsystem TMR, packet-based communication, with a high-performance 6-core MPSoC not very attractive and therefore a corner case. If this combination was still deemed necessary, a straight forward solution would be to maintain multiple isolated thread-assignment groups.

9.5.2 Simple Inter-Subsystem and Controller Networks

Many SPI and I2C implementations support multi-master shared bus operation, and it is possible to even create large and complex CAN-bus networks [335]. CubeSats often use these interface standards for low-speed inter-subsystem communication in simple CubeSat designs [39, 336]. While packet based interfaces offer far better scalability, reliability, and fault-mitigation properties for this purpose [337], in reality these concepts will remain in use aboard CubeSats for the foreseeable future. However, in contrast to interfacing with "dumb" endpoints ICs, these networks[2] usually consist of microcontrollers running satellite developer provided software. In this case, a better solution to de-replicating and obtain consensus within the system of our MPSoC's compartments is to make the subsystems aware of the replication.

[2]In CubeSat jargon often referred to as "buses".

A subsystem controller then can await receiving a second replica of a command sequence from a different master. Of course this does not solve the issue of a single compartment/master jamming or saturating the bus due to malfunction. However, most CubeSats using these interfaces as subsystem-bus currently usually also do not take actual meaningful countermeasures in this regard. This is technically possible, but requires entirely different network topologies [335, 337] than the simplistic single-level bus concepts used aboard CubeSats today [39].

9.5.3 Packet Switching and Routing On-Board Networks

For packet-based interfaces such as Spacewire [338], AFDX [94], CAN [55], or Ethernet [73], no hardware- or logic-side solution is necessary. There, packet duplication and integrity checking can be managed efficiently at the data link, network and transport layers (OSI layers 2 – 4 [339]). At the physical layer, Ethernet and thereof derived technologies such as AFDX [94] and TTEthernet [340] perform shared medium through collision detection and micro-segmentation with frame switching. Then, packet routing (L3) and de-duplication in software at the higher OSI layers can be deployed, e.g., in software. Today, this is common practice in relevant industrial applications such as AFDX and TTEthernet used in related fields such as atmospheric aerospace or safety critical automotive applications.

The FPGAs considered in our research provide an abundance of high-speed MGT transceivers. These are intended to support high-performance serial interfaces such as PCIe, or USB3 host interfaces [341], which may become attractive for CubeSat use in the future and have built in error correction support. Even the smallest XCKU3P part fields 16 such interfaces, and the location of these interfaces is in very attractive locations for using 2-3 of them isolated within each of our MPSoC's compartments [342]. In practice, this would allow for a very scalable, high-performance CubeSat inter-subsystem communication architecture [343] at little cost assuming a the satellite's high-level design takes this into account.

9.6 Implementation Considerations

The MPSoC architecture described in this chapter was developed for miniaturized satellite use, as an ideal platform for the software-FT approach described in Chapter 4. This architecture is not specifically dependent on utilizing ARM processor cores, but can be implemented with any FPGA-implementable soft-core. Our choice of the ARM platform was taken in part to allow thread migration between soft- and hard-cores (e.g., on Zynq Ultrascale+), maximum comparability to COTS mobile-market and embedded MPSoCs with secondary use aboard a major share of CubeSats. Especially for low-budget CubeSat users in research or university projects, standard vendor library cores such as Xilinx Microblaze may be an excellent alternative to our Cortex-A choice. These cores offer erasure coding and other basic fault tolerance features out of the box already, and performed rather well in radiation tests [331]. They are readily available and often even free of charge, especially to academics and non-commercial scientific research users.

We implemented a proof-of-concept on a Xilinx XCKU5P FPGA with modest resource utilization (28% LUTs, 33% BRAMs, 16% FFs, 5% DSPs) and 1.92W total

power consumption with Microblaze cores. In this 4-compartment design, each compartment was equipped each with one peripheral I2C master controller, one SPI master, as well as a dual-channel GPIO controller. Such an interface configuration is representative for most CubeSat applications, while AFDX, TTEthernet, and Spacewire are today not widely used aboard CubeSats.

This approach and architecture could very well be implemented on ASIC without reconfiguration and Stage 2, and we see this as a "big-space" variant of our approach. An ASIC implementation offers lower energy consumption, and allows higher clock rates due to reduced timing and shorter paths. If manufactured in an inherently radiation hard technology such as FD-SoI [144], it would be less susceptible to transients and more robust to permanent faults. Due to the drastically increased development cost and required manpower, the resulting OBC would not be viable for most miniaturized satellite applications (not anymore "on a budget").

9.7 Conclusions

The 3-stage FT approach combined with its MPSoC host system presented in this chapter is the first practical, non-proprietary, affordable architecture suitable for FT general-purpose computing aboard nanosatellites. It utilizes FT measures across the embedded stack, and combines topological with software functionality, utilizing only extensively validated standard parts. Thereby, we enable the use of nanosatellites in critical space missions, while the architecture allows trading processing capacity for reduced energy consumption or fault coverage.

An OBC relying upon this architecture can be facilitated with the minimal manpower and financial resources. The MPSoC can be implemented using only COTS hardware and extensively validated, and widely available library IP, requiring no proprietary logic or costly, custom space-grade processor cores. It offers a high level of resource isolation for each processor, utilizing architectural features originally conceived for ManyCore systems to achieve FT.

Each compartment functions as a stand-alone processing compartment with dedicated I/O, existing in its own clock domain and reconfiguration partition, thereby minimizing shared resources and reducing routing complexity. Compartments were purposefully designed to best support thread-level coarse-grain lockstep of weakly coupled cores, while allowing partial reconfiguration without stalling the rest of the system. The architecture was implemented successfully, and tested on current generation Xilinx Zynq/Kintex and Virtex FPGAs with 4, 6 and 8 compartments, and validated through fault-injection into RTEMS.