Software Fault Tolerance Methodology and Testing for the Embedded PowerPC

Mark Bucciero, John Paul Walters, and Matthew French
University of Southern California
Information Sciences Institute
Arlington, VA 22203
{mbuccier, jwalters, mfrench}@isi.edu

Abstract—In this paper we describe our software-based fault tolerance strategies for PowerPC devices embedded within Xilinx Virtex 4 FX60 FPGAs. Traditional FPGA fault tolerance techniques, such as scrubbing and TMR, cannot be applied to the embedded PowerPC. Our work targets scientific applications operating on space-based FPGA architectures consisting of an FPGA and a radiation-hardened controller. We use heartbeat monitoring, control flow assertions, and checkpoint/rollback to achieve high performance and low overhead fault tolerance. Our initial results show we are able to add our fault tolerance strategies with only 2% application overhead while recovering from 94% of the faults injected during testing.[12]

TABLE OF CONTENTS

1. INTRODUCTION

Recent generations of Xilinx FPGAs contain embedded PowerPC cores allowing domain experts an easy migration path for their existing C programs. This enables a rapid application development cycle where core functionality is quickly achieved by first porting applications to the PowerPC and code migration to the FPGA fabric is performed gradually, targeting performance-critical functions. This development cycle is especially attractive to the space community as it allows a low-risk spiral development path that yields higher performance. However, applications targeting the PowerPC cores are uniquely vulnerable to radiation induced Single Event Upsets (SEUs) because existing FPGA-based fault mitigation strategies are ineffective when handling upsets within the RISC core. Traditional FPGA fault tolerance [17—20] strategies can detect and correct errors within the FPGA's bitstream; however, the bitstream does not contain the complete state of the embedded PowerPC cores, and consequently, renders such strategies ineffective. Similarly traditional fabric-based fault injection tools are ineffective when used against the PowerPC [13—16]. Handling these upsets is critical for space applications where SEUs are common and computational resources are limited.

In this paper, we target the embedded PowerPC cores at scientific applications, which operate on traditional space-based FPGA subsystems, which utilize several FPGAs for processing and a radiation-hardened controller for bitstream scrubbing. These applications are more tolerant to data upsets and, to a limited extent, may trade reliability for increased performance. Rather than leveraging additional PowerPCs for redundancy, we utilize the additional cores to increase computational throughput first and then apply fault tolerance to the parallelized application. To that end, our primary goal is to detect and correct control flow and other catastrophic errors that would otherwise hang or crash the embedded PowerPCs. We ignore small, non-persistent, data errors that can be corrected in post-processing on the ground. The radiation-hardened controller can be used as a monitoring element for the PowerPCs to aid in system scheduling and recovery. If one node goes down, its task is reassigned by the radiation-hardened controller to one of the other processing elements while the impacted node recovers. In this type of architecture, the PowerPCs within the FPGA(s) create a mini-cluster of processing elements. One such space-based architecture is the NASA SpaceCube 1.0, which is deployed on the Naval Research Laboratory's Materials International Space Station Experiment 7 (NRL MISSE7).

The SpaceCube is made up of three main components: 2 Xilinx Virtex 4 FX60 FPGAs and 1 8-bit micro-controller implemented in an Aeroflex FPGA. Each Virtex 4 FX60 FPGA contains two embedded PowerPC 405 cores. With four PowerPC 405s and a radiation-hardened controller, the SpaceCube contains a mini-cluster of computation elements (Figure 1). For our initial research, the results presented in this paper target the simpler Xilinx ML410 development board, consisting of a single Virtex 4 FX60 FPGA.

Once an application is implemented on this mini-cluster of PowerPCs, our fault tolerance strategies can be applied. To achieve high performance and low overhead fault tolerance to the scientific application, we use heartbeat monitoring, control flow assertions, and checkpoint/rollback, working in concert to detect and correct errors. These fault tolerance strategies can be integrated after algorithm development has completed to minimize the impact on the application developer. Some

[1] 978-1-4244-7351-9/11/$26.00 ©2011 IEEE
[2] IEEEAC paper #1382, Version 2, Updated January 6, 2011

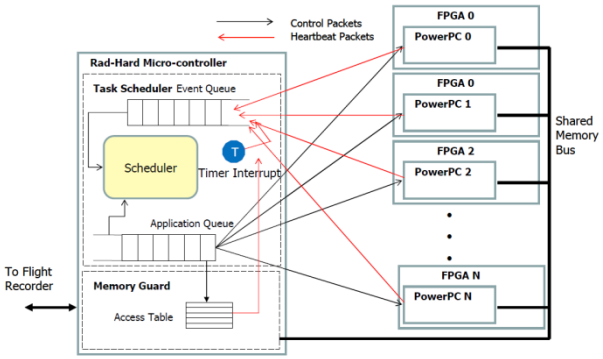

Figure 1: Block diagram of radiation-hardened controller scheduling task(s) to a set of PowerPCs.

techniques can be applied through pre-compiler directives and others by calling a set of methods in a user-level library at designated points in the application. This post-development flexibility allows fine-grained control of the methods to apply to the application and at which points during execution to apply them.

In this paper, we focus on describing these fault tolerance techniques and demonstrating them on a single PowerPC running a representative scientific application. To support this demonstration, we have developed a software based fault injector, called the Memory Sentinel and Injection System (MSIS), to corrupt the PowerPC general purpose registers, special purpose registers, and both the instruction and data caches. Our initial results show we are able to add these fault tolerance strategies with a low application overhead, while recovering from the vast majority of the faults that occur.

This paper is organized as follows. Section 2 describes some of the previous work related to FPGA fault tolerance, and to FPGA fault injection. Section 3 provides details about the embedded PowerPC and the bits that can be flipped by an SEU. Section 4 describes our approach to providing a level of fault tolerance for the Xilinx PowerPC 405. Section 5 details the MSIS, our method for software fault injection. Section 6 presents the data we have gathered from our fault injection experiments. Section 7 draws some conclusions and looks forward to some future research.

2. Related Work

When operating in a space environment, Xilinx SRAM based FPGAs, like other SRAM memories, are susceptible to radiation induced Single Event Upsets (SEUs). SRAM FPGAs are somewhat unique devices as SEUs can affect registers in either the configuration memory, which holds the state of the circuit, or the functional plane, which holds the user's registers. If a bit in the configuration memory changes, the function of the FPGA could be altered until it is reconfigured. Xilinx partial run-time reconfiguration allows for a small segment of the bitstream, called a frame, to be updated without impacting the remainder of the design. Using this feature, the frame containing the flipped bit can be restored to its intended function while leaving the rest of the design unchanged.

At the functional plane of the device, errors can manifest in two ways, directly if a register in the circuit being used is flipped, and indirectly, if a configuration memory bit associated with a part of the device being used is corrupted. Though scrubbing will correct the indirect faults, there is a window of opportunity based on the periodicity of the scrubbing in which faults may still occur. To account for the presence of functional errors, space-based FPGA systems employ triple modular redundancy (TMR) [2] to immediately mask faults and configuration errors until scrubbing can correct them. Each logical element in the design is triplicated, and a voter is added to decide on the correct result. A single upset may change the output of one of the three modules, but the voter ignores that output if the other two modules agree on the result. Tools to artificially inject a fault into the configuration memory can then be used to test the reliability of a design operating in a harsh radiation environment [3].

Recent trends in FPGA device architectures have steered away from homogenous sea of gates models and towards heterogeneous system on a chip architectures, adding features such as embedded processors, ethernet cores, and multi-gigabit transceivers. Unlike the other computational elements in the FPGA, the PowerPC 405 is not fully observable from the configuration memory. Scrubbing and TMR can be used to protect the logic surrounding the PowerPC, but cannot be used for the PowerPC internal functions, providing a significant hurdle for space-based use. Limited research has been done in this area, with the Simple Portable Fault Injector for the Embedded PowerPC (SPFI-ePPC) [4] as a wrapper around GDB, to test how the PowerPC responds to a fault, being the only known work in the field. SPFI-ePPC sets a breakpoint randomly during an application, changes a value, and resumes program execution. This method of fault injection is attractive for testing because it does not involve changing the HDL of a design. However, it can only modify registers and memory that are writable through GDB. The general and special purpose registers only account for a small percentage of the sensitive bits within the PowerPC, when the instruction and data caches are enabled (see Table 1 below). SPFI-ePPC provides a good first level analysis of how an application will respond to a bit-flip.

3. PowerPC 405 Details

In order to achieve successful fault mitigation strategies and understand fault injection results, it is first necessary to fully understand the variant of the PowerPC 405 utilized by Xilinx. The PowerPC 405 contained as a hard core in Xilinx Virtex 2 Pro and Virtex 4 FX devices is a 32-bit RISC, Harvard Architecture processor [5]. A block diagram of the PowerPC is shown below in Figure 2. The caches are each 16KB, 2-way set associative, with 8 32-bit words per cache line. The Memory Management Unit (MMU) is software controlled, but is, generally, only used by an operating system.

The PowerPC is an attractive computational element in space-based systems because of its computing power. Compared to the state of the art radiation hardened processor, the RAD750 (266 MIPS), the PowerPC 405 can compute nearly 3.5 times (900 MIPS) the number of computations per second. However, to use the non-radiation tolerant PowerPC 405 in a space-based system, it must be capable of identifying and recovering from a Single Event Upset (SEU). Without a recovery mechanism, an SEU within the processor could send it into an unknown or unrecoverable state. The exact details of the architecture used by Xilinx are proprietary, however based on a general knowledge of RISC architectures and our own experience implementing RISC architectures [12], we have derived an estimate of the sensitive bits within the PowerPC, shown in Table 1.

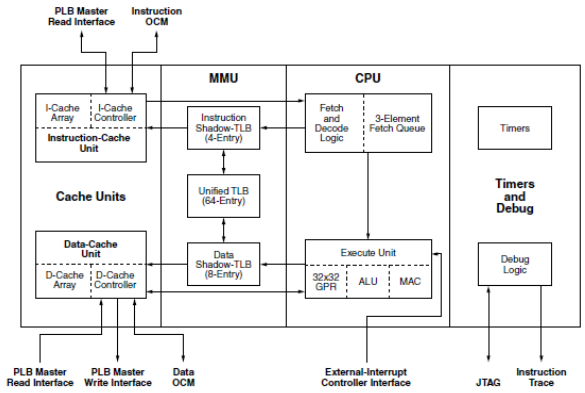

Figure 2: Xilinx PowerPC 405 block diagram.

As shown, the instruction and data caches account for more than 90% of the sensitive bits. Each PowerPC cache line does have a parity circuit. The parity of a cache line is verified whenever that line is accessed. If an SEU occurred within a cache line, the next time the line is accessed, a program exception is thrown because of its failed parity. However, due to an error in the PowerPC core [6], when the data cache parity circuit is enabled, it will immediately and continuously causes a program exception fault. This hardware error means cache parity checking cannot be enabled for use. Without a detection method, the caches become an important element to test for application response to an SEU.

4. Software Fault Tolerance

Now that we have at least a rudimentary understanding of the PowerPC 405 architecture, we can create fault detection and correction methods to mitigate the impact of SEUs on the PowerPC. The techniques described in this section include those used for error detection: heartbeat monitoring, control flow assertions, and watchdog timers. To mitigate the detected errors, we implement a user-level checkpoint and rollback library.

All of these techniques are designed to complement one another and to work in conjunction with a radiation-hardened

Table 1: PowerPC 405 Sensitive Bit Estimates

Feature	Size
Instruction Cache	16 KB + 64 control
Data Cache	16 KB + 64 control
General Purpose Register Set	32 x 32bit
Special Purpose Register Set	32 x 32bit
Execution Pipeline	10 x 32bit
ALU / MAC	~1,200 bits
Timers	3x 64bit
MMU	72 x 68bits
Misc	1024
Total	271,828 bits

controller. Heartbeats allow the radiation-hardened controller to monitor coarse-grained execution and status messages. Control flow assertions and watchdog timers are used by the executing PowerPC to ensure that execution continues in a predictable manner without skipping or repeating major code segments and to ensure that computational progress is being made. Finally, checkpointing and rollback are used by the executing PowerPC to periodically capture its state of execution. If a fault is found, the PowerPC may roll back to the most recent checkpoint before continuing computation. The ability of a PowerPC to restart from its most recent checkpoint avoids unnecessary wasted computation.

Nevertheless, fault tolerance has its trade-offs. A timer is used to send and receive heartbeats. The timer operates at user-defined intervals to allow users to balance overhead with finer-grained monitoring. Similarly, assertions add instructions to the application's execution path. The checkpoint and rollback library incurs the highest overhead, though as we will show, the overhead may be significantly reduced through the use of in-memory checkpointing.

Heartbeat Monitoring

The radiation-hardened controller is responsible for maintaining the state of the system and allocating computations to the different processing elements. In the SpaceCube 1.0 architecture, the radiation-hardened controller is implemented as a micro-controller in the Aeroflex FPGA. It acts as a computational supervisor to ensure the system is functioning as expected. The controller starts by assigning a task to a processor. Then, it observes the assigned application

for unexpected behavior. When an application completes, the next application is scheduled. If the application terminates unexpectedly, the controller decides the recovery method for the impacted processor.

Each processing element sends a set of informational messages, called heartbeats, to the controller, which contain data about the current state of the computation. The processing elements generate heartbeats at periodic intervals and for any major application event. A periodic message contains general information about the state of the computation, including a measure of the computational progress of the application. Similarly heartbeat events are sent when the computation begins or ends or when a failure mode is detected. Failure modes are detected by either control flow assertions or by PowerPC self-monitoring – see the sections below.

As an example, consider the case when the periodic heartbeat is no longer being sent from an application to the controller. The absence of a periodic heartbeat is an indication of a processor fault. When a fault occurs, the controller can decide to reassign the computation to another processor or to restart it on the same one.

Control Flow Assertions and Watchdog Timers

Control flow assertions enable the source code to perform a limited self-check at run-time to determine if an SEU has corrupted the program counter, a loop counter, etc [7]. Using this method, each PowerPC is capable of evaluating if the assigned computation is progressing as expected. If the processing element detects a control flow fault, the failure status is communicated to the control PowerPC using heartbeats.

Assertions can be implemented at either the compiler level or the source code level. We chose to insert the assertions at the source code level as a series of programmer directed pragmas, allowing the developer to specify critical code paths with minimal overhead.

The programmer adds assertions by inserting pairs of #pragma BEGIN var / #pragma END var statements at various points in the source code. A second utility transforms these pairs of statements into standard C code. Once assertions are added, control flow variables are checked at each #pragma END var statement for consistency.

We have implemented two forms of control flow assertions. The first ensures that program execution is progressing as expected. An assertion is raised if any of the control flow points are skipped or if the same point is crossed consecutively. The second form of assertion monitors the program to verify that the application is moving through the different control flow points. For example, if an application stops execution, the first assertion method will not be triggered since the control flow points have been executed in order. However, the second assertion method detects that the application is not making forward progress and raises an error.

The PowerPC built-in watchdog timer is used to ensure the application is still executing valid instructions. Using this method, a periodic interrupt clears the watchdog reset. If this interrupt is disabled, or otherwise interfered with, the watchdog timer will reset the PowerPC and its associated sub-systems. An application event heartbeat notifies the controller when the processor is reset, so an error can be logged.

Checkpoint and Rollback

Once a fault has been detected, it should be gracefully handled. Checkpoint and rollback allows software to cleanly recover to a previous state if an error has been detected. It has been successfully used, particularly, in the high performance computing (HPC) domain, to provide rollback recovery for long-running applications on failure-prone hardware.

There are several well-known checkpoint implementations. Plank et al. describe a well-known user-level checkpointing library libckpt [8]. Similarly Litzkow and Solomon describe a user-level checkpointing library for the Condor distributed processing system [10].

Perhaps the most common checkpoint library for Linux systems is the Berkeley Labs Checkpoint/Restart kernel module (BLCR) [9]. BLCR is a kernel module that supports a variety of architectures, including PowerPC. It has also been used to checkpoint distributed systems, such as MPI clusters [11]. Our embedded system does not currently use an operating system, so neither BLCR nor the user-level techniques described above, were suitable for our use.

Instead we developed our own user-level checkpoint and rollback solution for embedded PowerPC 405 cores.

We provide a straightforward API for our checkpointing library:

(1) checkpoint() captures the current state of the running application.

(2) restart() rolls back the application to the most recent checkpoint state.

Checkpointing is essentially a four step process:

(1) Pause the running application.

(2) Capture the memory segments and CPU registers.

(3) Write the captured data to storage.

(4) Resume the application.

From the user level, pausing the running application is equivalent to making a function call into the checkpointing library. Because the checkpointing library is active, no further

computations will occur[3]. In the case of parallel applications, we require the user to ensure that both PowerPCs are quiesced before invoking the checkpointing library. This ensures a consistent distributed state.

Capturing the memory contents is accomplished by reading the memory segment start and endpoints from variables assigned by the linker. Fixed sized memory segments are easily captured: the data section, SBSS, BSS, etc. can be read from memory directly and written to stable storage.

The register file may be captured at the C library level, using the setjmp() function. setjmp() captures all of the nonvolatile registers, including the stack pointer, and saves them into a platform-specific array, known as jmp_buf.

The stack and heap, however, are more complex. Recall that on most architectures, the PowerPC included, the stack grows downward. This means that, in memory, the address of the top of the stack is positioned earlier than the fixed stack bottom. We can easily derive the bottom of the stack from the linker symbols. The top of the stack, however, is by convention stored in general purpose register 1 on the PowerPC. Through inspection we found that the stack pointer, was being stored as the first element of jmp_buf. Using this value we can compute the top of the stack and capture only the used stack memory.

The heap, like the stack, is also a dynamic memory segment. We can easily derive the base of the heap using linker variables. However, finding the heap endpoint (known as the program break) must be done at runtime. To do so we employ the C library's sbrk() function call. The sbrk() function is typically used by malloc() to allocate additional heap space. It returns a pointer to the new program break. However, when called as sbrk(0), it returns the address of the current program break. We can use this to compute the heap boundaries and can store the heap to stable storage.

Once the full checkpoint, including memory, stack, heap, and registers have been captured we write the result to storage. Our solution supports checkpointing to flash memory, DDR memory, and BRAMs depending on the hardware platform being used and the application's requirements.

In Figure 3 we show the results of checkpointing a parallel SAR application to both compact flash and DDR memory. We have normalized the results to the non-fault tolerant case. Clearly, checkpointing to DDR memory is much more efficient than checkpointing to flash memory. Indeed, we are able to maintain 98% efficiency by checkpointing to DRAM, while maintaining only 78% efficiency when checkpointing to flash.

While checkpointing to flash certainly incurs a high overhead, there are advantages to its use. Checkpointing to flash will survive a power cycle, for example, while checkpoints to BRAM or DRAM will not. Checkpoints to flash do not consume limited embedded volatile memoryresources, unlike checkpoints to BRAM and DRAM. Ultimately, the choice between high performance in-memory checkpointing and high resiliency flash checkpointing must be made on a per-application basis.

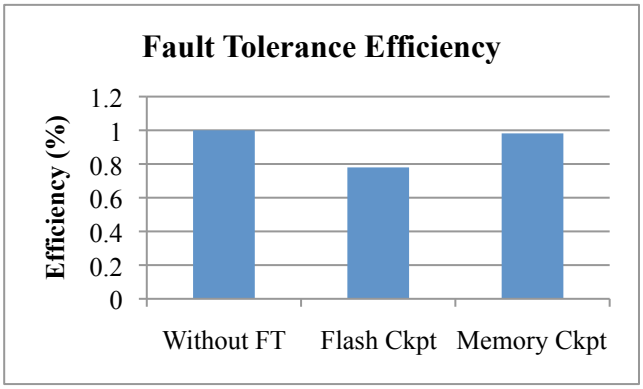

Figure 3: Checkpointing efficiency.

Rolling an application back to a previous checkpoint is as simple as calling restart() within the user's source code. At restart, the registers, memory segments, stack and heap are all restored to their previous values and locations. Memory is restored by reading the contents from a file, DRAM, or BRAM. Once memory is in place, registers are restored using setjmp's counterpart longjmp(). The restart() function never returns. Instead, after invoking longjmp(), application execution will appear to resume from the point that setjmp() was called during the checkpoint. From the application's perspective, the only difference between a checkpoint and a restart (at the setjmp() time) is that setjmp() returns 0 when called directly, and 1 when returned by longjmp().

While we can easily capture the state of the PowerPC, we are unable to automatically capture the state of external devices, UARTs, networks, etc. To checkpoint these devices, our checkpointing library allows users to pass pre- and post-checkpoint functions to the checkpoint() library function. These allow users to provide a custom function that will, for example, quiesce the network, close open files, or anything else, before the actual checkpoint occurs. After restarting the application, the post-checkpoint function is called immediately prior to returning to the user's application. This allows the user to restart networks, reopen files, etc.

5. Memory Sentinel and Injection System

The Memory Sentinel and Injection System (MSIS, pronounced em-sis) is a software-based fault injector for the PowerPC(s) within Xilinx FPGAs. Its purpose is to emulate an SEU by flipping any writable bit within the PowerPC - including the general purpose registers, special purpose registers, and both the instruction and data caches.

[3] To ensure consistency, users should temporarily pause any interrupts that may change system state.

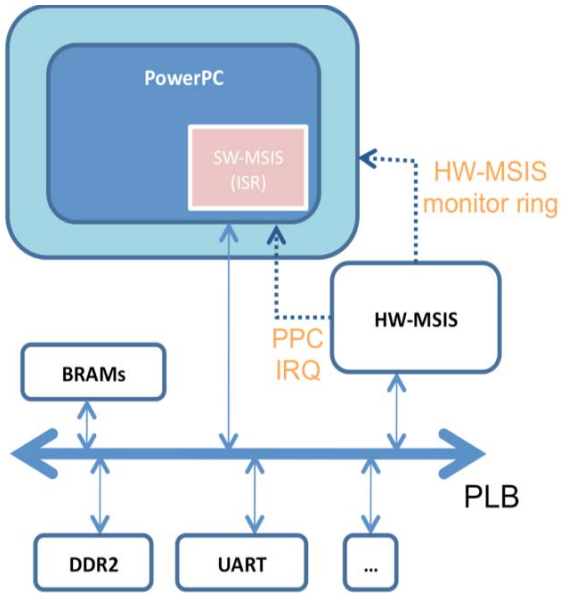

Figure 4: The MSIS can inject an SEU into the vast majority of the PowerPC sensitive bits and protect the read-only application memory.

In our current implementation, we are able to inject into the PowerPC's general purpose and special purpose register sets. We intend to extend this work to the PowerPC's data and instruction caches.

Being able to test how a bit-flip within the cache will impact an application is critical because the cache allows the application to run with maximum performance and accounts for a large cross-section of the PowerPC. It is assumed that the configurable logic will be protected from SEUs by scrubbing and TMR. A block diagram of the MSIS is shown in Figure 4 below. As shown, the MSIS is split into two main components: the SW-MSIS and the HW-MSIS.

The SW-MSIS runs as an interrupt service routine (ISR) and is responsible for performing the necessary steps to introduce an SEU (bit-flip) into the PowerPC. It can modify any software writable register and assist the HW-MSIS in modifying a cache line. The type of injection is chosen at random by a pseudo-random number generator. A main goal of the SW-MSIS is to limit how intrusive a fault injection is to the running application. For example, minimizing how much of the instruction cache is changed as a result of executing the SW-MSIS. To meet this goal, the SW-MSIS is compiled to a non-cacheable section of memory; and when the ISR starts, it disables the allocation of new cache lines on loads and stores, unless the injection is into cache. If the SW-MSIS injects a fault into the cache, the cache line needs to be reloaded from memory with one bit changed by the HW-MSIS.

The HW-MSIS acts as a timer to periodically inject a fault into the PowerPC and as a monitor to validate the bus transactions to/from the processor. This monitor ring is also capable of modifying data or bus transactions to/from the PowerPC. By changing the data going into the PowerPC, the HW-MSIS creates a difference between the data in the PowerPC cache and the application memory. This difference emulates a fault in the cache because the modified data is not stored in memory and the processor will use the cache data only for the life of the cache line.

By monitoring the PowerPC bus transaction, the HW-MSIS is also capable of providing a level of protection to the running system. If an SEU occurs within the PowerPC that creates a write to a read-only section of application memory, the HW-MSIS will prevent the operation from occurring. In this way, the HW-MSIS acts as a mini-Memory Management Unit (MMU). Unlike the MMU within the PowerPC, the HW-MSIS is implemented within the configurable fabric of the FPGA, where both scrubbing and TMR prevent an SEU from impacting the system. Because of this added level of protection, the MSIS should remain in a system it is used to test because it provides, with the fault injection disabled, additional fault detection to the system.

6. Application and Results

Our applications of interest are primarily Synthetic Aperture Radar (SAR) and Hyperspectral Imaging applications that have previously been provided by NASA. We have created a synthetic test application that is composed of the major elements of SAR and Hyperspectral Imaging. From SAR we have implemented a single precision FFT and complex multiply, and we have added a thresholding stage to the application to mimic the Hyperspectral application. The smaller test application was necessary in order to meet with the size limitations of our upcoming MISSE7 flight experiment, which does not provide us with any access to DRAM or other peripherals. Our ultimate goal is to correlate any faults observed on our MISSE7 experiment with injection tests performed using the MSIS. This required a common application.

The MSIS was used in a mode which only injected faults into the PowerPC register set. Neither the cache injection nor the memory guard features were used during this testing. We performed two trials: the first on a baseline flight application, and the second on the same flight application with fault tolerance enabled. We enabled heartbeats on the baseline test in order to allow unattended testing. In each trial we performed exactly 3060 injections. This number of injections is not exhaustive, as we are injecting randomly over both space (register location) and time (program counter value), but is statistically large enough to give us an estimation of trends within a reasonable amount of wall clock testing time (1 day ~ = 1,000 injections).

For baseline testing we broadly classify the results of each injection into one of three categories: good data, data error, or reset. A result of good data implies that after the injection, no difference in output was observed. A result of data error implies that after injecting the bit error, the output data had at least once difference from the known good result. Finally, a

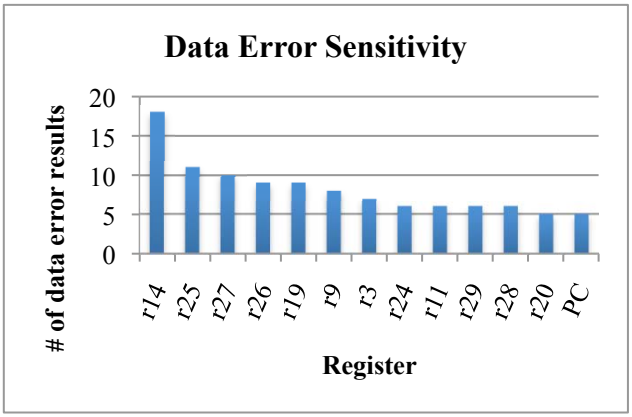

Figure 5: Baseline data errors.

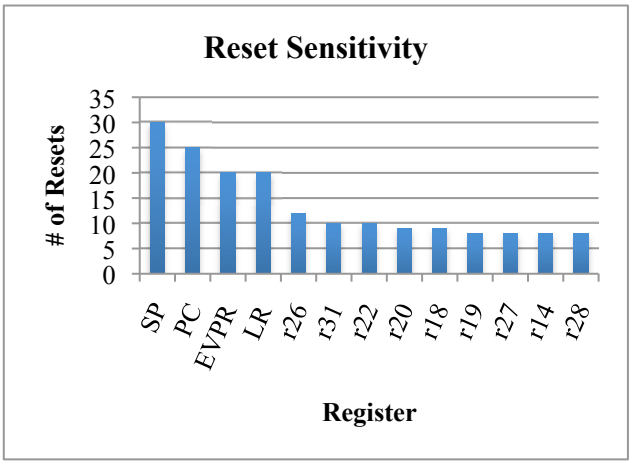

Figure 6: Baseline resets.

reset implies an error that would crash or otherwise hang the PowerPC.

When injecting into our fault tolerant design, we expanded the classification to track the number of rollbacks that resulted in good data as well as data error.

In Table 2 and Table 3 we present a summary of our fault injection results. As expected, the number of "good data (no action)" results remains consistent between our two trials. As we explain below, the data errors also remain largely consistent. Where our fault tolerance techniques have the most impact is in reducing the number of PowerPC hangs/resets.

Since our fault injector is an emulator running on actual hardware, a minimal heartbeat watchdog and reset was used to ensure the test was still active. This greatly facilitated batch testing of large numbers of injections. Therefore resets in the baseline case should be interpreted as an unrecoverable processor hang. In our fault tolerant experiment we enabled heartbeats, control flow assertions, watchdog timers, and checkpointing. Checkpoints were taken in the first iteration of the experiment. In this testing, reset indicates that the radiation-hardened controller will need to reboot the PowerPC

Table 2: Summary Baseline Injection Results

Result	*Percent*
Good Data (no action)	*86%*
Good Data (Rollback)	*0%*
Data Error (no action)	*5%*
Data Error (rollback)	*0%*
Reset	*9%*

Table 3: Summary Fault Tolerant Injection Results

Result	*Percent*
Good Data (no action)	*85%*
Good Data (Rollback)	*9%*
Data Error (no action)	*4%*
Data Error (rollback)	*0%*
Reset	*2%*

and did not have a checkpoint to roll back to, while rollback indicates that a checkpoint was found and used.

In Figure 5 and Figure 7 we can compare the histograms of data errors with respect to injection site observed between our fault tolerant and baseline results. In general we find that the number of data errors remains relatively constant regardless of the use of fault tolerance. This is to be expected as we target only upsets that cause PowerPC failures and application control flow errors. We leave data errors for post-processing on the ground.

Further, the top 5 most vulnerable registers are quite similar in both Figure 5 and Figure 7. Not only is general purpose register 14 the single most vulnerable register, but all of the top 5 most vulnerable registers are considered non-volatile registers by the PowerPC EABI (embedded application binary interface). The non-volatile registers are those that must be preserved through function calls. They are not used to pass function arguments or return values. It is therefore unsurprising that these registers are uniquely vulnerable in both the fault tolerant and baseline injection tests.

Note that it initially appears that the sensitivity of register 14 increases in our fault tolerant solution. In fact, this is an artifact of our randomized injection process. We actually injected nearly twice as many bit errors into register 14 in our fault tolerant solution. Yet, the number of data errors increased only slightly, from 18 errors in the baseline case to 25 in our fault tolerant design.

In Figure 6 and Figure 8 we present our reset sensitivity results. At the outset we can see that our fault tolerance solutions are reducing the number of PowerPC resets from 9% of the overall injections to 2%. This represents a substantial overall reduction in the number of PowerPC hangs crashes. The causes of these crashes are not surprising. For example,

in the baseline results, we see that the SP is the single most vulnerable register, followed by the PC. Injections into these registers commonly result in program exceptions due to the sensitivity of the program to these registers.

In the case of the fault tolerant results (Figure 8), we notice that we have introduced a sensitivity to the non-critical interrupts. This is because we rely on the non-critical interrupts to trigger heartbeat events, clear the watchdog, etc. If the non-critical interrupts are disabled a reset or rollback will occur either due to a series of missed heartbeats or an expired watchdog timer.

7. Conclusions and Future Work

In this paper we have described fault tolerance strategies for embedded PowerPCs within the Xilinx V4FX60 FPGA. To detect failures we have developed heartbeat monitors, self monitors, and control flow assertions. To mitigate the detected failures we have developed a user-level checkpointing library that allows an application to capture its state and to later roll back to the captured state.

We also described our fault injector named MSIS that we used in an injection campaign on a synthetic SAR and Hyperspectral-like application. We showed that our fault tolerance strategies improve the reliability of the embedded PowerPC cores by reducing the frequency of PowerPC resets by 7% while only adding about 2% overhead. This provides an interesting option for scientific applications that can correct small data errors in post processing in that with traditional TMR-based approaches, the overhead is on the order of ~200%, greatly reducing the amount of real-time science that can be achieved.

In the future, we will continue evaluating these fault mitigation strategies by improving our software-based fault injection (MSIS) and by testing at a radiation beam. The MSIS is currently used to inject faults into the PowerPC register set and is under test flipping bits with the instruction and data caches. We are also in the process of preparing for a radiation beam experiment. The beam is most representative way to emulate a real space environment while still on the ground. Since we are targeting sun synchronous Low-Earth Orbits, we are planning to test at a proton beam. By comparing the test results from the MSIS to the beam test, we endeavor to show that using the MSIS to inject bit flips is a representative way to test application response to the majority of SEUs within the PowerPC.

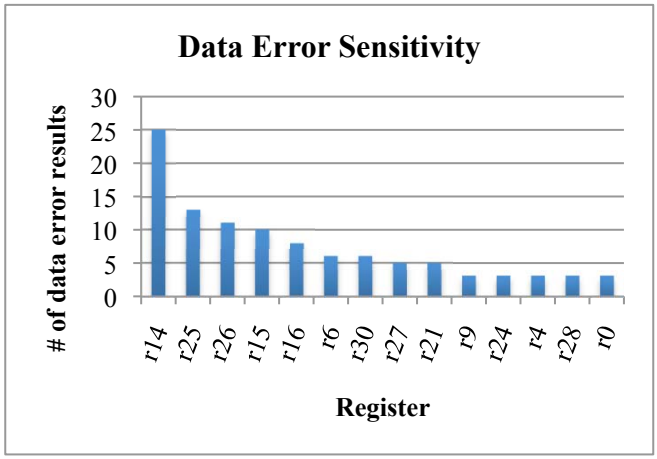

Figure 7: Fault tolerance data error sensitivity.

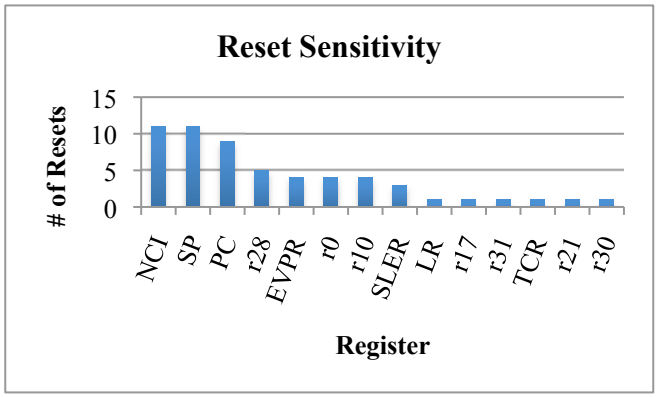

Figure 8: Fault tolerant reset sensitivity.

References

[1] Xilinx Application Note Web site http://www.xilinx.com/support/documentation/application_notes/xapp1088.pdf.

[2] Xilinx Application Note Web site, http://www.xilinx.com/support/documentation/application_notes/xapp197.pdf.

[3] M. Wirthlin, E. Johnson, N. Rollins, M. Caffrey, and P. Graham, "The reliability of FPGA circuit designs in the presence of radiation induced configuration upsets," in Proc. IEEE Symp. FPGAs for Custom Com- puting Machines (FCCM '03), Napa, CA, Apr. 2003.

[4] CHREC Web site http://www.chrec.org/pubs/SMCIT09_F6all.pdf

[5] Xilinx Web site http://www.xilinx.com/support/documentation/user_guides/ug011.pdf

[6] Xilinx Web site http://www.xilinx.com/support/answers/20658.htm

[7] R.Vemu, J. A. Abraham, "CEDA: Control-flow Error Detection through Assertions," iolts, pp.151-158, 12th IEEE International On-Line Testing Symposium (IOLTS'06), 2006.

[8] J. S. Plank, M. Beck, G. Kingsley and K. Li, "Libckpt: Transparent Checkpointing under Unix", Conference Proceedings, Usenix Winter 1995 Technical Conference, New Orleans, LA, January, 1995, pp. 213--223.

[9] J. Duell, P. Hargrove, and E. Roman.,"The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart." Berkeley Lab Technical Report (publication LBNL-54941), December 2002.

[10] M.Litzkow and M. Solomon, "SupportingCheckpointing and Process Migration Outside the UNIX Kernel", Usenix Conference Proceedings, San Francisco, CA, January 1992, pages 283-290.

[11] S.Sankaran, J. M. Squyres, B. Barrett, A.Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In LACSI Symposium, October 2003.

[12] J. Granacki, "MONARCH: Next Generation SoC (Supercomputer on a Chip)", *HPEC 2004*, Lexington, MA.

[13] E. Johnson, M. Caffrey, P. Graham, N. Rollins, M.Wirthlin,"Accelerator validation of an FPGA SEU simulator", IEEE Trans. onNuclearScience, vol.50, no.6, pp. 2147-2157, Dec. 2003.

[14] M. French, P. Graham, M. Wirthlin, L. Wang, and G. Larchev,"Radiation Mitigation and Power Optimization Design Tools forReconfigurable Hardware in Orbit", Proc. Earth-Sun System TechnologyConference, Hyattsville, MD, 28-30 Jun. 2005.

[15] L. Sterpone, M. Violante, "A New Partial Reconfiguration-Based Fault-Injection System to Evaluate SEU Effects in SRAM-Based FPGAs", IEEE Trans. on Nuclear Science, vol.54, no.4, pp.965-970, Aug. 2007.

[16] G. G. Cieslewski, A. D. George, A. M. Jacobs, "Acceleration of FPGA Fault Injectionthrough Multi-Bit Testing", ERSA 2010.

[17] Xilinx XTMR tool Web site, http://www.xilinx.com/ise/optional_prod/tmrtool.htm.

[18] B. Pratt, M.Caffrey, P. Graham, K.Morgan, M. J. Wirthlin, "Improving FPGA Design Robustness with Partial TMR", IEEE International Reliability Physics Symposium (IRPS), pp. 226-232, April 2006.

[19] M. French, P. Graham, M.Wirthlin, and L. Wang, "Cross Functional Design Tools for Radiation Mitigation and Power Optimization of FPGA Circuits", Earth Science Technology Conference, June 2006, Washington, D.C.

[20] C. Carmichael, M. Caffrey, A. Salaza, "Correcting single-event upsets through Virtex partial configuration", Xilinx Application Notes, XAPP216 (v1. 0), 2000.

BIOGRAPHY

Mark Bucciero *is an FPGA system architect with experience in embedded systems software and FPGA development. At USC/ISI, he is the lead FPGA architect for researching and developing novel radiation hardening by software techniques for embedded PowerPC processors within Xilinx devices. He pushed the state of the art in FPGA based systems utilizing the embedded PowerPC 405. He led the system development and hardware integration of 12 out of 16 FPGAs utilizing completely new hardware at ArgonST. He has a BS and MS in Computer Engineering from Virginia Tech.*

John Paul Walters *is a computer scientist at the University of Southern California's Information Sciences Institute, located in Arlington, VA. He received his PhD in computer science from Wayne State University in 2007, and his BA in computer science from Albion College in 2002.His research interests include fault tolerance, high performance computing, cloud computing, parallel processing, and many-core architectures.*

Matthew French *is a Project Leader at USC/ISI where he leads the Fine-grained Computing Group in research pertaining to FPGA fault tolerance, trust, and cognitive applications. He was Principal Investigator of the highly successful, NASA funded Reconfigurable Hardware in Orbit (RHinO) project, which first looked at many of the radiation and power issues of homogeneous SRAM-based FPGAs. He has over 20 papers and 3 patents in the areas of embedded processing and signal processing. He is a Senior Member of IEEE. He has a BS and ME in Electrical Engineering from Cornell University.*