JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 2, March 2005

Computing Requirements for Self-Repairing Space Systems

Phillip A. Laplante*
Pennsylvania State University, Malvern, Pennsylvania 19355

The United States recently announced ambitious plans for extended human and robotic exploration of the moon and mars. Coupled with efforts to complete the International Space Station, clearly mankind is destined to deploy increasingly more complex manned systems and for longer periods of time in Space. These systems, however, require fault-tolerant and highly adaptive real-time computer systems to control them. Fortunately, new technologies such as the field programmable gate array (FPGA) offer a flexible platform to develop robust, adaptable, self-repairing systems for space. But this new technology is itself vulnerable and therefore must be used with caution by augmenting existing practices with new techniques. The contribution of this paper is an overview and synthesis of recent developments in FPGA-based reconfigurable real-time system, and a discussion of reconfiguration issues in space applications. From these a system formalization, a proposed set of architectural and operating systems requirements, and suggestions for future work for FPGA-based self-repairing space systems are given.

I. Introduction

DESIGN of robust, fault-tolerant computer systems for space vehicles is considerably more difficult than for terrestrial ones because of the adverse operating conditions in space. These conditions are created by dramatic temperature variations, shock, vibration, electromagnetic interference, and transient failures induced by random collisions with charged particles. These rogue particles and the failures induced by them, called single event upsets (SEUs), produce effects that are similar to other kinds of transient noise, and can be either temporary or permanent. To deal with these challenges, NASA is increasingly relying on adaptable systems based on the field programmable gate array (FPGA).[1]

This paper considers the computing requirements of systems that can survive in space. The organization of this paper is as follows. First, a review of related work with respect to self-repairing systems, single event upsets, and reconfigurable architectures is presented. Next a discussion of the adverse effects of single event upsets (SEU) is given. Following is a brief discussion of the field-programmable gate array, though it is assumed that the reader already has familiarity with them. Next a review of certain experiences in fault-tolerant design for space missions is presented. Then, in this context, a set of high level hardware and software requirements for a fault-tolerant architecture for space applications is given. Finally, conclusions and suggested future work are offered. It should be noted that there are many good survey papers covering general fault-tolerant computing schemes, for example, Refs. 2 and 3. But the purpose of the present paper is to focus on the relatively narrow sub-specialty of self-repairing systems for space applications based on the FPGA.

II. Related and Previous Work

Self-repairing systems have long been sought for space applications by NASA and other space agencies. In fact, the notion of self-repair precedes the ambitions of space flight and is apparently due to Forbes and Stieglitz in the design of early computers. von Neumann, however, is attributed with proposing hardware triple redundancy, used particularly by NASA and in other mission critical applications for fault-tolerance.[4]

Received 3 May 2004; revision received 18 January 2005; accepted for publication 8 February 2005. Copyright © 2005 by the American Institute of Aeronautics and Astronautics, Inc. All rights reserved. Copies of this paper may be made for personal or internal use, on condition that the copier pay the $10.00 per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923; include the code 1542-9423/04 $10.00 in correspondence with the CCC.

*Associate Professor of Software Engineering, Engineering Division, Great Valley School of Graduate Professional Studies, 30 East Swedesford Road.

Much of the subsequent work in repairable computer systems has focused on detection and repair of discrete circuits. For example, Roth introduced two schemas for automatic detection, diagnosis and repair of failures in complex parallel computers. In the first schema, repair is done by deleting a processor in which a failure has been detected. In the second schema, individual devices are replaced with general purpose spares in a "PLA [programmable logic array] rendition of a regular design." Detection and diagnosis is done using the D algorithm, a hardware based approach for diagnosing stuck-at failures.[4] In short, this strategy involves fault identification and hot-swapping of spares.

For protection of the CPU, in particular, Dolev and Haviv introduced several self-stabilization algorithms that can handle soft errors (including SEU).[5] In their approach, they use an abstraction of the state transition graph to guarantee that any microprogram will not enter any dead states. If it does, the microprogram can be redesigned until all sequences can be shown to eventually converge to a fetch-decode-execute cycle upon experiencing a fault. Hence it can be used to insure that no microprogram can "hang-up" if affected by an SEU. Their approach can easily be adapted to FPGA configured microprocessors as well as any other device, such as A/D converters, executing a microcode program.

Laplante described the mechanisms used to protect the Space Shuttle Inertial Measurement Unit (IMU) from faults due to single event upsets. These strategies included hardware mechanisms such as error detecting and correcting codes, redundant hardware, and algorithm robustness. He also described software techniques for detecting spurious and missed interrupts due to SEU and for dealing with them once detected.[6]

Musliner et al. provided a framework for self-adaptive systems in hard real-time environments, which consists of five elements. The first is a subsystem that predictably executes real-time control plans. The second is a state-space planner and scheduler that use the system models to maintain a system "safe state." The third element is an adaptive mission planner that uses a reasoning engine to formulate long-range reconfiguration sets, absent failures. The fourth is a performance evaluator that tries to detect impending system failures and find run time efficiencies. Finally, an evaluator generator, dynamically adapts the performance evaluator. The last three elements, which are the unique contributions of this work, provide built-in model checking to optimize the task application set for a given mission phase.[7] This work is essential in the adaptive nature of the proposed system in that it provides a basis for establishing that dynamically reconfigurable task sets are possible, as well as providing a baseline onto which the fault-tolerant SEU mechanisms can be added.

In related work Walder and Platzner successfully built a prototype that dynamically loaded an existing set of networking and multimedia tasks including Advanced Encryption Standard (AES) decryption and audio stream decoding in an FPGA.[8] Their design incorporated a multitasking system functioning on relocatable hardware, and used a memory management unit to translate task requests to internal and external memory accesses. Device drivers and triggers where used to connect to external I/O. They also implemented a prototype using Xilinx technology.

Simmler et al. investigated "pseudo multitasking" in FPGA systems and showed how to share hardware resources among several tasks. This technique is similar to the multitasking on a single CPU supported by an operating system. In particular, they presented a way to enable task switching using an application independent technique to context switch using a bit stream. They also gave the requirements necessary for robust context saving and restoring for an FPGA coprocessor. Finally, an FPGA coprocessor implementation designed especially to support multitasking was described.[9]

Nollett et al. provided a theoretical framework for transparent complex hierarchical reconfiguration of an FPGA system.[10,11] They approach the problem of runtime reconfiguration by partitioning the FPGA into two tiles each acting as independently managed, run-time reconfigurable, soft computing resource. Hence, while operations are being conducted using the hardware resources on one of the tiles, reconfiguration can occur in the other. Communications between the tiles is conducted via a packet switched interconnection network. The authors note that one of the disadvantages of this approach, however, is the possibility of internal fragmentation of the FPGA space.

Burns et al. proposed the use of, in effect, an overlay manager using a virtual hardware manager, a transformation manager, a system configuration manager, and a device driver to hide the programming interface in an FPGA.[12] They proposed that the reconfiguration system for an FPGA architecture must provide the following:

1) The ability to reserve a chunk of FPGA resource.
2) The ability to transform a circuit, that is, to change its orientation or translate its position.
3) A rich high level symbolic representation for circuits that is suitable for transforming the representation into device specific programming data.
4) An architecturally correct representation of the FPGA device to support core algorithmic operations.[12]

More recently, Coyle et al. surveyed current research in self-repair of embedded systems, but specifically focused on non-real-time systems. They suggest that there are two kinds of self-repairing systems; those that provide *attributive repair*, which attempt to restore the attributes of the systems to some initial state, and those that provide *functional repair*, which attempt to restore the original functionality of the system. Their contribution also includes a UML model of some basic repair protocols, and a case study using a speech recognition system.[13]

Wigley and Kearney developed a self-repairing FPGA system using Java. They also characterized the basic services that should be provided by an operating system for reconfigurable computing: resource allocation, resource partitioning, application placement, and routing.[14] They introduced a suitable application loader and scheduler and suggested the need for an appropriate abstraction layer between the operating system and the application hardware. Finally, they offered performance metrics for the operating system. These were,

1) Execution time overhead introduced by the operating system itself.
2) Reduction in application specific performance induced by operating systems interference.
3) Fragmentation of the FPGA device as well as traditional operating systems fragmentation metrics.
4) Ease of application porting.
5) Ease of platform porting.

They also built a proof-of-concept prototype using a Celoxica FPGA board and using Java as the language for their operating system.

On the "pure" operating systems side, Mosse et al. examine the theoretical issues of missing deadlines in periodic, preemptive scheduled systems, due to SEUs and other transient faults. In particular, they present three algorithms that can tolerate a single missed task – a dynamic programming optimal solution, a near-optimal linear time heuristic, and a queue based solution.[15] However, missing a task is only one consequence of single event upsets.

Finally, the Adaptive Instrument Module (AIM), which resided in the Australian Fedsat 1 spacecraft, became the first space borne reconfigurable processor in 2002 (Ref. 16). Subsequently, a group at NASA/Johns Hopkins realized a prototype reconfigurable processing platform, Adaptive Data Analysis and Processing Technology (ADAPT), for development of adaptive functionality for space missions.[17]

Collectively, these and other related works provide a strong basis from which a set of design principles for self-repairing computers in space environments might be extracted.

III. The Threat of Single Event Upsets

Single event upsets occur in space systems when protons and heavy ions due to cosmic rays, solar flares, and passage through the van Allen belt cause a change to the state of some bi-stable device. Similar effects may also be seen in the wake of terrestrial nuclear events. Single event upsets can permanently damage a device through latch up, burnout, or gate rupture. SEUs may disrupt device operation for a few cycles, or for a single access only. In any case the effects of SEUs can hinder the correct operation of digital devices, presenting significant problems of reliable control.

The effects of SEUs include alteration of program memory, corruption of RAM or of a CPU register, and spurious or missed interrupts. These effects may cause negligible, minor, or catastrophic injury to the system, depending on when and where the bit is altered. Single event effects can also be seen in I/O circuitry, causing false or corrupted data to be input or output. As devices such as interrupt controllers, bus multiplexers, A/D converters and so forth contain registers and on-board RAM, they too are subject to SEU, though the small amount of on-board memory greatly reduces the likelihood of occurrence over short periods of time. A sampling of digital computer components and the deleterious effects of experiencing an upset are summarized in Table 1.

Devices have varying susceptibilities to upset based on part geometry, logic levels, and manufacturing process. The upset rates are not inconsiderable – rates of >3 upsets per million bits per day are not uncommon for certain SRAMs.[18]

Device manufacturers can harden parts by exposure to low level radiation to liberate any radioactive impurities that could create offending particles. However, this treatment diminishes the part's low-level radiation longevity, and does not eliminate the possibility of SEU altogether. Moreover, hardened parts are difficult to obtain. Shielding can be used to protect parts from SEU, but this again only lowers the upset rate, and the extra weight and volume of the shielding is a high price to pay. Therefore, for long duration missions (e.g. one or more years), where the risk of upsets increases linearly over time, the effects cannot be ignored. The most effective solutions, however, involve a combination of hardened parts and hardware of hardware and software fault-tolerance mechanisms.[19]

Table 1 Potential adverse effects of device failure due to single event upset or other transient failure

Device	Possible Adverse Effects
clocks	• false pulses • missed pulses
FPGA	• any effect possible depending on how device is configured
I/O circuitry	• bad data • missing data
interrupt controller	• spurious interrupts • missed interrupts • misprioritized interrupts
memory	• program corruption • data corruption
microprocessor other microprogrammed devices program counter	• device hang-up • program misexecution • jump program • mis-execute instruction • bad operand • latch-up

A variety of techniques employing hardware, software, or both have been developed to enable detection and recovery from an SEU. These schemes include use of error detection and correction chips, RAM scrubbing, coding, and robust algorithm design. A unique set of these and other techniques were developed for use on the Space Shuttle Inertial Measurement IMU computer and will be described later. But the capabilities of the FPGA suggest additional strategies for dealing with the SEU.

IV. Field Programmable Gate Arrays

Systems based on field programmable gate array (FPGA) technology are quickly replacing those based on application specific integrated circuits (ASIC). Field programmable gate arrays embody hundreds of thousands of reconfigurable gates per device that can be integrated to form system level solutions. SRAM based FPGAs are reprogrammable (even within the system) allowing rapid reconfiguration. Reconfiguration involves resetting the flexible hardware architecture into a new configuration. A formalized definition of reconfiguration is given in section 6.2.4.

Most FPGAs can be fully or partially reprogrammed, allowing for reconfiguration of selected areas of the device and leaving other areas intact. In space applications, in particular, the high densities and in-situ reprogrammability of SRAM based FPGAs are ideal.

The FPGA is not, however, without its disadvantages for use in high performance space applications. The time required to configure a device is not negligible, for example in the ADAPT system, approximately 1 second (Ref. 17). Moreover, the maximum FPGA device speed still lags behind that of ASICS and other silicon solutions. FPGA power consumption is also higher than that of similarly configured ASIC devices.[20]

Perhaps more importantly, a major challenge to FPGAs in space applications is their enhanced susceptibility to single event upsets. SRAM based FPGAs, particularly, are volatile devices and therefore are highly susceptible to single event upsets, even if they are radiation hardened.†[21] FPGAs are designed to be reconfigured and, unlike fixed devices, unused paths between components are left in place. These paths can then be energized through the long term effects of total ionizing radiation doses or by single event effects (collisions with charged particles In short, the strength of the FPGA, its easy reconfigurability, also is the source of its weaknesses. Still, the power of the FPGA is sufficiently great, that its weaknesses must be overcome.

†One NASA presentation jokingly notes that radiation hardened FPGAs are used as particle detectors.[19]

To cope with the SEU problem, FPGA manufacturers provide specific functionality. Xilinx, for example, provides an interface to its Virtex II series of FPGAs that confers special post-configuration operations for SEU mitigation. These operations allow for SEU detection and repair or reconfiguration of selected portions of the FPGA during continuing operation. In addition, a kind of RAM scrubbing operation is supported in which a particular logic block can be reloaded.[22] Here, by selectively scrubbing portions of the FPGA, severe SEU effects can be avoided.

V. Lessons Learned From Other Space Missions

The requirements for self-repairing systems go beyond mitigation of SEUs, because as noted, space missions are susceptible to other kinds of hazards, exacerbated by the remote and autonomous nature of the vehicles. To illustrate these hazards, consider three very different space missions. The first reviews the SEU protection mechanisms used in the Space Shuttle Inertial Measurement unit (IMU) control system. The second two cases describe software failures in recent Mars missions that, while unrelated to SEUs, highlight additional design challenges for the designer of space borne systems. The mention of these two systems further highlights a related constraint – NASA has mandated the use of off-the-shelf software where possible. In the two latter cases, shortcomings in the off-the-shelf software led to the system failures. In at least the latter case, the survivable FPGA based system might have overcome the weakness in the commercial software used.

A. Space Shuttle IMU Fault-Tolerant Design 1984

In the Space Shuttle IMU several strategies are used during system start up, in foreground processing, and in background processing to enhance fault-tolerance to SEUs and other transient hardware failures. Central to these strategies is the use of an error detection and correction chip (EDAC) based on a Hamming code. Using six syndrome bits to protect each 16-bit word, the device can detect and correct all single bit errors on the bus. It can also detect all 2-bit errors (and correct some) and even some 3-bit errors. Chips that use Reed-Solomon block coding can detect and correct multiple bit errors, but with more significant power and storage overhead. Software implementations of Reed-Solomon codes can also be used, but with significant processing overhead.[18]

The Hamming based EDAC chip does not correct errors in memory; they are corrected on the bus as they are fetched. Thus, in order to permanently correct the error in memory, the information, and the correct syndrome, is written back. This process is known as RAM scrubbing.

During system start up several tests are run as part of the built-in software test () suite. For example, a checksum is stored with the ROM based executable program and is recomputed to check for errors. The RAM area of memory is fully tested by writing and reading back, several patterns of bits into each cell. Then, the error detection and correction chip is fully tested. Finally, start up BIST of all system hardware is performed and the interrupt cycle structure and variables are initialized.

In each of the five periodic threads the associated interrupt signal needs to be validated before being serviced in order to avoid processing false interrupts. Accordingly, a redundant command (either a CPU readable discrete signal or memory-mapped input) is generated along with each interrupt request by the interrupting source. Each interrupt handler then checks the corresponding redundant signal when responding to an interrupt. If the interrupt is spurious, control can quickly be returned to the interrupted thread. Extra run time stack storage was provided to allow for such an occurrence. Similarly, any induced state where the interrupt controller cannot honor interrupts must be avoided. This is accomplished by continually refreshing the interrupt mask register and by continually re-enabling interrupts.

Missed interrupts are another potential SEU induced problem, which can be mitigated several ways such as by using faster rate threads to count the necessary interrupts in lower rate threads. In any case, the software design must be such that missing an occasional interrupt will not seriously compromise system performance.

SEUs can also cause the interrupt controller to incorrectly prioritize interrupts. Here the problem is solved by continually saving a copy of the previous interrupt status register in RAM. When an interrupt is serviced, the current contents of the status register are compared to the previous contents, which should always be lower. Any anomalies can be detected, and erroneous processing avoided. Finally, a foreground process monitors the "error detected" flag from the EDAC chip, and a count is kept and saved for post flight analysis.

While executing in background, several operations are performed to help minimize the potential effects of an SEU. First, as in initialization, a ROM checksum is computed to detect any damage to the program area of memory. RAM scrubbing is also performed, which protects against the small possibility that two SEUs occur in the same cell over the life of a mission -- an error which cannot always be corrected by the EDAC circuitry.[6]

B. Mars Pathfinder Rover Vehicle 1997

A notorious software failure occurred in a NASA unmanned space vehicle in 1997. The case involved the Mars Pathfinder Space mission's Sojourner rover vehicle, which was used to explore the surface of Mars. In this incident the MIL-STD-1553B information bus manager was synchronized with mutexes. Accordingly, a meteorological data gathering task that was of low priority and low frequency blocked a communications task that was of higher priority and higher frequency. This infrequent scenario caused the system to reset – a watchdog timer, designed specifically to protect against this kind of deadlock, overflowed causing the general system reset interrupt to be issued.[23]

The problem, however, would have been avoided if the priority inheritance mechanism provided by the commercial real-time operating system had been used. But it had been disabled. Fortunately, the problem was diagnosed in ground based testing and remotely corrected by re-enabling the priority inheritance mechanism.[24]

C. Mars Spirit Rover Vehicle 2004

A second high profile failure occurred in 2004, once again involving a Mars rover vehicle (named Spirit). This vehicle ran a special version of the popular VxWorks real-time operating system on a radiation hardened RAD6000 processor. In this case, communications with the rover was lost for several days while it was on the surface of Mars.

The cause of failure was file clutter in the on board flash memory. This, in turn, reduced the space available for the operating system, which had a very large footprint of 32 megabytes[‡], so that it could not load. The resultant fault triggered a system reset, but upon resetting, the operating system again failed due to insufficient flash memory. This scenario repeated until the batteries were drained sufficiently to trigger a "safe mode." Fortunately, the safe mode reboot loaded a subset of the operating system, which was able to load, run, and eventually allow command signals to be sent to the rover to clean up the file system, finally allowing a full operating system reboot.[25]

This very unfortunate situation might have been avoided in a self-repairing operating system if it had been able to recognize that certain files stored in flash memory were relatively unimportant, at least with respect to loading the full operating system.

The two Mars rover failures illustrate some of the difficulties in reliable control of space vehicles and highlight the need for robust, self-reparable systems. The self-repair aspect is crucial because, while in each case the situation that caused the failure was remedied, it would not have turned out so well had the vehicle been manned – the numerous systems resets that eventually resolved the problem might not have been possible. A more sophisticated solution would have involved system recovery without shutdown of any kind.

VI. Requirements for a Fault-Tolerant, Self-Repairing Architecture

Based on the peculiar environment of space, the characteristics of one main threat to the system (SEUs), the nature of FPGAs, and the lessons learned from past space missions, requirements for a fault-tolerant, self-repairing architecture are now given. These are organized as SEU protection mechanisms, a self-repairing hardware architecture, and operating systems requirements. The operating systems requirements encompass traditional requirements for embedded, hard, mission critical systems with the addition of special considerations due to the reconfigurable hardware.

A. SEU Protection mechanisms

Clearly radiation hardened devices and shielding should be used whenever possible in fault-tolerant design of space systems. But in addition, the mechanisms adopted in the Space Shuttle IMU, which were reviewed in section 5.1, provide a suitable baseline for protecting any space borne system, but not without costs.

For example, all random access and read only memory are protected by bus-corrected error detection and correction. This process detects and corrects the mostly likely SEU induced errors on the bus, but at the expense of slightly increased memory access times. In the case of the random access memory, scrubbing can be used to correct any errors detected in random access memory, which costs additional processor time. CPU latch up can be detected through a watch dog timer feature. But the costs for this protection are negligible increases in processor utilization and in power, board area, and weight due to the timer device. Self-stabilizing design using Dolev and Haviv's approach can be used to ensure that no FPGA based microprocessor or microprogrammed device will enter a death spiral.[5] Finally, spurious interrupts, missed interrupts, and mis-prioritized interrupts are handled through the use of redundant signaling but the cost for these redundancies is additional processing time, which though small, occurs during interrupt service routines and while interrupts are disabled, and therefore are non-negligible.

[‡] The real-time operating system for the Space Shuttle Inertial Measurement Unit required less than 64 kilobytes of memory.

The baseline SEU protection mechanisms for space borne FPGA systems and the costs of implementation just described are summarized in Table 2.

Table 2 SEU protection mechanisms and their cost (those marked with a "*" are negligible)

Adverse Effect	Remedy	Cost
corruption of RAM data	• EDAC chip • RAM scrubbing •	• increased memory access times • increased processor utilization •
corruption of ROM data	• EDAC chip • CRC checksum •	• increased memory access times • none •
corruption of PC	• none •	• none •
CPU latch-up	• watchdog timer •	• increased power, board area, weight* •
I/O circuitry	• none •	• none •
spurious interrupts	• confirmation flags •	• increased interrupt response times •
missed interrupts	• watchdog timer • counters •	• increased power, board, weight* • none •
misprioritized interrupts	• redundant status register check •	• increased interrupt response times* •
hang-up of microprocessor or other microprogrammed devices	• self-stabilization	• none (design time costs only)

It is noteworthy that in the Space Shuttle IMU system that the I/O circuitry and program counter did not have any explicit protection, largely because of the unavailability of a suitable hardware solution at the time. Implicit protection could have been provided through robust algorithm design. However, even such techniques as Kalman filtering or the rejection of statistically outlying inputs, for example, would eventually degrade if the I/O device was permanently damage. It is here that the reconfigurability of the FPGA can be used to enhance fault-tolerance.

B. Self-Repairing Space Systems Architecture

Fault-tolerant systems for space applications require more than SEU protection; they must be able to adapt to changing mission requirements and respond to system threats. The failures of the two Mars rover vehicles highlight the need for an adaptive fault-tolerant architecture that does not rely on resets to deal with hazardous situations.

Traditional real-time operating systems can provide a measure of reconfigurability by rearranging the task set. But FPGA-based systems can provide more flexibility through hardware reconfiguration.[22] Hence, while the Mars Pathfinder rover problem was solved by an operating systems adjustment, the Mars Spirit rover, which required more memory to avoid resetting, might have been able to obtain that memory in a reconfigurable system.

1. Hardware architecture

According to Wigley and Kearney a reconfigurable system must provide resource allocation, resource partitioning, application placement, and routing.[26] In the proposed architecture, these functionalities are provided by a FPGA subsystem, which reconfigures the hardware and software from an archive in response to mission events. The hardware subsystems are configured and the appropriate programs are loaded into the hardware subsystem by commands generated by hardwired logic embedded in the FPGA subsystem. The FPGA chip within the subsystem can be viewed as one or more *regions of interest*, which contain modules that are configured as appropriate devices or memory stores

The basic unit of reconfiguration is a *module*. A module contains one or more bitstreams that can configure a particular region of interest of the FPGA into devices. In terrestrial FPGA applications a non-volatile device such as flash memory contains the configuration bitstream for each device. Upon startup, a boot strap loader configures the volatile FPGA from the non-volatile store.

In the proposed architecture, a similar strategy is used. A radiation hardened, hard-wired boot strap loader loads modules from an archive in the secondary store. The archive contains both device configuration bitstreams and programs in executable form for the operating system and applications programs. To conserve board space and power, the contents of the archive are stored in a compressed form using a lossless compression scheme such as LZ77, which has been successfully employed in terrestrial FPGA applications.[27] Therefore, a decompressor works in conjunction with the boot strap loader (Fig. 1).

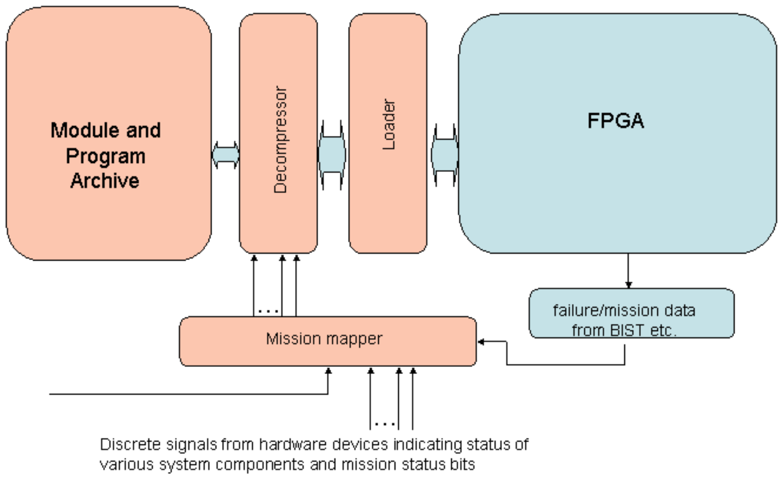
Module and Program Archive
Decompressor
Loader
FPGA
failure/mission data from BIST etc.
Mission mapper
Discrete signals from hardware devices indicating status of various system components and mission status bits

Fig. 1 High level view of the FPGA subsystem.

The archive is radiation hardened memory and protected with extra bits for an error detection and correction scheme such as the Hamming or Reed-Solomon codes.

A mission mapper, which is also a radiation hardened combinational logic circuit, is responsible for driving the reconfiguration of the system hardware and software based on various systems parameters.

Consider the following formalization of this architecture. Let R be the set of modules stored in the archive, where $R = \{R_1, R_2, ..., R_k\}$ (see note[§]). The compressed configuration bitstreams for all modules are denoted $R_1', R_2', ..., R_k'$. One of these modules, R_c, contains the bitstream configuration description of a soft processor. Others contain bitstream descriptions of I/O devices, memory management unit, interrupt controller, etc. Memory storage is configured in modules $R_{m_1}, R_{m_2}, ..., R_{m_l}$, with $l < k$. For convenience of notation we denote the memory modules $R_{m_1} \equiv M_1, R_{m_2} \equiv M_2, ..., R_{m_l} \equiv M_l$. For fault-tolerance it is highly desirable to separate operating system and application memory blocks and even blocks of memory between applications to prevent unprotected access.

As an example, consider a system consisting of the set of modules, *R*, depicted in Table 3.

[§] Except in the case where individual signals are involved, we use upper case letters with subscripts to denote the elements of this set and like sets because the elements themselves are sets.

Table 3 Partial set of modules for a reconfigurable space system.

Modules	Device	Memory provided
$R_1 = R_c$	CPU	NA
R_2	MMU	NA
R_3	Bus Mux	NA
R_4	A/D converter	NA
$R_5 = M_1$	Memory block 1	8KB
$R_6 = M_2$	Memory block 2	8KB
$R_7 = M_3$	Memory block 3	16KB
$R_8 = M_4$	Memory block 4	16KB
$R_9 = M_5$	Memory block 5	16KB
$R_{10} = M_6$	Memory block 6	32KB
R_{11}	Special purpose device 1	NA
R_{12}	Special purpose device 2	NA
R_{13}	Special purpose device 3	NA
R_{14}	Special purpose device 4	NA

Table 3 is incomplete in that there are numerous other devices that might need to be configured, such as interrupt controllers, timers, and so on. But Table 3 is provided as a simplified example to be described later.

2. Fault-tolerant bus architecture

In addition to protecting data in all bistable devices, data must also be protected as it moves through the system. Therefore, all internal and external busses employ a fault-tolerant bus architecture. It has been shown that an error detection system based on detecting invalid Manchester encoded data (a parity error) and then retransmittal is an effective scheme for EDAC on the bus.[18] If a reverse retransmission request channel (Fig. 2) is used for retransmission of any data that has been corrupted on the bus, then it will not place an undue burden on the more frequently used forward channel.[28]

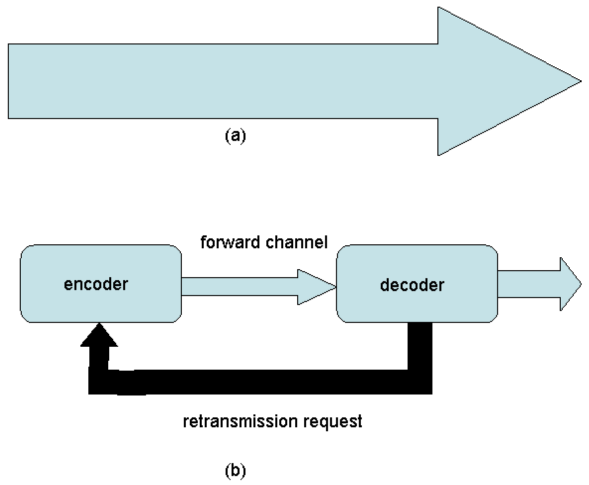

Fig. 2 (a) One direction of a fault-tolerant bus connection. (b) Detail showing coding system and a reverse retransmission request channel.[28] (Q3)

Another strategy that has been used for SEU tolerance involves optical busses. The SEDS MIL-STD-1773 fiber optic bus is a commercially available, SEU invulnerable bus based on optical transmitters and receivers. Such an approach has been used successfully in space.[18]

Depending on cost and other architectural considerations, either technique can be used to provide a robust, noiseless bus structure for communicating with all devices in the fault-tolerant systems.

3. Program storage

It is helpful to formalize some aspects of program storage in the archive. Let $P = \{P_1, ..., P_t\}$, be the set of all program code in uncompressed form. The set $P' = \{P'_1, ..., P'_t\}$ holds the compressed program files). Since the program code is divided into various operating systems and applications program code units, for convenience, we use mnemonics to represent the P_i. That is, suppose that the mnemonics OS_y, *and* APP_x represent the operating systems code and applications programs code respectively, then we have $P_1=OS_1, P_2=OS_2, ..., P_n=OS_n, P_{n+1}=APP_1, \ldots$ and so forth. To illustrate, consider the partial program set, *P*, depicted in Table 4.

Table 4 Partial set of programs for a reconfigurable space system.

Program	Mnemonic	Description
P_1	OS_1	Operating system code segment 1
P_2	OS_2	Operating system code segment 2
P_3	APP_1	Navigation
P_4	APP_2	Telecommunications
P_5	APP_3	Self-diagnostic
P_6	APP_4	Calibration
P_7	APP_5	Non-critical application 1
P_8	APP_6	Non-critical application 2

If program P_i resides in memory block M_j it is denoted $P_i \rhd M_j$. If the operator $|\ |$ denotes the size of, in bytes, of the operand (which can be either the size of a program P_i or the capacity of memory M_j), then clearly we must have $|P_i| \leq |M_j|$. This property can be used by the operating system during reconfiguration to determine if a program can fit into available memory, much as a traditional operating system would do page-frame fitting.

4. Reconfiguration capability

Based on the state of the system (including any known system damage) and the current mission phase, the mission mapper selects, decompresses, and loads the correct set of modules to be configured in the appropriate regions of the FPGA. Subsequently, the correct program set is selected, decompressed, and loaded into the memory modules that have been previously configured.

Suppose the *status set* $S = \{s_1, s_2, ..., s_v\}$ represent discrete signals from various devices or fault signals that can be set under program control by the various BIST programs. Then the mission mapper is a triple $\mathbf{M} = \{R, P, S\}$, which maps the current configuration and programs into a new one based on the status set. That is

$$\mathbf{M}: R \times R \times S \to R \times P$$

As an illustration, consider an initial system configuration consisting of a soft processor, memory management unit, memory to hold the operating system, memory to hold the program and data for application configuration "A", I/O devices, and scratchpad memory. Then the appropriate memory modules APP_{A1}, APP_{A2}, etc. would be loaded into the application memory store. A portion of the available FPGA may be unused and is left as spare for reconfiguration if needed (Fig. 3).

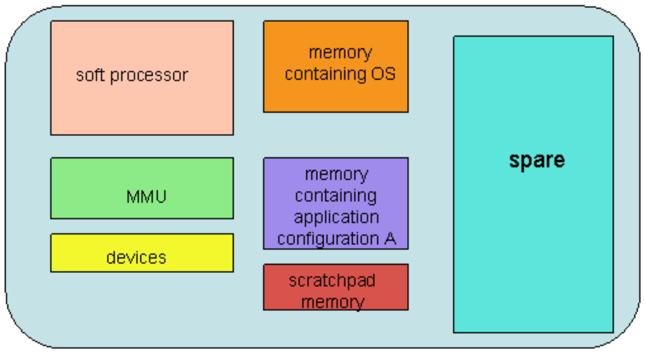

Fig. 3 Mission configuration "A" showing the regions of interest configured as devices based on the appropriate set of modules. The spare region is not configured.

Now suppose a new application set, configuration B containing code modules APP_{B1}, APP_{B2}, etc. is loaded. Such a reconfiguration due to a mission phase change (e.g. from launch to orbit) might be as shown in Fig. 4.

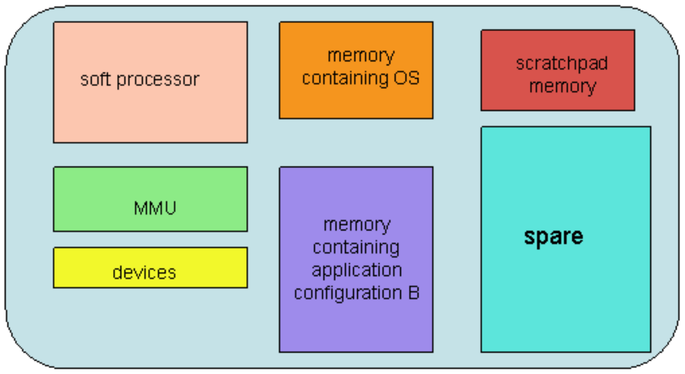

Fig. 4 Mission configuration "B."

The mission mapper can be implemented as a state machine. For example, consider the set of systems modules in Table 3, the set of programs in Table 4, and a set of mission phases corresponding to; calibration, diagnostic, pre-launch, launch, orbit, reentry, landing, and survival phases. In this case, the status set, S (omitting fault signals) represents the set of mission phases (calibration, diagnostic, and so forth). A corresponding mission mapper state machine in tabular form might appear as in Table 5.

Table 5 Partial mission mapper table (state machine) for a reconfigurable space system.

Mission/ Status Set	**Modules**								**Programs**								**Memory Blocks**				
	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	M_1	M_2	M_4	M_5	M_6
Calibration	■	■	■	■	■			■	■				■	■		■		■	■	■	■
Diagnostic	■	■	■	■		■			■	■	■	■	■	■	■		■			■	■
Pre-launch	■	■	■	■		■	■		■	■	■	■	■	■		■				■	■
Launch	■	■	■	■		■	■	■	■	■	■	■	■	■	■		■		■	■	■
Orbit	■	■	■	■		■			■	■	■	■	■	■		■	■	■	■	■	■
Reentry	■	■	■	■	■	■	■	■	■	■	■	■	■	■		■		■	■	■	■
Land	■	■	■	■			■		■	■	■	■	■	■	■	■		■		■	■
Survive	■	■	■	■					■		■	■								■	■

5. *Reasoning system*

The reasoning component of the mission mapper manages the configuration of all or part of the FPGA and must deal with a number of issues. This reasoning system is triggered by any change in the status set, either by an external command for change in mission phase or by an internal fault signal generated by the software built-in test.

First, the reasoning system must determine the correct set of hardware modules to be configured (via a hard wired finite state machine as in Table 5). Next, it must decompress and configure these components in the FPGA.

The mission mapper configures hardware modules in order of precedence so that the most important devices are configured first, followed by memory modules, which are configured as space permits. If sufficient configurable FPGA space is not available for either the requisite devices or minimal memory, then the mapper must attempt to load the survivability configuration. If this can not be loaded, then there will be a total systems failure. In this case a full reboot may be the only hope of reconstituting the system.

Suppose that S_i is an instance of the status vector, S, $R_j(i)$ the *j*th hardware module and $M_k(i)$ the *k*th memory module associated with S_i. Then the operation of the reasoning system can be illustrated by the following pseudo-code.

```
if state $S$ changes then
    for all $R_j(i)$ in $S_i$.                        // configure HW modules
        if $R_j(i)$ fits in available FPGA space
            configure $R_j(i)$
        else
            load survival configuration
    for all $M_j(i)$ in $S_i$.                        // configure memory
        if $M_j(i)$ fits in available FPGA space
            configure $M_j(i)$
        else
            if minimum memory not configured
                load survival configuration
```

Accordingly, the mission mapper circuit would be driven by signals designating the mission mode, hardware or software induced faults, and configure the hardware and software as determined by the logic of the table.

C. Enhanced Fault-Tolerant Services Promoted by Reconfigurability

At a minimum, a real-time control system for space applications should be able to tolerate both single event upsets and other kinds of transient failures using the redundant command mechanisms previously described. However, there are other protection mechanisms and synergies afforded by the FPGA that the operating system should exploit.

1. *Basic software requirements*

Certain software protection mechanisms should be employed to avoid unsafe states through interrupt controller protection, memory protection, and so on. Little research exists in this regard. However, several existing techniques might be adapted for this setting. For example, a modified Banker's algorithm could be used to detect when resources, such as memory, are becoming unavailable and will then cause the mission mapper to reconfigure the system accordingly. Such an approach might have proved successful in the Mars Spirit rover.

Another approach might use on-line model checking or software black boxes to collect execution time data that might indicate the need for reconfiguration. Nollett *et al* have already demonstrated the proof of concept for transparent complex hierarchical reconfiguration of a system for the FPGA.[10,11] Musliner et al. had proposed the adaptive mission planner and built-in model checker to ensure safe state behavior.[7]

2. *Determining when SEU errors occur and dealing with them*

Built in software testing can determine if an SEU has occurred and suggest an appropriate reconfiguration. For example, in an I/O device, it may be reasoned to be damaged if it outputs exhibit certain statistical properties, or if the device itself indicates a stuck-at or other persistent failure. In this case the I/O device is a candidate for reconfiguration.

In another case, if the BIST determines that the interrupt controller is latched up, or if spurious or missed interrupts are determined to be occurring too frequently, the interrupt controller may be reconfigured.

Finally, if the EDAC chip is issuing too many illegal instructions faults are issued by the processor, it might be likely that either the program counter or processor itself has been damaged, therefore, the processor might be reconfigured.

In any of these situation the operating system would throw an exception, set a particular bit in the status set S, and trigger an appropriate reconfiguration (Fig. 5) and as described in section 6.2.5.

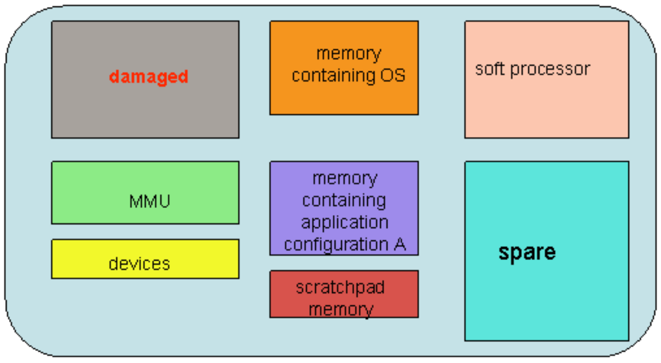

Fig. 5 A repaired configuration showing the soft processor remapped to another partition in the FPGA.

Other processors on board the system would need to take over essential functioning during the reconfiguration period. This would require extensive cooperation between operating systems amongst all of the system's processing subsystems, and is an area of open research.

Such system configuration might be realized as an orthogonal and parallel architecture employing three or more FPGAs attached to a common, but segmentable bus. In such a configuration one or more of the devices could take turns running different portions of the mission code while a "mission idle" FPGA could be designated the "surveyor". It would be loaded with code that would allow it to check the configuration of the other devices while they were running. The surveyor device could also load a test configuration into an idle device and then provide it with test inputs to check the completed functionality of the device and note failed configurable logic blocks or other components of the device. Such potential failures could be rechecked by another device before being permanently marked bad in the configuration store. Devices which were found to have completely failed could be isolated from the bus. Surveyor functionality would rotate between devices allowing a full periodic system check of the FPGAs and possibly their attached peripherals.

3. Survival mode

If so much of the FPGA is destroyed that a complete system cannot be configured, then if possible, a "survival mode" needs to be realized. Such a mode might allow for the support of some minimal set of hardware, operating system kernel, and applications software for one or more critical applications (e.g. life support, navigation, and communication), which might be sufficient to ensure the safe return of the vehicle and crew (figures 6 and 7).

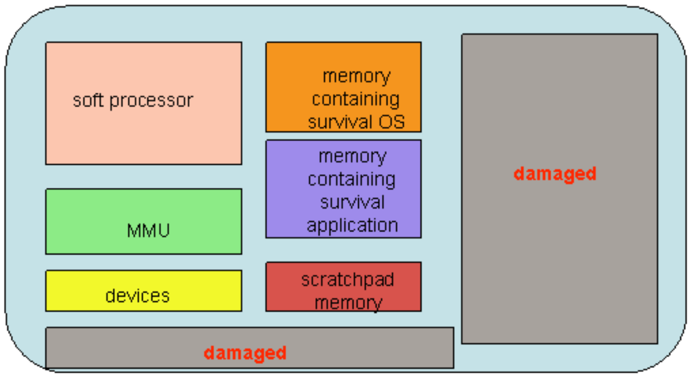
soft processor
memory containing survival OS
memory containing survival application
damaged
MMU
devices
scratchpad memory
damaged

Fig. 6 A damaged system in which only the basic hardware system is configured.

This survival version provides only minimal functionality sufficient to load the survival mode operating system and applications.

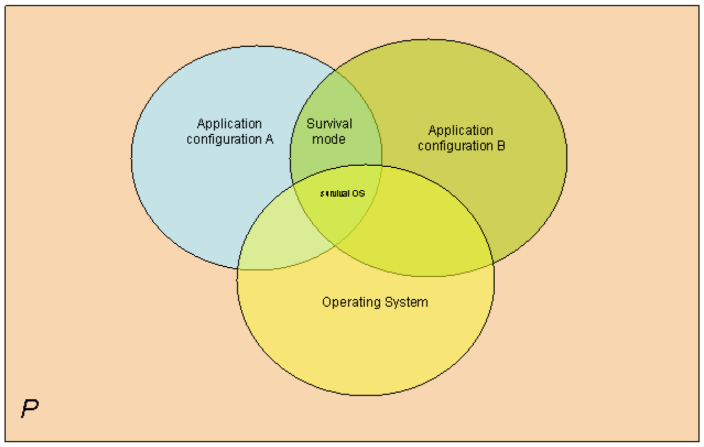

Fig. 7 The space of program code *P* showing two possible application configurations, operating system, essential operating system code, and survival mode system code.

Finally, in the case when the undamaged FPGA is insufficient to load the survival mode hardware then the system will have failed. In this case fault-tolerance would need to be provided by system duplication or through systems that could be reconfigured to replace the damaged system.

D. Real-Time Operating System Requirements

The operating systems requirements for FPGA based systems incorporate both traditional services and enhanced fault-tolerant services enabled by the hardware reconfigurability. A typical real-time operating system needs to provide a robust task scheduling and dispatching capability, intertask communications and synchronization mechanisms, and support for embedded devices. For hard real-time applications, the operating system must also provide predictable and bounded response times.

There are three strategies for providing real-time operating systems for FPGAs:

1) Build the operating system to run directly on the hardware (e.g. RealFast Corporation's Sierra 16® hardwired operating system for Xilinx devices)

2) Build the operating system to run on a soft processor (e.g. Xilinx's XMK® native operating system that runs on their MicroBlaze® soft processor).

3) Port a non-FPGA real-time operating system to run on an FPGA soft processor (e.g. Wind River Systems' VXWorks®, QNX's Neutrino OS®, Express Logic's ThreadX®, and Micrium's μ C/OS-II®).

Hardwired operating systems are the fastest but are generally the least flexible, and hence, adaptable. Commercial real-time operating systems that have been built for non-FPGA environments are feature rich and have a large user base, but their footprints are too large for most embedded space applications. Finally, most commercial real-time operating systems are cumbersome and contain far more functionality than is actually used. Therefore the second approach, to build an operating system to run on a soft processor, seems ideal for space vehicles because it allows for more easily built hardware interaction than the other two approaches.

The details of designing real-time operating systems for an embedded platform, such as a soft processor are well known[29] and need not be reviewed here. However, it is worth noting some basic principles for building such an operating system for hard, embedded applications.

1) Interrupt driven scheduling mechanisms for embedded real-time systems can be round robin, preemptive, or both. But for the most responsive and flexible system, both round robin and preemptive should be provided (e.g. tasks at same priority execute in round robin fashion). For adaptive purposes, the scheduling discipline should be such that it may be changed at any time.

2) Supported communications mechanisms should include semaphores and message queues, dynamic memory allocation (e.g. program overlaying), semaphores and message queues, and mailboxes.

3) Dynamic prioritization should be supported to deal with changing mission requirements, which can be handled by adjusting a register in the interrupt controller.

4) Multiple timers and external device interrupts should be supported.

5) The Priority Ceiling Protocol should be supported to prevent priority inversion and deadlocks.

6) A small footprint for the operating system should be sought.

This is by no means an exhaustive list, but is provided as an example of the principles to be embodied.

VII. Conclusions

In this paper, an overview of issues and history of fault-tolerant design for reconfigurable computing using FPGAs for space vehicles was given. From lessons learned from past space missions, architectural requirements for an adaptive hardware and software system were given. These recommendations incorporate a reconfigurable hardware platform based on the FPGA, SEU protection mechanisms, a fault-tolerant bus structure, and a mission driven reconfiguration scheme.

The proposed architecture does not require redundancy, and therefore is space, power, and time efficient. Alternatively, this architecture can be used to reconfigure a non-critical subsystem to take over for a more critical one; provided that they can communicate over a common bus structure and all input and output signals are available. Moreover, the recommended scheme can be used both as a survivability strategy and as a mission optimization strategy – only those mission components that are needed are configured at a given mission stage. This self-adaptive approach is useful not only in space missions but in terrestrial applications where nuclear events can create SEU-like effects.

Significant work needs to be done in further exploring adaptive operating systems that can make use of the reconfigurable hardware in ways that go beyond traditional operating systems protections. These areas of research have been mentioned previously. Moreover, though the fault-tolerance afforded by soft computing techniques and traditional distributed computing protection schemes have only been mentioned in passing here, they provide additional techniques that can and must be incorporated into any overall reconfigurable and survivable architecture.

Moreover, this paper has focused on the operation of one configurable subsystem. It has been suggested, however, that cooperation among multiple reconfigurable systems is needed to provide additional fault-tolerance and load sharing between critical and non-critical systems. Finally, further work in this area should be aimed at developing and testing prototypes of the proposed architecture under simulated operational conditions prior to deployment in unmanned vehicles, and eventually, manned space vehicles.

Acknowledgments

I would like to thank my graduate student, Steven Boyd for contributing to this work through prototyping and reading drafts of this paper. Thanks also to Kim Fowler of the Johns Hopkins University Applied Physics Lab for reading a draft of the paper and for bringing my attention to the AIM and ADAPT projects. Finally, thanks to the reviewers and associate editor for providing significant guidance in improving the paper overall.

References

[1]Stoica, Adrian, Keymeulen, Didier, and Lohn, Jason (ed.), *Proceedings of the First NASA/DOD Workshop on Evolvable Hardware*, IEEE Computer Society Press, 19-21 July 1999.

[2]Nelson, V. P., "Fault-Tolerant Computing: Fundamental Concepts," *Computer*, Vol. 23 , No. 7 , July 1990, pp.19-25.

[3]Gärtner, Felix C., "Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments," *ACM Computing Surveys*, Vol. 31, No.1, March 1999, pp. 1-26.

[4]Roth, J. Paul, *Mathematical Design*, IEEE Press, Piscataway, NJ, 1999.

[5]Dolev, Shlomi, and Haviv, Yinnon A., "Self-Stabilizing Microprocessor - Analyzing and Overcoming Soft-Errors (Extended Abstract)," Architecture of Computing Systems 2004, Lecture Notes in Computer Science 2981, Springer-Verlag, 2004, pp. 31-46.

[6]Laplante, Phillip A., "Fault-Tolerant Control of Real-Time Systems in the Presence of Single Event Upsets," *Control Engineering Practice*, Vol. 1, No. 5, Oct. 1993, pp. 9-16.

[7]Musliner, David J., Goldman, Robert P., Pelican, Michael J., and Krebsbach, Kurt D., "Self-Adaptive Software for Hard Real-Time Environments," *IEEE Intelligent Systems*, July/Aug. 1999, pp. 23-29.

[8]Walder, Herbert, and Platzner, Marco, "Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations, *Engineering of Reconfigurable Systems and Algorithms 2003*, 2003, pp. 284-287.

[9]Simmler, H., Levinson, L., and Männer, R., "Multitasking on FPGA Coprocessors," *Proceedings of the 10th International Conference on Field Programmable Logic and Applications*, Aug. 2000, pp. 121-130.

[10]Nollet, V., Mignolet, J.-Y., Bartic, T. A., Verkets, D., Vernalde, S., and Lauwereins, R., "Hierarchical Run-Time Reconfiguration Managed by an Operating System for Reconfigurable Systems," *Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms,* June 2003, pp. 81-87.

[11]Nollet, V., Coene, P., Verkest, D., Vernalde, S., and Lauwereins, R., "Designing an Operating System for a Heterogeneous Reconfigurable SoC," *Proceedings of the Reconfigurable Architectures Workshop 2003* [CD-ROM], April 2003.

[12]Burns, J., Donlin, A., Hogg, J., Singh, S., and de Wit, M., "A Dynamic Reconfiguration Run-Time System," *Proceedings of the IEEE Symposium Field-Programmable Custom Computing Machines 1997*, IEEE Press, April 1997, pp. 66-75.

[13]Coyle, E. A., Maguire, L. P., and McGinnity, T. M., "Self-Repair of Embedded Systems," *Engineering Applications of Artificial Intelligence*, Vol. 17, No. 1, Feb. 2004, pp. 1-9.

[14]Wigley, G., and Kearney, D., "The Development of an Operating System for Reconfigurable Computing," *Proceedings of the IEEE Symposium Field-Programmable Custom Computing Machines 2001* [CD-ROM], IEEE Press, April 2001.

[15] Mosse, Daniel, Mehem, Rami, and Ghosh, Sunondo, "A Nonpremptive Real-Time Scheduler with Recovery from Transient Faults and Its Implementation," *IEEE Transactions on Software Engineering*, Vol. 29, No. 8, Aug. 2003, pp. 752-767.

[16]Conde, R. F., Garrison Darrin, A., Luers, F. C., Jurczyk, S., Bergmann, N. and Dawood, A., "Adaptive Instrument Module - A Reconfigurable Processor for Spacecraft Applications," *Military and Aerospace Programmable Logic Devices Conference* [CD-ROM], 1999.

[17]Mills, Carl S., Hines, Glenn, Fowler, Kim R., Garrison Darrin, M. Ann, Conde, Richard F., and Eaton, Harry A. C., "Adaptive Data Analysis and Processing Technology (ADAPT) for Spacecraft," *NASA Earth Science Technology Conference (ESTC) 2003*, 24-26 June 2003. Available online at http://esto.nasa.gov:8080/conferences/estc2003/papers/ A1P1(Fowler).pdf (cited Feb. 2005).

[18]Seidleck, Christina M., LaBel, Kenneth A., Moran, Amy K., Gates, Michele M., Barth, Janet M., Stassinopoulos, E. G., and Gruner, Timothy D., "Single Event Effect Flight Data Analysis of Multiple NASA Spacecraft and Experiments; Implications to Spacecraft Electrical Designs," Oct. 1997. Available online at http://radhome.gsfc.nasa.gov/radhome/papers/chris.htm (cited Feb. 2005).

[19]LaBel, Kenneth A., "Programmable Logic in the Space Radiation Environment," Tutorial, 2002. Available online at http://klabs.org/richcontent/Tutorial/MiniCourses/radiation_mapld_2002/ Radiation_Course_2002_ MAPLD.hardcopy/pdf_files/ 0_Introduction.pdf (cited Feb. 2005).

[20]Altera Corporation, "Device Family Overview," 2004. Available online at http://www.altera.com/products/devices/ common/dev-family_overview.html (cited Feb. 2005).

[21]Katz, R., LaBel, K., Wang, J. J., Cronquist, B., Koga, R., Penzin, S., and Swift, G., "Radiation Effects on Current Field Programmable Technologies," *IEEE Transactions on Nuclear Science*, Vol. 44, No. 6, Dec. 1997, pp. 1945-1956.

[22]Carmichael, Carl, Caffery, Michael, and Salazar, Anthony, "Correcting Single-Event Upsets Through Virtual Partial Configuration," *Xilinx, Inc. application note*, June 2000. Available online at http://direct.Xilinx.com/bvdocs/appnotes/ xapp216.pdf (cited Feb. 2005).

[23]Liu, Jane, *Real-Time Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.

[24]Wind River Systems, "Customer Profiles: JPL / Pathfinder." Available online at www.windriver.com/ customers/profiles/jpl-pathfinder.html, last accessed 3/17/04.

[25]Fordahl, Matthew, "Commercial Software Aided Reboot on Mars," Yahoo News, Feb. 16, 2004. Available online at story.news.yahoo.com (cited Feb. 2005).

[26]Wigley, G., and Kearney, D., "Research Issues in Operating Systems for Reconfigurable Computing," *Proceedings of the International Conference on Engineering Reconfigurable Systems and Architecture*, 2002, pp. 10-16.

[27]Khu, Arthur, "FPGA Configuration Data Compression and Decompression," Xilinx whitepaper, 25 Sept. 2001. Available online at www.xilinx.com/bvdocs/whitepapers/wp152.pdf (cited Feb. 2005).

[28]Shanbhag, Naresh R., "Reliable and Efficient System-on-Chip Design," *IEEE Computer*, March 2004, pp. 42-49.

[29]Laplante, Phillip A., R*eal-Time Systems Design and Analysis, Third Edition*, IEEE Press/John Wiley & Sons, 2004.