Autonomous On-board Processing for Sensor Systems: High Performance Fault Tolerance Techniques

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

Abstract

A-OPSS fuses high performance reconfigurable processors with emerging fault-tolerance and autonomous processing techniques in order to realize high performance, autonomous sensor systems. By enabling fault-tolerant use of high performance on-board processing the utility of sensor systems will be greatly enhanced, achieving 10-100x decrease in processing time, which directly translates into more science experiments conducted per day and a more thorough, timely analysis of captured data. This research also addresses the ability to quickly react and adapt processing or mission objectives in real-time, by combining autonomous agents with reconfigurable computing techniques. Using A-OPSS, satellites, airborne or ground sensors will be able to perform high performance, fault tolerant computation, and develop situational awareness about their operating environment and tune or adapt the application algorithm such that they return the most useful and significant data to human and automated decision support systems. This paper focuses on the first year's efforts which revolve around developing robustness to radiation induced upsets that occur in the embedded PowerPC within Xilinx FPGAs. This work presents an analysis of the sensitive cross section of both the architecture and a SAR application parallelized over two PowerPC405s. We then introduce a series of fault tolerance software routines used by the High Performance Computing (HPC) community and adapt them to an embedded system, such as the NASA GSFC developed SpaceCube 1.0.

1. Introduction

Over the past decade, the physics of CMOS at advanced process node feature sizes has entered a trade space where total ionizing dose (TID) is no longer much of a concern for modern devices in relatively low radiation environments such as Low Earth Orbit (LEO), however Singe Event Upsets (SEUs) have become more prevalent. Researchers have focused on both hardware and software techniques to mitigate radiation effects in SRAM-based Field Programmable Gate Arrays in order to address SEUs and make these devices radiation tolerant. This has proved to be a boon to the space community as these processors can deliver processing performance several generations ahead of traditional radiation hardened by process devices, such as anti-fuse FPGAs.

As technology has progressed, FPGAs have evolved from homogeneous sea of gates architectures, to heterogeneous system on a chip architectures containing multipliers, multi-gigbit transceivers, Ethernet cores, and even embedded PowerPC processors. The embedded PowerPC in particular adds significant benefit to the space community as now scientists, who may not be VHDL experts, have an easy migration path for their existing C programs to an embedded space platform. This enables a rapid application development cycle where core functionality is quickly achieved and then 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. It is also easy to build embedded space platforms with several PowerPCs and scale performance as 2 PowerPCs are available in a single FPGA, and most current space hardware contains several FPGAs.

The downside however is that the embedded PowerPCs are susceptible to SEUs and new fault mitigation techniques must be developed in order to make use of these attractive features. Traditional FPGA fault tolerance strategies can detect and correct errors within the FPGA's bitstream;[i,ii] however, the bitstream does not contain the state of the embedded PowerPC cores, and consequently, errors within the PowerPC cannot be scrubbed. Additionally, there are two, not three, PowerPCs per FPGA, so traditional Triple Modular Redundancy (TMR) techniques are awkward to apply.

In this paper we describe our software-based fault tolerance strategies for PowerPC devices embedded within Xilinx Virtex 4 FX60 FPGAs. Our work targets *scientific* applications operating on traditional space-based FPGA architectures consisting of an FPGA and a radiation-hardened controller. These applications are more tolerant to data upsets and, to a limited extent, may trade reliability for increased performance in space. To that end, our primarily goal is to detect and correct control flow and other catastrophic errors that would otherwise hang or crash the embedded PowerPCs. We ignore small data errors that can be corrected in post-processing on the ground. We use heartbeat monitoring, control flow assertions, and checkpoint/rollback to achieve high performance and low overhead fault tolerance.

2. Target Platform and Applications

The application level fault tolerance strategies presented in this paper are generic enough to be utilized on any standard FPGA-based space platform, however as a proof of concept an engineering sample of the NASA SpaceCube 1.0[iii] is being targeted. The SpaceCube is made up of three main components critical to this study: 2 Xilinx Virtex 4 FX60 FPGAs and 1 8-bit micro-controller implemented in a radiation hardened Aeroflex FPGA. Each Virtex 4 FX60 FPGA contains two embedded PowerPC 405 cores. With four PowerPC 405s, the SpaceCube represents a small computing cluster and is nearly 12 times more powerful than the RAD750. For some building block experiments, this paper utilized the Xilinx ML410 development board, consisting of a single Virtex 4 FX60 FPGA allowing for a straightforward transition to the SpaceCube architecture. For the ML-410 test set up, a host PC communicating via UART currently plays the role of the Aeroflex micro-controller in the SpaceCube.As mentioned previously, these techniques are focusing on the domain of science applications. This pertains to all of the NASA decadal survey missions, but especially so to all that could greatly benefit from high performance on-board processing, such as DESDyni[iv], and autonomy, such as HyspIRI[v]. For this phase of the research, a synthetic aperture radar (SAR) application, representative of those being considered for DESDyni was utilized to implement the fault tolerant techniques on and measure performance. In future phases Hyperspectral imaging applications with autonomous scenarios will be considered.

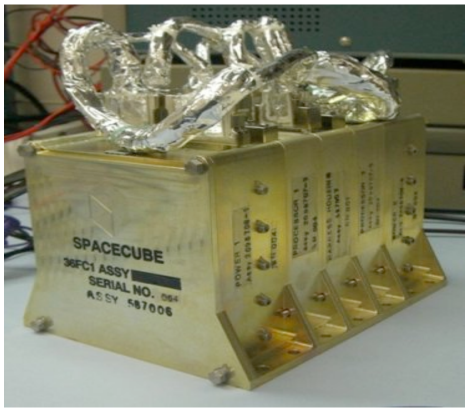

Figure 1 NASA's SpaceCube 1.0

These applications and missions yield a set of assumptions that will largely drive our fault tolerance strategies. First, by looking at science applications we can assume that non-persistent data errors that can be cleaned up via ground based processing are acceptable, especially when the trade off is to provide more data in a timely manner. Second, these missions are largely in sun-synchronous orbits, which represent a relatively low radiation environment, further relaxing the need for high overhead fault mitigation techniques. Third, these missions are not in continuous line of sight of a downlink station, which enables techniques such as out of order execution as there is a fair amount of system buffering. Finally, these applications tend to be streaming in nature, meaning they have a high degree of inherent parallelism that can be exploited.

The SAR application is representative of these scientific applications in that it exhibits large degrees of data parallelism over the compute-intensive FFT loops. When mapped to the SpaceCube architecture, this parallelism is exploited by dedicating a single PowerPC core to act as the master (performing coordination, file reads, etc.) and all other PowerPC cores acting as FFT-loop-level workers. This implementation will serve as our baseline as compared to versions with fault tolerance added.

3. PowerPC Analysis

It is first necessary to establish a theoretical sensitivity of the PowerPC to help establish an estimate of the upper bound of the sensitive cross section and set design goals for the fault tolerance techniques. Unfortunately, the exact number of registers and transistors in the embedded PowerPC is impossible to know without the original VLSI design, however an architectural analysis can provide data accurate enough for these purposes. Table 1 shows our estimate of the sizes of the key micro-architecture features within the PowerPC 405. The key point here is that the upper bound on the number of sensitive bits is about 141Kbits. For reference, the Xilinx V4 FX60 configurable logic has 21.3Mbits, or a potentially 151x larger cross section.

Table 1 PowerPC 405 Register Estimate

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	~140,880 bits

It is important to realize that the upper bound estimation is just an estimate and actual sensitivity will depend on the application, what architectural features it is using and how it uses them. For example, initially in SRAM-based FPGA studies, it was believed that the sensitive cross section was identical to the number of bits in the bitstream file it holds in memory. Additional experimentation eventually proved that only about 1/100 of those bits are ever used at any time, meaning the cross section was about 100x smaller than first believed and making a large difference in the reliability estimates for FPGAs. Similar considerations need to be taken into account when analyzing the PowerPC. For example, if an OS is not used, supervisory mode cannot be used, and therefore several special purpose registers will never be used. Similarly, caches and MMUs can be disabled entirely or used in reduced manners to limit sensitivities. A key characteristic of processors that is different than FPGAs is the time dynamic nature of registers, meaning that registers within the PowerPC may hold values that were at one time critical to an application's function, but have already been used, and will eventually get over-written by new input data. From a sensitive cross section analysis then, there are a number of cases where the time at which an SEU occurs is important, as an SEU could flip a bit in a register that has already been used and naturally get flushed by new incoming data and never show an impact on the results of the application.

Overall the manual analysis of the embedded PowerPC architecture provides some interesting insights. First, we expect the upset rate to be about 100x smaller than the FPGA fabric. Secondly, if the assumption that scientific applications are tolerant to non-persistent data errors holds, the amount of protection needed on large structures such as data caches is greatly reduced. Finally, the sensitive cross section is even further reduced as an SEU only has a small window of opportunity to impact a register while it is still critical to an application's functionality. All of these factors point towards developing a fault tolerant strategy that protects the control structures of an application first.

4. Parallelizing SAR

As described earlier, the parallel SAR application is implemented using a master-worker model. The master PowerPC is responsible for coordination, initialization, and I/O. The master and worker both compute independent FFTs in parallel. Communication and coordination between the two PowerPCs is achieved through the use of dedicated BRAMs. The BRAMs provide a simple mechanism for the implementation of barrier synchronization and an easy method for passing pointers between the two processors.

The primary obstacle to achieving performance in our implementation is the lack of hardware cache coherency between the two PowerPC cores. Disabling caching would have provided a straightforward implementation path at the cost of extremely poor performance. Using write-through caching and strategic cache flushing provided a modest improvement; however, performance still lags far behind write-back caching.

Enabling write-back caching required a thorough understanding of the memory and caching characteristics of the application. The solution is to use a combination of cache flushing and proper cache alignment. By cache-aligning each row of the shared arrays and flushing caches at synchronization points, we are able to avoid all cache coherency errors.

Figure 3 presents the results of the parallel SAR implementation with three caching options: no caching, write-through caching, and write-back caching. Speedup is greatly limited by not having the cache enabled; however, such an implementation provides performance that is more than 2x slower than a single processor with caching enabled. By enabling write-through caching the implementation is easier; however, the increased memory transactions nearly eliminate all parallel improvement. By enabling write-back caching we are able to achieve parallel speedup and make productive use of both processors.

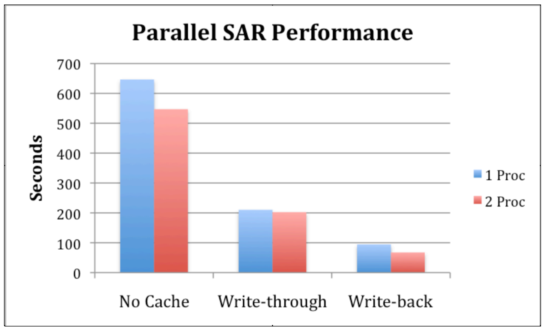

Figure 3: Parallel performance without caching, with

5. Fault Tolerance Strategies

The techniques described in this section focus on protecting the control flow of an application and include heartbeats, control flow assertions, and checkpointing and rollback. To minimize the impact on the application developer, our fault tolerance strategies can be implemented after algorithm development is completed. These methods can be applied with pre-compiler directives (e.g. control flow assertions) or by calling methods in a provided library (heartbeats and checkpointing/rollback) at designated points during application execution. This post-development flexibility allows the developer to choose which methods to apply to the application and at what points during execution to apply them.

5.1. Monitor and Heartbeats

For SpaceCube implementations, the radiation hardened Aeroflex micro-controller serves as an application monitor that is responsible for maintaining the state of the system and allocating computations to processors. It acts as an application supervisor to ensure the system is functioning as expected. The monitor starts by assigning a task to a specific PowerPC. Then, it observes the application for any unexpected behavior. If such a behavior is discovered, the monitor can decide to reassign that computation to another processing element or to restart the application on the same one.

The monitor receives heartbeat messages, which indicate the current state of the computation, from each processing element in the system. These heartbeat messages are sent to the application monitor at a regular interval. This interval can be chosen by the application developer and, upon expiration, triggers an interrupt for the PowerPC. If the monitor does not receive a heartbeat message from a PowerPC, or if a message arrives unexpectedly, it concludes that an upset has occurred and either a rollback or reset takes place.

Heartbeat messages also provide information about the state of the application and the processor. For example, when an application starts or ends, a unique type of heartbeat message is generated. Similarly, when an application detects some sort of failure, a different heartbeat message is sent to the monitor. The monitor verifies the contents of the heartbeat messages and decides on an appropriate course of action. The action taken can be decided at the system level and applied to the application. For example, if a computation is short, a checkpoint and rollback scheme may be unnecessary. In this case, when an error is detected, the computation can be restarted instead of rolled-back.

5.2. Control Flow Assertions

Control flow assertions enable source code to perform a limited self-check at run-time in order to detect SEUs corrupting the program counter, loop counters, etc.[vi] The advantage to using control flow assertions is that each PowerPC may evaluate its own progress without sending any messages to the rad-hard monitor. This improves performance and scalability because it reduces the bus interaction with the monitor, and only requires control flow messages to be sent to the monitor when an error occurs. The assertions may be implemented at either the compiler level or source code level. We chose to implement 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 user-designated points in the source code. A second utility then transforms the assertion statements into standard C code. Once assertions are added, control flow variables are checked at each **#pragma END *var*** statement for consistency. If an error is detected, a flag is set and subsequent heartbeats inform the monitor of the control flow error allowing corrective action to be taken either by restarting the computation or rolling back to a previous checkpoint. A global incrementing counter, unique to each assertion variable, is also maintained. This allows the heartbeat system to monitor application progress at a coarse grain. If the counters stop incrementing, the heartbeat monitor is alerted and corrective action is taken.

5.3. Checkpointing and Rollback

When a failure occurs, our primary corrective action is through checkpointing and rollback. A user-level library was developed that allows the user to snapshot a running computation through the use of a simple checkpoint() and rollback() API. The PowerPC's major memory segments are captured and stored either to a file or, for improved performance, to memory. After checkpointing, the heartbeat system notifies the monitor that a checkpoint has been taken, and program execution continues normally. The monitor records the checkpoint event so that in case of failure the monitor is able to instruct the application to either rollback (if a checkpoint has been taken) or restart computation (if a failure occurs prior to a checkpoint).

The major components of an application are easily checkpointable; however, there are some aspects that our checkpointing library is unable to snapshot. Computational states that exist within the FPGA logic, for example, are not automatically captured. Similarly all open files should be closed prior to checkpointing. To help programmers cope with these requirements we allow the user to easily extend the checkpointing process with user-defined callbacks. Callbacks allow the user to supply a function name (and argument) to the checkpoint() and rollback() API calls that will be called immediately prior to checkpointing and immediately after a rollback. The programmer can use the callbacks to close open files before checkpointing and reopen the files after a rollback.

6. Preliminary Data

Fault injection in an FPGA is traditionally performed by reading the bitstream, selecting a bit to inject an error into, flipping the bit, and running test vectors to see if the upset causes an error in the results. Due to the fact that the PowerPC is not represented in the bitstream, even error injection is difficult and proving to be a research topic. Currently the best method is to utilize JTAG based approaches which can interact with Xilinx's version of GDB to alter states in its debugger tools while executing an application. While these approaches yield some important data, they are only capable of injecting faults into the register set and cannot reach important features such as the caches, execution pipeline, ALU etc. Work currently underway under the A-OPSS project is expanding these fault injection techniques to consider these architecture features.

Until these new fault injectors are brought online, testing is limited to experiments using traditional JTAG fault insertion. Table 2 shows the results of register-based fault injection using the SPFI-ePPC fault injection tool[vii]. This initial test set of 100 trials is encouraging since the data suggests that for injections into the register set, the majority of failures do not manifest as silent data corruption. In fact, only 2% resulted in silent data corruption, while 89% resulted in no observable errors at the application level. The 8% of the injections that resulted in an observable application crash can be easily detected by the monitor.

Table 2 Register Set Error Injection Results

Injection Results	Percentage occurring
Data Corruption	2%
Crash	8%
Injection Failure	1%
No Effect	89%

Figure 3 represents the preliminary fault tolerance results measured on a Xilinx ML410 board. SAR is parallelized over 2 PowerPC cores and performance is measured against a baseline parallel SAR (no fault tolerance), parallel SAR with heartbeats and assertions, and finally parallel SAR with heartbeats, assertions, and a single checkpoint to either compact flash or DRAM. Heartbeats and assertions contribute little overhead and maintain 99% efficiency. Clearly, checkpointing to compact flash incurs a much higher overhead (78% efficiency) than checkpointing to DRAM (98% efficiency). The advantage to checkpointing to compact flash is that checkpoints survive device resets and reprogramming and DRAM usage is not increased. In practice these tradeoffs should be evaluated per-application.

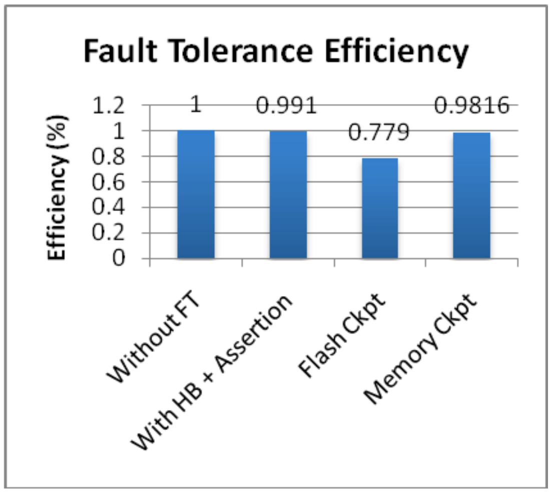

Figure 3: Efficiency of fault tolerance strategies normalized to parallel SAR (2 PowerPCs) without fault tolerance.

In Figure 5 we compare our AOPSS fault tolerance infrastructure to traditional duplication, TMR, and QMR (2 CPUs, 3 CPUs and 4 CPUs). We project the performance of 3 and 4 CPUs based on a 1.4x speedup for each doubling of processors (the same we achieved from our initial 2 CPU implementation). Clearly AOPSS demonstrates improved resource utilization over the existing techniques.

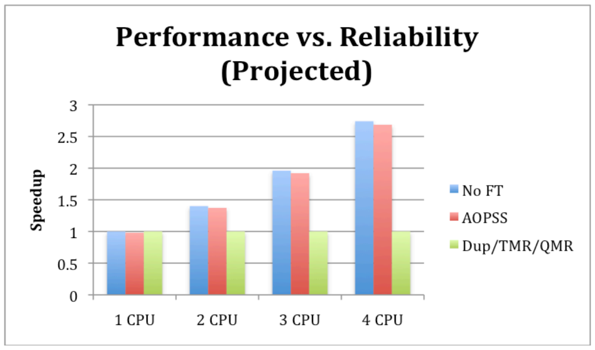

Figure 5: AOPSS performance vs. traditional strategies.

7. Conclusions and Future Work

A high performance and low overhead fault tolerance strategy targeting scientific applications on the SpaceCube 1.0 platform has been presented and initial steps towards experimental proof have been taken. In the upcoming year this work will further be enhanced by performing more fault injection testing using fault emulators capable of addressing more of the PowerPC architecture as well as performing radiation beam testing. With the fault tolerant architectural groundwork laid, the team will expand the application base to consider autonomous application scenarios, such as Hyperspectral image preprocessing and downlink prioritization.

8. References

[i] http://www.xilinx.com/esp/aero_def/see.htm
[ii] Brian Pratt, Michael Caffrey, Paul Graham, Keith Morgan, and Michael J. Wirthlin, *"Improving FPGA Design Robustness with Partial TMR"*, IEEE International Reliability Physics Symposium (IRPS) pp. 226-232, April 2006.
[iii] http://gsfctechnology.gsfc.nasa.gov/SpaceCube.htm
[iv] http://desdyni.jpl.nasa.gov/
[v] http://hyspiri.jpl.nasa.gov/
[vi] Ramtilak Vemu, Jacob A. Abraham, "CEDA: Control-flow Error Detection through Assertions," iolts, pp.151-158, 12th IEEE International On-Line Testing Symposium (IOLTS'06), 2006.
[vii] http://www.chrec.org/