Chapter 7

Reliable Data Storage for Miniaturized Satellites

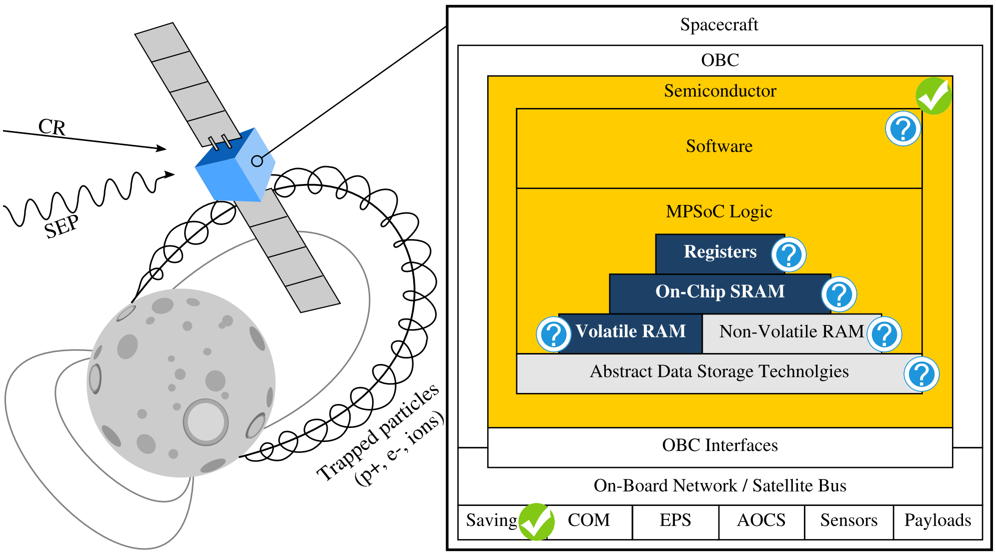
Memory Fault Tolerance

Reliable operation of an OBC can only be guaranteed if the integrity of the OBC's firmware, operating system, applications, as well as payload data can be safeguarded. Chapter 7 is therefore dedicated to discussing storage fault tolerance to answer RQ4. We discuss how the robustness of a nanosatellite's volatile memory components can be increased through software measures, as space-grade parts with strong erasure coding are not available to CubeSat designers. In the later parts of the chapter we cover integrity protection for data stored in commercial non-volatile memory ICs. The research presented in this chapter was published as finalist paper in the proceedings of the AIAA/USU Conference on Small Satellites (SmallSat) *[Fuchs15]. The sections related to MRAM and flash memory were published in the proceedings of the* International Conference on Architecture of Computing Systems (ARCS) *[Fuchs18] and the* International Space System Engineering Conference Data Systems In Aerospace (DASIA) *[Fuchs16].*

7.1 Introduction

Recent miniaturized satellite development shows a rapid increase in available compute performance and storage capacity, but also in system complexity. CubeSats have proven to be both versatile and efficient for various use-cases, thus have also become platforms for an increasing variety of scientific payloads and even commercial applications. Such satellites also require an increased level of reliability in all subsystems compared to educational satellites, due to prolonged mission duration and computing burden. Nanosatellite computing will therefore evolve away from federated clusters of microcontrollers towards more powerful, general purpose computers; a development that could also be observed with larger spacecraft in the past. Certainly, an increased computing burden also requires more sophisticated operating system (OS) or software, making software-reuse a crucial aspect in future nanosatellite design. In commercial and agency spaceflight, a concentration on few major OSs (e.g., RTEMS [248]) and processors (e.g., LEON3 and RAD750) has therefore occurred. A similar evolution, albeit much faster, can also be observed for miniaturized satellites.

To satisfy scientific and commercial objectives, miniaturized satellites will also require increased data storage capacity for scientific data. Thus, many such satellites have begun fielding a small but integrity-critical core system storage for software, and a dedicated mass-memory for pre-processing and caching payload-generated data. Unfortunately, traditional hardware-centered approaches to fault tolerance, also increase costs, weight, complexity and energy consumption while decreasing overall performance. Therefore, such solutions (shielding, simple- and triple-modular-redundancy – TMR) are often infeasible for miniaturized satellite design and unsuitable for nanosatellites. Also, hardware-based error detection and correction (EDAC) becomes increasingly less effective if applied to modern high-density electronics due to diminishing returns with fine structural widths. As a result of these concepts' limited applicability, nanosatellite design is challenged by ever increasing long-term fault coverage requirements.



7.1.1 Context and Application

Neither component level, nor hardware or software measures alone can guarantee sufficient system consistency. However, hybrid solutions can increase reliability drastically introducing negligible or no additional complexity. Software driven fault detection, isolation and recovery from (hardware) errors (FDIR) is a proven approach also within space-borne computing, though it is seldom implemented on nanosatellites. A broad variety of measures capable of enhancing or enabling FDIR for on-board electronics exists, especially for data storage. Combined hard- and software measures can strongly increase reliability.

This research was conducted as part of the MOVE-II CubeSat project based upon an ARM-Cortex processor as a platform for scientific payloads. To fulfill this role, the traditional CubeSat approach to reliability, risk acceptance, does not suffice. Hence, we designed MOVE-II's on-board computer (OBC) to guarantee data integrity using software side measures and affordable standard hardware where necessary. The capability to assure data integrity for program code and data is essential to then achieve fault-tolerance for data processing elements and at the system level.

After a detailed evaluation of potential OSs for use aboard MOVE-II, we chose the Linux kernel due to its adaptability, extensive soft-/hardware support and vast community. We decided against utilizing RTEMS mainly due to our limited software development manpower, the intended application aboard our nanosatellite MOVE-II, and the abundant compute power of recent OBCs.

7.1.2 Chapter Organization

Often, fault tolerance aboard spacecraft is only assured for processing components, while the integrity of program code is neglected. In the next section, we thus outline the importance of memory integrity as a foundation for fault-tolerant satellite computing and provide a view on the topic at a high level. To protect data stored in volatile memory, we present a minimalist yet efficient approach to combine error scrubbing, blacklisting, and error correction encoded (ECC) memory in Section 7.3. MOVE-II will utilize magnetoresistive random access memory (MRAM) [147] as firmware storage, hence, we developed a POSIX-compatible filesystem offering memory protection, checksumming and forward error correction. This filesystem is being presented in Section 7.4, can efficiently protect an OS- or firmware image and supports hardware acceleration. Finally, a high performance dependable storage concept combining block-level redundancy and composite erasure coding for highly scaled flash memory was implemented to assure payload data integrity, the resulting concept is outlined in Section 7.5. The final section of this chapter is used to discuss and wrap up the results obtained herein.

7.2 Data Integrity as Foundation of Fault Tolerance

The increasing professionalization, prolonged mission duration, and a broader spectrum of scientific and commercial applications have resulted in many different proprietary on-board computer concepts for miniaturized satellites. Therefore, miniaturized satellite development has not only seen a rapid increase in available compute power and storage capacity, but also in system complexity. However, while system sophis-

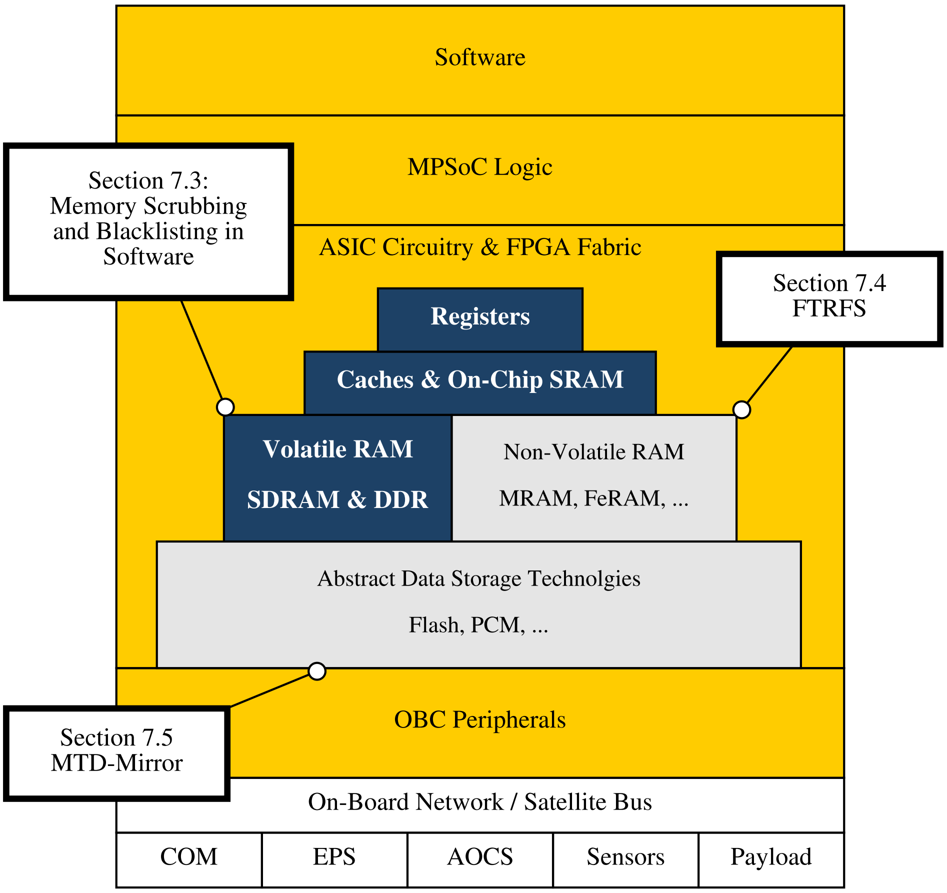


Figure 43: Data transiting or stored within components or system components shown in yellow and white may be corrupted due to radiation effects. Components depicted in blue can be safeguarded against data corruption using the concepts presented in the different sections of this chapter as indicated.

tication has continuously increased, re-usability, reliability remained quite low [249]. Recent studies of all previously launched CubeSats show an overall launch success rate of only 40% [41]. Such low reliability rates are unacceptable for missions with more refined or long-term objectives, especially with commercial interests involved.

As nanosatellites consist mainly of electronics, connected to and controlled by the OBC, achieving fault tolerance must begin with this component. Hence, an OBC's software and hardware must be designed to handle faults throughout a space mission, not if, but when they occur. fault tolerance can only be assured if program code and required supplementary data can be stored consistently and reliably aboard a spacecraft. Thus, data storage integrity must be assured first and foremost, without resorting to expensive, proprietary space-grade components that realize fault tolerance in hardware.

To enable meaningful fault tolerance, data consistency must be assured both within volatile and non-volatile memory, see Figure 43. Data is usually classified as either system data or payload data stored in volatile or non-volatile memory. The storage capacity required for system data may vary from few kilobytes (firmware images stored within a microcontroller) to several megabytes (an OS kernel, its and accompanying software). Very large OS installations and applications are uncommon aboard spacecraft and thus not considered in this chapter. Payload data storage on the other hand requires much larger memory capacities ranging from several hundred megabytes to many terabytes depending on the spacecraft's mission, downlink bandwidth or link budget, and mission duration. In addition, data and code will temporarily reside in volatile system memory and of course the relevant memories within controllers and processors (i.e. caches and registers) which again must satisfy entirely different requirements to performance and size.

Due to these varying requirements, different memory technologies have become popular for system data storage, payload data storage and volatile memory. In the following sections, we will discuss and develop protective concepts to ensure memory integrity aboard spacecraft with a special focus on our nanosatellite use-case. All these concepts can be implemented at least as efficiently to larger satellites, as size and energy restrictions are much less pressing aboard these vessels.

7.3 Volatile Memory Consistency

Inevitably, data stored will at least temporarily reside within an OBC's volatile memory and all current widely used memory technologies (e.g., SRAM, SDRAM) are prone to radiation effects [250]. As a straightforwards solution, some OBCs were built to utilize only (non-volatile) MRAM as system memory which is inherently immune to SEUs and therefore allows OBC engineers to bypass additional integrity assurance guarantees for RAM. However, MRAM currently can not be scaled to capacities large enough to accommodate more complex OSs. Thus, while miniaturized satellites often utilize custom firmware optimized for very low RAM usage, larger spacecraft as well as most current and future nanosatellites do utilize DDR or SDRAM. For simplicity, we will refer to these technologies as RAM in this chapter. However, it is not to be confused with the use of the term RAM in Sections 7.4 and 7.5 of this chapter, as in MRAM.

Radiation induced errors alongside device failover is often assured using error correcting codes (ECC), which have been in use in space engineering for decades. However, a miniaturized satellite's OS must take an active role in volatile memory integrity assurance by reacting to ECC errors and testing the relevant memory areas for permanent faults. To avoid accumulating errors over time in less frequently accessed memory, an OS must periodically perform scrubbing. In case of permanent errors, software should cease utilizing such memory segments for future computation and blacklist them to reduce the strain on the used erasure code. Assuming these FDIR measures are implemented, a consistency regime based on memory validation, error scrubbing and blacklisting can be established.

7.3.1 DRAM Corruption and Countermeasures

The fault profile for DRAM aboard CubeSats mainly includes two types of gradually accumulating errors: soft-errors (bit-rot) and permanent (hard) errors. Depending on the amount of data residing in RAM, even few hard errors can cripple an on-board computer: the likelihood for the corruption of critical instructions increases drastically over time. Therefore, to compensate for both hard and soft errors, ECC should be introduced [158].

Modern DRAM chips benefit strongly from feature size reduction and run at very high clock frequency, as a vast majority of a memory IC consists of memory cells. Soft errors there occur on the Earth as well as in orbit, due to electrical effects and highly charged particles originating from beyond our solar system. In case of such an error, data is corrupted temporarily but, and once the relevant memory has been re-written, consistency can be re-established. The likelihood of these events on the ground is usually negligible as the Earth's magnetic field and the atmosphere provide significant protection from these events, thus weak or no erasure coding at all is applied.

Hard errors generally occur due to manufacturing flaws, ESD, thermal- and aging effects. Thus, they may also occur or surface during an ongoing mission, further information on the causes for hard-faults in RAM is described in detail in [251].

By utilizing ECC, integrity of the memory can be assured starting at boot-up, though in contrast to other approaches ECC can not efficiently be applied in software [252]. Due to the high performance requirements towards RAM, weak but fast erasure codes such as single error correction Hamming codes with a word length of 8 bits are used [253, 254]. ECC modules for space-use usually offer two or more bit-errors-per-word correction. These codes require additional storage space, thereby reducing available net memory, and increase access latency due to the higher computational burden. Single-bit error correcting EDAC ASICs are available off-the-shelf at minimal cost, whereas multi-bit error correcting ones are somewhat less common and expensive. While such economical aspects are usually less pressing for miniaturized satellites beyond the 10kg range, nanosatellite budgets usually are much more constrained prompting for alternative, lightweight low-budget-compatible solutions. In the remainder of this section, we thus present a software driven approach to achieve a high level of RAM fault-coverage. We do so using commercial ECC paired with software measures, without expensive and comparably slow space-grade multi-bit-error correcting logic.

Ultimately, strong ECC is not a satisfying final solution to RAM consistency requirements due to inherent weaknesses of this approach to controller-faults, chip-level failure, and data-economical reasons in prolonged operation. Highly charged particles impacting the silicon of RAM chips can also permanently damage the circuitry of controller logic. In consequence, radiation can induce faults in control logic and other infrastructure elements of a memory IC, which there can causing SEFIs [255]. In contrast to hard and soft error in memory logic, SEFIs and permanent faults in controller logic can not be mitigated effectively through ECC. Instead, these should be mitigated at the system level, if this is possible. In Chapter 9, we show how this can be facilitated with commercial components. Otherwise, if no system-level mitigation is possible, the OBC remains prone to chip-level faults.

7.3.2 A Software-Driven Memory Consistency Concept

When utilizing ECC, memory consistency is only assured at access time, unless specialized self-checking RAM concepts are applied in hardware [256, 257]. Rarely used data and code residing within memory will over time accumulate errors without the OS being aware of this fact, unless scrubbing is performed regularly to detect and correct bit-errors before they can accumulate. The scrubbing frequency must be chosen based on the amount of memory attached to the OBC, the expected system load and the duration required for one full scrubbing-run [258]. Resource conserving scrubbing intervals for common memory sizes aboard nanosatellites range from several minutes up to an hour. Also, if a spacecraft were to pass through a region of space with elevated radiation levels (e.g., the SAA), scrubbing should be performed directly before and after passing through such regions.

As depicted in Figure 44, the DRAM integrity assurance measures usually realized in hardware in traditional space-grade components can be also be facilitated in software. We can construct a DRAM-integrity assurance regime using allocation-time memory testing, software-realized error scrubbing, and OS-side blacklisting of memory pages with defective blocks. All these elements can be realized in software using standard functionality, while a scrubbing tasks can be implemented within the OS's kernel, or even in userland. The specific implementation details therefore vary depending on what level this functionality is realized in.

Concept Overview

At a high level, this concept can be described as follows:

Bootup: During operating system bootup, the second stage bootloader or OS Kernel itself will execute platform bring-up code, may relocate the Kernel or RTOS code from storage into faster main memory. Subsequently, it will then prepare key OS data structure, and initialize core system functionality such as virtual memory, memory protection, a kernel console and logging, if available. All of these operations occur linearly, and require very little memory to be allocated. More memory intensive operations will occur past this point.

Blacklist: We add functionality to read a matrix of bad memory pages, where pages containing defective hard errors are marked. We can elegantly blacklist these memory pages by simply reserving them, thereby preventing them from being issued at a later stage. This is being done for performance and simplicity reasons, to avoid triggering ECC syndromes for known bad memory pages during operation, and performance costs. As the integrity of this bit-matrix is critical, it should reside in radiation-immune memory that does not suffer wear. Both FRAM and MRAM are viable technologies, due to small size of this memory, and simply redundancy for this memory can be realized as described in Chapter 9.

Operation: Once the bootup is completed, the Kernel will setup a suitable scrubbing task to periodically perform error scrubbing on main-memory associated memory regions. the OS will initialize flight software applications, and allocate memory for them.

Allocation: During operation of the flight software, whenever an application allocated memory, the OS will test the integrity of a memory page before issuing it to the consuming application. Should a memory page be discovered to be permanently defective, it will be left allocated but not issued. As we assume the availability of virtual memory, fragmentation of the memory map is a non issue. Memory allocation in most operating systems is an atomic operation, with interrupts being disabled during the operation. Hence, for the duration of memory allocation, no ECC syndromes will be

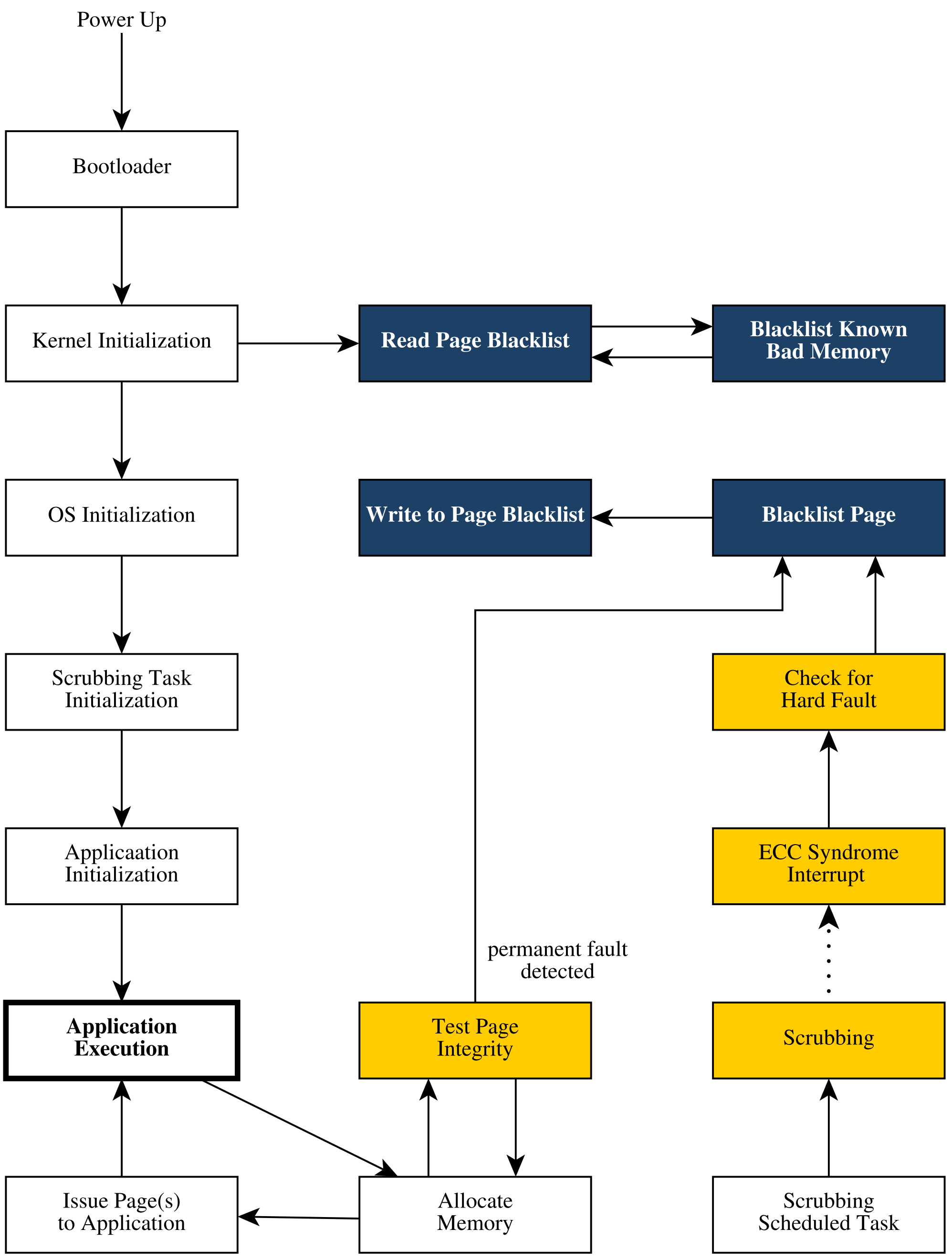


Figure 44: Integrity of volatile memory can be guaranteed if memory checking and ECC (yellow), as well as memory blacklisting (blue) are combined. Scrubbing must be performed periodically to avoid accumulating errors in rarely used code or data.

processed. At the end of allocation, in case an ECC syndrome interrupt is pending, syndromes for bad memory pages will be discarded.

Scrubbing: Periodically, the scrubbing task set up during OS initialization will read the entire DRAM address space, if hardware scrubbing is unavailable. This causes rarely accessed memory regions to be refreshed, preventing bit-upsets to accumulate there. The scrubbing application itself will not attempt to test if a page contains permanent faults, it just triggers ECC syndromes. It can be implemented in a variety of different ways, as described in Section 7.3.2.

Syndromes: We extend the functionality of the ECC syndrome handler, to not only determine if the ECC error was recoverable or not, and to respond to it in a suitable manner. Instead, we add functionality to test the relevant piece of memory to detect if the ECC error was caused by a soft or hard fault. In case of a hard fault, the relevant bit of the bad-memory matrix is flipped, and the page should no longer be used, as far as this is possible for already issued in-use memory. If desired, the syndrome handler can therefore consider ECC parameters in case multi-bit correcting ECC or Reed-Solomon block coding are used. Then, a minimum delta between hard errors in memory word and error correction capacity can be defined. This can help slow down the pace at which pages are discarded that contain faulty memory words.

Software-Implemented Scrubbing

In the case of a Linux Kernel and a GNU userland, a scrubbing task can most conveniently be implemented as a `cron`-job reading the OBC's physical memory. For this purpose, the device node */dev/mem* is offered by the Linux Kernel as a character device. */dev/mem* allows access to physical memory where scrubbing must begin at the device specific *SDRAM base address* to which the RAM is mapped. Technically, even common Unix programs like `dd(1)` could perform this task without requiring custom written application software.

Another possibility would be to implement a Linux kernel module using timers to perform the same task directly within kernel space. In this case, the scrubbing-module could also directly react to detected faults by manipulating page table mappings or initiating further checks to assure consistency. Execution within kernel mode would also increase scrubbing speed, allowing more precise and reliable timing.

Memory Checking and Blacklisting

Unless very strong multi-bit-error correcting ECC ($>$ 2 bit error correction) and scrubbing are utilized, ECC can not sufficiently protect a spacecraft's RAM due to in-word-collisions of soft- and hard errors as depicted in Figure 45. To avoid such collisions, memory words containing hard faults should no longer be utilized, as any further bit-flip would make the word non-recoverable [228]. Even when using multi-bit ECC, memory should be blacklisted in case of grouped permanent defects which may be induced due to radiation effects or manufacturing flaws as well.

Memory must also be validated upon allocation before being issued to a process. Validation can be implemented either in hardware or software, with the hardware variant offering superior testing performance over the software approach. However, memory testing in hardware requires complex logic and circuitry, whereas the software

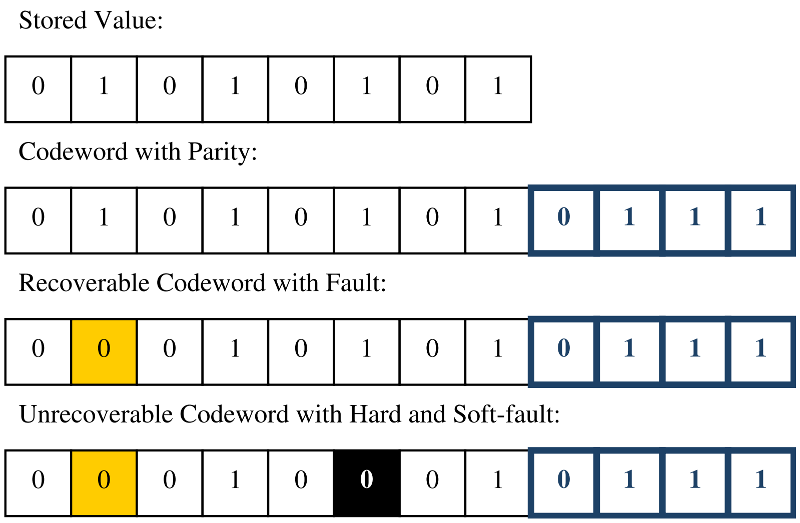


Figure 45: With single-bit correcting ECC-RAM, a word should no longer be used once a single hard-fault has been detected. Hard faults are depicted in black, soft faults in yellow, erasure code parity in green.

variant can be kept extremely simple. The Linux kernel offers the possibility to perform these steps within the memory management subsystem for newly allocated pages for ia32 processors already, and are currently porting this functionality to the ARMv7 MMU-code. In case the Linux kernel detects a fault in memory, the affected memory page is reserved, thereby blacklisted from future use, and another validated and healthy page is issued to the process. Therefore, we chose to rely upon this proven and much simpler software-side approach.

The ia32 implementation does not retain this list of blacklisted memory regions beyond a restart of the OS, though doing so is an important feature for use aboard a satellite. As memory checking takes place at a very low kernel-level (MMU code essentially works on registers directly and in part must be written in assembly), textual logging is impossible and persistent storage would have to be realized in hardware. An external logging facility implemented at this level would entail rather complex and thus slow and error prone logic, thus, a logging based implementation is infeasible. However, at this stage we can still utilize other functionality of the memory management subsystem to access directly mapped non-volatile RAM, in which we can retain this information beyond a reboot. Due to the small size required to store a page bitmap, it can be stored within a small dedicated FRAM/MRAM module, read by the bootloader and passed on to the kernel upon startup. This implementation can thus enable multi-bit-error correcting equivalent protection without requiring costly specialized hardware, while increasing system performance on strongly degraded systems.

7.4 A Radiation-Robust Filesystem for Space Use

The increased compute burden handled aboard modern nanosatellites also requires more sophisticated operating system (OS) software, which in turn results in increased code complexity and size [259].

For very simple computers, custom tailored OSs offer an excellent balance of size and functionality. However, development of proprietary OSs for unique custom computers has been abandoned in most of the IT industry, in favor of standard soft- and hardware reuse. This is still an ongoing process in spaceflight, though already producing a focus on a few types of radiation hardened processor platforms (e.g., LEON3, PPC750, RAD6000, see [260]) running common OSs [261, 262]. The same evolution has begun in nanosatellite computing, albeit much faster.

OSs popular in spaceflight such as RTEMS can consume less than 256KB of non-volatile (nv) memory [263], whereas Linux requires at least 2MB. If such a larger OS is used aboard a satellite, more sophisticated storage concepts are needed. Data must be stored permanently and consistently throughout the mission lifetime. Space missions often last between 5 and 10 years [264], but can reach 25 years or longer as discussed in Chapter 3. Thus a satellite's command and data handling (CDH), the on-board computer, must guarantee integrity and recover degraded or damaged data (error detection and correction – EDAC) over a prolonged period of time in a hostile environment. We consider a filesystem the most resource conserving and efficient approach, which also allows dynamically adjustable protection for the individual data structures. As Magnetoresistive Random-Access Memory (MRAM) [147] is widely used for radiation resistant data storage in nanosatellites, and therefore we developed FTRFS specifically for this technology.

7.4.1 Related Work and Preexisting File Systems

Filesystems often include performance optimizations such as disk head tracking, utilization of data locality and caching. However, most of these enhancements do not apply to storage technologies used in spaceflight. In fact, such optimizations add significant code overhead, possibly resulting in a more error prone filesystem and may even reduce performance.

Next-generation Filesystems, e.g., BTRFS, F2FS, and ZFS, are designed to handle many-terabyte sized devices and RAID-pools. Silent data corruption has become a practical issue with such large volumes [265]. Thus, these filesystems can maintain checksums for data blocks and metadata. Due to their intended use in large disk pools, they do also offer integrated multi-device functionality.

Multi-device functionality would certainly be advantageous, but neither ZFS nor BTRFS scale to small storage volumes. Minimum volume sizes are far beyond what current nanosatellite CDHs can offer. Technology scaling for the technologies strongly drives development of these file systems continuously towards larger volumes Hence, future development of these filesystems will require design decisions the conflict with the needs for spaceflight applications.

Filesystems for flash devices, similar to the memory technology itself, have evolved considerably over the past decade [266, 267]. Upcoming filesystems already handle challenges concerning potentially negative compression rates [268] or erase block abstraction, offer proper wear leveling and interact with device EDAC functionality (checksumming, spare handling and recovery). UFFS even offers integrity protection for data and metadata using erasure codes.

Most new flash-filesystems interact directly with memory[1], thereby are incompatible with other memory technologies unless flash properties are emulated. This introduces further IO and may result in unnecessary data loss, as flash memory is of course block oriented.

RAM filesystems are usually optimized for throughput or simplicity, often resulting in a relatively slim codebase. If designed for volatile RAM, these filesystem are optimized for simplicity and do not necessarily require a nondestructive unmount procedure. Non-volatile RAM filesystems access data in memory directly avoiding many of the indirection and abstraction layers required for more abstract memory technologies [269], while some even utilize in-line compression to increase storage capacity [269].

Except for *PRAMFS* [270], none of these filesystems consider memory protection to increase dependability. *PRAMFS* offers execute-in-place (XIP) support [271] and is POSIX-compatible, but offers no data integrity protection.

In contrast to flash memories RAM filesystems are not block based, but benefit from the ability to access data arbitrarily. Thereby, no intermediate block management is required and read-erase-update cycles are unnecessary. While simple block-layer EDAC would certainly be possible, structures within a RAM filesystem can be protected individually allowing for stronger protection.

Open source space engineering and CDH research is directed mainly towards testing radiation related properties of memory technologies [272, 273] and on NAND-flash in particular [274, 275]. At the time of this writing, we are unaware of advanced software-side non-flash driven storage concepts for space use.

7.4.2 FTRFS

We designed FTRFS as Fault-Tolerant Radiation-robust Filesystem for Space use. It is intended to operate efficiently with small volumes($\leq$4MB) and assure data integrity for critical firmware-related data stored within COTS MRAM components. To fulfill its purpose for storing a firmware image, it was designed to be bootable, and also to allow for the capacity to scale much to larger volumes than can be achieved with toggle-MRAM at the time of writing.

As base for this filesystem's fault model, we assume that computational correctness within the OBC itself can be assured. Furthermore, we assume that within the OBC, ECC is applied to CPU-caches and RAM so that upsets in in-transit data can be detected and mitigated before they are written to memory. A CPU running FTRFS must be equipped with a memory management unit with its page-table residing in ECC protected volatile memory. All other elements (e.g., periphery and ALUs), other memories (e.g., registers and buffers) and in-transit data are considered potential error sources.

Memory protection has been largely ignored in RAM-filesystem design. In part, this can be attributed to a misconception of memory protection as a pure security-measure against malware. However, for directly mapped nv-memory, memory protection introduces the memory management unit as a safeguard against data corruption due to upsets in the system [276]. Thus, only in-use memory pages will be writable

[1]in the case of Linux through the memory technology device subsystem (MTD)

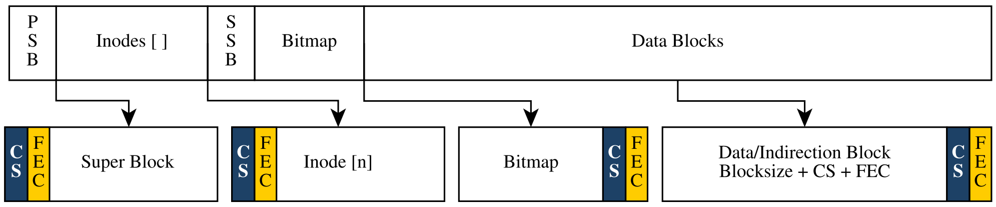


Figure 46: The basic layout of the presented filesystem. EDAC data is appended or prepended to each filesystem structure. PSB and SSB refer to the primary and secondary super blocks.

even from kernel space, whereas the vast majority of memory is kept read-only, protected from misdirected write access i.e. due to SEUs in a register used for addressing during a store operation.

FS-level data compression has been popular in size constrained filesystems. However, in our use case, well-compressible data, e.g., textual or binary log data, would reside in flash or PCM. Hence for a satellite's flight software firmware image will yield little gain, a and we therefore do not realize data compression as part of FTRFS, thereby allowing reduced code complexity and increasing performance.

After a detailed OS evaluation which was presented in [Fuchs12], we chose the Linux kernel as the base for our filesystem due to its adaptability, extensive soft/hardware support and vast community. We decided against utilizing RTEMS mainly due to our limited software development manpower. Further details on this evaluation including scoring data and a detailed description of the used criteria is available in [Fuchs12].

A loss of components has to be compensated at the software- or hardware level through voting or simple redundancy. Multi-device capability was considered for this filesystem, however it should rather be implemented below the filesystem level (e.g., via majority voting in hardware [277]) or as an overlay, e.g., RAIF [96].

The capability to detect and correct metadata and data errors was considered crucial during development. Based on the mission duration, destination or the orbit a spacecraft operates in, different levels of protection will be necessary. The protective guarantees offered can be adjusted at format time or later through the use of additional tools.

Due to the relatively restricted system resources aboard a nanosatellite, cryptographic checksums do not offer a significant benefit. Instead, CRC32 is utilized for performance reasons in tandem with Reed-Solomon encoding (RS) [229].

Metadata Integrity Protection

For proper protection at the filesystem level, in addition to the stored filesystem objects (inodes) and their data, all other metadata must be protected. Figure 46 depicts the basic layout. Although similar to *ext2* and *PRAMFS* [270], data addressing and bad block handling work fundamentally different. We adapt memory protection from the *wprotect* component of *PRAMFS*, as well as parts of the inode layout. *PRAMFS* is licensed under GPLv2 and based upon *ext2*.

The Super Block (SB) is kept redundantly, as depicted in Figure 46. An update to the SB always implies a refresh of the secondary SB, hence, hereafter no explicit reference of the secondary SB will be made. The SB also contains EDAC parameters

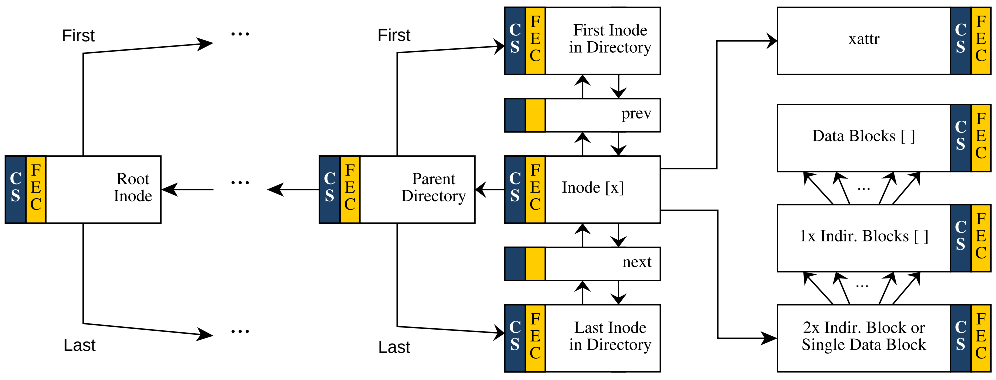

First
...
First
CS FEC First Inode in Directory
prev
CS FEC Root Inode
...
CS FEC Parent Directory
CS FEC Inode [x]
next
CS FEC Last Inode in Directory
Last
...
Last
xattr CS FEC
Data Blocks [ ] CS FEC
1x Indir. Blocks [ ] CS FEC
2x Indir. Block or Single Data Block CS FEC

Figure 47: Each inode can either utilize direct addressing or double indirection. Extended attributes are always addressed directly.

for blocks, inodes and the bitmap.

The SB is the most critical structure within our filesystem, and is static after volume creation. Its content is copied to system memory at mount time, thus it is sufficient to assure SB consistency the first time it is accessed.

As the SB contains critical filesystem information, we avoid accumulating errors over time through scrubbing. Thereby, the CRC checksum is re-evaluated each time certain filesystem API functions (e.g., directory traversal) are performed.

A block-usage bitmap is dynamically allocated based on the overhead subtracted data-block count and is appended to the secondary SB. The bitmap EDAC is also dynamically sized and must be stored beyond the compile-time static SB, even though placing it there would be convenient. Thus, the protection data is located in the first block after the end of the bitmap, see Figure 46. In case the bitmap is extended, the new part of the bitmap is initialized and then the error correction data is recomputed at its new location. We refrain from re-computing and re-checking the EDAC data upon each access, instead FEC data is checked before and updated after each relevant operation has been concluded.

Inodes are kept as an array. Their consistency is of paramount importance as they define the logical structure of the filesystem. The array's length is determined upon filesystem initialization and can change only if the volume is resized. As each inode is an independent entity, an inode-table wide EDAC is unnecessary. Instead, we extend and protect each inode individually.

Data Consistency and Organization

To optimize the filesystem towards both larger (e.g., a kernel image, a database) and very small (e.g., scripts) files, direct and double indirect data addressing are supported, as depicted in Figure 47. The filesystem selects automatically which method is used. Data protection requirements vary depending on block size, and use case. Thus FTRFS allows the user to adjust the protection strength for data blocks, as will be described in the next section.

Data block size cannot be arbitrarily decreased, as some Linux kernel subsystems assume them to be sized to a power of two. Instead, the filesystem internally utilizes larger blocks to include EDAC data, see Figure 48.

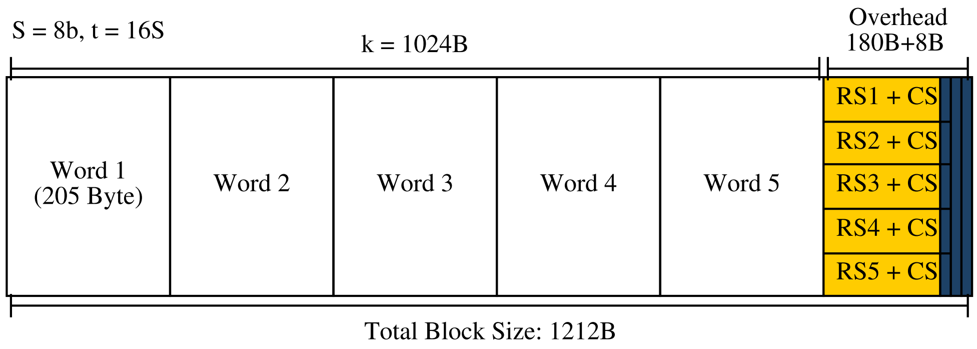


Figure 48: A data block subdivided into 5 subblocks. Separate checksums for the entire data block, EDAC data and each subblock are depicted in blue, which EDAC data is depicted in yellow.

Extended attributes (*xattr*) are deduplicated and referenced by one or more inodes, as depicted in Figure 47. Like in *PRAMFS*, *xattrs* are stored as data blocks, thereby we can treat these identically to regular data.

Nanosatellites, at least the non-classified ones, are not yet considered security critical devices. However, the application area of nanosatellites will expand considerably in the future [220]. An increasing professionalization will introduce enhanced requirements regarding dependability and security. Shared-satellite usage scenarios as well as technology testing satellites will certainly also require stronger security measures, which can be implemented using *xattrs*.

An *xattr* block's integrity is verified once its reference is resolved. Once all write access (in bulk) has been concluded, the EDAC data is updated.

Algorithm Details and Performance

Our primary design objective was to create a filesystem which could be used to store a full size-optimized Linux root FS including a kernel image safely over a long period of time within an 8MB volume. There are numerous erasure codes available that could be used to protect our filesystem, as discussed also by Wylie et al. in [102]. After careful consideration, RS was chosen due to the following reasons:

- The algorithm is well analyzed, and widely used in various embedded scenarios, including spacecraft.
- Highly optimized software implementations of RS encoder and decoder are available as part of standard libraries free of charge and are present in the Linux kernel.
- Open-source and commercial IP-cores are available to achieve hardware accelerations in an FPGA-based system, e.g. from opencores, from Xilinx, and via GRLIB.
- MRAM, while being SEU immune, is still prone to stray-writes, controller errors and in-transit data corruption. Misdirected access within a page evades memory protection and can then corrupt the filesystem, thus corrupted single-byte, 2, 4

and 8B runs will occur. RS relies upon symbol level error correction and can support symbols longer than 8 bit to then allow much larger codewords. This covers well the practical effects of faults will induce in commercial MRAM ICs.

RS decoding is computationally expensive, thus we split protected data into subblocks sized to 128B plus the user specified error number of correction-roots simplifying addressing and guaranteeing data alignment for power-of-two correction-root counts. Inodes and SBs can be fit into one single RS-code, while data block length does not result in extreme checking times. To skip the expensive RS decoding step during regular operation, a CRC32 checksum allows high-performance checking. The RS-code is only read in case the checksum is invalid.

Data blocks are divided into subblocks so the filesystem can make optimal use of the RS code length. For common block-sizes and error correction strengths, 5 to 19 RS codes are necessary, see Table 4 for information on expected overhead. The correction data is accumulated at the end of the data block. Checksums across the entire block's data, each subblock and the error correction data are also retained. The resulting data format is depicted in Figure 48. Protection can be enhanced further by performing symbol interleaving for the RS codes and the block data, at the cost of performance.

Filesystem traversal and data access will eventually slow down for strongly degraded storage volumes. As we immediately commit corrected data to memory, performance degradation is only temporary, assuming soft-faults.

Results and Current Status

FTRFS has been implemented for the Linux kernel. Due to its POSIX-compliance, it could easily be ported to other platforms. The memory protection functionality has been inherited from *PRAMFS*, the filesystem structure from *ext2*. We utilize the RS implementation of the Linux kernel, as its API also supports hardware acceleration.

Several components of the filesystem should undergo an optimization process, which will increase fault coverage capacity and read/write performance. Even though we have not yet conducted long-term benchmarking and performance analysis, the throughput degradation during regular operations is minimal: most modern mobile-market CPU cores can compute CRC32 within a few clock cycles due to hardware acceleration. We intend to publish additional performance and energy consumption

Data Structure	Size (B)	EC-Symbols per Word	Words per Block	Parity (B)	Overhead (B)	Overhead (%)
Super Block	128	32	1	32	68	53.13%
Inode	160	32	1	32	68	42.50%
Data Blocks	1024	4	5	20	68	5.86%
	1024	16	5	80	188	17.58%
	4096	4	17	68	212	4.98%
	4096	16	19	304	692	16.70%
Bitmap	1773	32	10	320	688	38.80%

Table 4: EDAC overhead for FS structures. 16MB volume size, 5% inodes, 1024B bock size

metrics, once testing has been concluded and basic optimizations have been applied and the OBC computer has been finalized.

Data is read and written once per access. It is good practice in critical scenarios and especially spaceflight to read and write data multiple times, or deploy more advanced consistency checking techniques [278]. These changes could be applied in bulk, through a macro, or compiler side.

The level of protection offered by FTRFS is adjustable during volume creation, or later by using a proprietary filesystem-tuning tool. RS has a long record of space use in CDH and communications. Thus, we know the algorithm offers efficient protection regarding our threat scenario. Once testing has been concluded, we will perform long-term performance analysis in a degraded environment. To benchmark the filesystem, data degradation can be introduced through fault injection.

Limitations and Advanced Applications

It is debatable whether journaling would increase FTRFS's reliability, as it usually helps safeguard filesystem consistency with slow storage media [279] due to power loss or disconnect. Spontaneous power loss for an OBC could also occur aboard a spacecraft due to EPS malfunction, but in most cases the practical effects of such an event can be handled differently at the design side. Spacecraft are battery backed and can utilize power electronics with a sufficient hold-back time to notify and gracefully shut down an OBC in case of EPS failure. All access in our filesystem happens synchronously, and MRAM still allows rapid access unlike classical mechanical disks. Hence, FTRFS can thus either conclude a pending write operation within the remaining active time, or the OS will have sufficient time to cancel pending writes in case the system has sufficient warning time. We therefore do not implement journaling.

Our filesystem implementation can currently not handle the failure of entire memory ICs holding the volume, or component-level SEFIs. However, FTRFS could be extended to support RAID-like features to compensate for device failure [277].

If data is stored with RS-symbol interleaving, an XIP mapping would technically be impossible. XIP could still perform mappings for non-interleaved data though, but thereby only the clear-text part of each RS code would be mapped and read. Via this memory mapping, integrity protection for stored file data would be ignored, unless we accept that a potential XIP mapping would allow program code to be loaded/executed without any integrity checking. Thereby, the integrity assumptions upon which FTRFS's concept is based would be violated and integrity could not be guaranteed for any executed program stored on the filesystem. Theoretically, data integrity could also be checked each time a mapping is established for a block. To perform these checks however, this data would have to be read in full, obsoleting the performance advantage and RAM conserving properties of XIP. XIP and filesystem-level data integrity protection can thus be considered mutually exclusive.

Permanent faults would cause fault effects to be corrected upon every access to a memory word, which is inefficient Fault in frequently accessed file system components (e.g., int the root inode), could therefore degrade the performance of FTRFS. In the current filesystem implementation, there is no functionality to avoid this behavior completely. Bad-block relocation is implemented within the filesystem, but only applied during file data write, truncate and allocation operations. This functionality could also be applied to file data read operations as well as for accessed inodes to increase robustness.

FTRFS could also operate on different memory technologies than MRAM, as long as data in this memory is directly addressable RAM or mapped through OS-kernel means (mmap). For more abstract memory technologies such as Flash, FTRFS is not an optimal solution and a block-based approach as described in the next section should be used.

7.5 High-Performance Flash Memory Integrity

Scientific and future commercial space missions as well as miniaturized satellites impose increasing demands on their on-board computer (OBC) systems, especially data storage devices [280]. They may require vast amounts of data to be stored, high throughput, and the possibility for concurrent access of multiple threads, programs or devices. While satisfying these requirements, storage systems must guarantee data integrity and the recovery of degraded or damaged data (error detection and correction – EDAC) over a prolonged period of time in a hostile environment. Consistent data storage becomes even more crucial for long-term missions (e.g., JUICE [281] and Euclid [282]) or in cases where highly scaled memory is used.

Legacy memory technologies can not be scaled for modern storage applications due to mass and energy restrictions or result in high complex storage systems. Thus, single-level cell NAND-flash memories (SLC), have become popular for high performance mass memory scenarios as they offer reasonably high packing density, and can be manufactured sufficiently radiation hardened. The chip-industry has moved on from SLC to multi-level cell flash memories (MLC) due to economical reasons. Therefore, SLC will become unavailable and will force future spacecraft storage concepts to rely upon MLC or entirely different memory technologies. While there are promising candidates [283] to fill this role in the long run, technological evolution does not yet allow, for example, non-volatile magnetoresistive RAM (MRAM) to be used as mass storage [272]. Phase change [284] or charge-trap based memory both would at present be usable as mass storage, but are not yet widely available in high density versions [157].

Traditionally, single-bit error correction, shielding, specialized manufacturing techniques, coarse structure width and redundancy are combined to enable radiation tolerant flash [285]. However, the protective level offered by such solutions is static and fixed at design-time and can result in high cost and low overall efficiency. For miniaturized satellites, cost and efficiency are crucial, thus, countermeasures must be implemented at a different level. With modern MLC-flash single bit error correction is insufficient and all-in-one solutions, such as file systems, tend to become very complex and difficult to debug. For future prolonged missions and larger storage arrays, more sophisticated and efficient EDAC concepts are required. Thus, we present an advanced high performance dependable storage concept based on composite erasure coding. As MLC-flash is also widely used aboard miniaturized satellites, and the authors are involved in developing such a satellite, MOVE-II, development was originally driven by nanosatellite requirements. However, the approach can be applied more efficiently to commercial applications where miniaturization imposed limitations do not apply. The concept could be implemented even more efficiently with very large volumes common in commercial spaceflight applications. It can be implemented entirely in software, with or without hardware acceleration, but also partially or fully in hardware.

Single- and Multi-Level Cell Flash

Each flash memory cell contains a single field effect transistor with an additional floating gate, the basic functionality of which is described further in Chapter 3. The state of a flash memory cell depends on whether the charge stored in the floating gate exceeds a specific threshold voltage (V_t). Hence, a flash memory cell is dependent on the capability of the memory cell structure to retain a charge. If the voltage exceeds the threshold, a cell can be read as programmed (0), else as erased (1), see Figure 49a. Single Level Cell Flash (SLC) cells can store one bit per cell.

The charge in an MLC cell can represent more than two states by introducing additional voltage thresholds. Assuming a four level cell, one can hold four states and represent two bits, as depicted in Figure 49b. The number of levels is not restricted to four, with 2^n states it is possible to encode n bits, but electrical complexity grows and the required read sensitivity and write specificity increase with the number of bits represented. Within nearly the same area of silicon, MLC flash memory thus allows a much higher packing density and the structure itself can be stacked and scaled well [286].

As the delta between voltage thresholds decreases due an increased number of state-levels, increased sensing accuracy is required for read operations, and more precise charge-placement on the floating gate is necessary. MLC memory is thus more dependent on its cells' ability to retain charge. In contrast to SLC, a state machine is required for addressing MLC memory which in turn increases latency and adds considerable overhead logic. Addressing in MLC flash can thus take multiple cycles and the state machine may hang or introduce arbitrary delays.

Due to a shifting voltage threshold in floating gate cells caused by the total ionizing dose, MLC flash memories are more susceptible to bit errors than SLC [153, 287]. Depending on the number of bits represented within a cell, a varying amount of data may thereby be corrupted by a single particle event, as depicted in see Figure 50. EDAC measures must thus compensate for more than single bit corruptions within a given word. Thus software or a filesystem must implement appropriate functionality to handle these effects in addition to erasure coding to safeguard from radiation.

Flash Memory Organization

NAND-flash memories are organized in blocks, consisting of multiple pages, in which cells are connected as NAND gates. In most NAND technologies, pages can be written and read individually, but only the block as a whole can be erased. The drawback

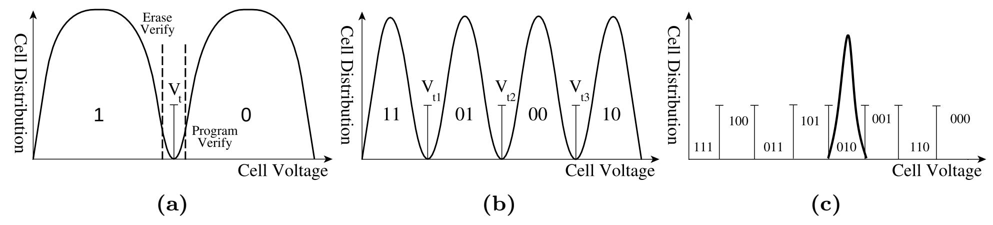


Figure 49: The voltage reference and threshold levels of SLC- flash cells (a) and MLC cells with 4 (b) and 8 voltage levels (c).

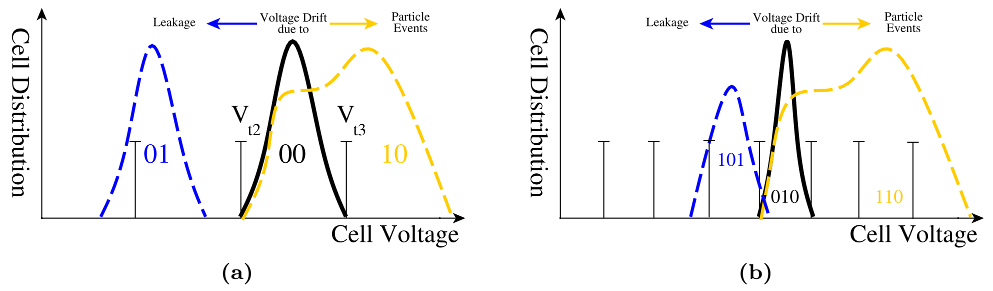


Figure 50: Radiation-induced bit upsets encountered in 4- (a) and 8-level (b) MLC cells.

here is that if a NAND-flash cell fails, the entire NAND block is affected. In NOR-flash, cells form NOR gates, which allow more fine grained read access at the cost of strongly increased wiring and controller overhead. Therefore, in order to appropriately handle NAND-flash block corruption, a filesystem must handle read/write and erase abstraction, as well as basic block FDIR. This is done through the introduction of an additional layer of functionality, the flash translation layer (FTL). When data is written to a flash block, partial erase operations are (usually) impossible and the entire block's previous content first has to be read and updated. Next, the block must be erased (by draining the block's cells' voltage) and may subsequently be programmed anew per page. Thus, read and write operations introduce different latency and make access to MLC flash much more complicated than to SLC due to the required addressing state machine.

To access data and handle special properties of flash efficiently, a filesystem has to interact with the memory device directly or via the OS's FTL. A flash filesystem must implement all functionality necessary to perform block wear leveling, read and erase block abstraction, bad-block relocation and garbage collection (depicted in blue in Figure 51) to prevent premature degradation and failure of a bank. The FTL acts as an interface between hardware specific device drivers and the filesystem, and can provide part of this FDIR functionality instead of the filesystem. In commercial SSD applications, this is handled by the SSD's controller and hidden from the OBC.

Over time, a flash memory bank will accumulate defective pages and blocks hve to and utilize spare pages and blocks to compensate. Traditionally, simple erasure coding (usually some form of cyclic block codes with large symbol sizes) is applied in software or by the controller to counter wear and charge leakage. Eventually, the pool of spares will be depleted, in which case the FTL or filesystem will begin recycling less defective blocks and compensate with erasure coding only, thereby sacrificing performance to a certain degree. For space use, the erasure codes' symbol size is usually reduced to support one or two bit correcting erasure coding, as corruption will mostly result from radiation effects. However, if this solution is applied for MLC-NAND-flash, block EDAC becomes very inefficient due to the occurrence of both single bit- and grouped errors, the latter being induced by SEUs affecting multiple cells in highly scaled memory.

Majority Voting for Flash Memories

While voting is technically still possible for MLC-flash, it is severely constrained by the additional circuitry, logic and strongly varying timing behavior. Voting would have to be implemented for the addressing state machine as well, otherwise it could stall the entire voting circuit or permanently disable its memory bank. Due to the varying timing behavior of NAND-flash and the more complex logic, the resulting voter-circuit thus becomes more error prone. The added logic also requires more energy and reduces overall performance. Of course the slowest memory bank or block also dictates performance of the voting circuit.

7.5.1 The MTD-mirror Middleware Layer

As outlined in the previous sections, error correction is crucial for current data storage based on NAND-flash. To enable future dependable MLC-NAND-flash based data storage solutions for space flight applications, existing EDAC functionality can be adapted and improvements added where necessary. Thus, we developed a storage system to satisfy the following requirements:

1. Efficient, fast data storage on MLC mass-memory.
2. Integrity protection and error correction with adjustable strength, to allow optimization according to mission duration, environment and type.
3. Efficient handling of direct and indirect radiation effects on the memory as well as the control logic.
4. Protection against device failure.

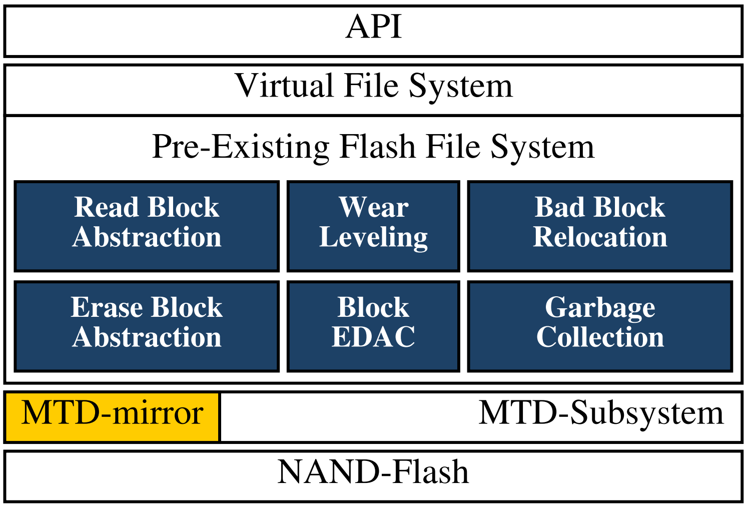


Figure 51: Memory access hierarchy for an MTD-Mirror set. Flash-memory specific logic is depicted in blue and partially resides within the FTL. Required modifications to enable the concept are depicted in yellow.

5. Low soft- and hardware complexity: While a certain level of complexity is acceptable for commercial spaceflight applications, it is crucial in microsatellite design.

6. Universal filesystem support and interactivity.

We consider these requirements to be met best through enhanced EDAC functionality as FTL-middleware. At this level, RAID-like features and checksumming can be combined most effectively with a composite erasure coding system. As our use case includes a Linux based OBC, we implemented MTD-mirror on the memory technology device (MTD) FTL subsystem of the Linux Kernel. The solution is depicted in Figure 51. Any unmodified flash-aware filesystem can be deployed on top of the MTD-mirror set. By utilizing mirroring (RAID1) and distributed parity (RAID5/6) we can therefore protect against device, bank and block failure. Within this section we focus on mirroring, as the basic concept is very similar to distributed parity sets.

To safeguard against permanent block defects, single event functional interrupts, radiation induced programmatic errors and logic related problems, we apply coarse symbol level erasure coding. As this is insufficient to compensate for radiation effects, silent data corruption and bit flips are compensated using bit-wise error correction. The solution was implemented in the FTL, as the required logic can still be kept abstract and device independent while it can profit significantly from hardware acceleration. The FTL-middleware also provides enhanced diagnostics, as no further abstraction is introduced.

Alternative Approaches

EDAC and device independence could also be provided by an filesystem directly, which we showed for MRAM with FTRFS in Section 7.4. A Flash filesystem such as UFFS could be extended to handle multiple memory devices and EDAC, or FTRFS could be modified to handle flash memory. Even though possible to implement, such an all-in-one filesystem would be complex and error prone.

Device independence could also be added on top of an existing flash filesystem as a separate layer of software [96], see Figure 52. Within a RAIF set, increased protective requirements could be satisfied with additional redundant copies of the filesystem content. The underlying individual filesystems would then have to handle all EDAC functionality and escalate fatal errors and unrecoverable file issues to the set, as RAIF by itself does not offer any integrity guarantees beyond filesystem or file failure.

Since RAIF only reads from underlying filesystems, it is prone to filesystem-metadata corruption which can result in single block errors failing entire filesystems. Additionally, Flash-filesystems usually rely upon parameter-fixed block based error correction and do not offer configurable protection for different filesystem structures, which is at best sub-optimal for space use.

A file damaged in different locations across the set's filesystems would become unrecoverable as RAIF would discard information regarding the location of damage to a file and in the best case would forward a defective copy to the application. It would therefore inhibit error correction and may even cripple recovery of larger files. While RAIF could be adapted to handle these issues, the resulting storage architecture would again become very complex, difficult to validate and debug. As RAIF implements filesystem redundancy, its storage efficiency will furthermore be inferior to distributed parity concepts such as the more advanced variants of the presented concept. As a pure software layer without the possibility to interact with the devices, hardware acceleration of RAIF would be impossible.

Device Failure and SEFI Protection

In contrast to RAIF, RAID can been applied efficiently to storage architectures and has been used previously aboard spacecraft (e.g., in the GAIA mission) [288]. However, these were based on SLC (see Section 7.5) and only relied on RAID to achieve device failover through data mirroring (RAID1) and distributed parity (RAID5/6) [288, 289]. As RAID itself does not offer any integrity guarantees beyond protection against read device failure, designs usually rely upon the block level hardware error correction provided by the flash memory or controller or implement simple parity only.

The main issue encountered with plain RAID setups is the absence of validation for a block or group of blocks. RAID merely retains redundant copies of data – parity – which can be used to restore lost data. RAID foresees that a data block is either unrecoverably lost (signaled by a read error) or fully intact; it is thereby prone to silent data corruption encountered in flash memory [72]. As the basic RAID concepts do not utilize checksumming to verify integrity, corrupted data will be read and used even if sufficient parity or valid copies were available. However, once checksumming and forward error correction is added to RAID levels, they can be utilized aboard spacecraft efficiently.

The even distribution of bit-errors would be troublesome for symbol based erasure codes traditionally applied for flash block EDAC. Utilizing RAID on top of block based erasure coding is thus insufficient for protecting MLC-NAND-flash.

RAID functionality usually would be implemented as a block layer. This is certainly

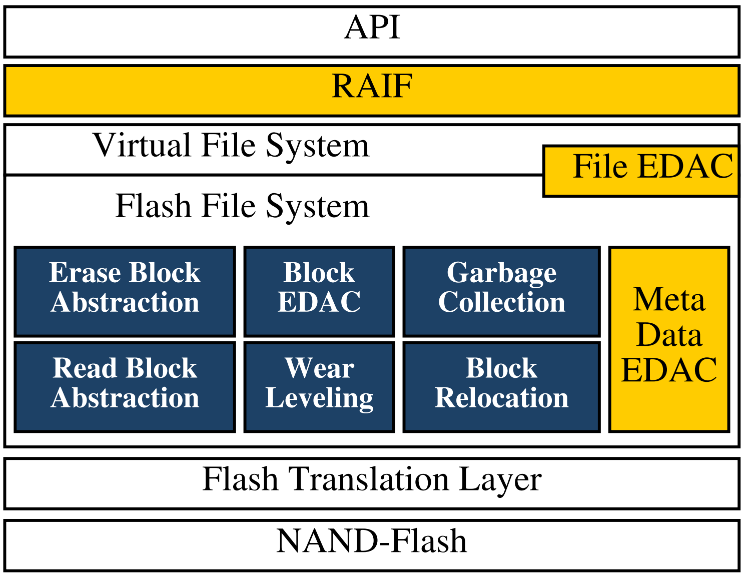


Figure 52: Memory access hierarchy for an enhanced RAIF based concept with added filesystem level error correction.

possible also for flash memory, however it would hinder the file system from performing block EDAC and wear leveling. While block abstraction would still be possible even on top of a block layer on-top of the FTL, other high-level filesystem functionality would be denied device access, depicted in red in Figure 53. These functions would then have to be implemented at a much lower level within the access hierarchy, introducing further code overhead and reducing EDAC efficiency.

RAID-like functionality could however also be implemented as a middleware within the FTL as depicted in Figure 51. As such, it can interact both with the underlying flash memory as well as the filesystem and the rest of the FTL, without requiring alterations to either. Such middleware can remain pervious to filesystem operations requiring direct interactivity with the underlying flash and at the same time allow device failure protection to be combined with enhanced erasure coding. RAID can therein be implemented with comparably little effort. Validation, testing and analysis can thus be simplified as all implementation work can be concentrated into an FTL middleware module.

Block-Level Consistency

MTD-mirror's block consistency protection is depicted in Figure 54 and includes two checksums and error encoding layers. Thus, it implements a concatenated/composite

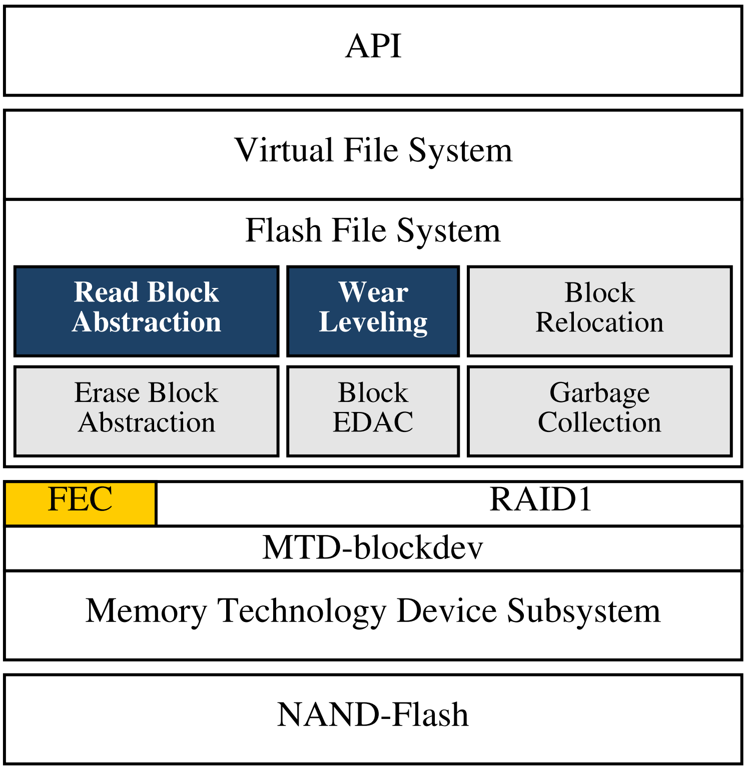


Figure 53: RAID prevents EDAC and wear leveling functionality withing a flash-filesystem from being implemented. Affected elements are colorized in gray.

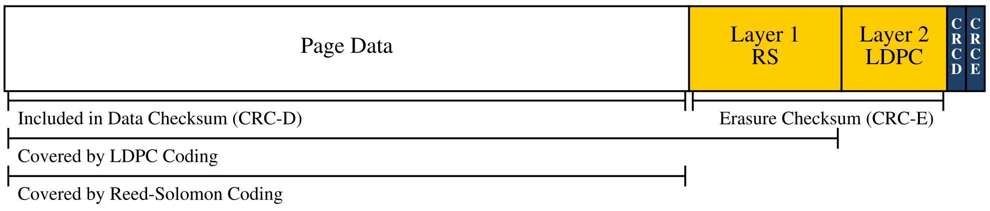


Figure 54: The layout of an MTD-mirror page. Added erasure code correction information is depicted in yellow, checksums in blue.

erasure code system. The data checksum allows bypassing decoding of intact data, which will often be the optimistic default case. The second checksum can be used for error-scrubbing of erasure data and prevents symbol drift of the RS-layer. Even though CRC16 could be considered sufficient for most common page and block sizes, we utilize a 32-bit checksum to further minimize collision probability at a minimal compute overhead.

Protection against Multi-Bit Upsets

The first layer of erasure coding is based on relatively coarse symbols and protects against data corruption induced by stray writes, controller issues and multi-bit errors. As data on NAND-flash is stored in pages and blocks of fixed length and the coding layer should protect against corruption up to 8 byte length (`int64_t`), Reed-Solomon (RS) erasure coding [103] was selected. We chose to rely on the RS block code as the algorithm is well analyzed, and widely used with NAND-flash memory and in various embedded scenarios, including spacecraft. Optimized software implementations, IP-cores and hardware acceleration are available.

Erasure coding with coarse symbols is efficient if symbols are largely or entirely corrupted, but shows weak performance when compensating radiation-induced bit-rot, to which MLC is comparably prone. SEUs will be evenly distributed across the memory and will thus equally degrade all data of a code word, corrupting multiple code symbols with comparably few bit errors. Therefore, RS is applied at the page level, instead of the block level to allow more efficient reads and avoid access to other pages within the same block to retrieve erasure coding parity. RS parity is therefore stored within each page, together with a checksum for the page and the parity. RS encoding and decoding can be should parallelized due to the small word sizes in hardware.

Bit-Level Erasure Coding

Previous radiation-tolerant OBC storage concepts often relied upon convolution codes as these allow efficient single-bit error correction. However, as error-models become more complex (2-bit errors as in MLC), codes complexity increases and efficiency diminishes. Therefore, a second level of erasure coding using Low-Density Parity Check Codes (LDPC) [290] was added to counter single or double bit-flips within individual code symbols of the first level RS code. LDPC was chosen as it is efficient with very small symbol sizes (1 or two bit), offers superior performance compared to convolution codes [291], and allows iterative decoding [292]. Only if RS decoding fails, the set resorts to LDPC. LCPC can then support recovery of slightly corrupted

RS-symbols and parity. Thereby otherwise unrecoverable data can be repaired by salvaging damaged symbols which can drastically increase recovery rates on radiation-degraded memories.

Although LDPC codes benefit from longer code word lengths, Morita et al. [293] show that the gain from a 4KB code to a 32KB code can be negligible. For systems where buffer memory is scarce, it may therefore be of advantage to use comparably small codes and sacrifice a bit of LDPC performance. Thus, an LDPC word size between 3 and 4KB offers solid LDPC performance without requiring very large words and thereby enable fast iterative decoding.

Joint Iterative Decoding using Soft-Output Shorter code words also enable joint iterative decoding [294] using soft-output for both LDPC and RS codes. LDPC can be adjusted to output not only plain copy of the expected original code word, but can also yield the decoder's certainty about each bit's value. Equally, an RS decoder could be extended to handle such soft-input. Then, it could attempt decoding to decode multiple variants of a corrupted word using different uncertain positional values from the LDPC soft-output.

In practice, this allows us to produce linear composite erasure coding system, which we depict in Figure 55. However, decoding does not have to happen linearly: A hardware-implemented LDPC decoder has a considerable logic footprint, while RS decoding can be parallelized. Hence, it may be desirable to construct such a composite system by paralellizing RS decoding.

As depicted in Figure 55, a closed feedback-loop that inputs the soft message output $R(Y)$ of the LDPC decoder into RS can be constructed. The system iterates between RS and LDPC decoding until either decoder can reconstruct a valid code word. To tackle the issue of the thereby variable timing behavior, the number of iterations can be limited or a timeout can be defined.

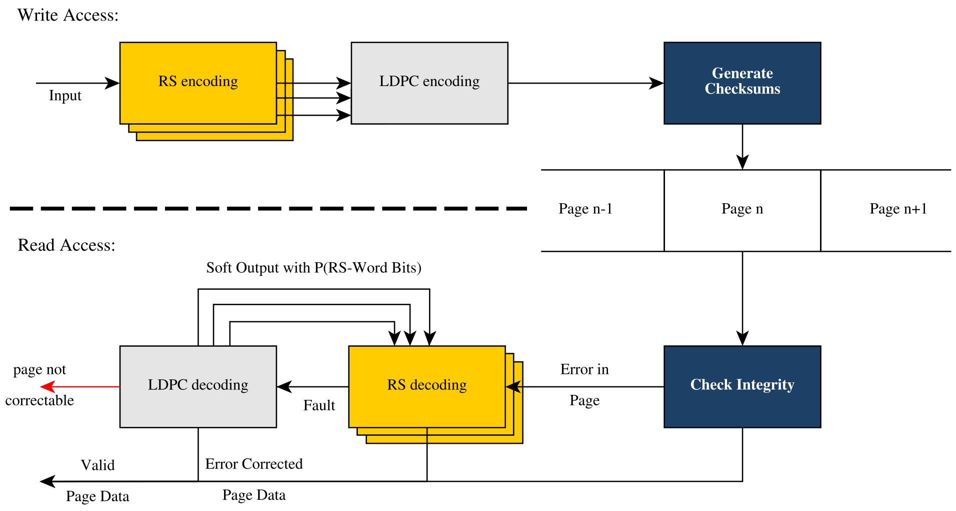


Figure 55: Joint iterative decoding using LDPC soft-output with added parallelization (triple-arrows). RS encoding can be parallelized to increase write-throughput. Speculative RS-decoding could be utilized to reduce LDPC iterations by performing multiple parallel RS-decoding attempts with different values for low-certainty bits.

Error Handling Runtime Behavior In case the checksum does not match the plain block data, an MTD-mirror set will first attempt to retrieve an intact copy of the data from another memory of the RAID-set. If this fails, or all other blocks are invalid as well, erasure decoding for the damaged block is attempted. As multiple copies of the erasure code parity data and checksums are available, the set can also attempt repair using fields of different blocks in the hope of obtaining a consistent combination of block-data. This behavior can allow recovery even of strongly degraded data or permanently defective blocks.

As RS hardware-acceleration is readily available in our use-case, we apply the two FEC layers in order (Figure 54). However, the sequence can be chosen based on the individual system design, the used algorithmic parameters, the available acceleration possibilities and phase of the mission. An important aspect for this decision is the expected level of degradation of the utilized flash memory due to radiation, thus the occurrence of single bit errors. If severe bit-rot is expected or higher order density MLC is used, the LDPC-layer should be applied prior to RS decoding. Thereby, the increased probability of the second FEC layer failing to recover data is accepted in the hope of achieving a sufficiently high amount of intact code symbols.

7.5.2 Advanced Applications

In this section, we focused on describing a storage solution based on RAID1 for simplicity reasons. While the logic required to implement this storage solution is relatively simple, more advanced distributed parity RAID concepts offer increased mass/cost/energy efficiency due to overhead reduction. Thus, we have been working to adapt and expand MTD-mirror to benefit from such more advanced architectures.

There has been prior research on adding checksumming support to RAID5 in [288, 289], though utilizing RAID5 directly would introduce certain problematic aspects. Error correction information in RAID5C can either be stored redundantly with each block, introducing unnecessary overhead, or as single copy within the parity-block. While this would increase the net storage capacity, a single point of failure would be introduced for each block group. If the parity block was lost, the integrity of data which was protected by this block could no longer be verified. Instead, RAID5 can be applied to data and error correction information independently, only requiring one extra checksum to be stored with each block.

RAID6, however, can be implemented almost as-is, with error correction data and checksums being stored directly on the two or more parity blocks associated with each group. There are also promising concepts for utilizing erasure coding for generating parity blocks by themselves, thereby obsoleting simple hamming-distance based parity coding [97,295]. Further research on this topic is required and may enable optimization for flash memory and radiation aspects similar to the ones described in this paper.

7.6 Conclusions

In this chapter we presented three software-driven concepts to assure storage consistency, each specifically designed towards protecting key OBC components: a system for volatile memory protection, FTRFS to protect firmware or OS images and MTD-mirror to safeguard payload data. All outlined solutions can be applied to different OBC designs and do not require the OBC to be specifically designed for them. They can be used universally in miniaturized satellite architectures for both long and short-term missions, thereby laying the foundation to fault tolerance at the system level. In contrast to earlier concepts, none of the approaches requires or enforces design-time fixed protection parameters. Both can be implemented either completely in software, or as hardware accelerated hybrids. The protective guarantees offered are fully run-time configurable.

Assuring integrity of core system storage up to a size of several gigabytes, FTRFS enables a software-side protective scheme against data degradation. Thereby, we have demonstrated the feasibility of a simple bootable, POSIX-compatible filesystem which can efficiently protect a full OS image. The MTD-mirror middleware enables reliable high-performance MLC-NAND-flash usage with a minimal set of software and logic. MTD-mirror is independent of the particular memory devices and can be entirely based on nanosatellite-compatible flash chips by utilizing FEC enabled RAID1 and checksumming.

Neither traditional hardware nor pure software measures individually can guarantee sufficiently strong system consistency for long-term missions. Traditionally, stronger EDAC and component-redundancy are used to compensate for radiation effects in space systems, which does not scale for complex systems and results in increased energy consumption. While redundancy and hardware-side voting can protect well from device failure, data integrity protection is difficult at this level. A combination of hardware and software measures, as outlined in this chapter, thus can increase robustness, especially for missions with a very long duration. Thereby, a low-complexity satellite architecture can be maintained, thereby error sources reduced, while testability and throughput can be increased.