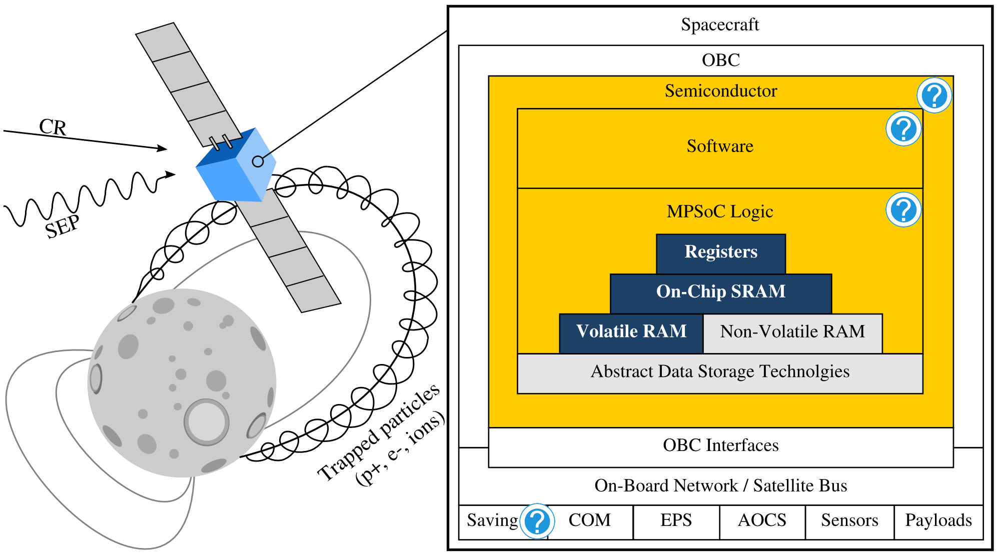
Chapter 4

A Fault Tolerance Architecture for Modern Semiconductors

Stage 1 & Architecture Overview

In this chapter, we describe a non-intrusive, integral, flexible, hardware-software-hybrid approach which enables the use of modern multiprocessor system-on-chips (MPSoCs) for spaceflight without violating application constraints. We introduce a co-designed system architecture utilizing three interlinked fault tolerance measures. To drive this architecture, we propose a coarse-grain thread-level lockstep implemented in software, and describe our implementation in detail in this chapter. We provide benchmark results for the lockstep, which allows very pessimistic worst-case performance overhead measurements. The technological feasibility of this architecture is demonstrated through implementation of a basic proof-of-feasibility MPSoC implementation.



4.1 Introduction

Modern embedded technology is a driving force in satellite miniaturization, contributing to a massive boom in satellite launches and a rapidly evolving new space industry. Micro- and nanosatellites (100-1kg) have become increasingly popular platforms for a variety of commercial and scientific applications, due to an excellent balance of performance and cost. However, this class of spacecraft suffers from low reliability, discouraging its use in long, complex, or high-priority missions. The OBC related electronics constitute a much larger share of a miniaturized satellite than they do in larger satellites. Thus, per component, they must deliver better performance and consume less energy. Therefore, due to cost considerations, miniaturized satellite OBCs are generally based upon processors manufactured in fine-feature-size technology nodes, such as those used in mobile embedded devices.

Traditional hardware-based fault tolerance (FT) concepts for general-purpose computing, however, are ineffective for modern, highly scaled systems-on-chip (SoCs), becoming a prime source of malfunctions aboard miniaturized satellites [2]. Larger satellites, too, are limited by the constraints of traditional ways to achieve fault tolerance for space applications, as these prevent larger satellites from harnessing the benefits of modern processor designs, and multiprocessor-SoCs (MPSoCs). Also, these hardware-based FT-measures can not handle varying performance requirements during multi-phased missions and mega-constellations [187]. Software-based FT measures rapidly evolved due to efforts of the scientific community, and are effective for modern embedded hardware. However, these advances have largely been ignored by the space industry, as well as closely related fields such as atmospheric aerospace, as they were researched only in theory, but rarely meant for implementation. While many of these concepts include innovative ideas, major implementation obstacles and fundamental issues remain unaddressed. Often, prior research makes impractical assumptions towards the platform or application environment, ignores fault detection, recovery from failover, or other real-world constraints. Many concepts also attempt to uphold safety and availability, e.g., for atmospheric aerospace use, but not computational correctness. To the best of our knowledge, no integral and practical solution to utilizing modern MPSoC-based systems within high-priority space missions has been developed to date.

There is a wide gap between academic research towards novel FT concepts and their practical application in spacecraft OBCs. Satellite computers for control purposes are still largely based upon architectures developed decades ago, while theoretical research has not achieved the level of maturity necessary to bridge this gap. Thus, neither traditional hardware- nor software-based FT solutions could offer all the functionality necessary to improve the reliability of state-of-the-art embedded SoCs in miniaturized satellite OBCs. Other concepts promise excellent FT guarantees in theory, but require complex architectures that often do not address the specific challenges of computers flying in space. Innovations are especially needed in general-purpose computing, as OBCs must execute a broad variety of applications efficiently.

This approach was developed for a 4-year European Space Agency (ESA) project with two industrial partners. Due to the interdisciplinary nature of this project, other aspects of this approach and its hardware implementation are described further in Chapters 5 – 10.

In the next section, we discuss related work, and how the design constraints and challenges outlined in Chapter 3 are up until the time of writing are addressed in fault-tolerant OBC design. Section 4.3 contains a brief overview of the multi-stage approach, its limitations, terminology, as well as the application model and requirements. Each stage is described in the subsequent sections, with the supervision concept explained in Section 4.4.4. Section 4.7 then introduces briefly an MPSoC architecture specifically designed as a platform for this FT concept. Performance and checkpoint reliability are discussed in Section 4.8, followed by conclusions.

4.2 Related Work

Radiation challenges OBC fault coverage constantly and throughout a mission and affects all of an OBC's components depicted in Figure 21. Traditionally, FT is enabled through circuit-, RTL-, core-, and OBC-level voting, which is costly to develop, difficult to validate, maintain, and slow to evolve [88,104,132,188–190]. Software takes no active part in fault-mitigation, as faults are suppressed at the circuit level, preventing the effective assessment of a processor's health. Circuit- and RTL-voting are effective for microcontrollers and very small SoCs, while core-level voting requires logic unavailable in COTS systems. Modern embedded COTS MPSoCs consume very little energy. But to achieve FT using hardware-side measures, arrays of synchronized high-frequency voters or core-lockstep in hardware are necessary. As voting and core-level lockstep at GigaHertz clock rates are non-trivial, it has been implemented only at considerably lower frequencies with non-COTS hardware [88, 190–192].

In general, hardware-voting based MPSoC designs are static and non-adaptive, as the entire design's fault coverage properties are highly chip-specific [193]. All these components are single-vendor solutions, often with walled-garden ecosystems with vendor lock-in. FT MPSoCs for space use contain retrofitted TMRed single-core processors, e.g., [104], or are unique, experimental solutions for specific satellite missions [194,195]. In contrast to these solutions, modern MPSoCs also allow considerably more software design freedom due to the available compute resources, thereby reducing the required development time and complexity. For scientific instrumentation and low-priority CubeSat missions, COTS-based MPSoCs and FPGA-SoC-hybrids have been utilized, but these are not suitable for critical satellite control applications within miniaturized satellites [196]. Ground-based FT applications do not consider the specific threat-scenario and application environment, physical constraints, and thermal design constraints [5, 197]. Instead, we propose to use software-side functionality to assure FT for conventional, non-fault-tolerant processor cores.

First concepts involving coarse-grain lockstep are promising [198–200], but do not address the specific challenges to FT in space [201]. FT using thread-level very-long-instruction word architectures [202, 203] has also been explored, though the approach still requires pipeline-level voters in hardware. Most implement checkpoint & rollback or restart, which makes them unsuitable for spacecraft command & control applications [204], others ignore fault-detection [205, 206], or require external, infallible fault detection entities with deep knowledge about application-intrinsics [207] but no concept of how this could be obtained. Often, faults are assumed to be isolated, side-effect free and local to an application [208] and/or transient [199, 200, 205], which voids their effectiveness for space applications. Many prior concepts entail high performance- [209], resource-overhead [210, 211], or impose severe design constraints on applications and the OS [198, 199]. To be effective in the space environment, an FT approach must be based upon forward-error-correction and the implementation complexity must be

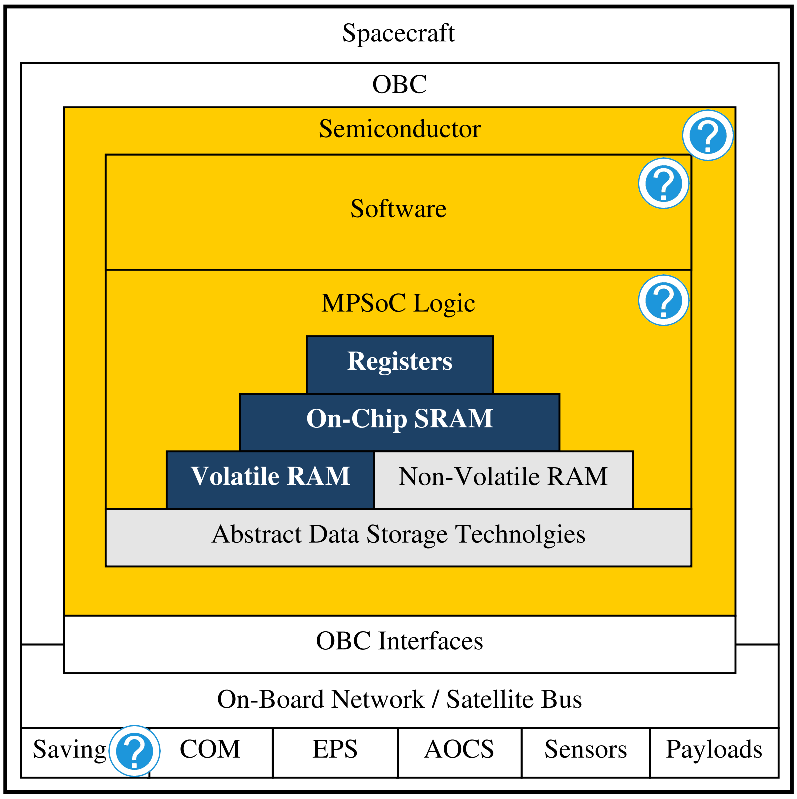


Figure 21: A component-level view of a satellite OBC. The multi-stage fault tolerance architecture proposed in this chapter covers faults affecting MPSoC, semiconductor infrastructure, logic as well as software (yellow). Volatile memory (blue) and non-volatile memory (gray) can well be protected using error correction coding and is described in Chapter 7.

low, and must be suitable for general-purpose computing and impose little or no constraints on the application software. Changes to the OS infrastructure must be platform portable, code-wise localized, and individually verifiable.

[199, 200, 208] implement voting through OS invasive measures, can not handle multi-threaded applications and consider the OS and stored program code to be fault-free. [201] requires no modifications to the application software whatsoever, but can only assure availability in a networked application architecture. An acceptance of these constraints does not allow for adequate FT in a space mission scenario, and thus we propose that application and OS instance must be able to fail arbitrarily without impacting the residual system. In this case, fault propagation between application instances also becomes a non-issue. Considerable research has been directed towards FT real-time scheduling and mixed critical software-FT systems, though only at a theoretical level [212–214]. As a consequence, no implementable, software-driven FT concept for modern embedded- and mobile-market MPSoCs in space exists, creating a gap between the described prior research on software- and hardware-FT based implementations.

4.3 Fault Tolerance through Software

This approach consists of three fault-mitigation stages:

Stage 1 is implemented entirely in software and provides fault-detection through coarse-grain lockstep to enable self-testing, and can be implemented in COTS MPSoCs.

Stage 2 improves medium-term reliability through FPGA reconfiguration, and enables long-term fault coverage using alternative configuration variants. It utilizes Stage 1's fault detection capabilities.

Stage 3 extends the lifetime of a degraded OBC by utilizing mixed criticality to assure fault coverage for high-criticality threads. It enables the OBC to automatically sacrifice performance or fault coverage of lower-criticality threads in favor of higher-critical applications, thereby maintaining a stable core system.

The presented concept is flexible and the individual stages are modular, as Stage 2 or 3 can be omitted depending on the OBC and mission. Our approach is designed for generic COTS MPSoCs, as these are readily available in a variety of performance classes at low cost. In the architecture described in Section 4.7, we place processor cores within isolated compartments. We consider it an ideal platform for our approach. In MPSoCs without a compartments, *compartment* can be substituted for *processor core*, and the differences in fault coverage are discussed in Section 4.7.

Terminology

Fault detection in our approach is based upon sets of compartments running two or more lockstepped copies of application threads. We refer to such a group of lockstepped threads as a *thread group*. Timing-compatible thread groups can be combined and executed on the same set of compartments, and are then referred to as a *compartment group*.

The relation between these is visualized in Figure 22. A thread group can realize a varying level of replication to achieve majority voting (thread 0 in the figure), error detection (thread 1), or even individual execution. One compartment may be host to

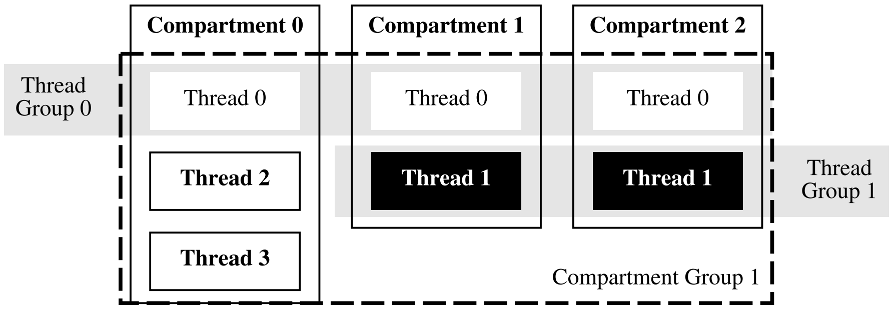


Figure 22: Schematic illustration of the relation between compartments running applications as threads, thread groups, replication, and timing-compatible compartment groups.

multiple thread groups threads may be unassigned from it, or newly assigned to it at runtime using conventional thread and process management functionality of the OS.

A compartment group periodically executes a *checkpoint routine*, which computes checksums for all active threads and compares them with the other compartments in the group (*siblings*), thereby enabling a majority decision or error detection. The time between checkpoints (the *checkpoint frequency*) is defined by the threads in a compartment group and can be modified at runtime. All lockstep-relevant information is stored in *state memory*, a compartment-dedicated memory segment which is read-only accessible by compartments.

Application Requirements

The OS only has to support interrupts, wake-up timers, and a multi-threading capable scheduler. To the best of our knowledge, such functionality is available in most widely-used RT- and general-purpose OS implementations. Virtual memory support is required to enable performance-efficient multi-threading. Furthermore virtual memory simplifies thread-management, context switching, and thread isolation, benefiting overall fault tolerance.

The only requirement for applications is interruptable at application-defined points in time, during which checkpoints can be executed. As there is no efficient, uniform approach to assess the health of threads, we rely upon applications assessing their own health-state. A thread can provide four callback routines to the OS, which are executed during compartment initialization and by the checkpoint handler:

- an *initialization routine*, to be executed on all compartments at bootup;
- a *checksum callback*, used to generate a checksum for comparison with siblings,
- a *expose state callback*, exposing all thread-state relevant data to synchronize a sibling with a compartment group; This data can either be placed directly in the compartment's state memory, or as a reference to structures in main memory.
- and an *update state callback*, which is executed on a compartment that needs to synchronize its state to a compartment group.

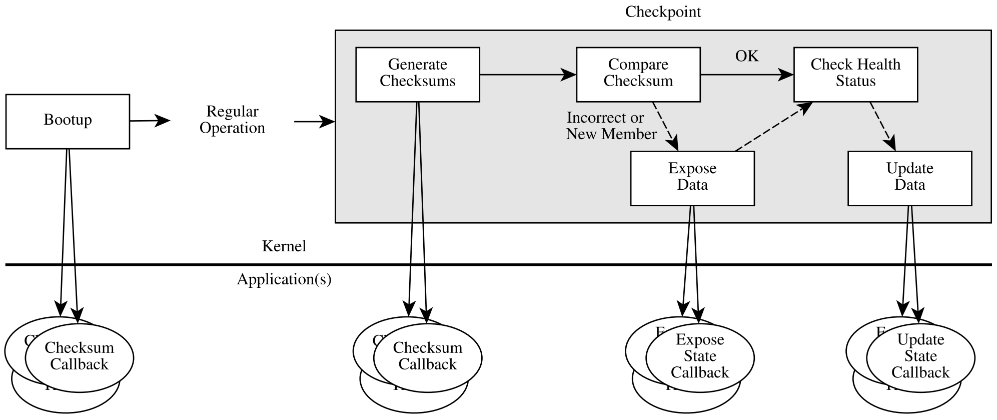


Figure 23: High-level time diagram for the execution of application provided callback functions during the operation on an on-board computer.

Figure 23 depicts where and how these callbacks are used during the regular operation of the lockstep. Some of the callbacks may be omitted, e.g., for applications not requiring bootstrapping or with an already exposed state. The checksum computation and state synchronization callbacks are intentionally placed within the domain of the application developer. This enables decisions about an application state to be taken by the entity with the best knowledge of the individual thread and the means to determine which data is relevant to the system and application state, and must be preserved.

Threads can be executed in an arbitrary order within a lockstep cycle as long as their state is equivalent during the next checkpoint. However, interrupting an active application at a random point in time is usually undesirable. We avoid thread-synchronization issues [198] by enabling the application developer to define comparison points where the application will yield control to the checkpoint handler. If an application requires real-time scheduling, the tightness of the RT guarantees depends upon the time required to execute these callbacks. Communication between thread-groups and compartment-groups is of course possible and will remain reliable, as long as the receiving application is aware that it will receive multiple message replicas. To prevent faults from propagating through IPC channels, a thread can compare the received messages.

Limitations

This approach guarantees system state consistency and control flow correctness after each checkpoint, and for all past checkpoint periods. It also assures computational correctness before the last checkpoint, but can not actively prevent faults from occurring during the ongoing checkpoint cycle. Thus, if one compartment experiences a fault, incorrect results may be propagated outside the system, even though the damage caused to the OBC will be corrected during the next checkpoint, and system state consistency will be asserted. This limitation is inherent to coarse-grain lock-stepping concepts, but could be elevated at the thread-level somewhat using finer-grain event hooking, e.g., system-call hooking [199]. However, this workaround requires in-depth modifications to the OS kernel and development toolchain, is thus non-portable and difficult to maintain, while still not solving the underlying conceptional limitation.

Related research, however, does show that a solution at the system-design level is much better suited to prevent fault-propagation of transient faults between checkpoints using simple I/O voting [201]. Traditional hardware-FT approaches used in space computing are strong for assuring non-propagation of faults across interfaces using hardware-side voting, but can not protect the control-flow and system-state consistency efficiently. While the system state and system-level fault tolerance are assured by Stage 1, and long-term system resilience are safeguarded in Stages 2 and 3, we can utilize simple I/O voting to prevent fault-propagation for compartment groups. Performing I/O voting on interface is already a common practice in satellite computing, as considerable effort is put into providing interface redundancy aboard larger satellites. Small satellites, especially CubeSats, usually can not spare the additional energy, space and mass required for interface replication. For such spacecraft, I/O voting can be implemented on-chip using library IP cores.

4.4 Stage 1: Short-Term Fault Mitigation

Stage 1 offers software-controlled, thread-level, distributed majority voting and fine-grain fault logging within any COTS MPSoC with three or more processor cores. The objective of Stage 1 is to detect and correct faults at each checkpoint to assure computational correctness, control-flow consistency, and a consistent system state after each checkpoint. To do so, Stage 1 requires a processor guaranteeing sequential consistency.

Instead of exerting direct control over the MPSoC, a supervisor can assure FT indirectly, as fault coverage and control are distributed and enforced by the compartments themselves. In consequence, the supervisor does not require any knowledge about the executed application threads, an individual compartment's state, or other OBC intrinsics. The thread group assignment within an MPSoC can be reconfigured freely at runtime to implement different voting configurations. Thus, the described approach can exploit parallelization to improve reliability, throughput, or minimize power consumption, thereby allowing the system to adapt to multi-phased missions with varying performance requirements.

4.4.1 Thread-Based Self-Testing

The program flow of this stage is depicted in Figure 24 and described subsequently. It can be implemented within an existing scheduler and an interrupt service routine (ISR). A practical example for compartment fault handling and recovery, and an overview over how the supervisor interacts with the system are provided at the end of this section.

Bootup & Initialization

After bootup, a compartment first executes basic self-test functionality to assure integrity of compartment-local IP-cores and memory. Each thread's initialization routine is executed on all compartments to allow faster state-update in case a new thread-group is added to a compartment. When being assigned to a compartment, a thread will register its desired checkpoint frequency and its checksum, expose/update callback routines. After the threads have been initialized, each compartment will set a periodic timer to initiate checkpoints. As depicted in Figure 24, a compartment will execute its first checkpoint immediately after the MPSoC has been fully rebooted, to assure that application and OS initialization were successful. If only this individual compartment was rebooted, it can thus return to the spare compartment pool to replace a faulty core in the future.

Checkpoint Start

A checkpoint is triggered by a timer interrupt or externally by the supervisor. A thread can delay a checkpoint until it has reached a viable state for checksum comparison by disabling interrupts, thereby deferring interrupt processing. The checkpoint ISR saves the existing system state, loads the actual checkpoint handler, performs a context switch to kernel mode, and invokes the checkpoint handler.

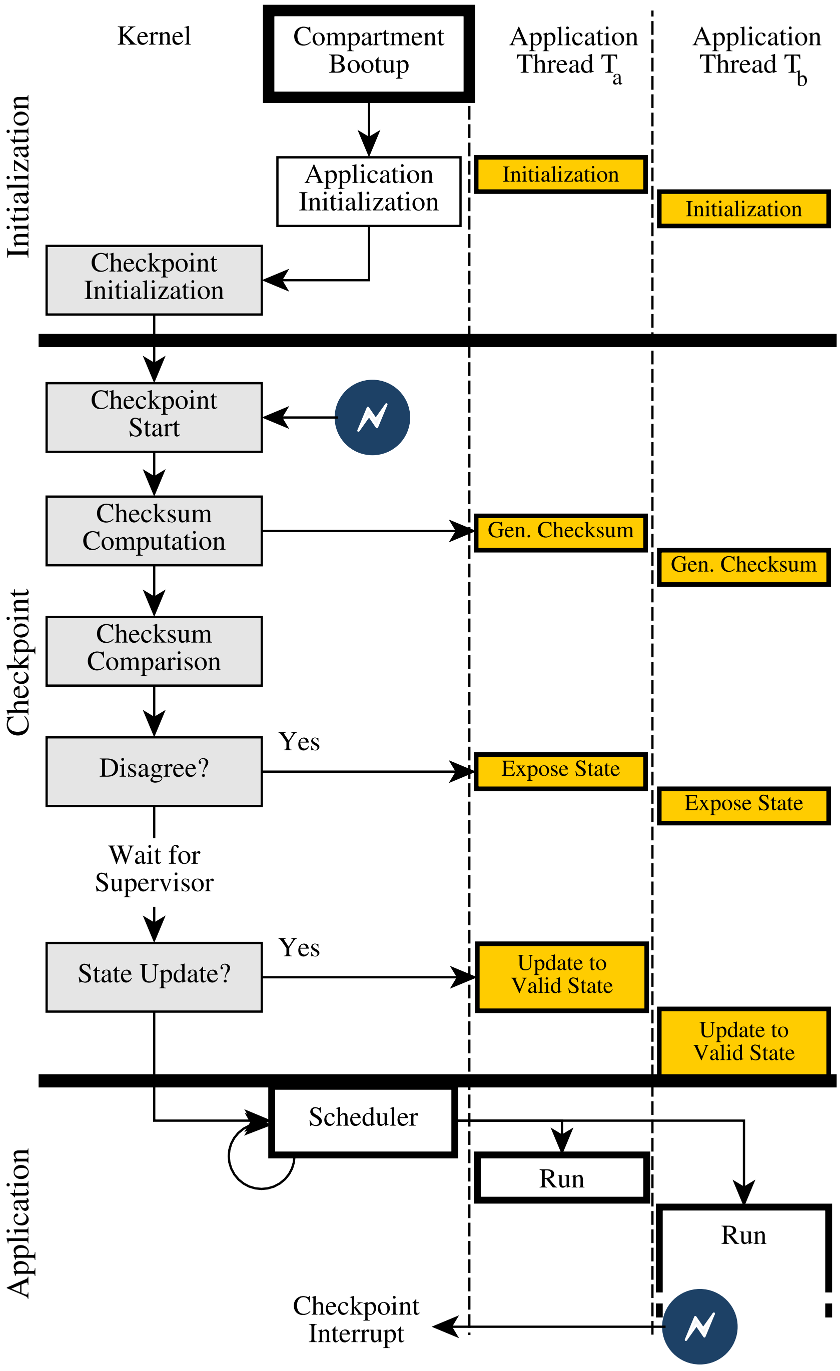


Figure 24: The execution cycle of a compartment during Stage 1. All code necessary for implementation is highlighted in gray, callbacks in yellow.

Checksum Computation

The checkpoint handler invokes each active thread's checksum callback scheduled for checking. As not all threads in a compartment group require the same checking frequencies, not all active threads will be validated during each checkpoint. This checksum callback returns a representation of the application thread's internal state as checksum or hash generated from thread-private variables and other internal application state. The checksum format is compile-time defined, and must be chosen based on FT needs. The algorithm used to generate this checksum is up to the application developer. Each checksum is stored in the compartment's local state memory and thereby exposed to the other compartments. If no checkpoint routine can be provided, a checksum is computed by the checkpoint handler for an application-defined memory range. This memory range can be utilized by the application to deposit state-relevant data passively, e.g., through linker scripts or pre-processor macros. A non-continuously running application can also deposit its results in state memory or return a checksum upon exit.

Prior concepts required deep modifications to the OS to allow a proprietary central health-management entity to retrieve this information directly [198,205], or utilized no application-internal information [200, 201, 211]. Instead, this approach enables us to utilize application-intrinsics to assess the health-state of the system, without requiring any knowledge on the applications. The time required to generate checksums can be minimized by adapting the application code, e.g., by retaining computational by-products which would usually be discarded.

Checksum Comparison

Once all checksum callbacks have been executed, a compartment will monitor its group members' state memory segments until another compartment is ready for comparison. It will do so until it has compared its checksums with all siblings, or the system designer's compartment-group deadline expired. Compartments will usually begin comparing its checksums with siblings immediately or wait only briefly, as delays are mainly induced due to varying memory latency or malfunctions. If it detects a checksum mismatch or a sibling violated the deadline, the compartment will stop comparing checksums and report disagreement with that compartment to the supervisor.

Thread Disagreement & State Propagation

If a compartment detected a checksum mismatch, it executes the expose state callback routine of all threads in the affected compartment group. This callback can be omitted if all state-relevant data is already in state memory, e.g., for non-continuous running applications. The checkpoint routine will adjust the checkpoint's timer if a new thread group was added to the compartment group, and return control to the scheduler.

State Update and Thread Execution

The scheduler will check three conditions during regular operation: if any thread-group is active, the compartment was newly added to a compartment group, or requires an update. Idle compartments sleep until the next checkpoint and can be woken up by the supervisor to reduce energy consumption and fault-potential. In case a compartment must update a thread-group's state from a sibling, the relevant update callback will be executed for each thread. Compartments that have detected disagreement with one of their siblings will delay execution for a compartment-group-wide grace period, to allow a sibling to retrieve a state-copy from state memory. Once a compartment has updated its state using a sibling's data, application processing continues. The other compartment group members will also wake up after the grace period and continue executing threads. This concludes the lockstep cycle.

4.4.2 A Practical Example

Figure 25 depicts a quad-core MPSoC with a single compartment group and three members. A fault has occurred during the second lockstep cycle on compartment C_2, which is subsequently replaced with the idle compartment C_3. C_3 must retrieve a copy of the state of its threads T_a and T_b from another valid sibling. The replaced compartment, C_2, can subsequently be tested for permanent defects by the OS and the supervisor.

4.4.3 Checkpoint-Frequency & Real-Time Capabilities

The level of fault coverage is mainly dependent on the checkpoint frequency. During a checkpoint, the computationally most costly operations are the application checksum callbacks, the expose/update callbacks and a new compartment's update callback. Each of these operations involves a context switch and may imply a varying level of data being read or written. Thus, the performance overhead and fault tolerance capabilities are mainly based upon actual applications checked, as this actual checkpoint handler code is rather trivial. In general, a higher checkpoint frequency implies more time will be spent in checkpoints, finer grained fault-detection are possible, thus better fault coverage.

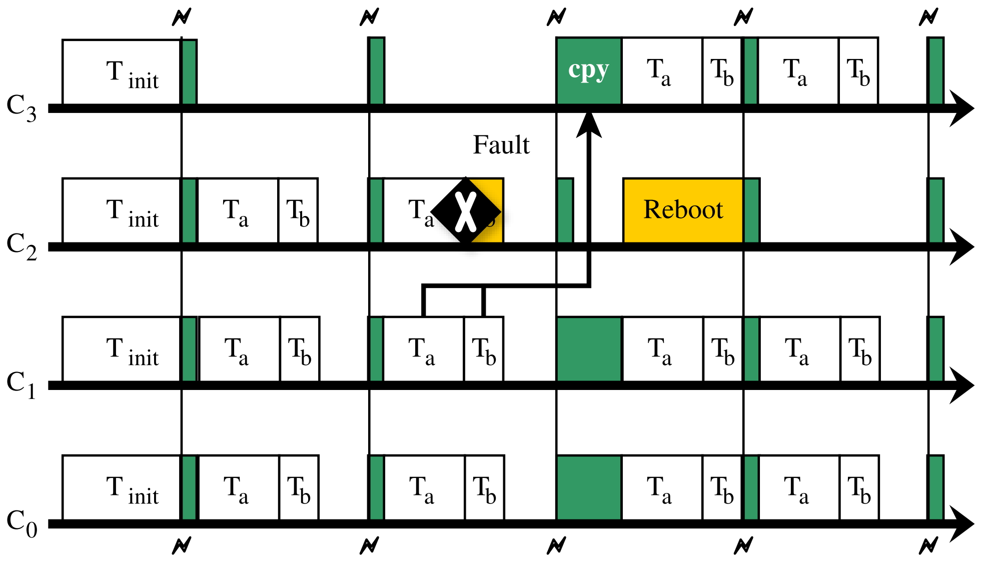


Figure 25: Compartment initialization and a complete Stage 1 lockstep cycle.

In our implementation, interrupts are deferred during a checkpoint, thus applications are not serviced and will not process I/O, thereby affecting the level of real-time capabilities the MPSoC can offer. However, though this can be worked around using a more elaborate interrupt handling concept, e.g., using interrupt prioritization or filtering. Real-time capabilities are thus directly dependent on the MPSoC, and application implementation characteristics, with the OS infrastructure playing a minor role. For complex applications with a large state, a lower checkpoint frequency, however, also implies a larger difference in state. Hence, more data must be copied between compartments to achieve thread-synchronization requiring additional time. Thus, a larger state also requires more time for execution, potentially more complex data structures, thereby implying longer expose- and update-callback.

Overall, the performance of OBCs executing less complex applications with little state will improve with lower checking frequencies. For such OBCs, more checkpoints imply more computational overhead. With more complex applications, there is considerable optimization potential to find a sweet-spot between checkpoint frequency and application-state size. However, performance is strongly dependent assuring that high-quality callback-routines are provided by the application developer.

4.4.4 Supervision

The supervisor is connected to the MPSoC through a multiplexed bus-interface, where each line signals agreement with another compartment. Fine grained disagreement reporting does not significantly improve fault coverage and constrains scalability of the MPSoC. As depicted in Figure 26, the supervisor only reacts to disagreement between compartments, otherwise remaining passive. It maintains a fault-counter for each compartment, and acts as a system-reset inducing watchdog timer for the MPSoC. To resolve 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. Stage 1 alone can not reclaim defective compartments beyond programmatically avoiding the use of defective peripherals, memory pages or processor functionality. Thus, Stage 2 will attempt to repair compartments to prevent resource exhaustion.

In contrast to existing FT solutions, faults can be reported by each compartment individually, because fault detection is decentralized. As this functionality is implemented at the kernel level, we can utilize the OS's powerful logging and diagnostics facilities, instead of relying upon the supervisor to provide a minimal useful level of logging. Diagnostics can thus be enriched with application-level information. Thereby, defect assessment accuracy can be improved compared to prior FT-approaches, enabling more sophisticated debugging without requiring live-interaction.

Our lockstep is effective with very low checkpoint frequencies, requiring few checks in second intervals. Hence the supervisor is no performance bottleneck for the system as a whole. Therefore, high-performance MPSoCs can be well supervised using pre-existing discrete COTS supervisors. COTS MPSoCs will utilize an external supervisor, while ASIC, FPGA and FPGA-SoC-hybrid based MPSoCs can implement this functionality in reconfigurable logic. An off-chip supervisor can be used for ac-

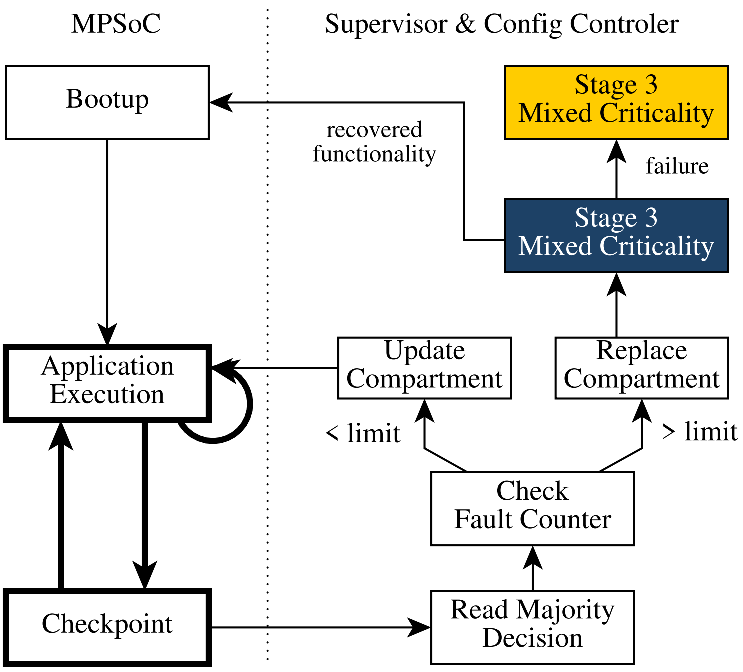


Figure 26: A compartment's and supervisor's program-flow and their interactions. Stage 1, 2 and 3 logic are indicated in white, blue and yellow respectively.

tive compartment health-management and FPGA reconfiguration, enabling the use of FPGA reconfiguration. See Chapter 10 for further details the supervisor interface.

4.5 Stage 2: MPSoC Reconfiguration & Repair

The previous stage can compensate faults as long as healthy compartments are available to replace defective compartments. In all existing hardware-side FT implementations, resource exhaustion is mitigated through over-provisioning (adding more spares). Over-provisioning of compartments naturally is inefficient and curtails system scalability, but is certain due to the static, unchangeable nature of existing ASIC based solutions. This will inevitably result in resource exhaustion, and has not been solved in prior work.

Stage 2 is designed to perform active compartment health management and test, repair, validate and recover faulty compartments, thereby tackling this fundamental limitation. In FPGA-based systems transient faults can corrupt the stored configuration of programmed logic, thus induce permanent effects within the running configuration [215, 216]. However, even if a logic cell is damaged permanently the residual highly-redundant FPGA fabric will remain intact and can be re-purposed [217]. It could be repaired with differently routed, functionally equivalent configurations.

The main issue preventing prior research from utilizing FPGA reconfiguration to increase FT of general purpose computing architectures is a lack of non-invasive, flexible circuit level fault detection. As efficient fault-detection for configurable logic is an unresolved issue, Stage 2 relies upon fault-detection by Stage 1.

The functionality of Stage 2 is depicted in Figure 27. 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 will repeat this procedure with differently routed configuration variants, potentially avoiding or repurposing permanently defective logic.

Assuming a MPSoC architecture outfitted with compartments (see Section 4.7) is used, compartments are topologically isolated. Thus, reconfiguration of just one compartment will not impact the other compartments and allow the OBC to recover a compartment in the background. If reprogramming was unsuccessful or fabric-level faults persist, the supervisor will repeat the previous step with differently routed configuration variants. Partially defective logic cells can be re-purposed, while other cells can be avoided entirely, if no other usage is possible. Other elements of the FPGA fabric can be treated equivalently. The supervisor can also attempt full reconfiguration implying a full reboot of all compartments.

Stage 2 can also test different on-chip memories, the processor cores, and peripheral controllers through external interconnect access ports (e.g., an AXI-bridge). If the OBC is implemented on an ASIC or with a COTS MPSoC, a widely available low-level debug and testing interface such as JTAG can be utilized for the same purpose. Further details on reconfiguration and error scrubbing with a microcontroller-based

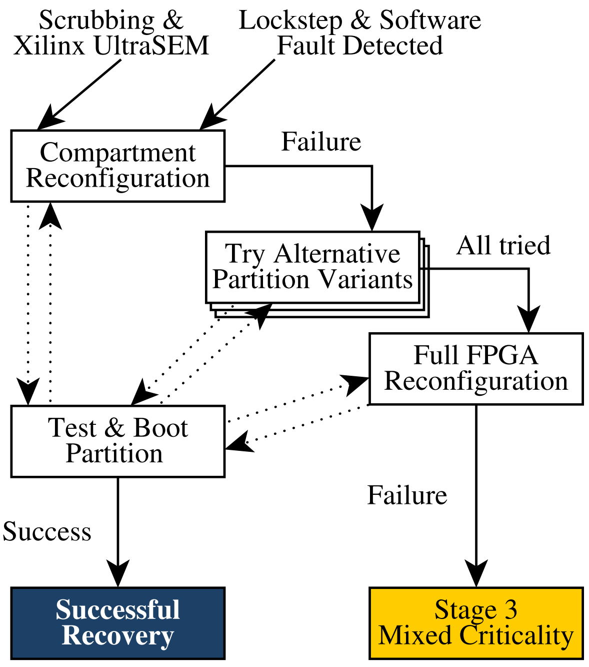

Scrubbing & Xilinx UltraSEM
Lockstep & Software Fault Detected
Compartment Reconfiguration
Failure
Try Alternative Partition Variants
All tried
Full FPGA Reconfiguration
Test & Boot Partition
Failure
Success
Successful Recovery
Stage 3 Mixed Criticality

Figure 27: The objective of Stage 2 is to recover defective compartments and other logic through partial and full FPGA reconfiguration. If this is unsuccessful as well and no further spare processing capacity is available to handle future faults, Stage 3 is activated to find a more resource conserving application schedule, replenishing the spare resource pool.

proof-of-concept implementation for a nanosatellite are available in Chapter 5.

If a defunct compartment can not be repaired through automated reconfiguration, additional diagnostic information can be used for further analysis. The operator can utilize this information to conduct fault analysis on the ground, to craft a suitable replacement configuration to avoid these areas. Of course, this implies extreme development effort but for many higher-priority space missions, the loss of a spacecraft may be more costly than the engineering costs for saving the mission.

4.6 Stage 3: Applied Mixed Criticality

Stage 3 utilizes thread-level mixed criticality to extend an OBC's lifetime once the previous stages have depleted all spare resources. Its primary objective is to autonomously maintain system stability of an aged or degraded OBC at short notice to avert loss-of-mission and loss-of-subsystem, even if an OBC approaches the end of its lifetime. The operator can then define a more resource conserving satellite operations schedule, sacrifice link capacity, or on-board storage space. Thus, dependability for high-criticality threads can be maintained by reducing compute performance, throughput, or increasing latency of lower-criticality applications.

The criticality of applications executed on an OBC can be differentiated by the importance of the controlled subsystem or relevance for commandeering the spacecraft. Performance degradation or even a loss of lower-criticality tasks aboard a satellite is in general preferable to a loss of system stability for key applications. As thread groups can be added and removed from compartment groups, and multiple compartment groups can coexist in the same MPSoC, individual threads can also be migrated between compartment groups [206]. Furthermore, the checkpoint frequency of a compartment group can be reduced to increase a compartment's computational capacity, or it can cease servicing low-priority interfaces.

The supervision logic is extended to reallocate thread-groups across the system based upon the thread's priority. Hence, if Stage 2 failed to reconfigure the OBC, the supervisor can generate new compartment-group assignments for threads with high priority and will attempt to retain existing assignments. Eventually, all healthy compartments will be saturated with threads, and no further assignments will be possible. Then, it can either allocate more mappings, providing lower-priority threads with less processing time to maintain availability, reduce the checking frequency, or leave them inactive. The OBC developer can decide at design time, which applications would benefit most from continuous operation with reduced performance or reliability, and which can be forgone.

In practice a satellite operator can use this functionality also to dynamically adjust the performance of the MPSoC mid mission. This is achieved by adapting the distribution of applications across compartments, the level of replication of application threads, and the processing time allocated to individual application threads. The three properties, thus, are in competition to each other, as depicted in Figure 28. This capability is analogous to the powersaving capabilities present in today's mobile devices and consumer desktop computers, where performance and energy consumption objective compete. An optimal combination of these objectives exists only in theory, but in practice would be very costly to obtain. For practical use, a set of "good enough but non-optimal" can be achieved as at runtime autonomously using heuristics. Further information on Stage 3 including dynamic thread-mapping, as well as performance,

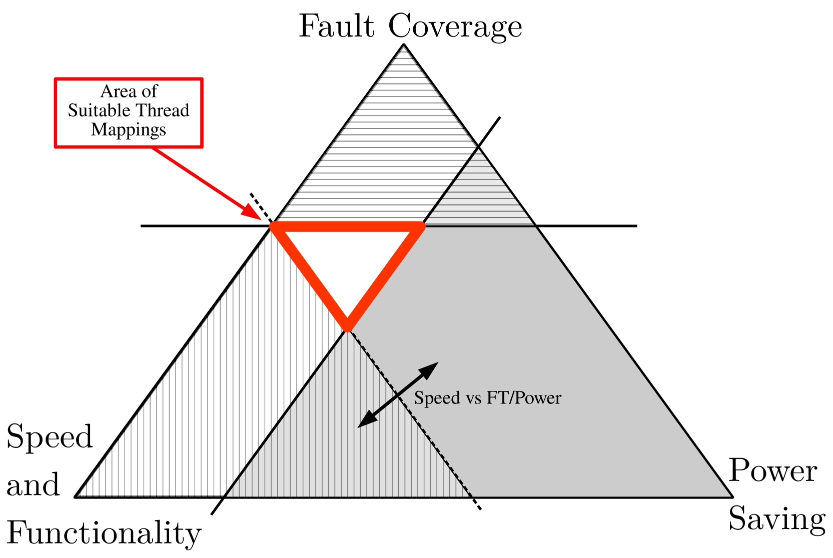


Figure 28: Our architecture allows the system properties of fault tolerance, performance, and energy consumption of an OBC to be adjusted at runtime. The spacecraft operator can prioritize one of these objectives, e.g., to achieve minimum energy consumption by sacrificing processing speed, while maintaining a given level of fault tolerance.

energy and robustness optimization at run-time is available in Chapter 6.

In Figure 29, initially two compartment groups are executed on one MPSoC with 6 compartments. The first group consists of T_a and T_b executed on $C_0 - C_2$, to perform highly-critical platform management and control tasks. The second group performs payload data handling tasks and is initially run on $C_3 - C_5$, and runs its lockstep at half the frequency as the higher critical group mentioned before. It consists of two threads, with T_c acting as payload subsystem driver task of medium criticality, and a computationally expensive low-criticality application T_d performing data compression. In the first checkpoint cycle, a fault occurs on C_5 which is detected after this group executes its first checkpoint. No spare processing capacity is left to replace the failed core with directly. C_2, however, still has sufficient spare capacity to accommodate T_c, but not T_d. T_c is migrated to a separate, new compartment group and executed on compartments 2 – 4, thereby maintaining strong FT. The lower-criticality task T_d remains degraded. Therefore, T_d will continue to run in DMR mode on the intact cores C_3 and C_4, which only allows fault-detection in the future.

4.7 Platform Architecture

Our multi-stage FT-approach is in principle platform independent and can be implemented within any multi-threading capable OS supporting interrupts and timers. For most COTS-MPSoC based nanosatellites in a LEO orbit, stage 1-3 alone offer sufficient fault coverage. Aboard such spacecraft, MPSoC interfaces are either unprotected or protected programmatically and outside the MPSoC (e.g., using EDAC chips or by resolving SEFIs through power cycling). Aboard larger, more critical spacecraft

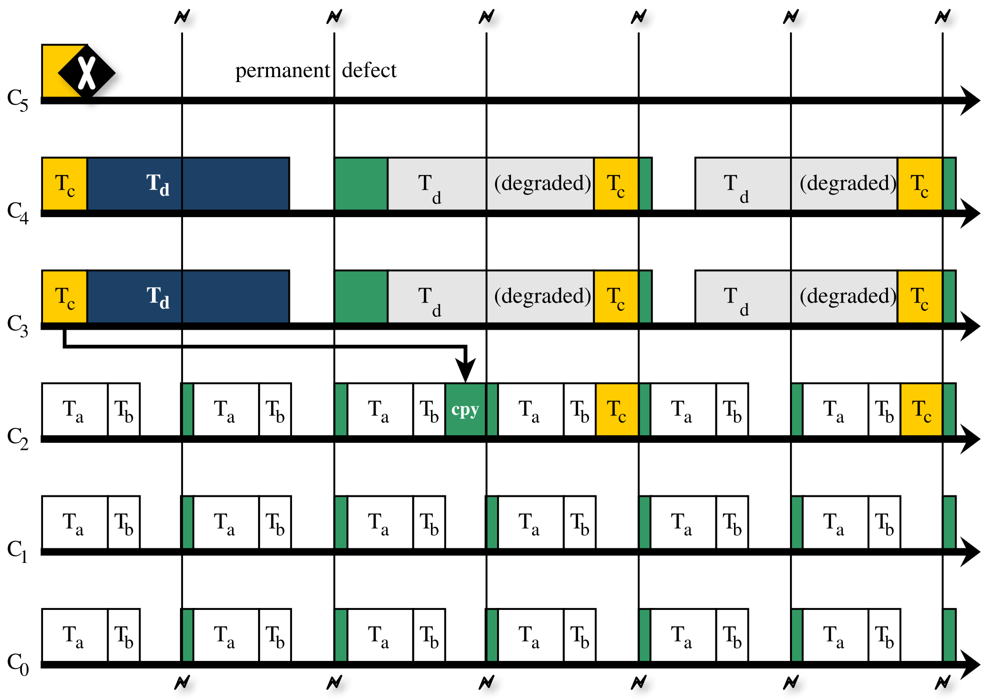


Figure 29: If no healthy spare compartments are available, the Stage 3 can split defunct compartment groups and uphold FT guarantees for high-criticality threads. The necessary adjustment to the checkpoint frequency on compartment 2 is omitted for simplicity.

such faults can not be accepted, and OBC interfaces are usually implemented redundantly at great effort. This redundancy is inherent to our approach with due to the compartment-based architecture, and we developed an MPSoC platform capable of surviving the loss of peripheral devices and permanent, non-resolvable defects in interfaces.

4.7.1 MPSoC Architecture Concept

This MPSoC can be implemented in full using library IP available with standard industry FPGA or ASIC design tools without custom FT components. We have implemented our MPSoC prototype with Xilinx Vivado standard IP, AXI Interconnects, for low-tier ARM Cortex-A processor cores to be provided by one of our industrial partners. For common space applications, size-optimized cores such as the Cortex-A32, -A35 and A5 offer an excellent balance between performance, universal platform support and logic utilization. The architecture minimizes shared logic, compartmentalizes compartments, and offers a clearly defined access channel between compartments for sharing checkpoint-results and application-state. We are aware that most miniaturized satellites do not require such a high degree of fault coverage, and often can not afford the added hardware complexity and development effort.

The MPSOC depicted in Figure 30 follows a compartmentalized architecture. The software run on the individual processor cores is strongly isolated from each other. It is meant to be implemented within an FPGA to counter resource exhaustion when mitigating faults in Stage 1. It utilizes simple redundancy to compensate for SEFIs, but does not contain radiation-hard or FT processor cores or custom logic. Each compartment is equipped with a processor core, an interrupt controller (IRQ in the figure), a dedicated on-chip memory slice used as state memory, and several peripheral interfaces through the local interconnect. Compartments are connected through an I/O memory management unit (IOMMU) and a global interconnect to main- and non-volatile memory. They can not access the local interconnect of other compartments to prevent interference and minimize shared logic. This compartmentalized architecture benefits from partial reconfiguration, as compartments can be placed strategically on an FPGA's fabric along partition borders. Our approach and this architecture support multi-FPGA and -ASIC MPSoCs without adaptation, thereby improving scalability and resilience against FPGA-level SEFIs.

The ECC-protected dual-port state memory in each compartment holds the current compartment-status, thread assignments, as well as the checksums and state information. One interface is connected to the compartment's local interconnect, while the second port is read-only accessible via the global interconnect. The state memory is inherently redundant, as threads are executed on at least two compartments. The

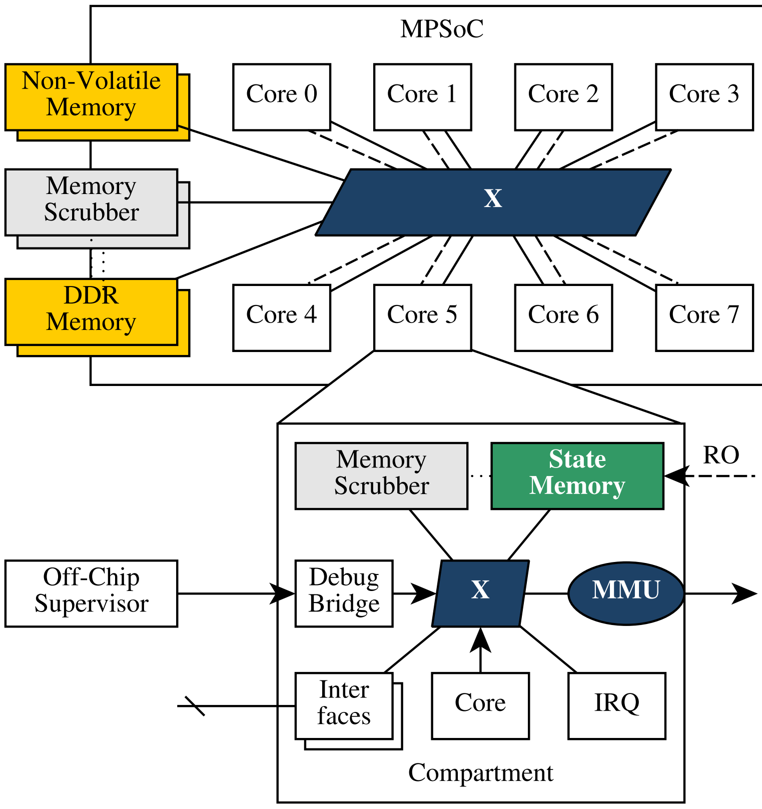


Figure 30: A simplified representation of the presented MPSoC with memory controllers highlighted in yellow, scrubbers in green, and interconnect in blue. A dedicated interface on each compartment allows supervisor access.

shared main memory is redundant to safeguard from SEFIs affecting the compartment-shared interface. Both instances are ECC protected and connected to the global interconnect. The main memory is split into several segments: each compartment has write-access to its own segment, and can read the global shared code segment. ECC-fault syndrome interrupts for main memory are handled by the supervisor. We perform error-scrubbing on these memories to avoid accumulating bit-flips due to transient and permanent faults. The scrubbing frequency should be set depending on the actually used memory technology, production node and mission parameters. Non-volatile memory is implemented redundantly as well. Our prototype is designed to utilize radiation immune MRAM and PCM [197] and we realize advanced FT for these memories as described in Chapter 7. Each compartment's main memory segment, state memory, and non-volatile memory are mapped to the same compartment-local address ranges. At the thread-level, the address-space in each compartment is thus identical, making application and OS code location independent and allowing compartments to share binaries. Further implementation details are available in Section 4.10.

4.7.2 Feasibility

We developed an early MPSoC design based on the this architecture utilizing exclusively library-IP. Instead of ARM cores, this quad-core demonstration design includes Xilinx MicroBlaze processor cores, as these are more available to the general public. It targets standard FPGA development boards and is equipped with a single shared DDR4 main memory controller, and 2MB on-chip BRAM program memory. This reduced design was implemented successfully using the Xilinx Vivado Design Suite and Stage 1 was implemented using FreeRTOS and using the Xilinx SDK toolchain.

Each compartment is outfitted with data and instruction caches, an interrupt controller, a UART interface, state memory and an additional local memory for storing compartment-private information, and a GPIO controller to signal agreement between compartments. All compartment-local memories are equipped with ECC, as this increases logic size of the relevant memory controllers, and includes two additional interrupts for each connected memory. We could achieved full timing closure at 250MHz core frequency on VCU118 and KCU116 development kits, though the clock frequency

Resource	Utilization	Available	**Utilization %**
LUT	68,705	1,182,240	**5.81%**
LUTRAM	9,235	591,840	**1.56%**
FF	92,536	2,364,480	**3.91%**
BRAM	810	2,160	**37.48%**
DSP	27	6,840	**0.40%**
IO	163	832	**19.59%**
BUFG	17	1,800	**0.94%**
MMCM	6	30	**20.00%**

Table 3: Resource utilization of the quad-core demonstration MPSoC on a Xilinx VCU118 development board. The on-chip program memory and DDR4 memory controller disproportionately inflate BRAM utilization.

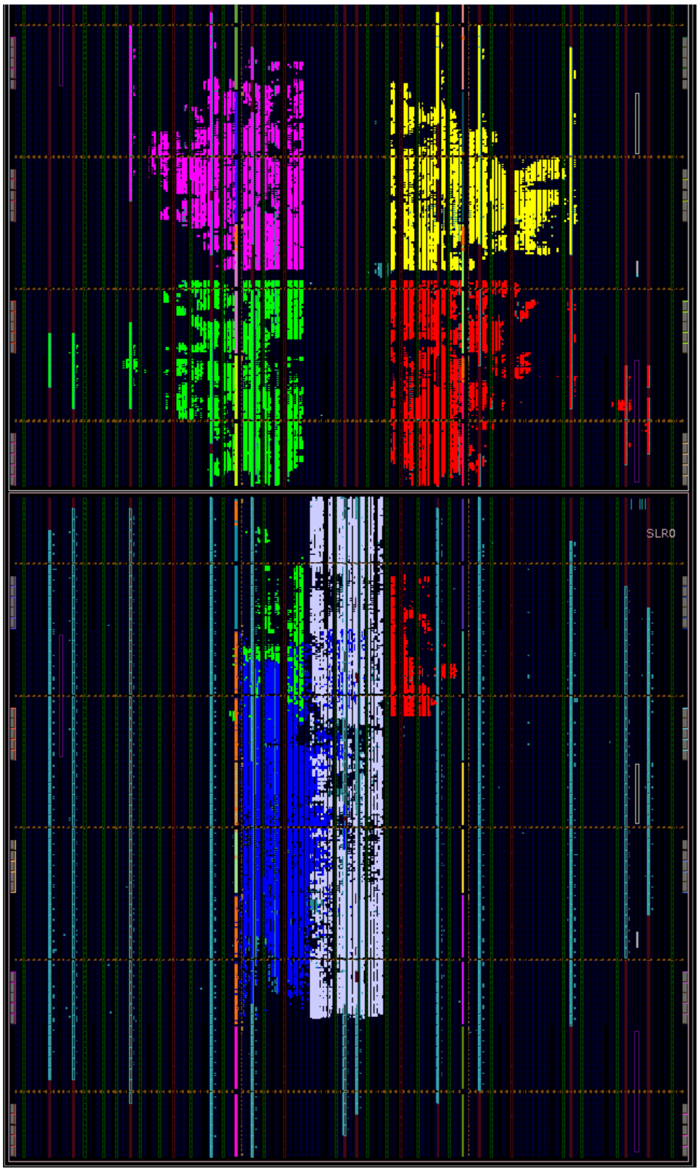


Figure 31: Logic placement of the demo-MPSoC on a VCU118 development board running 4 Compartments: green, red, yellow, pink; Global Interconnect: white; Xilinx DDR4 controller: blue; Program Memory: teal.

was selected to achieve a simple design, not an efficient or fast one. If additional time was invested into timing optimization and clocking, the clock speed can be drastically increased. Additional information regarding the compartment and SoC layout are available in Chapters 9 and 10.

Fabric utilization based upon the Xilinx Virtex VCU118 Development Kit is depicted in Figure 31. Due to the use of on-chip program memory and the DDR4 memory controller, BRAM utilization is inflated compared to the MPSoC described previously.

Resource utilization is indicated in Table 3, with more details given in Section 4.10. Stage 2 and 3 do not require additional FPGA logic.

This design's very low logic usage shows that the architecture itself can be scaled to 8 and more compartments comfortably, and most current-generation FPGAs offer an abundance of unused resources for Stage 2. With current-generation FPGA platforms, Stage 2 will thus not only be able to recover defective compartments using spare resources, but could even place multiple compartments as cold or hot spares. The Microblaze cores utilized here for demonstration purposes can directly be replaced with more powerful processor cores, assuming the necessary peripheral IP is added as well (e.g., an ARM GIC instead of the MicroBlaze Interrupt Controller).

4.8 Discussions

The reliability of each individual compartment's voting decision can be weak, and an individual compartment can report false (dis)agreement with its siblings. Our approach takes into account that any software or hardware component associated within a compartment can fail arbitrarily. Such failure is mitigated through a distributed decision, which is taken based on each compartment's perspective of its siblings. Thus, this approach does not require the checksum logic to compute correctly, and we assume that faults may occur at any time during the lifetime of a compartment. As compartment groups usually consist of three or more compartments, the likelihood of false-disagreements or non-reported disagreement is insignificant. To mask such a fault, multiple faults would have to coincide in a majority of compartments within the same compartment group during a single checking period and induce the same fault. The probability for such an event is extremely low, except at very high radiation levels. Even in such situations, such faults would be detected after the subsequent checkpoint with near certainty.

Prior research proves the conceptual effectiveness of thread-based FT [88,200] and software-based FT combined with simple I/O voting [201]. Also, the detailed FT capabilities of a platform utilizing our approach are influenced by the actually used FPGA, ASIC or COTS-MPSoC design. These imply mainly design decisions and a varying acceptance of single-points-of-failure. Schedulability, timing conformity, and deadlock-avoidance have been extensively researched in literature, e.g., in [210]. Thus, what remains to be shown is the runtime performance overhead induced by the presented approach, as the main objective of our research is to enable the efficient use of high-performance mobile-market COTS MPSoCs within satellite computers. To achieve worst-case performance estimations, we developed a naive, unoptimized implementation of the Stage 1 of our approach, as the others do not affect the runtime performance of the MPSoC. This naive implementation shows a median-best performance degradation of 9% and median-worst degradation of 26% on compartments with a single processor core. Further information on the conducted tests is available in Section 4.10, as well as performance measurements for 6 different application scenarios modeled after the NASA/James Webb Space Telescope's Mid-Infrared Instrument (MIRI) [219].

As prior thread-level FT implementations [199, 200, 208] are based upon fundamentally different concepts, only address transient faults within a very limited scope, and are deeply embedded into proprietary OS, their fault coverage and performance can not be directly compared. However, the measured performance overhead does fall within the same range as measured in [199], and we also observe comparable average-case performance. To put these measurements into context, even a 50% slowdown on modern MPSoCs will offer a factor-of-5 performance increase over state-of-the-art radiation-hardened processor designs, thereby showing a favorable cost-vs-benefit trade-off.

4.9 Conclusions

In this chapter, we presented the first practical and integral multi-stage approach to fault-tolerant (FT) general purpose computing for spaceflight use. The approach explicitly does not utilize radiation-hardened or hardware-FT processor cores and utilizes no central MPSoC-internal voting logic. It can thus be implemented within COTS MPSoCs or alternatively entirely with non-FT, standard library IP-cores available in FPGA or ASIC design software. In contrast to prior research, the presented approach considers the full and realistic fault-model for space computing, and operates within real-world constraints. The approach does not require failure-free components within an MPSoC or in the OS, and does not leave conceptual gaps, e.g., regarding fault detection and recovery. It is not based upon traditional radiation-hardened processor cores and does not achieve fault tolerance through hardware-measures.

We showed that our approach is programmatically simple and requires little custom code, which can also be implemented in most pre-existing multi-threading capable OS. Faults can be detected and mitigated using application provided routines, enabling decisions about an application's integrity to be taken by the application developers themselves. As a consequence, the system designer no longer must struggle to assess the health of each individual application's state, and instead can focus on determining an optimal solution to problems at hand. It allows flexible fault-detection, mitigation and recovery within COTS MPSoCs, laying the foundations for FT computing aboard miniaturized satellites, and helping to bridge the gap between theoretical embedded research and practical implementation in the space industry. While remaining flexible, and inducing only a minimal performance overhead, the presented multi-stage approach offers time-bounded real-time guarantees.

The approach can be well complemented with several other reliability-improving measures which were integrated into the outlined reference MPSoC architecture. Preliminary benchmark results of an unoptimized implementation show a low performance overhead, suggesting a beyond factor-of-5 performance increase over state-of-the-art radiation-hardened processors for space use. Our approach allows the host platform to scale vertically (more powerful processor cores and more interfaces per compartment) as well as horizontally (more compartments), with virtually any modern processor core. Thereby, we aim to increase acceptance for software-side FT approaches in the space industry, building trust in hybrid hardware-software architectures. Thus, our approach is the first integral, real-world solution to enable the fault-tolerant application with modern MPSoC designs for critical satellite control applications, thereby enabling the use of such SoCs in future high-priority space missions.

4.10 Annex: Worst-Case Performance Estimation

To achieve worst-case performance estimations, we developed an unoptimized implementation of the first stage of our approach in C to be run in user-space. The provided benchmark results were generated based on code derived off a special CCD readout program used for space-based astronomical instrumentation. The application was executed with a varying amount of data processing runs in a compartment group at the indicated checking frequencies, and without protection for reference.

4.10.1 Implementation Outline

This implementation was written in approximately 800 lines of user-space C-code including benchmark facilities. It utilizes system calls and the POSIX threading library to simulate compartments and thread management. Thread-management at this level is computationally much more expensive than if performed bare-metal or in kernel-code. A bare-metal implementation within an operating system reduces this performance overhead drastically. This implementation therefore allows very pessimistic benchmarking, which can yield a baseline for the lockstep's performance cost. The implementation also serves as an excellent simulator to validate the correctness of the described logic, and allows better debugging than on the actual MPSoC implementation.

4.10.2 Test Application

Synthetic, widely used benchmark suites are unsuitable to benchmark OS-level functionality. Thus, we derived a demo-application off an astronomical instrumentation application. We chose to utilize the background scenario of scientific computing, as devices for scientific instrumentation are usually better documented. The program flow of our demo application is based on the NASA/James Webb Space Telescope's Mid-Infrared Instrument (MIRI) described in [219]. This program continuously reads three 16-bit 1024x1024 false-color sensor arrays, stores, and processes the results. It averages multiple captured frames to optimize the instruments exposure time and avoid pixel saturation, or to capture faint astronomical sources [219].

4.10.3 Methodology and Test Setup

The setup simulates an MPSoC three compartments executing the described demo application, and measures performance of the application executing within a compartment. For each plot in Figure 32, 100 measurements were taken of the real-time necessary to process 600 1-Megapixel frames with subsequent processing runs. Data heavy modes indicate a high amount of post-processing runs, whereas compute-heavy modes indicate lower per-thread workload.

- Very Compute Heavy: 60000 Postprocessing Runs
- Compute Heavy: 75000 Postprocessing Runs
- Balanced Compute Heavy: 90000 Postprocessing Runs
- Balanced Data Heavy: 105000 Postprocessing Runs

- Data Heavy: 135000 Postprocessing Runs
- Very Data Heavy: 150000 Postprocessing Runs

Benchmark results were generated on a Intel Core I7-2600K Sandy Bridge-based system with a host kernel's scheduling frequency of 1kHz (`CONFIG_HZ_1000`). Hyper-Threading and SpeedStep was disable to avoid interference between threads. Binaries were compiled with `GCC 6.3.1 (20161221)` without compiler optimization (`-O0`).

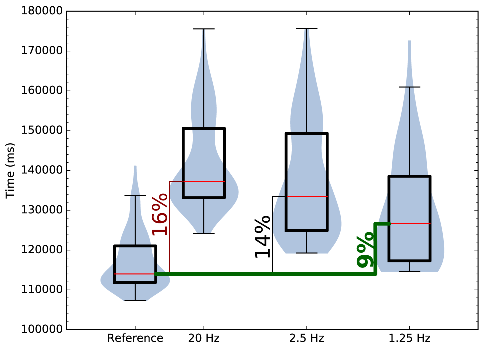
4.10.4 Results

This naive implementation of our approach at the application level on Linux shows median-best performance degradation of 9% and median-worst degradation of 26%, which are also indicated in Figure 32a and e in bold. Across all test runs, we measured on average 80% worst-case and 95% best-case performance compared to the unprotected reference runtime. The violin plots – shadows around the box-plots – indicate the distribution of the measurements to depict the accumulation of the individual measurements.

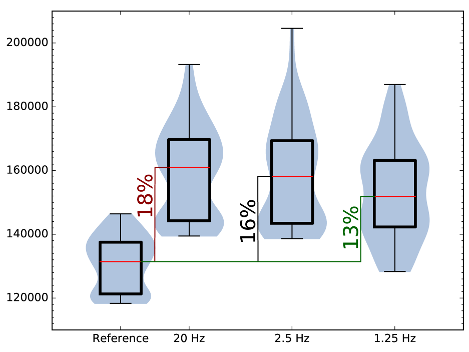
As expected, the performance varies depending on workload, with data-heavy tasks a-c showing better performance. This too was expected as Stage 1's code consists mainly of integer operations, binary comparisons, load/stores, and jumps. Better performance can be expected in a more optimized implementation at the kernel level due to a reduced computational cost of operations that in userland require system calls. To put these measurements into context, even a 50% performance degradation on modern MPSoCs will offer a factor-of-5 performance increase over state-of-the-art radiation-hardened processor designs.

Assuming an average performance degradation between 10% and 20% at such extreme checking frequencies, our approach can thus allow a modern MPSoC to perform better than comparable state-of-the-art hardware-voting based processor solutions, while requiring no proprietary processor design, offering full software-control at a fraction of the development effort and costs. And in contrast to existing hardware-based fault tolerance solutions, our architecture does not struggle against feature-size reduction, but scales up with technology and benefits from more modern production nodes.

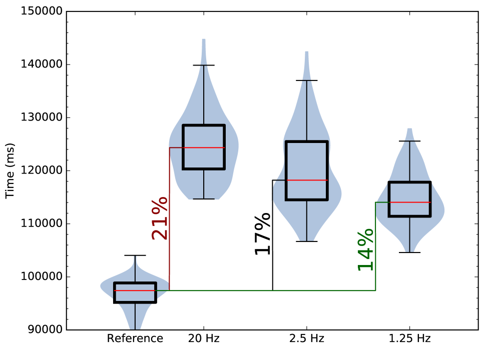
The lockstep was run with very high checkpoint frequencies (20hz, 2.5hz and 1.25hz) which during normal operation will most likely never be used. For most LEO applications, we expect that checkpoints would be run only every 5 to 10 seconds. Furthermore, system calls and thread-management on high-performance mobile-market processor cores can be much less costly than when run on desktop hardware. Realistically, this would implying very little performance cost ranging from 0.5% to 2% overhead.



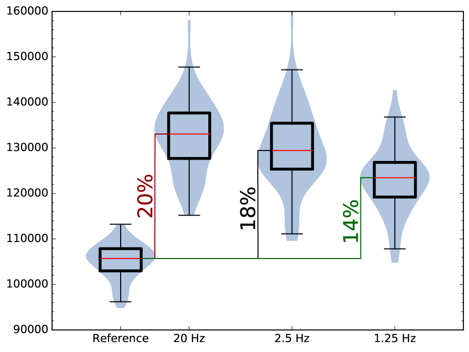
(a) Data-Heavy



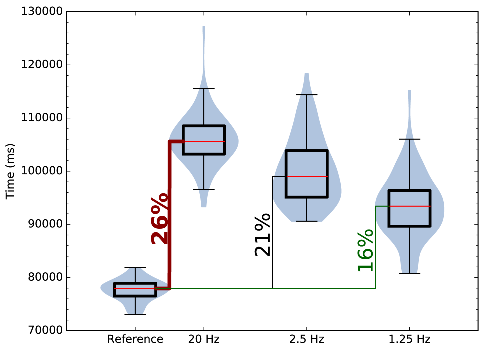
(b) Very Data-Heavy



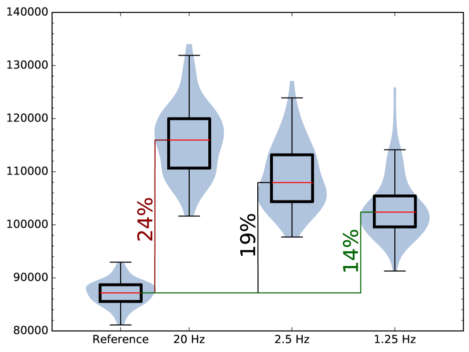
(c) Balanced Compute-Heavy



(d) Balanced Data-Heavy



(e) Very Compute-Heavy



(f) Compute-Heavy

Figure 32: Performance measurements of 6000 runs for processing 100 1024x1024 pixel CCD frames with different checkpoint frequencies and workloads.