Software Fault-Tolerant Techniques for Softcore Processors in Commercial SRAM-Based FPGAs

Nathaniel H. Rollins and Michael J. Wirthlin
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering
Brigham Young University, Provo, UT. 84602
nhrollins@byu.edu, wirthlin@ee.byu.edu
Keywords - Software fault-tolerance, SRAM-based FPGA, SWIFT, Checkpointing, Hardware fault-injection

1 Introduction

This paper implements software fault-tolerant techniques on a softcore processor implemented in a commercial SRAM-based FPGA for use in space-based applications. These software techniques include a modified version of software implemented fault tolerance (SWIFT) [8], consistency checks, and checkpointing [6]. To evaluate the reliability and costs of using software techniques to protect a softcore processor, two popular hardware-based techniques are used for comparison: duplication with compare (DWC) with checkpointing, and triple modular redundancy (TMR) with checkpointing. All of these techniques are implemented on the LEON3 softcore processor - the processor used by the European space agency (ESA) [5].

In this paper we protect the LEON3 softcore processor against SEUs at the cost of time instead of area. This goal is accomplished by using software fault-tolerant techniques. This study shows that specific software mitigation techniques can detect and recover from 99% of all configuration upsets to the LEON3 processor at a third the area cost of TMR, and for only a 1.8x performance cost.

2 Background

Microprocessors are an important part of most space-based applications, but the processors used in space are generally slow, expensive, and inflexible [3]. Processors used in space-based applications must be able to handle the harsh radiation-filled environment provided in space, thus radiation-tolerant processors are required. Making a processor radiation-tolerant usually means making an *existing* processor resistant to radiation. These processors, called radiation hardened (rad-hard by process) processors, are very expensive, and are often one to two decades old [3], and are therefore larger and slower than current commercial processors.

As an alternative to an older expensive rad-hard processor, more recent *softcore* processors have been used in space. In contrast to rad-hard processors, softcore processors are fast, flexible, and less expensive. These processors are called softcore processors because they are implemented in a field-programmable gate array (FPGA) fabric that allows the processor design to be altered. In contrast, a hardcore processor is one that is unalterable after it is implemented. Hardcore processors are implemented as application-specific integrated circuits (ASICs). Softcore processors are implemented in the logic of an FPGA.

FPGAs are a popular option for space-based applications because of their flexibility, reprogramability, and low application development costs. They can be reprogrammed while in-orbit to adapt to changing mission needs or correct design errors. For example, the Mars rovers use FPGAs for their motor control and landing pyrotechnics [7]. Also the Australian FedSat satellite uses FPGAs in its high performance computing payload [4].

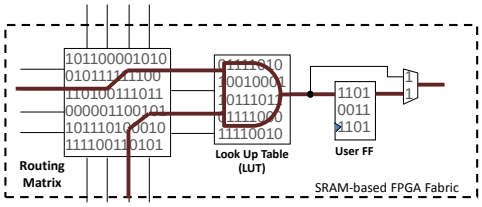
One of the biggest advantages that SRAM FPGAs provide is the ability to be reconfigured, even while in-orbit. This ability is provided by an FPGA's configuration memory which controls all of the logic and routing on the device. Changing the hardware design that runs on the FPGA is done by changing the contents of the configuration memory. **Table 1** shows that for the FPGA device used in this study (Xilinx Virtex4 SX55), there are about 3x more configuration memory bits than user memory bits in the entire device.

Configuration Bits	16 804 608	73.9%
User BlockRAM Bits	5 898 240	25.9%
User Flip-Flops	50 560	0.2%

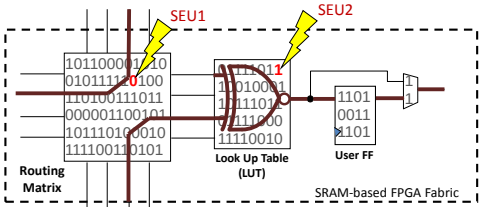
Table 1: Memory bits for the Xilinx Virtex4 SX55 device.

The flexibility and reprogramability that FPGAs provide for space-based applications comes at a price - SRAM-based FPGAs are inherently sensitive to the effects of faults caused by high-energy particles. These single event upsets (SEUs) can occur not only in FPGA user memory bits but also in FPGA configuration bits. In other words, an SEU can cause a lasting effect in the user logic or routing. **Figure 1** shows an example of two SEUs in an FPGA configuration memory. The first SEU changes the routing, causing a floating input in user logic. The second SEU changes the user logic, causing incorrect output.

This work was supported in part by the I/UCRC Program of the National Science Foundation under the NSF Center for High-Performance Reconfigurable Computing (CHREC).



(a) A design operating correctly.



(b) The two SEUs shown causes a bit in the SRAM fabric and a bit in the LUT to flip. The first bit controls routing to the design, and the second controls logic.

Figure 1: Single event upsets (SEUs) can cause permanent effects: permanent faults are repaired only when the FPGA is reconfigured.

A lot of research has identified hardware techniques to protect FPGA designs in the presence of SEUs. Triple modular redundancy (TMR) and configuration memory scrubbing [1] are one of the most popular ways to protect FPGA designs. TMR works by triplicating a design and voting on the outputs of the three modules. TMR voters detect and mask a faulty module. FPGA configuration scrubbing refers to the correction of upsets within the configuration memory. While a design runs on an SRAM-based FPGA, a scrubbing unit continually checks for upsets in the configuration memory. This is done by reading the contents of configuration memory and comparing it against a *golden* copy of the memory bitstream. Alternatively, CRC values of a portions of the bitstream are calculated and compared to the known CRC values. It is common for configuration scrubbers to be employed when SRAM-based FPGAs are used in radiation-filled environments such as space. Although TMR and configuration scrubbing is widely used, it is very expensive in terms of area.

As an alternative to these expensive methods, software fault-tolerant techniques can be used in softcore processors [2, 6, 8]. Software fault-tolerant techniques are a popular way of making traditional processors reliable in the presence of upsets. Software techniques come at the cost of performance instead of area. The main purpose of a software fault-tolerant technique is to protect processor memory elements from upsets. In a softcore processor these techniques must do more than protect user memory elements - these techniques must also detect and correct processor faults caused by upsets in the FPGA logic and routing.

This study differs from other studies using software techniques [2] and is different than other studies using a fault-tolerant LEON3 processor [5]. These previous studies use an ASIC processor implementation, and thus focus on protecting only the user memories (row two in Table 1). This study uses software techniques to protect logic and routing of the LEON3 processor (the FPGA configuration memory bits - row one in Table 1). Although this study does protect the memory bits using parity, the focus is on using software techniques to protect FPGA configuration bits.

3 LEON3 Softcore Processor

The softcore processor used in this study is Aeroflex Gaisler's LEON3 [5] processor. This study focuses on only core of the LEON3 which includes the 32-bit processor core, hardware multiplier and divider, 1 Kbyte direct-mapped instruction and data caches, interrupt controller, and main memory.

The different parts of the LEON3 that are protected are shown in **Figure 2**. The fault detection coverage that each reliability technique provides cannot be divided as clearly as is shown in the figure so Figure 2 shows the *primary* protection technique for each processor unit. The figure also shows the primary detection technique for routing between the processor blocks. Checkpointing does not show up in the figure since it is not a detection technique. Checkpointing is the recovery method used for all processor units.

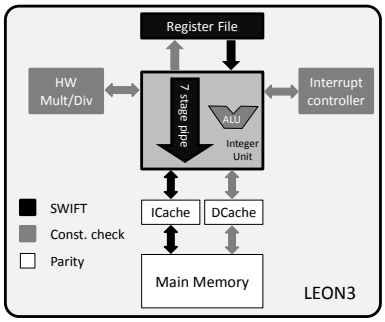


Figure 2: The primary fault detection methods for each LEON3 unit.

In this study programs run on the processor without an OS. Each program is written in C and compiled to assembly code using the SPARC gcc compiler. The assembly code is manipulated to add control-flow monitoring (part of SWIFT) and interrupt service routines for checkpointing and consistency checks. An assembler then creates the VHDL memory files required to synthesize the FPGA design.

4 SW Fault-Tolerant Techniques

The goal of this study to use software reliability techniques to create a fault-tolerant LEON3 processor for a commercial SRAM-based FPGA. This section introduces these software techniques (SWIFT, consistency checks, and checkpointing), and shows how each of them detect or correct upsets in the LEON3.

4.1 SWIFT

Software implemented fault tolerance (SWIFT) [8] is a software reliability technique that, like most software fault-tolerant techniques, is designed to detect upsets in the processor core memory elements and registers. The memory elements intended for protection include the register file, pipeline control, and other processor state. This study extends the intended use of SWIFT to protect the processor logic and routing.

There are two parts to the traditional SWIFT technique. The first part protects against data upsets through assembly language instruction duplication. A true SWIFT instruction duplication implementation in the LEON3 requires instruction set architecture (ISA) changes and incurs a very large performance overhead. So instead of a true implementation this study uses register file complement duplicate with compare (CDWC) to protect data. This is actually a hardware technique, but it is only implemented on the register file, and the area cost it incurs is very small. Although CDWC doesn't necessarily provide the same data protection that true SWIFT code duplication does, it requires no ISA changes and has no performance penalty.

The second part of SWIFT uses software control-flow monitoring to protect against control errors. Instructions are added at the beginning of each code block to compare a dynamic signature with the static signature of the block. If the dynamic signature ever differs from the block signature, the block was entered erroneously and a control flow upset is detected. Control-flow monitoring is effectively implemented for a small performance overhead.

4.2 Consistency Checks

Performing consistency checks is a well known software reliability technique used to detect upsets in memories and functional units. This technique works by periodically executing a set of instructions that verify the correctness of memories and functional units. Instead of verifying the correctness of the units themselves, this study uses consistency checks to detect upsets in the logic and routing leading to the units. These checks are executed as an interrupt service routine that is executed at regular intervals.

4.3 Checkpointing

Checkpointing is the fault-recovery technique used in this study. There are different amounts of state that can be saved in a checkpoint. At the very least, the register file contents and processor state registers must be saved. Other memory hierarchies may also be saved such as cache and main memory. The LEON3 uses write-through caches, so saving cache contents in a checkpoint is unnecessary. Instead the caches are simply invalidated. In this study, the main memory is also included as part of the checkpoint. Saving the contents of main memory can be expensive in terms of area and performance. But in this study it is saved without a performance penalty since main memory is implemented with on-chip block RAMs (BRAMs). Regardless of the size of the main memory, the time required to save or restore the contents of memory will be constant since all of the BRAMs that make up the main memory can be written-to simultaneously. Using on-chip BRAMs for main memory however, limits the potential size of the main memory.

5 HW Fault-Injection Results

This section discusses the reliability and costs of the software reliability techniques presented in this study. To evaluate reliability, hardware fault-injection is performed. Fault-injection works by flipping every FPGA configuration bit as a program executes on the LEON3 processor (over 16 million flips and program executions).

Before a bit is flipped, the program is run once to completion in order to ensure that at least one checkpoint has been taken. Next, the program is run for a random number of cycles before the bit is actually flipped. After a given bit is flipped, the system is observed while the program continues to execute to see how the program is affected. Finally the bit is repaired. To perform this fault-injection SEAKR's XRTC board is used.

In the fault-injection tests each of the FPGA memory bits is classified as being either required for architecturally correct execution (**ACE**) or unnecessary for architecturally correct execution (**unACE**). In other words, upsetting an ACE bit causes the program to run incorrectly, and upsetting an unACE bit does not hinder correct program execution. Upsets to ACE bits are further classified as being either detectable, recoverable errors (**DRE**), detected, unrecoverable errors (**DUE**), or silent data corruption (**SDC**) bits. When a DRE bit is upset, the upset is detected and successfully repaired. In other words checkpointing succeeded. When a DUE bit is upset, it is detected, but unsuccessfully repaired. In other words checkpointing failed. When an SDC bit is upset, it goes undetected, thus checkpointing is never even attempted. Ideally all upsets will be classified as DRE. Since SDC upsets aren't even detected, they are undesirable.

As a way of evaluating the relative effectiveness and cost of using software techniques to protect a LEON3 processor, two other LEON3 designs are implemented. These LEON3 processors are protected with hardware-based techniques: DWC with checkpointing and TMR with checkpointing. DWC with checkpointing is used for comparison since it is a popular processor reliability technique [6]. TMR with checkpointing is included in the comparison because TMR is the most common SRAM-based FPGA design reliability technique.

In order to provide the average percentage of ACE and unACE bits for each LEON3 processor design, **Table 2** shows approximate configuration and user memory bit usages for each design. The percentages shown in parentheses are

with respect to the values in Table 1.

Approximate LEON3 Memory bit usage			
LEON3 Design	Config Bits	BRAM Bits	FFs
SW Tech & Check	1 844 248 (11%)	423 936 (7%)	1318 (3%)
TMR & Check	4 780 069 (28%)	829 440 (14%)	3677 (7%)
DWC & Check	3 036 476 (18%)	608 256 (10%)	2670 (5%)

Table 2: Approximate configuration memory and user memory usage for each LEON3 design.

Hardware fault-injection is performed for a set of programs and the average ACE and unACE configuration bit percentages for each of the LEON3 designs is shown in **Table 3**. The table shows that for the LEON3 processor protected using software techniques, 7% of the FPGA bitstream bits corresponding to the processor are ACE bits. Note that this 7% is not a percentage of *all* the bits in the DUT (Table 1), but is a percentage of only the those bits in the DUT that correspond just to the LEON3 processor design (column two of Table 2). Of the upsets to those ACE bits, 92% are detected and successfully corrected, an additional 4% are detected but unsuccessfully corrected, and the remaining 4% are undetected. Overall, only 1% of upsets to *all* the bits corresponding to the LEON3 processor cannot be both detected and corrected.

HW Fault-Injection Results (Average)				
LEON3 Design	unACE	ACE		
		DRE	DUE	SDC
SW Tech & Check	93%	92%	4%	4%
TMR & Check	91%	93%	6%	1%
DWC & Check	92%	74%	23%	3%

Table 3: Average LEON3 bitstream bit classification percentages for the three LEON3 processors.

The area costs for the different LEON3 processor protection schemes are shown in **Table 4**. The cost is measured with respect to an unprotected LEON3 processor. The table shows that one of the huge advantages of using software techniques is the very low area cost they incur. The area cost of the LEON3 processor protected with software techniques is about a third the cost of using TMR, and about half the cost of using DWC.

Area Costs			
LEON3 Design	Slices	BRAMs	DSPs
SW Tech & Check	1.43x	1.92x	1.00x
TMR & Check	3.76x	3.75x	3.00x
DWC & Check	2.37x	2.75x	2.00x

Table 4: Area costs for the LEON3 processor designs.

The true cost of using software techniques to detect and recover from upsets in a processor is revealed in **Table 5**. The table shows how many more clock cycles are needed on average to execute a program compared to running the program on an unprotected LEON3. It also shows the average increase in code size. On average, the LEON3 protected with software techniques requires 1.8x more clock cycles than an unprotected processor.

Performance Costs (Average)		
LEON3 Design	Code Size	Run Time
SW Tech & Check	1.20x	1.76x
TMR & Check	1.01x	1.03x
DWC & Check	1.01x	1.03x

Table 5: Average performance and code size costs for the three LEON3 processors.

6 Conclusion

Software fault-tolerant techniques can detect and recover from 99% of upsets to a LEON3 processor at a third the area cost of TMR for a mere 1.8x performance cost. Although the user memories are protected, the hardware fault-injection evaluations inject faults into configuration memory only (which controls processor logic and routing).

References

[1] Correcting single-event upsets through Virtex parital configuration. Technical report, Xilinx Corporation, June 1, 2000. XAPP216 (v1.0).

[2] F. Abate, L. Sterpone, C. Lisboa, L. Carro, and M. Violante. New techniques for improving the performance of the lockstep architecture for SEEs mitigation in FPGA embedded processors. *Nuclear Science, IEEE Transactions on*, 56(4):1992 –2000, 2009.

[3] K. Anderson. Low-cost, radiation-tolerant, on-board processing solution. *Aerospace Conference, 2005 IEEE*, pages 1–8, March 2005.

[4] A. Dawood, S. Visser, and J. Williams. Reconfigurable FPGAS for real time image processing in space. *Digital Signal Processing, 2002. DSP 2002. 2002 14th International Conference on*, 2:845–848 vol.2, 2002.

[5] J. Gaisler and E. Catovic. Multi-Core Processor Based on LEON3-FT IP Core (LEON3-FT-MP). In *DASIA 2006 - Data Systems in Aerospace*, volume 630 of *ESA Special Publication*, July 2006.

[6] G.-L. Park and H. Y. Yong. A new approach for high performance computing systems with various checkpointing schemes. *The Journal of Supercomputing*, 33:65–78, 2005. 10.1007/s11227-005-0221-3.

[7] D. Ratter. FPGAs on Mars. *xCell Journal*, (50), August 2004.

[8] G. A. Reis, J. Chang, and N. Vachharajani. Software-controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization*, 2(4):366–396, December.