Expecting the Unexpected - Radiation Hardened Software

Peter C. Mehlitz[1]
Computer Sciences Corporation/NASA Ames Research Center, Moffett Field, CA 94035

John Penix[2]
NASA Ames Research Center, Moffett Field, CA 94035

Radiation induced Single Event Effects (SEEs) are a serious problem for spacecraft flight software, potentially leading to a complete loss of mission. Conventional risk mitigation has been focused on hardware, leading to slow, expensive and outdated on-board computing devices, increased power consumption and launch mass. Our approach is to look at SEEs from a software perspective, and to explicitly design flight software so that it can detect and correct the majority of SEEs. Radiation hardened flight software will reduce the significant residual risk for critical missions and flight phases, and enable more use of inexpensive and fast COTS hardware.

I. The Space Radiation Challenge

Operating electronic devices in space leads to a high probability of random, spurious defects caused by various radiation sources. Radiation-induced failures can be caused by *Single Event Effects* (SEEs - single protons or heavy ions hitting the computing device), or by accumulated absorption of radiation leading to degradation over time (Total Ionizing Dose - TID). In the context of this paper we focus on the most likely case of SEEs causing probabilistic memory errors, namely transient Single Event Upsets (SEUs - "flipped bits"), and potentially permanent failures like Single Event Latchups (SELs - "stuck bits"). A thorough analysis of radiation effects on electronic devices can be found in the "Single Event Criticality Analysis Report" [1].

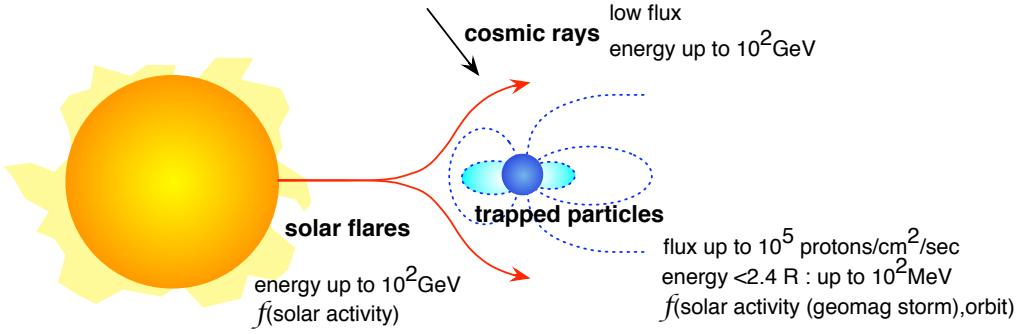

Figure 1. Radiation sources

Due to the different sources of radiation (trapped particles in magnetic belts, solar flares, cosmic rays and - possibly - nuclear power in future spacecraft), the probability of encountering SEEs varies vastly with mission profile, solar activity, electronic devices[18] and spacecraft design. For lower earth orbits (<2.4 earth radius), disrupted operations due to SEEs are a common problem.

---

[1] Intelligent Systems Division, Robust Software Engineering Group, M/S 269-2, <pcmehlitz@email.arc.nasa.gov>

[2] Intelligent Systems Division, Robust Software Engineering Group, M/S 269-2, <John.Penix@nasa.gov>

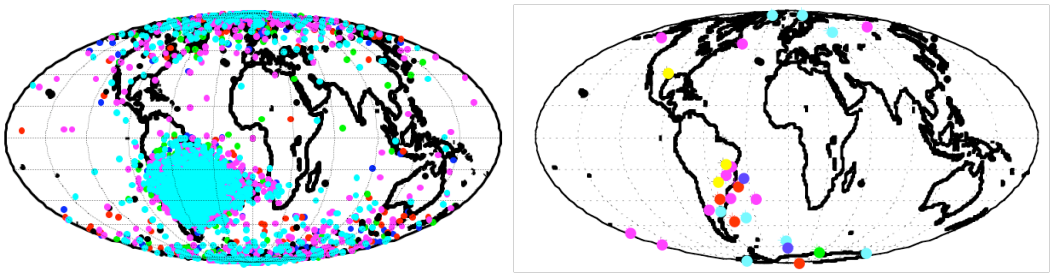

Figure 2. Corrected single bit and uncorrected multi bit errors during GP-B mission

The diagrams above show occurrences of bit failures that were encountered during the Gravity Probe B mission[2], with points marking detected SEEs and colors representing the different on-board computers. The left diagram displays transient single bit failures that were correctable. The right diagram shows uncorrectable multi-bit failures, occurring roughly once per 40 days per computer, causing a reboot about every 90 days (depending on the amount of used memory).

The data shown in these diagrams does not represent a GP-B specific problem. The Satellite News Digest website[3] lists about 40 anomalies attributed to radiation during autumn 2003, which does not include military satellites. Effects can range from temporary functional disruptions to a complete loss of mission.

II. Conventional Hardware Based Approaches

Traditionally, radiation effects have been treated primarily as a hardware problem, to be mitigated by shielding, radiation hardened processors, and Error Detection And Correction (EDAC) memory.

Shielding significantly adds to launch weight, and does not protect against high energy particles like heavy ions (as observed by Gravity Probe B). Materials that are effective for protecting electrical devices can also cause secondary radiation that is harmful for humans. Shielding can even cause higher risk of failure due to increased energy transfer of slowed down particles[4]. Shielding is especially problematic for very small spacecraft (pico spacecraft).

Radiation hardened processors require more power, and are substantially slower than COTS processors, lagging behind in terms of performance by about a decade - an effect known as the “commercial hardware gap”. More recent radiation hardened Single Board Computers (SBCs) achieve redundancy at the board level, but at the cost of size and power. Again, both are critical for pico spacecraft.

EDAC memory uses hardware implemented Error Correction Codes (ECCs), and is usually configured for “single bit correct, multi-bit detect” modes. In order to avoid accumulation of bit errors that cannot be corrected, a “scrubber” software component performs cyclic scans of the memory using idle processor time. Since such a scrubber competes with the application for processor cycles, the scan rate - and hence protection level - goes down if the flight software computational demands go up.

All these hardware measures are static by nature, have to be designed for the worst case, and do not take into account the fact that radiation threat levels vary significantly with flight phase and solar activity. In addition, the radiation problem will get aggravated by future advances of COTS hardware [5]. Due to higher chip densities and lower power levels, the vulnerability against SEEs is expected to increase. Finally, it is important to note that even if radiation hardened hardware is used, there is a significant residual risk. Operations statistics of missions like Gravity Probe B [2] and ARGOS [6] showed a higher rate of memory errors in rad-hardened devices than expected.

III. Benefits of a Software Based Approach

Given the shortcomings of current hardware protection mechanisms, our approach is to create flight software that is resilient against SEEs. By increasing the probability of relevant SEE detection, reducing the amount of potential data loss, and decreasing required recovery time, we expect to achieve a twofold effect

1. increase robustness
2. decrease dependency on specialized hardware

If robustness can be sufficiently increased by means of suitable, hardware independent software design, this would enable more use of commercial off the shelf (COTS) hardware, with significant advantages for a variety of missions. In detail, our work is motivated by the following goals:

(1) Increase on-board computing performance

The performance gap between COTS and radiation hardened processors is typically at least one order of magnitude; it lags COTS by about one decade. While not all spacecraft functions require significantly more computing power, the general need for computing performance in upcoming missions is definitely increasing due to

- more and increasingly complex mission science
- longer mission durations - requiring dynamic adaptation to unforeseen mission events by uploading/reconfiguring software over a long mission lifetime (for which it is good to start with some performance margin)
- more complex system monitoring, such as Independent System Health Management (ISHM), due to very high reliability requirements for human and robotic space flight missions with extended duration

It should be noted that the same arguments hold for increased memory capacity of COTS components.

(2) Reduce power consumption and form factor

Traditional radiation hardening requires shielding and hardware redundancy, resulting in more weight and power consumption. New radiation-hardened SBCs use redundant COTS processors to reduce the performance gap, but still require about twice the power of COTS SBCs. Due to this redundancy, radiation hardened SBCs are also considerably bigger. Both power and form factor are especially critical for pico spacecraft (for example, for scouting missions) that provide only very limited power sources and available space.

(3) Adaptivity

By using flexible software measures, the static nature of hardware-only solutions can be overcome to dynamically adapt protection levels when needed, including short term (SEE) and long term (TID) risks:

- SEE related - protection levels can be ramped up during critical flight phases (orbit, entry) and periods of increased solar activity. During non-critical flight phases, dynamically reduced software protection allows COTS performance gains to be available for more science.
- TID related - material degradation due to absorbed radiation means the likelihood of failure increases over time. To a certain degree, software measures can compensate for this by dynamically increasing protection levels based on observed failures and absorbed TID, effectively extending the lifespan of science missions.

(4) Reduce residual risk on radiation hardened hardware

Some mission profiles impose significant residual risk even if radiation hardened hardware is used (for example, orbits that cross the South Atlantic Anomaly). Most of the proposed software measures can also be used to enhance radiation robustness for these missions.

(5) Cost reduction by widespread COTS hardware usage

Direct cost savings by using COTS hardware generally are lost to increased spacecraft qualification costs due to missing reliability data for COTS components. Widespread use of tested COTS hardware in a variety of missions would make it possible to realize the potential savings, both in terms of hardware and development costs. Cost benefits for pico spacecraft should be immediately available, with a dramatic impact on feasibility of new missions.

The potential benefits of COTS hardware usage can be illustrated by comparing costs and performance of the RAD6000 (the prevalent radiation hardened processor used in space flight), and the current Intel P4:

processor	*per chip costs*	*clock frequency*	*transistors*
RAD6000	> $200,000	25 MHz	~1e^6
Intel P4 *(COTS)*	$500	> 2 GHz	~55e^6

Figure 3. Comparison between radiation hardened and COTS processors

Even though mission costs depend on many factors, the acquisition cost savings of a single radiation hardened processor is enough to enable a whole CubeSat pico spacecraft to be built and launched.

While we do not expect that software-only measures will make radiation hardened hardware obsolete in general, we do anticipate the risk can be reduced enough to enable use of COTS hardware in substantially more missions. Certain spacecraft with tight power, weight and size constraints (for example, pico spacecraft) will not be feasible without this capability, software based radiation protection can be considered as an enabling technology for such missions.

IV. The Radiation Hardened Software Project

To look at radiation effects from a software perspective, we first need to categorize the types of potential failures. SEEs can be classified as transient or permanent, correctable or uncorrectable, affecting memory chips or processors. Single Event Upsets (SEUs) of memory chip cells represent the most likely scenario, resulting in randomly "flipped bits" that can be corrected by rewriting. While the rewrite is not time critical in itself, detection and correction has to occur before SEUs accumulate. Depending on number of affected bits and type of device (for example EDAC memory), the error might either be on-the-fly correctable (using Error Correction Codes - ECCs), or might require re-computation by means of function restart.

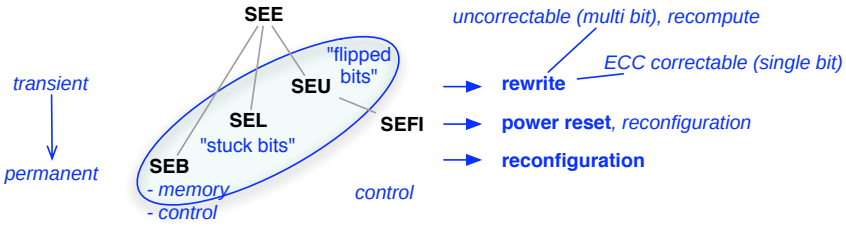

Figure 4. SEE classification

A special case of SEUs are Single Event Function Interrupts (SEFIs), causing a device to enter an undefined state due to an SEU in a control circuit. SEFIs require a power reset to recover, which has to be triggered by a secondary hardware device that monitors the progress of the affected processor. Single Event Latchups (SELs) are caused by high current states that have the potential to overheat and destroy a device, unless a power reset is not performed in time. Single Event Burnouts (SEBs) destroy a device by a high current state in a power transistor. SELs and SEBs can manifest themselves as "stuck bits", i.e. require power resets and potential reconfiguration to avoid reusing permanently damaged memory locations.

The radiation hardened software project is mostly focused on temporary and permanent memory failures in memory chips and processor cache.

Robustness of flight software depends on a number of system layers, including application, programming environment, operating system, and hardware. To achieve overall robustness, all layers ultimately have to be analyzed, but our initial focus is on software components that can be designed according to our needs - the application layer. This includes error simulation capabilities and potentially flight experiments to validate our approach. Consequently, the project consists of three parts:

1. a library for radiation hardening (RHS library) with corresponding application design guidelines,
2. a flight experiment to test a set of applications that were hardened using the RHS library against non-hardened variants in a real space environment,
3. a software simulator to reproduce the measured results in a lab environment, optimize the RHS library, and apply it to other missions and applications.

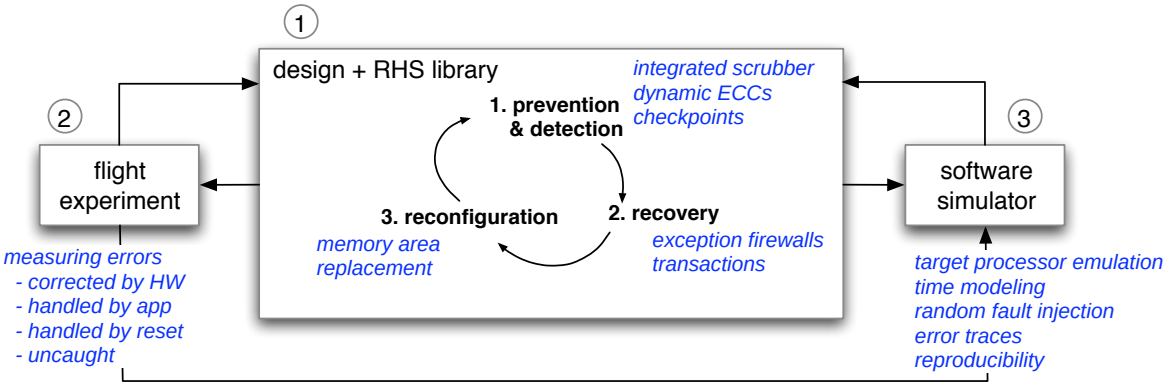

Figure 5. RHS project components

A. RHS Library

Our primary research focus is to coalesce known techniques into a homogenous software design that is resilient against probabilistic errors, while maintaining an acceptable runtime overhead. The goal is to increase overall robustness from an engineering perspective; a revolutionary new way to detect errors would be useless if it is not backed up by adequate measures to recover from a detected error. We will spend some effort investigating new hardening techniques, but the majority of the work will go into creating a balanced approach.

The underlying execution model of the application is a sequence of transactions - functional units that either completely succeed, or are completely reverted. A transaction can be further subdivided into a fallback point, a check point, and potential rollback actions. Each of these transaction components has an associated runtime cost, the goals being to

1. minimize nominal path cost overhead ($C_s + C_c = \min$)
2. minimize time to recovery for (off nominal) rollback actions ($D_r = \min$)

The challenge is to achieve sufficient time-to-recovery while keeping the runtime costs of frequent checks and fallback point creation at an acceptable level.

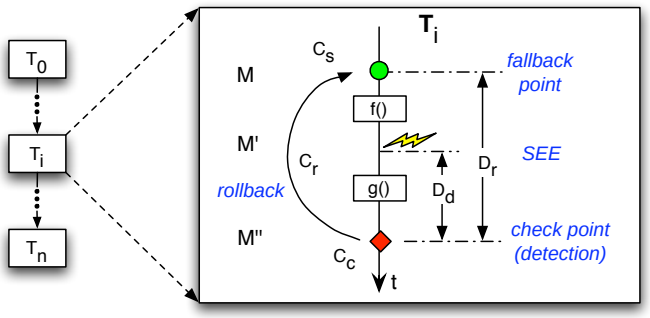

Figure 6. Transaction schema

Based on this model, we consider three different areas of hardening functionality:

(1) Prevention and Detection

This mainly refers to so called “memory scrubbers”, which are tasks that periodically scan EDAC memory to prevent accumulation of bit errors. The memory is usually configured to detect multi bit failures, but to correct only single bit errors. It is therefore crucial to maintain a certain scan rate to avoid accumulation. Prevention in this sense refers to correcting SEEs by means of hardware and software that is transparent to the application. Existing

scrubbers are usually transparent by being implemented as isolated idle tasks, but therefore also compete with the application for processor cycles.

Since memory that is not used by the application does not need to be scanned for SEUs, and memory that is currently accessed by a radiation-hardened software application does not need to be re-scanned by the scrubber, we expect improvements of the effective memory scan rate by integrating/interfacing the scrubber with the application. As a first step, we will use heap information about allocated objects to direct the scrubber (which also has to check the heap structure itself).

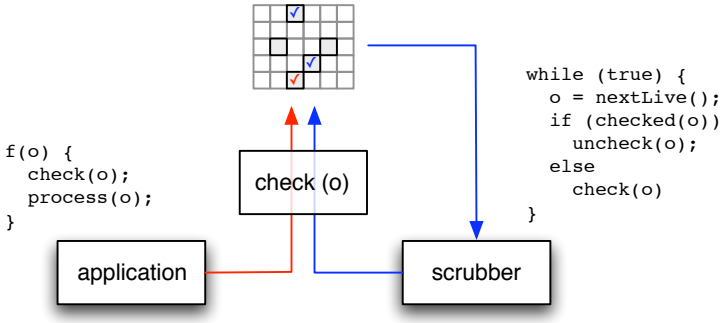

Figure 7. Integrated memory scrubber

Since SEEs are stochastic, this mechanism can be optimized by using generational data and time stamps to avoid frequent re-scanning in application loops, or delayed checks by the scrubber. While the target property is to scan all live objects within a certain amount of time, scrubber and heap overhead have to be kept low to avoid introducing new vulnerabilities.

Not all memory areas have to be periodically checked, and some critical memory objects should even be correctable in case of multi-bit errors, to avoid time consuming resets and permanent loss of data. These error conditions must be recognized as close as possible to the point where the incorrect memory value is first used, synchronously to normal program execution.

In this case granularity, i.e. frequency of checks, is the primary factor in balancing error probability and runtime costs. Corresponding methods do not only use time and space information like the intelligent scrubber, but also semantic information about relevant memory objects. Factorizing data into contiguous memory areas that can be associated to their processing functions is an important task within an RHS design.

Critical object integrity can also be checked and corrected by the application, transparently using software-only Error Correction Codes (ECCs like Reed Solomon[7]) on contiguous memory objects (an approach that is especially appealing for object oriented programming languages like C++). The amount of parity bytes can be dynamically adjusted to the respective threat scenario (for example, ramped up during solar storms or SAA fly-over). Originally developed for robust communication protocols, ECCs are a good example of reuse of existing technology in the new context of radiation hardening. This method would also handle errors in processor memory, like cache and registers, which are otherwise not amenable to scrubbers.

Processor memory errors can also cause control flow inconsistencies, detectable by techniques like control flow checking with software signatures[8]. Checks can be performed at basic block level, function calls, or dedicated check points to reduce runtime costs. Based on the target processor architecture, control flow checking overhead has to be weighted against the probability of illegal instruction exceptions (software checks for spurious jumps are not justified if the probability to reach them due to prior illegal instructions is too low).

Static memory errors (ROM) can be detected with checksum comparison prior to reads, which again depends on appropriate granularity, balancing the frequency of access with the size of the relevant object.

Checks can be type based, directly or indirectly (checksum) value based, or check state based (for example to verify protocols with automatons [17]). They can be performed explicitly upon entry and exit of certain functions, as general invariants (evaluated automatically for a set of functions) or at locations where data consistency conditions have to be met. Check conditions should be written in the most concrete way, using exact comparison for boolean and integer values where possible.

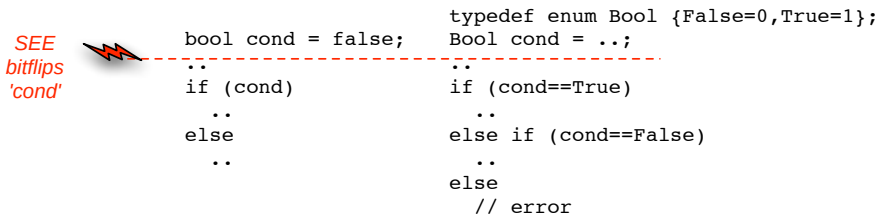

Figure 8. Concrete conditional expressions

If object oriented programming languages are used, it is strongly recommended to use Programming by Contract[14] as an inheritance-safe way to specify properties. Temporal properties like protocols can be verified with Dynamic Assertions[17] - a technique that uses pre- and post-conditions to add, evaluate and remove dedicated check objects.

It should be noted that many of the mentioned techniques also serve a dual purpose to help detect programming errors during testing, especially with respect to memory corruption of non-reference data values.

(2) **Recovery**

Once a defect has been detected, the system needs to take immediate action to get back to a safe state. This requires some type of exception handling and program state restoration (re-execution from a point with valid, consistent data), which includes the following topics:

- identification of permanent hardware damage (relevant hardware tests) and/or conditions requiring power reset, like SELs.
- robust control transfer with a fallback restart mechanism and exception "firewalls" (standard exception handling might depend on stack integrity, but a failure might lead to more severe consequences during recovery if the stack is corrupted)
- transaction-based design of applications, to identify resources that have to be released, and portions of consistent data that have to be restored in case of failure

Control flow transfer to the nearest restart point can be performed with error return values, language specific exception handling, non-local jumps, or signals. If exception handling is used, care has to be taken that a corrupted stack does not aggravate the problem. Non-local jumps and signal based exception handling can be used as a fallback control transfer mechanism.

Resource release is usually a less important problem in embedded applications than it is in multi-user/multi-tasking operating systems, unless the failed function leaves devices in a temporary state. If the number of resources is small enough, a resource list can be kept to keep track of required release actions. In case of language supported exception handling, the *resource acquisition is initialization* [15] (RAII) technique can be used.

Data restoration is preferably done by function related temporary work areas that are discarded upon failure. Using stack variables for temporary storage of changed data values is a suitable way to achieve this, leading to a functional programming style. For larger transaction scopes, software design patterns encapsulating operations into dedicated objects can be employed (for example the *Command* pattern[9]), possibly utilizing Design-for-Verification[16] (D4V) techniques.

The major challenge is to get the system back into a consistent state that requires the least amount of re-computation and restoration, i.e. minimizes the time until full operation is resumed. Time to recovery mostly depends on suitable partitioning of data areas that have to be closely associated to corresponding functions, which is analogous to transactions in database systems. Transaction based processing is likely to be the major application design factor.

(3) **Reconfiguration**

After the initial recovery is done, and the program is restarted from a safe point, actions might have to be taken to avoid follow-up defects caused by permanently damaged hardware (for example, "stuck bits" in certain memory areas). This can be done by marking and replacing unusable memory, possibly involving relocatable heap objects (i.e. pointer replacement), using similar techniques like automatic dynamic memory management systems[10]. Memory pools are also suitable to avoid permanently damaged memory areas, requiring less system overhead at a potentially increased amount of lost memory[11].

To ease reconfiguration, all required devices that can be kept redundant should only be referenced indirectly. Functions should check for the status of required devices prior to execution.

B. Flight Experiment

Past missions provide sufficient statistical data to derive realistic SEE scenarios, and some missions - like ARGOS[6] - even did test certain software failure protection mechanisms, but since the RHS project aims at an engineering solution, we have to show effective increase of overall robustness of radiation hardened flight software. In addition, we cannot assume early adopters of the RHS library in live missions without prior trust building measures. For these reasons, we intend to conduct a flight experiment within the RHS project that validates our approach.

The Gravity Probe B (GP-B) spacecraft[12], operated by Stanford University, seems to be an ideal candidate. The original GP-B science mission will be completed in September 2005, which opens up the unique opportunity to use one of the science computers (Squid Readout Electronics - SRE) for dedicated SEE tests in a real space environment. GP-Bs 640km polar orbit ideally passes through the most radiation-relevant areas (poles, South Atlantic Anomaly), which caused a high number of SEEs with a significantly larger than expected number of uncorrected multi-bit errors during GP-Bs main flight phase [6]. The spacecraft is fully operational, and the flight experiment would not use critical hardware (attitude control is performed by a different computer).

The GP-B SRE computer is equipped with 3 MB of EDAC memory in single bit detect, double bit correct mode, the bit correction being done by firmware (i.e. can be altered or switched off).

With this configuration, we will be able to measure

- SEE occurrence (single and multi-bit)
- memory scan rates (isolated and intelligent scrubber)
- number of errors corrected by scrubbers (isolated and intelligent)
- number of resets (with/without RHS library)
- number of transaction-restarts (with RHS library)
- number of undetected errors (with/without RHS library)

In addition to statistical SEE data produced during GP-Bs main operations phase[2], we intend to use also a separate set of test applications. The rationale behind this is to guarantee functional equivalence between hardened and raw variants with minimal redesign effects - measuring the runtime overhead of radiation hardening techniques in relation to robustness gains (reduction in resets and unrecognized errors) is the major goal of our project. In addition, we also want to enable efficient detection of unrecognized errors (wrong computational results) due to non-scanned memory, for which there is no statistical data.

C. SEE Software Simulator

While we expect a concept validation of radiation hardened software from the flight experiment data, differences in mission profile, spacecraft design and flight software require a more flexible way to adapt and optimize hardening techniques to specific mission requirements. To provide this flexibility, we will create a software-based simulation environment for SEEs with the following capabilities:

1. target processor modeling - to run original, executable flight software without modification of its memory layout or instruction set
2. time modeling - to simulate realistic defect rates
3. random fault injection - including
   - temporary and permanent memory faults (SEUs and SELs), affecting static-, heap- and stack- memory segments
   - register corruption
   - code corruption
4. program traces - to analyze defect propagation from injection to detection
5. exact reproducibility of defects - to generate traces and enable further defect analysis

To achieve these requirements, we intend to use an existing processor emulator that is interfaced to a debugger, and extend it by random failure injection and replay functionality. An architecture that separates program inspection (user interface) from execution (processor emulation and fault injection) is preferable to avoid redundant work. There is a sufficient number of existing hardware emulations, including all major processor architectures. The open sourced Gnu Debugger system[13] is a viable candidate for this work.

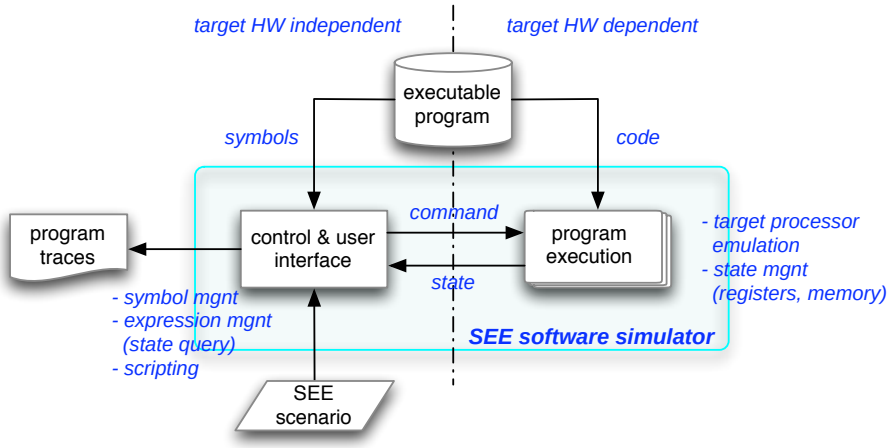

Figure 9. SEE simulator

V. Conclusions

Radiation is a serious problem for computer systems operated in space. In this paper, we have presented the Radiation Hardened Software project as a novel approach to cope with this threat. The project is still in it's initial phase, being proposed to several NASA programs. We plan for a three years duration, during which we intend to create a robust, validated radiation hardening infrastructure for flight software applications, to be used in conjunction with specific design guidelines. From an engineering perspective, this is only the first part in an continuous effort that ultimately has to include tests and recommendations regarding programming environments and operating systems that are radiation robust. We consider it to be a good starting point, since it will enable us to develop, refine and validate the software construction principles of how to make flight software resilient against probabilistic memory errors, and will give us the analysis tools (SEE simulator) to be used in the following phases.

References

[1] Kenneth LaBel et al, "SEECA - Single Event Criticality Analysis", GSFC report, Feb. 1996, http://radhome.gsfc.nasa.gov/radhome/papers/seecai.htm

[2] Brandon D. Owens, Michael E. Adams, William J. Benzce, Gaylord Green, Paul Shestople, "The Effects of Radiation Events on Gravity Probe B", *IEEE Transactions on Nuclear Science* (to be published)

[3] Satellite News Digest: "Chronology of Satellite Failures", URL: http://www.sat-index.com/failures/index.html?http://www.sat-index.com/failures/timeline.html

[4] J.H. Adams, "The variability of single event upset rates in the natural environment," *IEEE Trans. on Nuclear Science*, vol. NS-30, no. 6, Dec. 1983.

[5] S. Mitra, N. Seifert, M. Zhang, K. Kim, "Robust System Design with Built-In SoftResilience", *IEEE Computer*, Feb. 2005

[6] M.N. Lovellette, K.S. Wood, D.L. Wood, J.H. Beall, P.P. Shirvani, N. Oh, E.J. McCluskey, "Strategies for Fault-Tolerant, Space-Based Computing: Lessons Learned from the ARGOS Testbed", *IEEE Aerospace Conference Proceedings*, Big Sky, MT, 2002

[7] S. Lin, D.J. Costello Jr., "Error Control Coding: Fundamentals and Applications", Prentice Hall, 1983.

[8] N. Oh, P. Shirvani, and E. McCluskey, "Control Flow Checking by Software Signatures," *IEEE Trans. Reliability*, Mar. 2002

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns - Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1994

[10] R. Jones, R. Lins, “Garbage Collection - Algorithms for Automatic Dynamic Memory Management”, Wiley, 1996

[11] A. Alexandrescu, “Modern C++ Design: Generic Programming and Design Patterns Applied”, Addison-Wesley, 2001

[12] Gravity Probe B, http://einstein.stanford.edu/

[13] Free Software Foundation, “GDB - The GNU Project Debugger”, http://www.gnu.org/software/gdb/documentation/

[14] Bertrand Meyer: Object-Oriented Software Construction, Prentice Hall, 1988

[15] Bjarne Stroustrup, “The C++ Programming Language - Second Edition”, Addison-Wesley, 2000

[16] P. Mehlitz, J. Penix, “Design for Verification - Using Design Patterns to Build Reliable Systems”, *Proc. of 6th Workshop on Component-Based Software Engineering*, 2003.

[17] P. Mehlitz, J. Penix, “Design for Verification with Dynamic Assertions”, *Proc. of 29th Annual NASA/IEEE Software Engineering Workshop (SEW), 2004*

[18] A. Campbell, P. McDonald, K. Ray, “Single Event Upset Rates in Space”, *IEEE Transactions on Nuclear Science*, vol. 39, no. 6, Dec. 1992