Enabling Dependable Data Storage for Miniaturized Satellites

Christian M. Fuchs*† Advisors: Martin Langer* and Carsten Trinitis†
*Institute for Astronautics †Chair for Computer Architecture and Organization
Technical University Munich, Boltzmannstr. 3/15, 85748 Garching, Germany
{christian.fuchs, carsten.trinitis, martin.langer}@tum.de

Abstract

We present storage integrity concepts developed for the CubeSat MOVE-II over the past two years, enabling dependable computing without relying solely upon hardened special purpose hardware. Neither component level, nor hardware- or software-side measures individually can guarantee sufficient system consistency with modern highly scaled components. Instead, a combination of hardware and software measures can drastically increase system dependability, even for missions with a very long duration. Dependability in the most basic sense can only be assured if program code and required supplementary data can be stored consistently and reliably aboard a spacecraft. Thus, to enable any form of meaningful dependable computing, storage integrity must be assured first and foremost. We present three software-driven concepts to assure storage consistency, each specifically designed towards protecting key components: a system for volatile memory protection, the filesystem FTRFS to protect system software, and MTD-mirror to safeguard payload data. All described solutions can be applied to on-board computer design in general and do not require systems to be specifically designed for them. Hence, simplicity can be maintained, error sources minimized, testability enhanced, and survival rates of miniaturized satellite increased drastically.

I. 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 dependability 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 [1]) 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 achieving dependability of these components, especially radiation-hardening, can also drastically 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 dependability requirements.

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 drastically increase system dependability.

The authors are involved in developing the nanosatellite MOVE-II based upon an ARM-Cortex processor as a platform for scientific payloads. Hence, we designed MOVE-II's on-board computer (OBC) to guarantee data integrity using software side measures and affordable standard hardware where necessary, as traditional approaches for achieving dependability do not suffice for such a system. 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.

Often, dependability 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 dependable computing and provide a view on the topic at the grand scale. 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 III. MOVE-II will utilize magneto-resistive random access memory (MRAM) [2] as firmware storage, hence, we developed a POSIX-compatible filesystem (FS) offering memory protection, checksumming and forward error correction. This FS is being presented in Section IV, 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 V.

II. Memory Integrity as a Base for Dependability

The increasing professionalization, prolonged mission durations, 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 sophistication has continuously increased, re-usability, dependability, and reliability remained quite low [3]. Recent studies of all previously launched CubeSats show an overall launch success rate of only 40% [4]. Such low reliability rates are unacceptable for missions with more refined or long-term objectives, especially with commercial interests involved. As nanosatellites mostly consist of electronics, connected to and controlled by the OBC, achieving an elevated level of system dependability must begin with this component.

Besides extreme temperature variations and the absence of atmosphere for heat dissipation, the impact of the near-Earth radiation environment are crucial in space computing and system design. About 20% of all detected anomalies aboard satellites can be attributed to high-energy particles from the various sources, also in OBC-related components [5], [3]. Depending on the orbit of the spacecraft and the occurrence of solar particle events, its on-board computer will be penetrated by a mixture of high-energy protons, electrons and heavy ions. Physical shielding using aluminum or other material can reduce certain radiation effects. However, sufficient protection would require unreasonable additional mass for shielding.

In Low Earth Orbit (LEO), the radiation bombardment will be increased while transiting the South Atlantic Anomaly (SAA) [6]. Earth's magnetic field experiences a local, height-dependent dip within the SAA, due to an offset of the spin axis from the magnetic axis. In this zone, a satellite and its electronics will experience an increase of proton flux of up to 10^4 times (energies > 30 MeV) [7]. This flux increase results in a rapid growth of bit errors and other upsets in a satellite's OBC. In the case of MOVE-II, the full functionality of the command-and-data-handling subsystem, thus also its OBC, is required at all time due to scientific measurements being conducted from one of the satellite's possible payloads, even though planned maintenance outages (e.g. reboots) are acceptable. This scientific payload should measure the anti-proton flux within the SAA, whose properties are subject of scientific debate.

Currently, dependable computing on satellites is based mainly upon radiation tolerant special purpose hardware, as the cost of space electronics and software is usually dwarfed by a satellite's launch, testing and validation costs. Such components usually are significantly more expensive than commercial off-the-shelf (COTS) hardware. In part, this can be attributed to a thorough selection process performed for such components, but pricing is designed for aerospace and spaceflight applications equipped with vast budgets for long-term projects. Also, these components usually require more energy, but offer comparably little compute power due to decreased clock frequencies and smaller caches. These drawbacks mainly originate from a primary reliance upon increased structural width of the silicon, besides more resilient manufacturing techniques and materials. For miniaturized satellite use, especially in nanosatellites, the prices commonly charged for such components are prohibitively high, often making their use entirely infeasible.

Regardless if hardened processing components can be utilized, an OBC must compensate for radiation induced displacement damage, latch-up and indirect event effects which can not be mitigated technologically. Therefore, both soft- and hardware must be designed to handle these issues, not if, but when they occur. However, this fact has been largely ignored

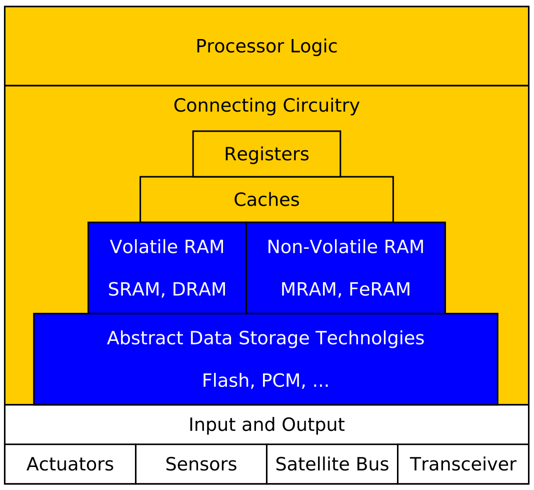

Fig. 1. A satellite's memory hierarchy including input/output functionality (depicted in white) and the processor. Data transiting or stored within elements shown in yellow may be corrupted arbitrarily. Components depicted in blue must be safeguarded against corruption, i.e. using the concepts presented within this paper.

especially in nanosatellite design, resulting in comparably cheap energy efficient federated OBC concepts based upon highly fragile clusters of microcontrollers. This aspect certainly is undergoing a fundamental shift also due to externally induced mission requirements and an increasing level of professionalization [8].

Dependability in the most basic sense can only be assured if program code and required supplementary data can be stored consistently and reliably aboard a spacecraft. Thus, to enable any form of meaningful dependable computing, storage integrity must be assured first and foremost. In the future, new manufacturing techniques such as FD-SOI [9] will enable relatively radiation-tolerant, cheap application processors of-the-shelf [10], reinforcing the need for dependable data storage solutions. Therefore, this research was motivated by finding solutions to assure dependability without fully relying on rad-hard processors and TMR.

Different storage technologies vary regarding the energy-threshold necessary to induce an effect and the severity of its consequences. Various types of Single Event Effects (SEEs), the destructive ones being the most relevant, are well described in [11]. Some novel memory technologies (e.g. MRAM [2], PCM [12]) have shown inherent radiation tolerance against bit-flips, Single Event Upsets (SEUs), due to their data storage mechanism [13], [14]. Due to a shifting voltage threshold in floating gate cells caused by the total ionizing dose, flash memories become more susceptible to bit errors the higher they are scaled. Highly scaled flash memories are more prone to SEUs causing shifts in the threshold voltage profile of one or more storage cells as well [15]. All these memory technologies are sensitive to Single Event Functional Interrupts (SEFIs) [16], which can affect blocks, banks or entire circuits due to particle strikes in the peripheral circuitry.

To enable meaningful dependable computing, data consistency must be assured both within volatile and non-volatile memory, see Figure 1. 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 highly uncommon aboard spacecraft and thus not considered in this paper. 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 dependable data storage 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.

III. 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 [17]. 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 small satellites and very simple nanosatellites often utilize custom firmware optimized for very low RAM usage, larger spacecraft as well as most current and future nanosatellites do rely upon SRAM, DRAM or SDRAM. For simplicity, we will refer to these technologies as RAM within this section. However, it is not to be confused with the use of the term RAM in Sections IV and Vof this paper, 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

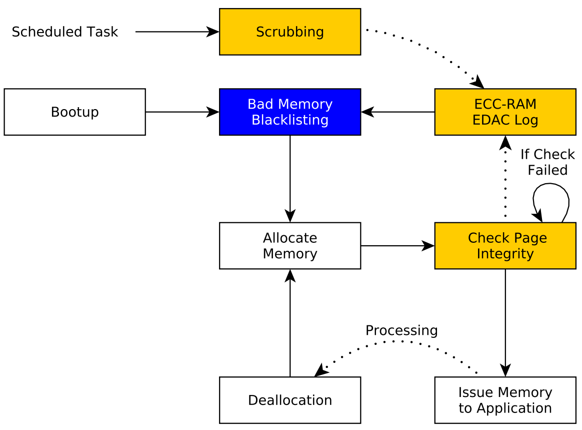

Fig. 2. Integrity of volatile memory can be guaranteed if memory checking (yellow), ECC and page-wise blacklisting (blue) are combined. Scrubbing must be performed periodically to avoid accumulating errors in rarely used code or data.

used erasure code. Assuming these FDIR measures are implemented, a consistency regime based on memory validation, error scrubbing and blacklisting as depicted in Figure 2 can be established.

A. Memory Corruption and Countermeasures

The threat scenario for RAM 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. Soft errors 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 very weak or no erasure coding is utilized. 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 [18]. Highly charged particles impacting the silicon of RAM chips can also permanently damage circuitry. Therefore, to compensate for both hard and soft errors, ECC should be introduced [19].

By utilizing ECC-RAM 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 [20]. 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 [21], [22]. 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 drastically more 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. Below, we thus present a software driven approach to achieve a high level of RAM integrity without expensive and comparably slow space-grade multi-bit-error correcting ECC modules. Ultimately, stronger ECC for RAM is not a satisfying final solution to RAM consistency requirements due to inherent weaknesses of this approach during prolonged operation.

B. A Memory Consistency Assurance Concept

When utilizing ECC, memory consistency is only assured at access time, unless specialized self-checking RAM concepts are applied in hardware [23], [24]. 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 [25]. 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.

Scrubbing tasks can be implemented in software within the OS's kernel, but could be initiated by a userland application as well. 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 system 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.

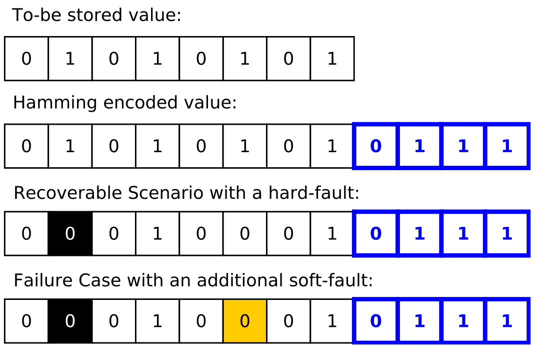

Fig. 3. With single-bit correcting ECC-RAM, a word should no longer be used once a single hard-fault has been detected. Hard faults are depicted in black, soft faults in yellow, erasure code parity in green.

C. 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 3. 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 [26]. 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 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 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.

IV. FTRFS: A Fault-Tolerant Radiation-Robust Filesystem for Space Use

While MRAM can not yet offer the capacity necessary for payload data, it is an excellent choice for storing OS data due to its SEU immunity. However, even then the OBC is still prone to SEFIs, stray-writes, processor- and controller errors as well as in-transit data corruption. FTRFS (fault-tolerant radiation-robust filesystem for space use) was designed for small volumes ($\leq$4MB), but can also manage significantly larger volumes up to several gigabytes in size (depending on the data block size). Erasure coding is applied to protect against in-transit data corruption and stray-writes within memory pages, which can not be prevented using memory protection. Cyclic redundancy checksums (specifically CRC32) are utilized for performance reasons in tandem with the Reed-Solomon (RS) erasure code [27]. Even though CRC16 could be considered sufficient for most common block sizes, we utilize a 32-bit checksum to further minimize collision probability at a minimal compute overhead.

Most modern processors support memory protection, also for directly mapped memory such as MRAM, thereby enable a powerful safeguard against data corruption due to processor and controller faults. However, memory protection has been largely ignored in RAM-FS design, which, in part, can be attributed to a misconception of memory protection as a pure security measure. For directly mapped non-volatile memory, memory protection introduces the memory management unit as a safeguard against data corruption due to upsets in the system [28]. In a scenario where MRAM is used, only in-use memory pages will be writable 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.

As the volatile memory's integrity can be assured by the previously described functionality, we assume ECC to be applied to CPU-caches, thus faults in these deceives are considered detectable and possibly correctable at runtime. A computer running FTRFS must be equipped with a memory management unit with its page-table residing in integrity-assured memory to enable run-time dynamic memory protection. All other

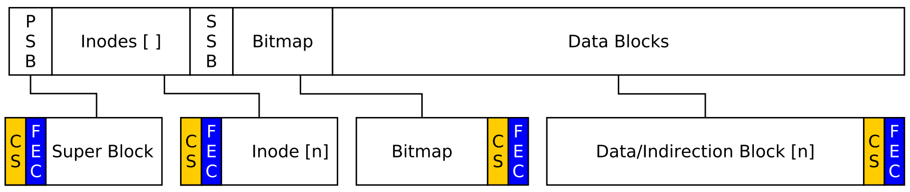

Fig. 4. The basic layout of FTRFS: EDAC data is appended or prepended to structures, depending on if a structure is of compile-time fixed size or not. PSB and SSB denote the primary and secondary super blocks. The root inode is statically assumed to be the first inode.

elements (e.g. periphery and compute units), memories (e.g. registers) and in-transit data in buffers are considered potential error sources. 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 FS, however it should rather be implemented below the FS level [29] or as an overlay, e.g. RAIF [30].

A. Approach Outline and Alternatives

We consider an FS the most portable and efficient approach to combine these features, as the resulting solution should also be bootable while maintaining flexible EDAC capabilities. RAM FSs are not block based and benefit from the ability to access data arbitrarily. Data structures holding information about the physical (e.g. inode count and volume size) or logical (e.g. directory structure and permissions) FS layout require relatively little space compared to file data. In contrast to block-based erasure coding, FS level measures enables stronger erasure coding for these structures requiring minimal extra space. Also, block-based coding would introduce abstraction, thereby additional code and complexity. A block-based approach would also sacrificing the random access possibility on MRAM, requiring more complicated locking. We initially intended to utilize an existing FS instead of implementing and maintaining one, and therefore conducted an in-depth review of potential alternatives.

Silent data corruption has become a practical issue with nowadays common many-terabyte sized volumes. Therefore, next-generation FSs, e.g. BTRFS [31] and ZFS [32], and some modern RAID [1] solutions [33] can maintain checksums for data blocks and metadata, but none of these FSs scale to small storage volumes. All of these FSs include throughput enhancing functionality like caching and disk head tracking, to optimize data access and utilize locality. However, these enhancements do not apply to random-access memories and add significant code overhead, thereby reducing performance while at the same time making FS implementations more error prone. Minimum supported volume sizes are far beyond what current miniaturized satellites can offer.

[1]Redundant Array of Independent Disks

FSs for flash devices [34], [35] already handle wear leveling[2] and support device EDAC functionality (checksumming, spare handling and recovery). However, these FSs require interaction with the memory technology or the flash translation layer (FTL) [3], thereby are incompatible with other memory technologies. This introduces further input/output-load and may result in data corruption, and, as flash memory is block based, these FSs would suffer similar drawbacks as a pure error correcting block layer.

RAM FSs are usually optimized for throughput or simplicity, but are usually designed for volatile memory, thus do not even include nondestructive unmount procedures. Few RAM FSs for non-volatile exist [36], [37], but none of these FSs are designed with dependability in mind, albeit PRAMFS which however so does not offer data integrity guarantees.

Thus, in the absence of a potentially reusable FS, we decided to develop FTRFS based on a layout similar to the simple and space conserving extended 2 (ext2) FS. Aligning FTRFS to this FS's layout enabled significant code reuse especially regarding concurrency, locking and permission handling. We adapted PRAMFS's memory protection functionality for FTRFS and introduced erasure coding.

B. Metadata Integrity Protection

Efficient error detection and correction of meta information and data was considered crucial during development. The protective guarantees offered by the FS can be adjusted at format time or later through the use of external tools to the mission duration, destination and the orbit a spacecraft operates on. For proper protection at the logical level, in addition to the stored FS objects (inodes) and their data, all other information must be protected as well. Thus, we borrow memory protection from the *wprotect* component of *PRAMFS*.

Although the basic FS layout is inspired by *ext2* data addressing and bad block handling work fundamen-

[2]keeping all parts of a memory device at an evenly used, level
[3]The memory technology device subsystem (MTD) in Linux

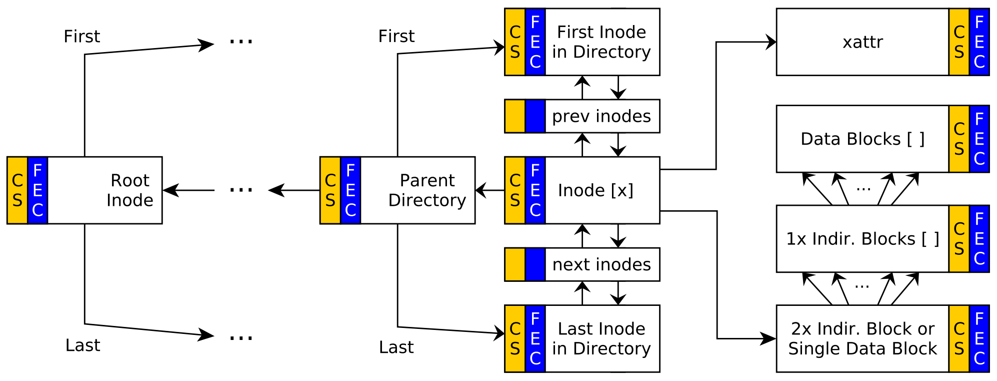

Fig. 5. Each inode can either utilize direct addressing or double indirection for storing file data. An inode may possess extended attribute and contains a reference to its parent directory. Each directory's child inodes are kept within a double linked list maintained by the parent.

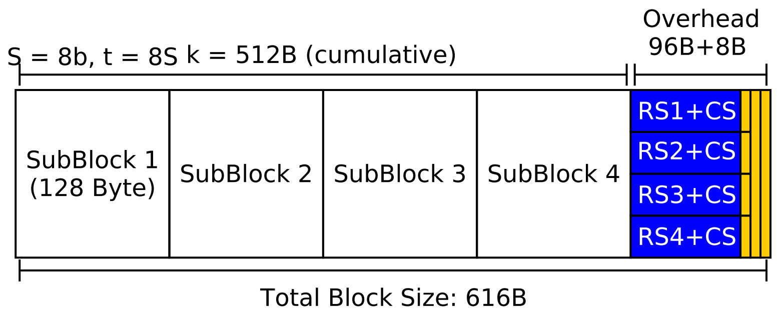

Fig. 6. A 512B data block subdivided into 4 subblocks using example RS parameters. Checksums for the entire data block, EDAC data and each subblock are depicted in yellow, FEC data in blue.

tally differently. *The Super Block* (SB) is kept redundantly, as depicted in Figure 4 and an update to the SB always implies a refresh of the secondary SB. The SB contains EDAC parameters for blocks, inodes and the bitmap, and is static after volume creation. We avoid accumulating errors over time through scrubbing and trigger this functionality upon execution of certain FS functionality (e.g. directory traversal). Using MRAM, the FS does not suffer from radiation induced bit-rot and errors thereby can only accumulate during usage, additional time-triggered scrubbing during periods of little or no volume access is unnecessary.

A block-usage bitmap is appended to the secondary SB and allocated based on the overhead subtracted data-block count. Thus, the protection data is located in the first block after the end of the bitmap, see Figure 4. We refrain from re-computing and re-checking the entire bitmap upon each access, as file allocation or truncate related operations can imply up to hundreds of consecutive alterations to the bitmap.

Inodes are kept as an array, each representing a file, directory or other FS object. Their consistency is of paramount importance as they define the logical structure of the FS. As each inode is an independent entity, an inode-table wide FEC segment is unnecessary.

Most modern FSs utilize tree-based structures or triple-indirection to organize data into blocks or extends (chunks of data extending over a given number of blocks or size). These structures are comparably fragile and space inefficient for small files, as a significant base-amount of blocks must be set aside only for addressing (e.g. 1 block to hold the actual data, 3 blocks for triple-indirection). To optimize the FS towards both larger (e.g. a kernel image, a database) and very small (e.g. scripts) files, direct data addressing and double indirection are supported, as depicted in Figure 5.

Nanosatellites are not yet considered security critical devices. However, the application area of nanosatellites will expand considerably in the future and include security critical applications [8]. 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 extended attributes (*xattrs*). As *xattr* blocks contain various records and different individual permissions, the block's integrity is verified once before an operation and it is updated after all write access (in bulk) has been concluded. *xattrs* are treated like data blocks but are deduplicated and referenced directly within inodes.

C. Algorithm Details and Performance

There are numerous erasure codes available that could be used to protect a full size-optimized Linux root FS including a kernel image safely over a long period of time. After careful consideration, Reed-Solomon (RS) was chosen mainly due to the following:

- Cyclic block codes show excellent performance for multi-bit error correction. RS is particularly well analyzed, and widely used in various embedded scenarios, including spacecraft. Optimized software implementations, IP-cores and ASICs are available, guaranteeing universal availability.
- MRAM, while being SEU immune, is still prone to stray-writes, controller errors and in-transit

Data Structure	Size (B)	Correction Symbols/Code	# Codes	Correction Total (B)	Overhead (B)	Overhead (%)
Super Block	128	32	1	32	76	59.38%
Inode	160	32	1	32	76	47.50%
Bitmap	1543	16	14	224	448	29.09%
Data Blocks	1024	4	8	32	104	10.16%
	1024	8	8	64	168	16.41%
	4096	4	32	128	392	9.57%
	4096	8	32	256	648	15.82%

TABLE I
EDAC OVERHEAD FOR DIFFERENT FS STRUCTURES. BITMAP FOR: 16MB FS, 1024B BS WITH 8B CORRECTION CAPABILITY PER 128B DATA

data corruption. Misdirected access within a page evades memory protection and corrupts the FS, thus grouped errors will occur. RS relies upon symbol level error correction, which is precisely the kind of corruption the FS must correct.

RS decoding becomes computationally more expensive with increasing symbol size, thus it is important to utilize small symbols while choosing large enough symbols to enable reasonably long codes. The protected data is subdivided into sub-blocks sized to 128B plus the user specified number of correction-roots using a comparably small symbol size of 8 bits. 8 bit sized symbols enable high-performance RS decoding and byte-alignment simplifies addressing, while Inodes and SBs fit into single RS-codes. To skip the expensive RS decoding step during regular read operations, data- and FEC integrity are verified using CRC32 checksums. Data blocks (Figure 6) are divided into subblocks so the FS can make optimal use of the RS code length, while correction data is accumulated at the end of the data block. For common block-sizes and error correction strengths, 4 to 32 RS code words are necessary, see Table I FS overhead.

V. High-Performance Flash Memory Integrity

To enable larger payload mass-storage, highly scaled memory is required and NAND-flash (see Section V-A) is currently the most popular technology to fill this role. Even though flash may eventually be replaced by (radiation tolerant) phase change memory (PCM) in the long run, it is not yet available in high-density versions. Thus we must focus our attention on flash as the only viable mass storage technology, until high-density PCM becomes available in the future. For this memory type, software must compensate for wear and various translation and abstraction layers, making all-in-one solutions like FSs very complex and error prone. Thus, more sophisticated EDAC concepts are required, as simple redundancy, parity or erasure coding are insufficient. A protective concept efficient also for highly-scaled flash memory must therefore be based upon the special properties of flash and its architecture. Therefore the initial parts of this section are dedicated to flash memory organization and an analysis of why simple voting is insufficient in this case. We used these results to construct an EDAC concept specifically to handle highly scaled multi-level cell NAND-flash according to the these requirements:

1) Efficient data storage on MLC flash memory.
2) Integrity protection and error correction strength adjustable to varying mission parameters.
3) Effective handling of radiation effects on the memory as well as the control logic.
4) Protection against device failure.
5) Low soft- and hardware complexity.
6) Universal FS and OS support.

We consider these requirements to be met best with FTL-middleware, which is similar to a self contained extension or a plugin. Therein, RAID-like features and checksumming can be combined effectively with a composite erasure coding system. We implemented MTD-mirror as part of the MTD subsystem of the Linux Kernel. By utilizing mirroring (RAID1) or distributed parity (RAID5/6) we can therefore protect against device, block and page failure.

To handle permanent block defects, single event functional interrupts, radiation induced programmatic errors and logic related problems, we apply coarse symbol level erasure coding. As outlined below, this measure would be insufficient to compensate for radiation effects, silent data corruption and bit flips. The solution is to be implemented within the FTL, therefore it can still be kept abstract and device independent while it can also profit from hardware acceleration, additionally providing enhanced diagnostics.

A. *Single- and Multi-Level Cell Flash*

Each flash memory cell consists of a single field effect transistor with an additional floating gate. Depending on the voltage applied between the gate's source and drain, electrons are pumped into or out of the floating gate. A cell's state is thus dependent on whether or not a specific threshold voltage level is exceeded. a cell can be read as programmed (0) if the threshold is a exceeded, or erased (1), see Figure 7a.

A Single Level Cell Flash (SLC) cell can thereby store one bit using one threshold. In an Multi Level Cell (MLC) gate, additional thresholds are used to differentiate between multiple values, see Figure 7b. With 2^n voltage levels and $2^n - 1$ thresholds, it is possible to encode n bits, on the same piece of silicon, MLC can thus allow a much higher packing density. However, electrical complexity grows and the required

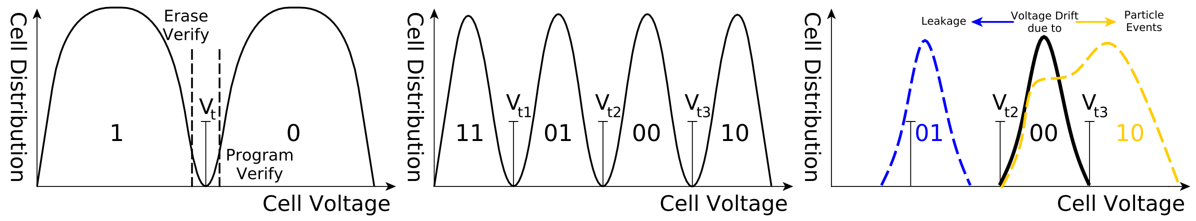

Fig. 7. The voltage reference and thresholds of SLC- (a) and MLC (b) cells. (c): Bit-flips for the value 00 due to leakage (01) and radiation effects (10).

read sensitivity and write specificity increases with the number of bits represented.

As the voltage-delta between levels decreases, increased precision is required for sensing and charge-placement, resulting in MLC memory being more dependent on its cells' ability to retain charge. A state machine is required for addressing MLC memory which in turn increases latency and adds considerable overhead logic, as multiple addresses are not mapped to one gate. This comparably complex state machine may thus hang or introduce arbitrary delays with operations requiring multiple cycles and varying latency.

Environmental temperature variations change the leakage current of the silicon, draining the floating gate of both SLC and MLC flash over time [38]. Software, e.g. the flash translation layer (FTL), must impose appropriate countermeasures against these effects in addition to high energy radiation related degradation. MLC flash memories are also more susceptible to bit errors than SLC [39] due to a shift voltage threshold in floating gate cells caused by the total ionizing dose. Also, highly scaled flash memories become increasingly prone to single event upsets (SEU) causing shifts in the threshold voltage profile of cells, referred to as multiple bit upsets [40]. A varying amount of data will thereby be corrupted through one SEU for which EDAC measures must be adapted, depending on the number of bits represented within a cell [41]. Currently widely used single-bit error correcting EDAC measures are thus insufficient to protect MLC consistency.

Data in flash memory can not read freely due to the layout of the cell circuitry. NAND-flash memory is organized in blocks and pages, in which cells are connected as negated AND circuits, forming NAND-gates. If connected as NOR gates, random-read-access is possible at the cost of strongly increased wiring and controller overhead is possible, curtailing so called NOR-flash's data storage density. Partial writes to flash are impossible and the entire block's previous content must be read and updated in RAM, afterwards the block can be erased (by draining the block's cells' voltage) and programmed anew. Hence, read and write operations induce different timing behavior and make access to MLC-flash much more complicated than to SLC-NAND- or NOR-flash due to the addressing state machine. With either technology, a flash FS and the FTL must handle basic block FDIR as well as erase-block and (for NAND-flash) read-page abstraction. A flash FS must thus implement all functionality necessary to handle failure of memory blocks and pages to extend memory life. It must perform block wear leveling, read and erase block abstraction, bad-block relocation and garbage collection to prevent premature degradation and failure of pages and blocks. The FTL can implement parts of this FDIR functionality for the FS interfacing with hardware specific device drivers.

Over time, a flash memory bank will accumulate fully defective blocks and utilize spare blocks to compensate. Eventually, the pool of spares will be depleted, in which case the FTL or FS will begin recycling less defective blocks and compensate with erasure coding only, thereby sacrificing performance to a certain degree. Traditionally, erasure coding is applied in software or by the controller to counter defects due to wear and bit-flips due to charge leakage. For simplicity, cyclic block codes with large symbol sizes are utilized, though the latest generation of solid-state-drive controllers has begun employing more sophisticated codes. For space use, the symbol size is reduced to support one or two bit correcting erasure coding, as corruption will mostly result from radiation effects. However, block EDAC becomes inefficient due to the occurrence of grouped errors and SEUs affecting multiple cells in highly scaled memory [42].

B. *Alternative Approaches*

While voting and triple-modular-redundancy are 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. However, due to the varying timing behavior of NAND-flash and the more complex logic, the resulting voter-circuit would thus become more error prone, require more energy and reduce overall performance.

Like with FTRFS, the outlined requirements could also be met using a flash FS. UFFS particularly would be a prime candidate to be extended to handle multiple memory devices and enhanced EDAC, however, the resulting all-in-one FS would be complex and error prone. Device independence could also be added on top of regular flash FS as a separate layer of software [30], see Figure 8. Within a RAIF set, increased protective requirements could be satisfied with additional redundant copies of the FS content. The underlying individual FSs would then have to handle all EDAC functionality, as RAIF by itself does not offer any integrity guarantees beyond FS or file failure.

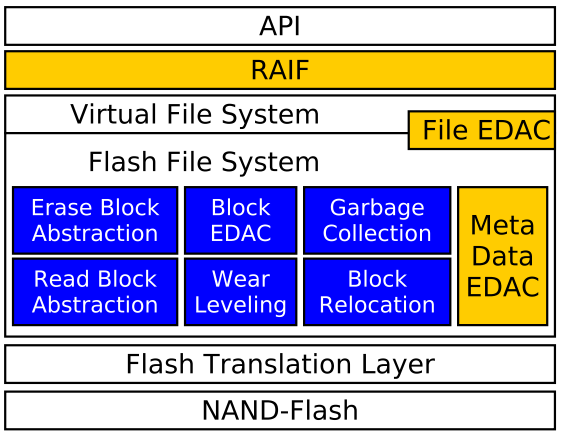

Fig. 8. Memory access hierarchy for a RAIF based system with added error correction. Extensive modifications to various components are be required; affected elements are depicted in yellow.

RAIF sets, however, are prone to FS-metadata corruption which can result in single block errors failing an entire FS, as RAIF only reads from underlying FSs. Files damaged across all of a RAIF set's FSs would become unrecoverable and would forward a defective copy to the application, instead of combining multiple damaged copies into a correct one. RAIF therefore actively inhibits error correction and may even cripple recovery of larger files. While RAIF could be adapted to this issue, the resulting storage architecture would again become highly complex, difficult to validate and debug. As RAIF implements simple FS redundancy, its storage efficiency will always be lacking compared to distributed parity concepts. Being a pure software layer without the possibility to interact with the devices, hardware acceleration of RAIF would be impossible.

RAID can be applied efficiently to space-borne storage architectures and has been used previously aboard spacecraft (e.g. in the GAIA mission), in contrast to RAIF [43]. These architectures, however, were designed for SLC (see Section V) and only relied on RAID to achieve device fail-over through data mirroring (RAID1) and distributed parity (RAID5/6) [44], [43]. However, they usually rely upon the block level hardware error correction provided by the flash memory or controller or implement simple parity only. The different distribution of bit-errors on MLC flash, can not be addressed using these concepts and coarse symbol based erasure coding is insufficient.

C. The MTD-Mirror Middleware Layer

RAID-like functionality could be implemented as a middleware within the FTL as depicted in Figure 9, thus the software can interact both with the FS and the rest of the FTL, without requiring alterations to either. Such middleware can remain pervious to FS operations and allows device failure protection to be combined with enhanced erasure coding as 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

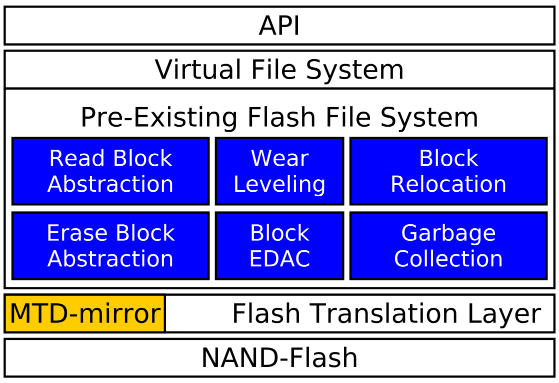

Fig. 9. Memory access and data flow hierarchy for an MTD-Mirror set. Flash-memory specific logic is depicted in blue and may partially reside within the FTL. Required modifications are indicated in yellow.

an FTL middleware module.

The functional layout of MTD-mirror's block consistency protection is depicted in Figure 10 and is based upon a serial concatenated (composite) erasure code system. Like in FTRFS, a data checksum allows bypassing decoding of intact data. The second checksum is used to prevent symbol drift of the erasure coding layers. The first erasure coding layer is based on relatively coarse symbols and protects against data corruption induced by stray writes, controller issues and multi-bit errors similar to FTRFS, due to which RS [45] was selected as well.

Erasure coding with coarse symbols is efficient if symbols are largely or entirely corrupted, but shows weak performance when compensating radiation-induced bit-rot. SEUs statistically will equally degrade all data of a code word, corrupting multiple code symbols with comparably few bit errors, due to which previous storage concepts often relied upon convolution codes. However, as error-models become more complex (2 or more bit-errors in MLC), convolution code complexity increases, and storage efficiency diminishes. To handle single or double bit-flips within individual code symbols of the first level RS code, a second level of erasure coding using Low-Density Parity Check Codes (LDPC) [46] was added. LDPC is efficient with very small symbol sizes (1 or two bit), allows high-speed iterative decoding [47] and offers superior performance compared to convolution codes [48]. LCPC supports recovery of slightly corrupted RS-symbols and parity, but due to more costly decoding is only being used in case RS fails for repairing individual damaged symbols. This runtime behavior can thereby drastically increase recovery rates on radiation-degraded memories. The set can also attempt repair using data from different blocks in the hope of obtaining a consistent combination of block-data, as multiple copies of the erasure code parity data and checksums are available.

We chose to apply the two FEC layers in order (Figure 10) as hardware-acceleration of RS is readily available to us. Hence, the sequence should be determined based on the available acceleration possibilities and mission parameters. If severe bit-rot is expected or higher order density MLC is used, the LDPC-layer should be applied prior to RS decoding.

D. Optimization & Future Work

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. There has been prior research on adding checksumming support to RAID5 in [44], [43], though utilizing RAID5 directly would introduce issues. Error correction information in RAID5C [44] 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

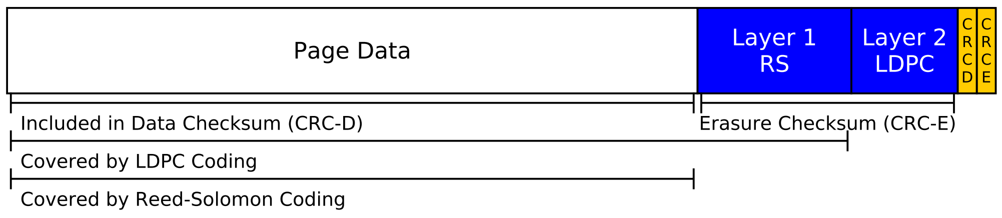

Fig. 10. The simplified layout of an MTD-mirror write-block without to read-page indication. Added erasure code correction information is depicted in yellow, checksums in blue.

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 per 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 [49], [50]. 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.

VI. Results and Current Status

For testing our volatile memory protection system, we are currently using the memcheck functionality within a virtualized ia32-demonstrator. Once the memcheck port for ARM has been completed, we will switch the demonstrator to the ARMv7 architecture and can thereby re-use the existing setup for testing and debugging.

FTRFS is currently undergoing testing and has been implemented using the RS implementation of the Linux kernel, as its API also supports hardware acceleration. Due to its POSIX-compliance, the FS could easily be ported to other platforms. An in-depth description and analysis of FTRFS has been published as part of the computer science conference ARCS2015 proceedings [51]. While the FS has been tested and logically validated, the code should be optimized which will result in a drastic performance increase. The FS's overall performance depends strongly upon the utilized hardware, as synthetic benchmarks are not representative for different OBC implementations. An in-depth analysis will be conducted once a feature complete OBC implementation for MOVE-II is available. Data degradation during metering will be introduced using fault injection. However, artificial fault injection is usually not considered sufficient to prove the efficiency of a fault-tolerance concept for space-use. An identical test-model of our satellite's OBC including the FS will undergo testing using various radiation sources before launch. Results will be made available once these tests have been conducted.

The high level architecture of MTD-mirror has been implemented for RAID1, though we have begun working towards enabling more advanced distributed parity concepts. The concept originally begun as a spin-off from FTRFS using a fault-tolerant block layer for more complex memory than MRAM. MTD-mirror therefore also serves as a showcase for how innovative concepts developed for miniaturized satellites can influence commercial and agency space flight: the concept's design was in part influenced by the payload data storage requirements of the JUICE and Euclid missions. A related paper on how the MTD-mirror layer can be adapted to very large storage volumes common to these missions was published at DASIA2015 [52], which however is not intended directly for nanosatellite use. FTLs like Linux-MTD exist for most modern OSs or are built into the base kernel for those supporting flash memory directly. Besides API adaptions to MTD-mirror, little additional work is required for porting it to other OSs.

VII. Conclusions

In this paper we presented three software-driven concepts to assure storage consistency, each specifically designed towards protecting key 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 increased system dependability. 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 FS 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 paper, can thus drastically increase system dependability, even for missions with a very long duration. Thereby, simplicity can be maintained, error sources minimized, testability can be increased and throughput maximized.

ACKNOWLEDGMENT

The author would like to thank G. Furano and G. Magistrati of ESTEC, G. Williamson of Maxwell Technologies and P. Lombardi of Syderal SA for their encouragement and invaluable feedback. Special thanks to Carla Höhn and Nicolas Appel for their feedback and support.

REFERENCES

[1] RTEMS Development Team, "The real-time executive for multiprocessor systems RTOS," project website: www.rtems.org.

[2] R. Katti *et al.*, "High speed magneto-resistive random access memory," 1992, US Patent 5,173,873.

[3] J. Bouwmeester and J. Guo, "Survey of worldwide pico-and nanosatellite missions, distributions and subsystem technology," *Acta Astronautica*, 2010.

[4] M. Swartwout, "The first one hundred CubeSats: A statistical look," *JoSS 2pp*, 2014.

[5] S. Bourdarie and M. Xapsos, "The near-earth space radiation environment," *IEEE Transactions on Nuclear Science*, 2008.

[6] J. Heirtzler, "The future of the south atlantic anomaly and implications for radiation damage in space," *Journal of Atmospheric and Solar-Terrestrial Physics*, 2002.

[7] J. Schwank *et al.*, "Radiation Hardness Assurance Testing of Microelectronic Devices and Integrated Circuits," *IEEE Transactions on Nuclear Science*, 2013.

[8] D. Evans and M. Merri, "OPS-SAT: An ESA nanosatellite for accelerating innovation in satellite control." Spaceops, 2014.

[9] J. Schwank *et al.*, "Radiation effects in SOI technologies," *IEEE Transactions on Nuclear Science*, 2003.

[10] M. Kochiyama *et al.*, "Radiation effects in silicon-on-insulator transistors fabricated with 0.20 μm FD-SOI technology," *Nuclear Instruments and Methods in Physics Research*, 2011.

[11] ECSS, "Calculation of radiation and its effects and margin policy handbook," 2010.

[12] F. Chen, "Phase-change memory," 2014, US Patent App. 14/191,016.

[13] G. Tsiligiannis *et al.*, "Testing a Commercial MRAM Under Neutron and Alpha Radiation in Dynamic Mode," *IEEE Transactions on Nuclear Science*, 2013.

[14] J. Maimon *et al.*, "Results of radiation effects on a chalcogenide non-volatile memory array," in *IEEE Aerospace Conference*, 2004.

[15] S. Gerardin *et al.*, "Radiation Effects in Flash Memories," *IEEE Transactions on Nuclear Science*, 2013.

[16] D. Nguyen *et al.*, "Radiation effects on MRAM," in *Radiation and Its Effects on Components and Systems*. IEEE, 2007.

[17] L. Z. Scheick, S. M. Guertin, and G. M. Swift, "Analysis of radiation effects on individual dram cells," *IEEE Transactions on Nuclear Science*, 2000.

[18] A. Hwang *et al.*, "Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design," *ACM SIGPLAN Notices*, 2012.

[19] K. Gupta and K. Kirby, "Mitigation of high altitude and low earth orbit radiation effects on microelectronics via shielding or error detection and correction systems," 2004.

[20] D. Dopson, "SoftECC: A system for software memory integrity checking," Ph.D. dissertation, Massachusetts Institute of Technology, 2005.

[21] R. Goodman and M. Sayano, "On-chip ECC for multi-level random access memories," in *IEEE CAM*, 1989.

[22] D. Bhattacharyya and S. Nandi, "An efficient class of secded-aued codes," in *I-SPAN'97 Proceedings*. IEEE, 1997.

[23] Y. You and J. Hayes, "A self-testing dynamic RAM chip," *Electron Devices, IEEE Transactions on*, 1985.

[24] D. Callaghan, "Self-testing RAM system and method," 2008, US Patent 7,334,159.

[25] J. Foley, "Adaptive memory scrub rate," 2012, US Patent 8,255,772.

[26] K. Suzuki *et al.*, "Birthday paradox for multi-collisions," in *ICISC*, 2006.

[27] S. Wicker and V. Bhargava, *Reed-Solomon codes and their applications*, 1999.

[28] S. Suzuki and K. Shin, "On memory protection in real-time OS for small embedded systems," in *IEEE Workshop on Real-Time Computing Systems and Applications*, 1997.

[29] S. Su and E. Ducasse, "A hardware redundancy reconfiguration scheme for tolerating multiple module failures," *IEEE Transactions on Computers*, 1980.

[30] N. Joukov *et al.*, "RAIF: Redundant array of independent filesystems," in *IEEE MSST*, 2007.

[31] O. Rodeh *et al.*, "BTRFS: The Linux B-tree filesystem," *ACM TOS*, 2013.

[32] O. Rodeh and A. Teperman, "zFS – a scalable distributed file system using object disks," in *IEEE MSST*, 2003.

[33] M. Baker *et al.*, "A fresh look at the reliability of long-term digital storage," in *SIGOPS Operating Systems Review*, 2006.

[34] J. Engel and R. Mertens, "LogFS-finally a scalable flash file system," in *12th International Linux System Technology Conference*, 2005.

[35] S. Qiu *et al.*, "NVMFS: A hybrid file system for improving random write in nand-flash SSD," in *IEEE MSST*, 2013.

[36] M. Stornelli, "Protected and persistent RAM filesystem," project website: pramfs.sourceforge.net.

[37] N. Edel *et al.*, "MRAMFS: a compressing file system for nonvolatile RAM," in *MASCOTS 2004*. IEEE, 2004.

[38] K. Young *et al.*, "SLC vs. MLC: An analysis of flash memory," *Whitepaper, Super Talent Technology, Inc.*, 2008.

[39] F. Irom *et al.*, "SEEs and TID results of highly scaled flash memories," in *IEEE Radiation Effects Data Workshop*, 2013.

[40] S. Gerardin *et al.*, "Radiation effects in flash memories," *IEEE Transactions on Nuclear Science*, 2013.

[41] Y. Cai *et al.*, "Error patterns in MLC NAND flash memory," in *IEEE DATE2012*.

[42] F. Irom *et al.*, "Effects of scaling in SEE and TID response of high density NAND flash memories," *IEEE Transactions on Nuclear Science*, 2010.

[43] S. Zertal, "A reliability enhancing mechanism for a large flash embedded satellite storage system," in *IEEE ICONS*, 2008.

[44] B. Kroth and S. Yang, "Checksumming RAID," 2010.

[45] J. Plank *et al.*, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Software: P&E*, 1997.

[46] E. Kurtas *et al.*, "System perspectives for the application of structured LDPC codes to data storage devices," *IEEE Transactions on Magnetics*, 2006.

[47] M. Lentmaier *et al.*, "Iterative decoding threshold analysis for LDPC convolutional codes," *IEEE Transactions on Information Theory*, 2010.

[48] K. Andrews *et al.*, "The development of Turbo and LDPC codes for deep-space applications," *IEEE Proceedings*, 2007.

[49] M. Song *et al.*, "A low complexity design of Reed-Solomon code algorithm for advanced RAID system," *IEEE TCE*, 2007.

[50] P. Sobe, "Reliability modeling of fault-tolerant storage system-covering MDS-codes and regenerating codes," in *ARCS*, 2013.

[51] C. Fuchs *et al.*, "FTRFS: A fault-tolerant radiation-robust filesystem for space use," *ARCS2015*.

[52] ——, "A fault-tolerant radiation-robust mass storage concept for highly scaled flash memory," *DASIA2015*.