Fault-tolerant satellite computing with modern semiconductors

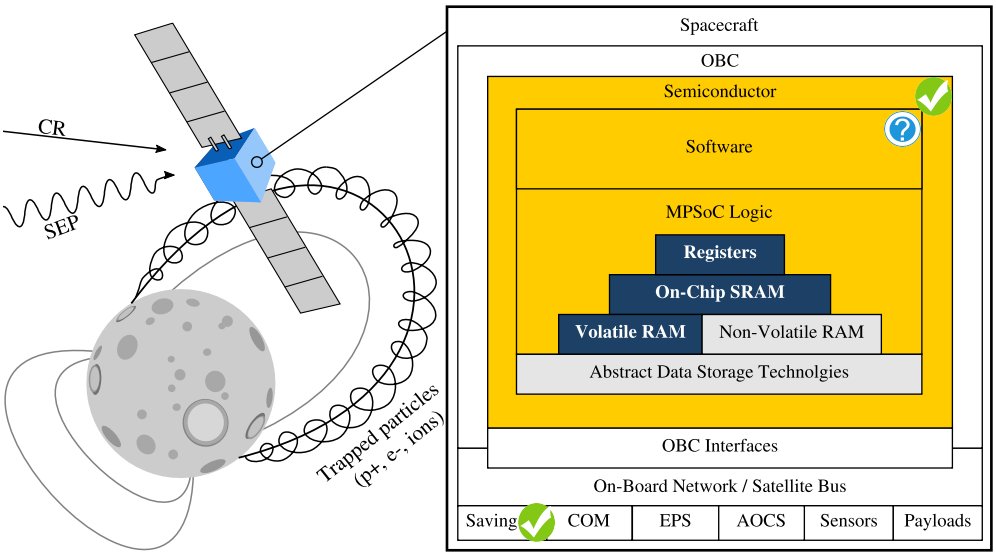
Fuchs, C.M.

Chapter 6

Mixed Criticality and Resource Pooling

Stage 3

In this chapter, we discuss the third and final stage of our fault tolerance architecture. Stage 3 enables satellites of all weight classes to more efficiently handle accumulating permanent faults, and to age gracefully instead entering a degenerate state, thereby answering RQ3. We show how this functionality in conjunction with mixed criticality properties of a satellite's on-board computer can be exploited to improve robustness and efficiency. By modifying the application mapping within the MPSoC and adjusting thread-replication at runtime, the system can dynamically trade compute performance for functionality, robustness, and energy consumption at runtime. Considering our architecture as a whole, the the mechanisms discussed in this chapter exist in software and utilize extensively architectural properties of our MPSoC.



6.1 Introduction

Satellite miniaturization has enabled a broad variety of scientific and commercial space missions, which previously were technically infeasible, impractical or simply uneconomical. However, very small satellites such as nanosatellites and sometimes even microsatellites ($\leq$100kg) are currently not considered suitable for critical and complex multi-phased missions, as well as high-priority science applications, due to their low reliability. On-board computer (OBC) and related electronics constitute a large part of such a spacecraft's mass, yet these components lack often even basic fault tolerance (FT) functionality. Due to budget, energy, mass and volume restrictions, existing FT solutions originally developed for larger spacecraft can in general not be adopted. Nanosatellite OBCs also have to cope with drastically varying workload throughout a mission, which traditional FT solutions can not handle efficiently. Therefore, we developed a novel FT approach offering strong fault coverage, which was implemented fully using only a single FPGA with commodity processor designs, and library IP.

This architecture can protect generic applications with an arbitrary structure, can adapt to varying performance requirements in longer multi-phased missions, and can adapt to a shrinking pool of processing capacity similar to a biological system, efficiently handling aging effects and accumulating permanent faults. As major parts of our approach are implemented in or directly controlled by software, a spacecraft operator can configure the OBC to deliver the desired combination of performance, robustness, functionality, or to meet a specific power budget. To offer strong fault detection, isolation and recovery (FDIR), we combine software-side fault detection and mitigation and configuration scrubbing with various other FT measures across the embedded stack, enabling strong, low-cost FT with commodity hardware, while exploiting FPGA reconfiguration to mitigate permanent faults.

The next two sections contain background information, and a discussion of related work. In Section 6.4 a brief overview over the three stages of our approach is provided. Our proof-of-concept OBC-design is described in Section 6.5, with the functionality of each FT-stage outlined in the subsequent sections. How this approach can improve efficiency of OBC in spacecraft of all weight classes, spare resource utilization and fault coverage, is discussed in Section 6.6. Section 6.7, introduces *performance profiles* allowing a system-on-chips (SoC) to trade compute performance for energy efficiency, robustness, and functionality at runtime. Our approach provides advantages to spacecraft of all weight classes, and can be implemented also within distributed systems, for which further applications and improvements are discussed in Section 6.8.

6.2 Background

Tasks which would be handled by multiple dedicated payload and subsystem processing systems aboard a larger satellite, are usually handled by just one COTS-based command & data handling system in nanosatellites. These utilize mobile-market and embedded SoCs with one or more cores (MPSoCs), SDSoCs [40], or FPGAs [237]. Due to manufacturing in fine technology nodes, such chips offer superior efficiency and performance as compared to space-grade OBC designs, but are also non-FT[1]. These SoCs consist mostly of extensively tested and optimized standard logic, reused, supported,

[1]Exceptions to this rule received uncommonly abundant funding, are technology demonstration for FT concepts, or custom failover designs.

and evolved continuously by several industries and used daily by countless developers. In contrast, most radiation-hard-by-design (RHBD) processors cores, and SoCs manufactured in more robust manufacturing processed (RHBM) are crafted almost artisanally at high cost by few designers with little commercial stimulus for optimization. Their cost, energy consumption and mass often exceed such a spacecraft's global power budget, total mass, and almost always its overall project budget. Therefore, we developed a hybrid FT-approach based upon only COTS components, library IP, and existing software, instead of artisanal processor designs and proprietary instruction set architectures.

Existing hardware voting based FT solutions are design-time static and can tolerate a fixed number of failures within a voter setup, which can not be changed at runtime. Critical biological systems instead consist of independent, cooperating cells or clusters of similar functionality with a high degree of inherent redundancy and self-healing capabilities. Damage to a single cell is compensated by the remaining cells, and a complete breakdown of functionality occurs only due severe damage to the system at a broader scale. Our approach combines various FT techniques to mimic such behavior at the logic and SoC level, through FPGA reconfiguration and software-controlled thread migration within a globally share pool of processor cores, enabling graceful aging. The replication level, hence fault coverage capabilities, and various other parameters can be adjusted at runtime, while spare capacity can be reused to run background and lower-criticality applications instead of remaining idle.

In small feature-size chips, the energy threshold above which highly charged particles can induce faults in digital logic (single event effects - SEE) decreases, while the ratio of events inducing multi-bit upsets (MBU), and the likelihood of permanent faults in logic and memory increases. Increased fault coverage of hardware-FT based concepts on such chips through additional FT-circuitry therefore implies diminishing returns, preventing an application of traditional RHBD/RHBM concepts [104, 132] to mobile-market SoCs. Total ionizing dose, however, becomes less of a problem with finer technology nodes, and recent generation FPGAs also show decent latch-up performance [142,143]. FPGAs have drastically improved FDIR potential [238] despite being more vulnerable to transients, as radiation-induced upsets in the running configuration can be corrected via reconfiguration with alternative configuration variants [105].

6.3 Related Work

Fine-grained, non-invasive, and scalable fault detection in FPGA fabric is challenging, and subject of ongoing research [239, 240], and often is simply ignored in scientific publications [241]. Most FPGA-based FT-concepts rely on error scrubbing, which has scalability limitations for complex logic [239, 242], unless special-purpose offline testing is utilized [243]. In the future, memory-based reconfigurable logic devices (MRLDs) [244] may allow programmed logic to be protected like conventional memory, and thus would drastically simplify fault detection. If manufactured using phase/polarity-change memory instead of charge-based technologies, MRLDs could further increase robustness, but the memory technologies themselves are only emerging at the time of writing. In this chapter, we thus present an approach to general-purpose FT computing that compensates for faults across the embedded stack and through partial FPGA reconfiguration. We realize fine-grained fault detection at the software level, and perform scrubbing only as an auxiliary measure in the background

to increase robustness of our SRAM-based FPGA platform.

Hardware voting today is used exclusively for protecting simpler FT processor cores at the microcontroller level [88,104], and for accelerators [245] supporting application code with tightly constrained program structure. Hence, the application of this hardware-centered approach has become a technical dead-end for protecting widely used application processor designs intended for general-purpose computing, while accelerators by themselves would only assure FT for computation and data offloaded to such a device. In our research, however, we seek to deliver strong fault coverage for general purpose computing, and aim to efficiently protect even larger and more complex modern application processors, such as those widely used in mobile market and embedded devices.

Mobile market processors can run at gigahertz clock rates, for which hardware-side voting or instruction-level lockstep are non-trivial, hence, hardware voting approaches have been implemented only at lower clock rates [88,191,192]. For comparison, today's highly optimized COTS library IP achieves clock speeds comparable to traditional FT-processor designs on ASIC even on an FPGA, without requiring manual fine-tuning. We instead utilize software-driven coarse-grain lockstep to achieve fault detection, and maintain consistency between cores, requiring no vast arrays of synchronized voters, while utilizing COTS IP.

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]). However, to implement sophisticated and efficient thread migration, fault-detection must be facilitated at the OS or application-level without falling back to design space exploration. Coarse-grain lockstep of weakly coupled cores can do just that, 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], and entail 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.

6.4 System Overview & Requirements

Coarse-grain lockstep is one among several measures used in our hybrid FT approach to facilitate forward-error-correction (FEC) and deliver strong fault coverage. Our approach consists of three fault mitigation stages:

Stage 1 utilizes coarse-grain lockstep for fault detection. It generate a distributed majority decision between processor cores.

Stage 1 utilizes time-triggered checkpoints to autonomously resolved faults corrupting the state of applications. It facilitates re-synchronization and thread migration in case of repeated faults, enabling strong **short-term fault coverage**.

Stage 2 assures the integrity of programmed logic by interfacing with Stage 1 and functionality such as Xilinx SEM. Its objective is to assure and recover

the integrity of processor cores and their immediate peripheral IP through FPGA reconfiguration, thereby **counteracting resource exhaustion**.

Stage 3 handles resource exhaustion and re-allocates processing time within the system to **maintain stability of critical applications and functionality in a degraded system**.

These Stages form a closed loop and implements FDIR in several steps as depicted in Figure 38. Additional information on Stage 1's thread-level coarse-grain lockstep, beyond what is briefly described in Section 6.5.1 are available in Chapters 4.

Stages 1 and 3 can be implemented separately on a generic MPSoC in low-end nanosatellites (e.g., 1U CubeSats). Then, they would provide a level of system-level robustness which otherwise would be only be achievable through proprietary hardware-FT solutions, without requiring the use of an FPGA.

For larger spacecraft, we complement this functionality with a compartmentalized MPSoC architecture for FPGA as outlined in the next section. It allows the system to recover defective compartments through reconfiguration, and enables it better handle permanent faults.

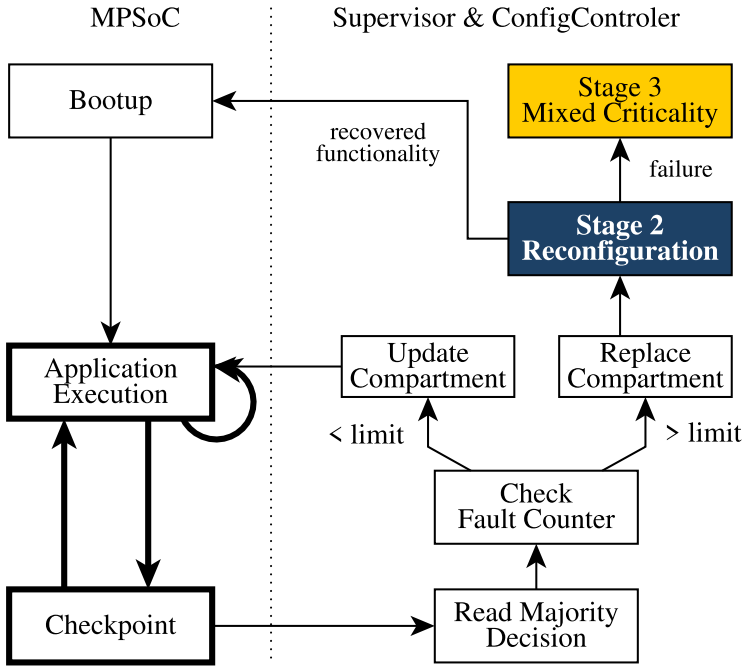


Figure 38: Stage 1 (white) implements a continuous checking loop, which facilitates fault coverage through thread-level synchronization and migration between compartments. Stage 2 (blue) can recover faulty compartments using reconfiguration. In case of resource exhaustion, Stage 3 (yellow) adapts the thread allocation to best utilize the remaining processing capacity.

6.5 System Architecture Review

Figure 39 depicts a simplified version of our MPSoC design. It follows a multi-core-like architecture with each compartment containing a processor core, local interconnect, and peripheral IP-cores and interfaces. A debug bridge allows supervisor access to each compartment, e.g., to perform introspection for testing purposes or to trigger a reset. The only globally shared resources are a set of redundant main memory controllers and non-volatile (nv) data storage. Code in nv-memory can be shared between compartments, while widely used DDR and SDRAM controllers are too large to instantiate for each compartment, and would require an excessive number of I/O-pins. Hence, our MPSoC architecture consists of isolated SoC-compartments accessing shared main memory and operating system code, in contrast to the conventional MPSoC designs, where cores share most infrastructure and peripherals.

Each compartment's checkpoint-related information is stored in a dedicated on-chip dual-port BRAM memory (*validation memory*) and exposed to other compartments, to allow low-latency information exchange between compartments without requiring inter-compartment cache-coherence or access to main memory. Validation memory is

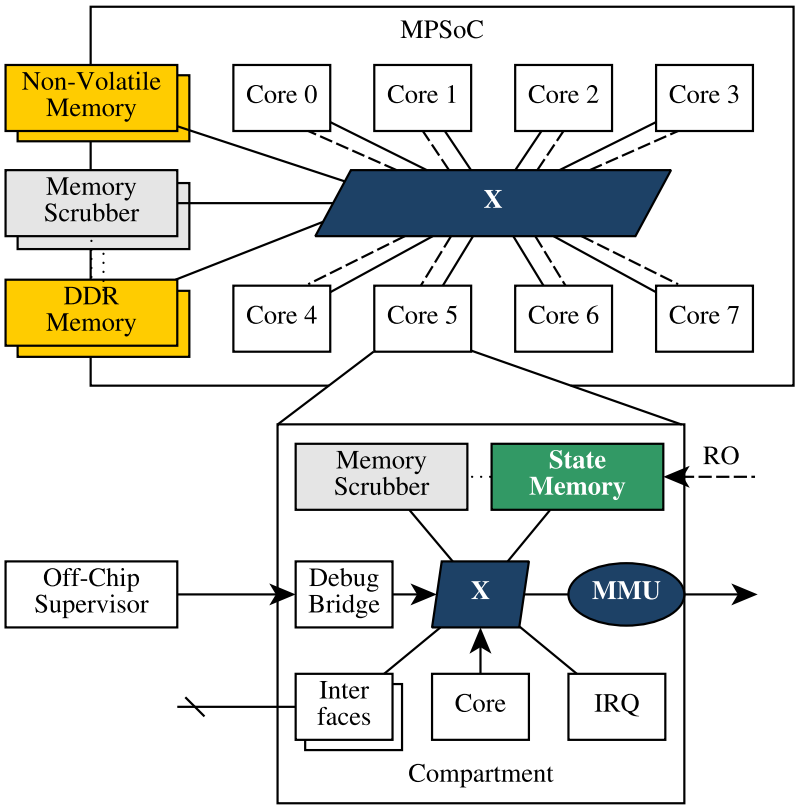


Figure 39: A high-level topology diagram of our compartmentalized MPSoC architecture with memory controllers highlighted in yellow, and interconnect-logic in blue. A debug-bridge on each compartment allows supervisor access. Access to each compartment's validation memory is possible read-only through the global interconnect.

writable through the compartment-local interconnect, and is read-only accessible by other compartments.

The address space layout on each compartment, including mapping of peripherals and interfaces within the address space are identical. Each compartment can access its own main memory address segment, which is mapped to the same address range on all compartments. Additionally, main memory in its entirety (all memory segments) is read-only accessible system wide, to simplify state synchronization between compartments.

During a checkpoint, the state of all threads mapped to a compartment is compared and synchronized with its siblings. To do so, the checkpoint handler executes an application-provided callback function for all pending threads, producing checksums generated from thread-private data structures. Checksums are stored in the compartment's local validation memory and thereby exposed to the other compartments, and then compared with the other compartments in the system. In case of disagreement, the compartment signals disagreement with that sibling and executes synchronization callbacks for all affected threads. If necessary, it then also executes relevant update callbacks and then resumes application execution. An more detailed description of these mechanisms as well as benchmark results for an astronomical application are described in Chapter 4.

6.5.1 Stage 1: Short-Term Fault Mitigation

The objective of Stage 1 is to detect and correct faults within a compartment, and assure a consistent system state through checkpoint-based FEC. It is implemented as sets of compartments running two or more copies of application threads (siblings) in lock step. Checkpoints interrupt execution, facilitating the lockstep and enforcing synchronization, allowing thread assignment within the system to be adjusted if required, as depicted in Figure 38.

This approach enables us to utilize application intrinsics to assess the health state of the system without requiring in-depth knowledge about the application code. The supervisor just reads out the results of the compartments' decentralized consistency decision. Threads can be scheduled and executed in an arbitrary order between two checkpoints, as long as their state is equivalent upon the next checkpoint.

We avoid thread synchronization issues due to invasive lockstep mechanisms [198] by merely reusing existing OS functionality without breaking existing ABI contracts. Therefore, we can continue relying upon pre-existing synchronization mechanics such as POSIX cancellation points[2] and their bare-metal equivalents (e.g., in RTEMS *RTEMS_NO_PREEMPT* or the POSIX API). Stage 1 can even deliver real-time guarantees, and the tightness of the RT guarantees depends upon the time required to execute application callbacks. In our RTEMS/POSIX-based implementation, we utilize priority-based, preemptive scheduling with timeslicing, allowing threads to delay checkpoints until they reach a viable state for checksum comparison.

Checkpoints are time triggered, but can also be induced by the supervisor through an interrupt, e.g., to signal that new threads have been assigned. Thus, the OS only has to support interrupts, timers, and a multi-threading capable scheduler. To the best of our knowledge, such functionality is available in all widely used RT- and general purpose OS implementations.

[2]E.g., sleep, yield, pause, for further details, see IEEE Std 1003.1-2017 p517

A fault resolved during a checkpoint may cause the affected compartment to emit incorrect data through I/O interfaces, an inherent limitation to coarse-grain lockstep [199]. For many very small nanosatellite missions this is acceptable, as the use of COTS components requires incorrect I/O to be sanitized anyway. In contrast, larger spacecraft already utilize interface replications or even voting, usually requiring considerable effort at the interface level to facilitate this replication. Our approach combined with the previously described MPSoC architecture inherently provides interface-level replications by design, no longer requiring extra measures to be taken. Additional protection is therefore only needed for space applications where non-propagation of incorrect I/O is required but interface replication is undesirable, i.e., due to PCB-space constraints aboard CubeSats or unchangeable subsystem requirements. For packet-based interfaces such as Spacewire, AFDX, CAN, or Ethernet, no hardware-side solution is necessary, as data duplication can be managed more efficiently at OSI layer 2+. This approach today is widely used as part of real-time capable FT-networking [94]. Other interfaces like I2C and SPI allow a simple majority decision per I/O line, which can be implemented on-chip through FIFO buffers, as the remaining on-compartment interfaces have low pin count and run at relatively low clock frequencies.

6.5.2 Stage 2: Tile Repair & Recovery

Stage 1 can not reclaim defective compartments, eventually resulting in resource exhaustion. Therefore, in Stage 2, we recover defective compartments through reconfiguration to counter transients in FPGA fabric. To do so, the supervisor will first attempt to recover a compartment using partial reconfiguration. Afterwards, the supervisor validates the relevant partitions to detect permanent damage to the FPGA (well described in, e.g., [218]), and executes self-test functionality on the compartment to detect faults in the compartment's main memory segment and peripherals. If unsuccessful, the supervisor can repeat this procedure with differently routed configuration variants, potentially avoiding or repurposing permanently defective logic.

As compartments are placed along partition borders in our MPSoC architecture, compartments can be recovered in the background without interrupting the rest of the system. The supervisor can also attempt full reconfiguration implying a full reboot of all compartments. Further details on reconfiguration and error scrubbing with a microcontroller-based proof-of-concept implementation for a nanosatellite are available in Chapter 5. If both partial- and full-reconfiguration are unsuccessful and all spare resources have been exhausted, Stage 3 is utilized to assure a stable system core to enable operator intervention.

6.5.3 Stage 3: Applied Mixed Criticality

Stage 3 autonomously maintains system stability of an aged or degraded OBC. When considering a miniaturized satellite's OBC, we can differentiate individual applications or parts of flight software by criticality. At the very least, we will find software essential to a satellite's operation, e.g., platform control and commandeering, as well as other applications of various levels of lower criticality. If the previous stages no longer have enough spare processing capacity or compartments to compensate the loss of a compartment, this stage utilizes thread-level mixed criticality to assure stability of core OBC functions. To do so, it can sacrifice lower criticality tasks in favor of providing compute resources to reach the desired replication level for critical threads.

Dependability for higher-criticality threads can efficiently be maintained by reducing compute performance or reliability of lower-criticality applications. Lower-criticality tasks may be executed less frequently or on fewer compartments, thereby reducing functionality or fault coverage for these tasks, retaining resources for higher-criticality threads. This decision is taken autonomously, and the operator can then define a more resource conserving satellite operation schedule at a spacecraft level, e.g., sacrifice link capacity, or on-board storage space, to make best use of the OBC in its degraded state.

6.6 Spare Resource Pooling

This FT approach enables FT even for very small satellites, but provides benefits for spacecraft of all weight classes. To increase fault coverage in traditional hardware voting FT systems, additional cores and spares must be provisioned, while compute performance can be increased by utilizing higher-performance processor cores and adding more hardware voting instances. This is done at design time, requiring over-provisioning, and can not be changed throughout a mission. Cores are hardwired to a specific instance, therefore, an instance will degrade once its spares are exhausted, even if idle spares were available elsewhere.

In contrast, our approach is not based on hardwired voting instances, as applications are mapped to a global pool of compartments with a given replication level. Our approach does utilize spare resources too, but spare compartments and conventional compartments are identical. Hence, spare compartments do not have to remain idle, and unused processor capacity becomes a spare resource that can be re-purposed. Thus, the fault coverage capabilities of the system are no longer dependent on the distribution and location of permanent faults within the system, increasing overall robustness.

As applications can be migrated between compartments, low criticality threads and background tasks can be assigned to utilize free spare capacity. These lower-criticality threads can be de-scheduled in favor of higher-criticality applications, if needed. Spare capacity can also be used to increase FT for threads, which usually would be executed without majority voting or separately due to resource constraints. We can distribute a defective compartment's workload to other compartments, to best take advantage of the remaining system resources.

The best target compartments and to-be-evicted threads are not determined ad-hoc, but before a fault actually occurs, to reduce the time spent in a checkpoint. We can maintain one replacement strategy for every compartment, due to the low compartment and thread counts common in space applications today[3]. Subsequent to a fault, these strategies are recomputed to consider the now reduced processing capacity of the system. As thread assignments are not controlled by the supervisor, but only adjusted, threads may exit, fork or create new child threads. Therefore, an update to adjust these strategies to the currently running threads is also triggered based on the fault counter of Stage 2. Even if a fault occurs immediately after the current

[3]The main application for our architecture is platform control. ManyCore-systems with hundreds of cores would allow too many combinations, but they will not be applied to satellite platform control in the foreseeable future. For dedicated payload data processing, this may be different, but our interest in this thesis is mainly platform control and unified satellite data handling aboard miniaturized satellites.

checkpoint, these strategies will only be needed at the next checkpoint. Therefore, this is a background operation which can be handled by the supervisor, allowing the OBC to resume processing immediately.

Figure 40 depicts a six compartment MPSoC running four applications of different criticality. A fault has occurred in compartment 3, which has been marked as permanently defective, and there are multiple recovery solutions:

- Affected threads could be relocated to a compartment running lower-criticality applications, replacing them as depicted in Figure 40a. For example, the threads previously run on compartment 3 can be migrated to compartment 6, replacing lower criticality thread-copies previously run there. This requires compartment 6 to copy the state of its newly assigned threads from compartment 1 or 2, at the cost of executing the lower-criticality applications redundantly instead of with majority voting.

- Instead of entirely de-scheduling one instance of each lower criticality threads, the clock frequency on two compartments could be increased, allowing one of each high-criticality thread to be migrated. In Figure 40b, this is depicted by moving the threads from the failed compartment to compartments 5 and 6 without de-scheduling instances of the low criticality threads. This is possible as coarse-grain lockstep only requires an equivalent state between siblings upon reaching a checkpoint and no cycle-accurate synchronization. Most modern embedded and mobile-market cores support frequency scaling.

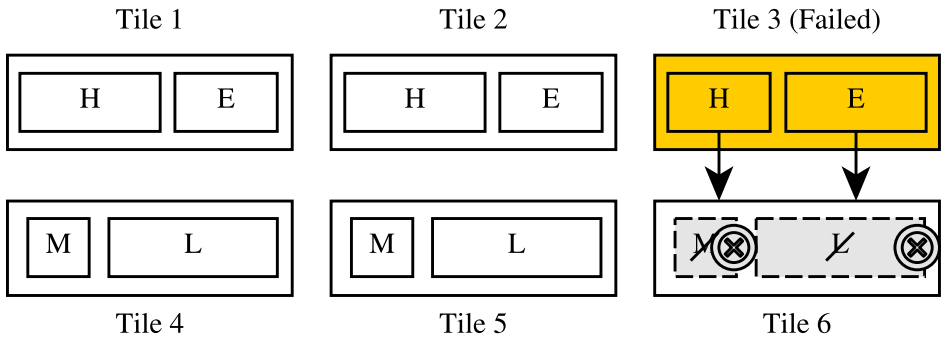
- Another possibility would be to instead increase the clock frequency of just one compartment, if sufficient additional processing capacity can be made available that way.

- Finally, in contrast to increasing the clock frequencies of individual compartments, compartment 4-6's schedulers could also assign less processing time to the lower-criticality tasks as shown in Figure 40c. Due to timing implications for real-time applications, this may only be possible for sporadic tasks, and background applications, which do not require a fixed amount of processing time. Also, to guarantee equivalent work is conducted for the medium and lower-criticality threads, the schedulers on 3 instead of just 2 compartments would require adjustment, wasting processing capacity in Tile 4 and 6. However, during this idle time, Tile 4 could be deactivated to reduce energy consumption.

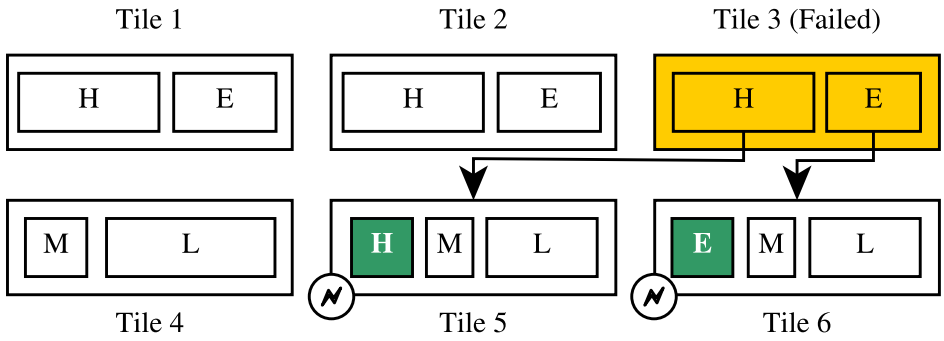
The ideal recovery strategy depends on the current performance requirements towards the OBC. Additional thoughts on this aspect are discussed, e.g., in [241], where different replacement strategies are described at a more mathematical level for video streaming applications. In the next section, we therefore discuss a heuristic approach to find near-best solutions to calculate this decision autonomously and rapidly, considering different performance requirements.

6.7 Adapting to Varying Mission Requirements

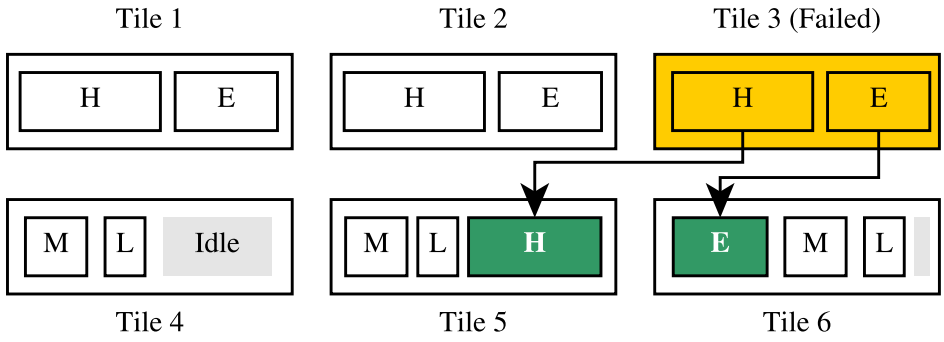
The approach described in the previous sections allows an OBC to meet a desired power budget, maximize fault coverage, processing power, or even functionality. Hence, the spacecraft can better fulfill its scientific or commercial mission, and increase the



(a) Migration by low-criticality thread pruning.



(b) Migration through clock-speed increase.



(c) Migration through processing time reduction.

Figure 40: A hexa-core MPSoC running 4 threads of mixed criticality (**E**ssential, **H**igh, **M**edium, and **L**ow), where compartment 3 (yellow) suffered a hard fault. To retain majority voting for the higher criticality threads, different recovery strategies can be facilitated through, without directly requiring spares.

spacecraft's lifetime. Theoretically, all we need to do is find the ideal set of thread mappings which fulfill our desired trade-off between processing capacity, FT, and minimal energy consumption. These three performance objectives can be visualized as depicted in Figure 41, and viable mappings can be found in the inner area outlined in red.

These three objectives oppose each other, and fully dynamic performance optimization at runtime is non-trivial and costly. Prior publications in computer science (e.g., [241, 246])approaches such issues with computationally expensive optimization algorithms to find the ideal solution, or design space exploration to find a large set of near-best and chose the optimal solution either at runtime [241] or design time [246]. The latter defeats the purpose of run-time flexibility and adjustment. While design space exploration at runtime is infeasible due to the limited processing capacity of a supervisor, unless tight constraints are placed upon applications regarding structure and functionality [241]. In practice, however, we do not have to find the singular "best possible" solution when recovering from a fault, instead we just need a "good enough" solutions yielded by a heuristic algorithm [247]. Once the system has been stabilized, ample time will be available to further optimize the thread mapping and usually this is done by the operator or flight software. The code of this algorithm is depicted in Algorithm Listing 1.

To facilitate a heuristic approach, we first reduce these three competing objectives to a set of *performance profiles*, examples of which are given in Table 42. In each

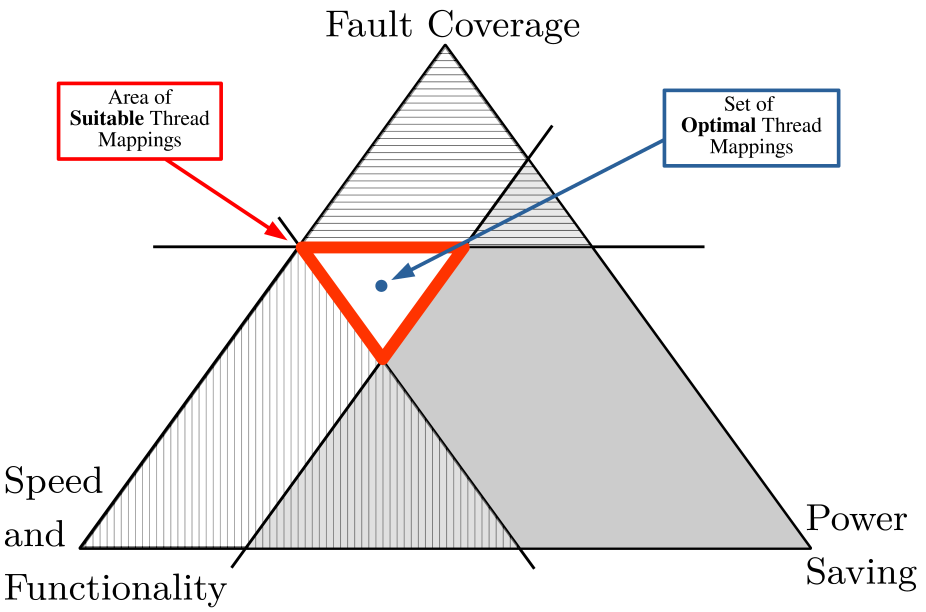


Figure 41: An MPSoC utilizing the presented approach can trade speed, energy efficiency, and fault coverage at run-time. We utilize *performance profiles* for each objective to facilitate a heuristic solution, which is located somewhere within the red highlighted area. This is an approximation of one or multiple "ideal/optimal" thread-mappings, which can be computed only with more processing time, through design-space exploration solution space (brute force).

profile, criticality classes (essential - low) are assigned one or multiple execution modes: separate execution with de-scheduling allowed, separate, redundant, majority voting, or with more cores, e.g., to enable Byzantine voting (referred to as NMR, TMR, DMR, separate, and de-schedule in Table 42). Duplicate assignments allow threads to be mapped in either mode, to enable mode reduction in case of resource constraints. For example, when running in the robustness profile, essential applications are always assigned the desired number of cores, while high-criticality applications are at least TMRed (depending on available resources). Other applications are preferably executed TMRed, but may be executed also DMR to retain fault detection, in case of resource exhaustion, instead of entirely de-scheduling lower criticality threads. Depending on mission requirements, the operator can then select the most suitable performance profile from a set of pre-generated at runtime, or could draft a new one.

To map threads, we build a new mapping for a task using the strongest desired execution mode. We evaluate if this exceeds the available power budget (energy profile) or processing capacity. If so, we begin reducing the execution mode of tasks beginning with the last mapped and therefore lowest-criticality thread. If successful, we append the mapped thread to a list and proceed with the next thread. To minimize the amount of de-scheduled and mode reduced threads, we can sort threads of same criticality based on required processing capacity. Thereby, computationally expensive threads are reduced in execution mode first, freeing up larger amounts of processing resources.

If not all threads could be mapped, we can de-schedule lower-threads exceeding the compute capacity, energy constraints, or allocate less processing time to specific applications system. Once no further mode or processing time reductions are possible due to real-time guarantees, we cease mapping new threads to uphold fault tolerance guarantees for this reduced core system. As final step, we traverse the list from the start and increasing execution mode to undoing mode reductions for as many threads as possible. The supervisor itself only has to execute the latter part of this algorithm and perform mode and processor time reduction, or de-schedule the lowest criticality threads. It does not have to actually generate all these mappings as it does not enforce

Mode	Performance	Power Saving	Robustness	Functionality
NMR	E - - - ↑	E - - - ↓	E H M L ↑	E - - - ↑
TMR	E H M L	E H M L	E H M L	E H M L
DMR	- H M L	- - M L	- H M L	E H M L
Separate	- - - L	- - - L	- - M L	E H M L
Deschedule	- - - -	- - - -	- - - -	- H M L

Figure 42: *Performance profiles* with threads of different criticality levels (**E**ssential, **H**igh, **M**edium, **L**ow) being assigned different replication levels to enable fault detection or different voting configuration through thread replication. Arrows indicate the strategy used for choosing mappings. E.g., In the Power Saving profile, all threads are first mapped in their highest desired replication level, and then reduced beginning with the lowest priority threads until the system's thread mapping allows a given energy consumption threshold to be surpassed. In the Performance or Robustness profiles, we instead attempt to achieve the highest level of thread-replication that is possible with the given available processor compartments. In the Functionality profile, we wish to retain a stable setup for essential application, even if this requires lower criticality threads to be de-scheduled.

ALGORITHM 1: Pseudo-Code of the Thread-Allocation Heuristics

Input: T_i: List of Threads, P: performance profile, C: Set healthy Cores
Output: M: List of mapped thread-groups

```
1  for T_i from T_0 to T_n do
       // Attempt to create a mapping for the thread
2      replication_level = getDesiredReplication(P, T_i)
3      thread_group = makeGroup(T_i, replication_level, C)
4      thread_mapping = getTargetCores(thread_group, C)
5      if isValid(thread_mapping) then
6          AppendGroup(M, thread_group, targets)
7      else
           // Failure, try to map with lower replication
8          lowest_replication = getLowestAllowedReplication(P, T_i)
9          while replication_level is not lowest_replication do
               // reduce replication level and retry
10             replication_level = getLowerReplication(P, T_i)
11             thread_group = makeGroup(T_i, replication_level, C)
12             thread_mapping = getTargetCores(thread_group, C)
13             if isValid(thread_mapping) then
14                 AppendGroup(M, thread_group, thread_mapping)
15                 goto line 1 // break out of nested loop and continue

           /* Insufficient compute capacity available in the system. E.g., too many
              compartments failed. Attept to reduce the replication level of an early
              mapped higher priority application to free compute capacity.          */
16         for M_i from M_i to M_0 do
17             t = getThread(M_i)
18             others_replication = getCurrentReplication(P, t)
19             lowest_replication = getLowestAllowedReplication(P, t)
20             while others_replication is not lowest_replication do
                   // Reduce replication for next higher priority group and retry
21                 tryReduceReplication(P, M, M_i, others_replication, C)
22                 thread_mapping = getTargetCores(thread_group, C)
23                 if isValid(thread_mapping) then
24                     AppendGroup(M, thread_group, targets)
25                     break
               // Can not reduce mapping, try to reduce earlier mapped thread
           // Too-few compute resources, de-schedule and try to map next thread
```

thread assignment in the system and only intervenes if necessary.

This algorithm also provides all mechanisms necessary to minimize the amount of active processor cores, and as threads can be concentrated to as few compartments as possible, maximizing the number of clock-gated cores. Individual tasks could also signal preference for reduced processing instead of a mode reduction as the approach itself is computationally inexpensive.

6.8 Discussions

We implemented the MPSoC architecture described in Section 6.5 using Xilinx Kintex and Virtex FPGAs as well as the Zynq SDSoC platform [40], as these are relevant for our target missions. However, for larger satellite platforms, this approach and architecture could very well be implemented on ASIC, and we see this as a “big-space" variant of our approach. An ASIC implementation would have lower energy consumption, and allow higher clock rates due to tighter timing and shorter paths, and be less susceptible to transient faults. If manufactured in an inherently radiation hardened technology such as FD-SoI [144], the system as a whole would be considerably more resistant to transient faults. Stage 2 would then be reduced to testing and validate compartments, while no longer being able to recover faulty compartments containing defective logic, but strong fault coverage of SEEs would be improved due to RHBM.

Overall, an FPGA implementation offers stronger FDIR capabilities, better coverage for permanent faults, and high flexibility at low cost, while the ASIC variant could offer better system performance and radiation tolerance due to RHBM. Custom ASIC development of course is expensive and time-consuming, thus, the resulting implementation would not be a viable solution for most miniaturized satellite applications, and therefore not in the scope of this technology development project.

The relaxed cost, energy, and size constraints aboard larger spacecraft allow an implementation of our approach spanning multiple FPGAs. Compared to a single-chip implementation, a multi-FPGA MPSoC variant offers better scalability due to easier routing, can tolerate chip-level defects, and SEFIs to the globally shared memory controllers, these can be distributed to different FPGAs. Replicated thread-instances could then also be distributed across FPGAs, offering non-stop operation while one of the FPGAs undergoes full reconfiguration. However, our proof-of-concept is focused on a single-FPGA based prototype for nanosatellite use.

Our project is focused on payload data handling and platform control for miniaturized spacecraft, and therefore accelerator cores supporting computational offloading are outside the scope of our research. Nonetheless, it is possible to also protect accelerator systems using this approach, yielding at least similar benefits. The structure and type of applications usually executed on accelerators is tightly constrained as compared to general purpose platform control, simplifying lockstep replication and thread-mapping. Especially synchronization for real-time applications and the impact of live-migration between compartments or state-updates on a faulty compartment, become much simpler if fully deterministic application behavior is assumed, as would be the case for computational offloading.

Our existing MPSoC design utilizes an AXI interconnect, but we plan to rework our MPSoC to instead use a NoC between compartments and shared memory controllers. The existing interconnect implementation allows low-latency communication, but has

a large footprint, and is difficult to route[4] for larger compartment counts (without optimization, we successfully placed 8 compartments). A NoC instead allows not only better scalability and easier routing, but also enables the implementation of a broad variety of FT concepts such as [93].

Tiles have direct read-only access to another compartment's memory segment to allow rapid thread migration and allow real-time capacity. However, direct access to shared main memory is not necessary to facilitate Stages 1-3. The data exchange required to facilitate thread migration could very well be implemented using IPC or through sockets, when considering complex networked architectures. In distributed systems, our approach could thus manage threads across multiple nodes sharing data when required, at the cost of higher latency.

We developed this approach to guarantee FT for opaque threaded applications on POSIX-compatible RTOS and general purpose operating systems such as RTEMS and Linux. However, the same functionality can also be applied to virtualized, voted systems and to runtime based platforms. It would be very well imaginable to implement Stage 1 within MicroPython or a hypervisor, and instead vote on Python scripts or virtual machines.

6.9 Conclusions

To the best of our knowledge, the on-board computer (OBC) design presented in this chapter is the first practical, non-proprietary, and affordable fault tolerance (FT) approach suitable even for very small spacecraft. It offers strong fault coverage, using just commercial-off-the-shelf hardware, library IP, and commodity processor cores, requiring only a single FPGA and a microcontroller based supervisor. The software-side FT approach outlined in Stage 1 is non-invasive to applications and the OS, therefore existing software can be reused and extended easily, while retaining real-time capabilities. The research presented in this chapter covers the entire FDIR loop, and does not ignore or make unrealistic assumptions regarding fault detection.

Our approach enables the re-use of existing development tools and IP designed for mass-produced mobile-market applications, taking an important step towards departing from the artisanal development approach in today's space computing. Instead of requiring new technologies to be re-invented constantly and maintained at high cost, the FT mechanisms presented in this chapter are flexible, which can adapt and grow with the development of computer and processor technology.

We do not just enable FT for a satellite class which so far has been considered unreliable, but also enhance the fault coverage capabilities of OBCs in larger spacecraft, and other applications with similar constraints and fault profile. Our approach facilitates majority voting through dynamic, replicated thread groups mapped to the available processor cores dynamically at runtime, instead of hardwiring them. Thus, all processing capacity, including spares, are part of a shared resource pool. Therefore, spare resources can be used more efficiently, and allowing idle compute capacity to be used productively until it is needed for fault coverage. An OBC running the presented hybrid hardware-software FT approach can adapt to varying mission requirements regarding adjusting the OBC transparently at run-time, trading processing capacity for reduced energy consumption or increased fault coverage.

[4]We can still achieve a functional implementation meeting timing constraints at several hundred megahertz, but the interconnect PBlock becomes disproportionately large.