Fault Tolerant Computing In Space Environment And Software Implemented Hardware Fault Tolerance Techniques

Ugur YENIER

Department of Computer Engineering
Bosphorus University, Istanbul

Abstract

Reliable computing in critical tasks is a log-term issue in computer systems. Two major fields of research are fault avoidance techniques and fault tolerance techniques. Even though these techniques can be used together, in many cases they are in opposing sides in system design. Fault tolerance techniques are also separated in two major fields, hardware redundancy and time redundancy. In this paper, I am going to compare the fault avoidance techniques with the purely software implemented fault tolerance techniques from with the experiments carried out in a special space mission, the ARGOS project.

Introduction

Electronic systems and computers are a part of our daily lives and critical missions. We rely on computers for many critical tasks like air control, medical equipments, banking operations etc.

Reliability, availability and serviceability (RAS) are the major factors in consideration in system design to provide continuous correct operation. In non-critical tasks, clearly, RAS is a major factor in customer satisfaction.[1]

There are two main approaches to dealing with faults. Fault avoidance and fault tolerance.

Fault avoidance techniques try to reduce the probability of fault occurrence, while *fault tolerance* techniques try to keep the system operational despite the presence of faults. Since faults cannot be completely eliminated, critical systems always employ fault tolerance techniques to guarantee high reliability and availability.

Redundancy is the base for fault tolerance. While introducing extra components (hardware redundancy) or extra execution time (time redundancy) or combination of both, we aim to provide redundancy. Major consideration issues in redundancy is cost, area, weight, speed and power consumption, limiting the amount of fault avoidance that can be introduced into the system. Clearly, the most fault-prone components should be considered as a target for redundancy first.

Faults can be either permanent or temporary. Temporary errors are usually caused by interference in the environment of the system, causing a transient effect that is eliminated within time without any other interference. Radiation is a major source of interference, especially in space missions. These errors are called Single-Event Upsets (SEUs). SEUs are major cause of concern in space missions, which are also observed at ground level. A single particle can upset multiple adjacent nodes in a circuit and cause a *Multiple-Bit Upset* (MBU). MBUs can occur in up to 10% of SEU events [1]. Therefore, they should be considered in the design of an error detection and correction technique.

Permanent faults happen when a part fails and needs to be replaced. A system can be designed with spare parts to automatically replace the failed part, or it can be designed so that it can continue its operation without the failed part, perhaps at lower performance, a technique called *graceful degradation*. The latter case is possible when the task of the failed part is not essential or can be carried out by another part. This usually happens when redundant parts are present in the system to provide higher performance.

Radiation is a major source of faults in space environment. Radiation hardening is a well known

technique used to reduce the sensitivity of the components to radiation. Major drawback of the technique is that, the components are very expensive and lag behind today's commercial components in terms of performance. Therefore there is a strong motivation to build low-cost, high performance, fault tolerant systems that can perform in the space environment. Unhardened *commercial off-the-shelf* (COTS) components, as in many other computing areas, are a challenging compotator against the specially designed components. The major drawback of COTS components is that they lack or have limited fault avoidance and fault tolerance features. Software-implemented hardware fault tolerance (SIHFT) techniques are proposed to provide low-cost solutions for enhancing the reliability of these systems without changing the hardware.

Advanced Research and Global Observation Satellite (ARGOS) [2] project is an extensive project to determine, to what extent commercial electronics, e.g., microprocessors and RAMs can be used in space environment. Project targets to design an experiment to collect data onboard an actual satellite so that it can be possible to compare radiation-hardened components with COTS components and evaluate different fault tolerance techniques, and also to develop techniques for fault tolerance that are software based and do not require any special hardware.

The ARGOS Project: Experimental Results

The Stanford ARGOS project [2] is an experiment that has been carried out on the computing test-bed of the NRL-801: *Unconventional Stellar Aspect* (USA) [8] experiment on the *Advanced Research and Global Observations Satellite* (ARGOS) that was launched in February 1999. The ARGOS satellite has a Sun-synchronous, 834- kilometer altitude orbit with a mission life of three years. The USA Experiment, one of the eight experiments that are flying on the ARGOS satellite, is primarily a low-cost X-ray astronomy experiment.

The objective of the test-bed in the USA environment on ARGOS is to comparatively evaluate the different approached to reliable computing in the space environment, including the radiation hardening of components, architectural redundancy and software fault tolerance techniques. This goal is met by flying processors and comparing performance in orbit during the ARGOS mission.

- The *Hard board* uses radiation hardened chip set, features a self checking processor pair configuration
- The *COTS board* uses a microcontroller from IDT, uses only commercial off-the-shelf (COTS) components and has no hardware error detection mechanism except for parity for cache memories.

Unfortunately it is stated that due to the difference in clock frequencies and the presence of cache memory on the COTS board, the COTS board is about 25 times faster than the Hard-board.

The two boards are operated simultaneously in orbit. It is the first example of a so-called "McCluskey test" [6], i.e., the simultaneous operation of commercial and hardened processors of the same class in the same orbital environment. Earlier experiments to gather fault-tolerance data, had very limited scope. These experiments either implemented only one fault-tolerance technique or collected very limited data, or they artificially injected faults, which may not fully represent the condition in an actual space environment. In ARGOS project, data have been collected in an actual space environment, thereby avoiding the necessity of relying on questionable fault injection. The LEO polar orbit of ARGOS presents a variety of radiation environments that are encountered during this mission which is a good opportunity, providing a rigorous test.

Boards on the satellite run special purpose test algorithms and store results, and then send the results back to the mission center at an available downlink window. It is also possible to upload new applications to test different techniques on the fly.

Hard Board

The Hard board has hardware EDAC for memory and a self-checking processor pair. Moreover, the data and address buses have parity bits. Upon a mismatch between the master and shadow processors, an exception is generated leading to a system halt and reset. Uncorrectable memory errors or parity errors also lead to system halt.

Eight errors occurred in the first program and nine in the second for the data sizes and periods shown in Table 1. These errors are concentrated in the SAA and polar cusps where the radiation levels are high as shown in Figure 1.

There has been an exception that led to a system halt. All other errors have been detected and corrected be test programs and the execution continued.[3]

Program	Data Size (KB)	Running Period (days)	Number of Errors
Memory Test	256	140	4
	512	349	4
Sine Table Generation	128	191	3
	512	250	9

Table 1- Errors in Hard Board

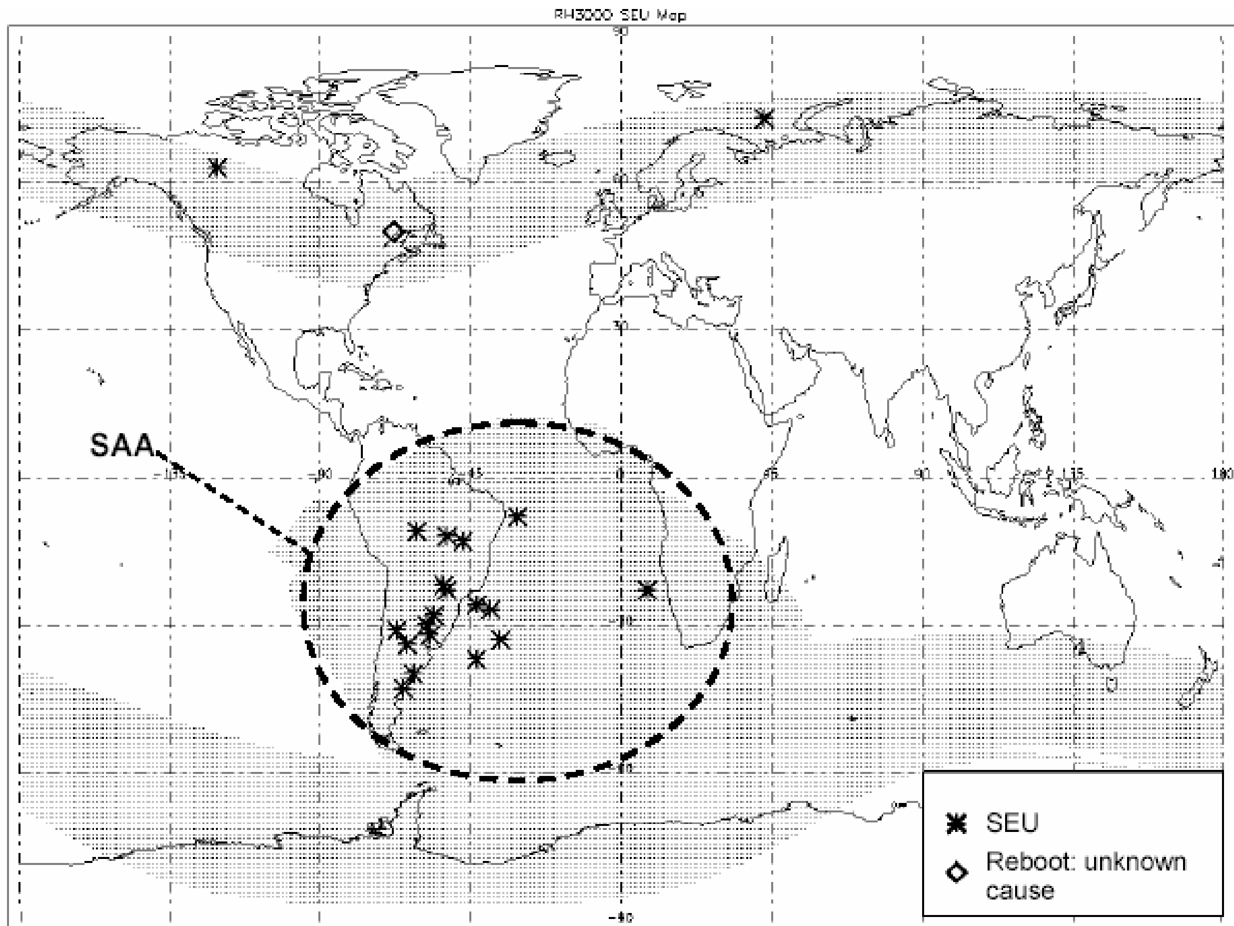

Figure 1- Geographical map of the upsets in the Hard board

COTS Board

First the software fault tolerance techniques that have been used in the ARGOS test bed are introduced in this section. These techniques include, Software Implemented Error Detection and Correction Codes, Error Detection by Duplicated Instructions, Control Flow Checking by Software Signatures and Watchdog Timer. Each section covering a specific technique includes test results obtained from other independent studies which are presented in different papers.[4] [6] [7] [9]

Later the experiment results from ARGOS are compiled together to form the conclusion of the performance of the combination of the techniques.

Software Implemented EDAC [9]

Introduction

Transient errors in memory chips are well-known, long considered reliability issues in computer systems. To prevent this problem Error Detection and Correction (EDAC) codes — also called Error Correcting Codes (ECCs) — are the dominating solution to this problem. Typically the bus is extended to accommodate extra bit(s) (parity bits) and encoding and decoding circuitry is added to detect and correct memory errors. The architecture requires extra hardware therefore introduces extra cost to the system. Usually COTS system does have no or very simplistic EDAC coding schemes. This limitation has to be solved by using another form of redundancy.

Software EDAC techniques do not require extra hardware to implement EDAC but presents protection for code and data that resides in the main memory. SEUs in main memories usually manifest themselves as bit-flips.

Software EDAC only addresses to find solution to transient errors, permanent errors are out of scope.

1. The objective is to devise a scheme to protect the data residing in main memory. The data that are protected by software EDAC are fetched and used by the processor in the same way as unprotected data are fetched and used. The EDAC program runs as a background task and is transparent to other programs running on the processor.
   Moreover, the protected data bits have to remain in their original form, to make the scheme transparent to the rest of the system. This requires the use of a systematic code such that data bits are not changed and are separable from the EDAC check bits.
2. If the same protection that is provided by hardware is to be provided by software, each read and write operation done by the processor has to be intercepted. However, this interception is infeasible because it imposes a large overhead in program execution time. Therefore, for software-implemented EDAC, only *periodic scrubbing* is done. In periodic scrubbing, the contents of memory are read periodically and all the correctable errors are corrected. If memory bit-flip errors are not corrected by the periodic scrubbing before program is executed, other software-implemented error detection techniques (e.g., assertions, *Error-Detection by Duplicated Instructions* [6], or *Control-Flow Checking by Software Signatures* [7]) are used to detect the errors. When an error is detected, a scrub operation is enforced before the program is restarted.
3. The space used for check bits reduces the amount of memory available for programs and data. Therefore, the *check-bit overhead* (# check bits / # data bits) must be as low as possible.

Multiple Error Correction

There are two possible ways that a multiple error occurs. Either multiple SEUs occur before the scrubbing is done, or a single SEU causes MBU. In the former case, scrubbing frequency should be adjusted according to the SEU rate to avoid exceeding the correction capability. In the later case, since multiple errors correspond to memory cells that are physically adjacent, the memory layout should be considered when designing an EDAC scheme. If the design is such that the physically adjacent bits belong to separate codewords, these errors can be corrected. To achieve this, the designer of the EDAC scheme needs to know the mapping of physical bits of the memory structure, to the logical bits in memory address space (location of the bits in a programmer's view of the memory). This mapping is determined by the system-level structure and the chip-level structure.

Conclusion

It has proved to be very effective in enhancing the availability of the system. Without software EDAC, the system works an average of 2 days before it crashes due to SEU corruptions in programs and needs a reset. With software EDAC this average was increased to about 20 days which is still short but is an order of magnitude improvement.

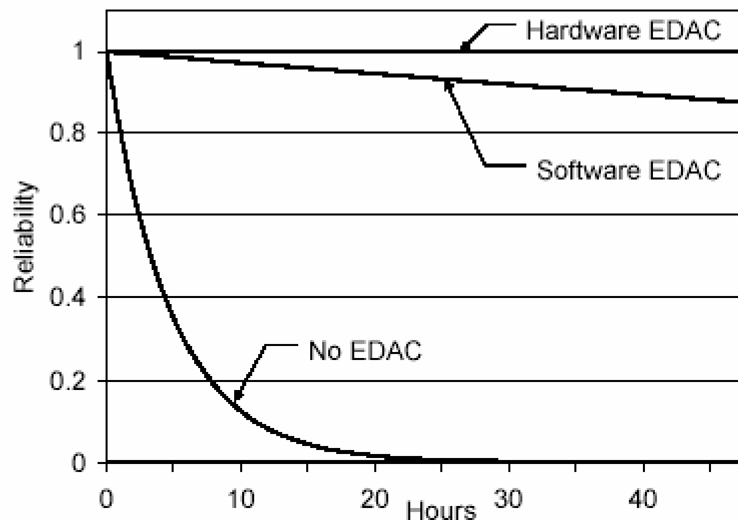

Figure 2- Software and hardware EDAC reliability comparison

It can be seen from Figure 2 that, hardware EDAC is outperforming software EDAC. This is due to the fact that hardware scheme performs the checking process on the fly for each data or code word read from the memory in oppose to the scrubbing process of software scheme.

EDDI [6]

Introduction

In Error Detection by Duplicated Instructions (EDDI), the instructions are duplicated during compilation and different registers and variables are used for the new instructions. This pure software technique is especially useful when designers cannot change the hardware, but they need dependability in the computer system. The *Control Flow Checking by Software Signatures* (CFCSS) [7] technique can be used with EDDI to increase the fault coverage.

EDDI can be used to detect faults that are caused by bit-flips in memory, or that can be modeled as bit-flips in memory. For example in the operation of transfer from the memory to the data bus, a bit can be corrupted. This error can be modeled as a bit-flip in memory. Transient errors in control logic, address and date buses, functional units can cause the intermediate value of the computation to be incorrectly output. These errors can be detected by EDDI.

Time redundancy reduces the amount of extra hardware at the expense of extra hardware. The basic concept in time redundancy is to repeat computations and compare output results. If the errors are detected then the computations can be performed again to obtain the correct results. However time redundancy techniques suffer from execution time over head and performance loss in the system.

The idea of error detection by instruction duplication is to duplicate the instructions, variables and registers used in the control flow. *Master instructions* are the originals in the source code and the *shadow instructions* are the duplicated corresponding ones. General registers and memory are portioned into two segments to avoid overlapping store operations. After the master and shadow instructions are executed, the results are compared by the *comparison instruction*. In a correct execution the results are the same. If results are not the same, the computation is repeated to obtain the correct output. Below is an example of a simple duplicated addition operation.

```
ADD R3, R1, R2            ; master instruction
ADD R23, R21, R22         ; shadow instruction
BNE R3, R23, gotoError    ; comparison instruction
```

Suppose registers R1, R2 and R3 are the master registers and R21, R22 and R23 are the shadow registers that contain the same value as R1, R2 and R3 respectively. If the values in R1 and R21 are the same and the values in R2 and R22 are also the same, then the two sums in R3 and R23 should be equal. Otherwise, an error has occurred during the operations. A conditional branch instruction BNE R3 R23 gotoError (branch to gotoError if R3 and R23 are different) performs a comparison and transfers control to an error handler if an error has occurred.

To obtain the best performance out of the system, the comparison instruction should be performed right before register store operations or a branch or jump instruction. After the comparison the results are stored to the memory to free up the registers. It is not necessary to compare intermediate results that can propagate to the final result, since the final result will be corrupted by the error in the intermediate result anyway.

A *basic block* is a branch-free sequence of instructions; no jumps into or out of the block except for the first and last instruction of the block. A *storeless basic block* is a sequence of instructions in which there is no store instruction except for the last instruction and the last instruction may be a store instruction or a branch instruction. A storeless basic block is always inside a basic block. Within a storeless basic block, shadow instructions are scheduled to maximize resource utilization by attempting to use idle resources, which are not used by master instructions. If the last instruction of a storeless basic block is a store instruction, a comparison instruction is placed before the store to compare the master and shadow register values that will be stored in memory.

Results

This section contains the experimental results from [6] that indicate the EDDI performance.
First, the source files are compiled and the target machine codes are generated without error detection instructions. A fault injector forced one bit- flip in the code segment of the machine code. For each iteration in the simulation, the location of the bit-flip is determined by a random number generator. The result of 500 iterations is shown in Table 2.
The numbers in the second row of the table (incorrect result) indicate the number of faults that cause the programs to produce incorrect results that look like correct ones to the observer. The third row means that erroneous result is repeatedly produced because the fault creates an infinite loop in the program. In the fourth row, the processor does not respond to the observer, so we have to manually stop the processor. The number in the fifth row shows the percentage of faults that are detected by the operating system in the machine. A segmentation fault and failed assertion are examples of faults detected by the operating system. Sometimes, despite the bit-flip in the code segments, the programs produce correct results. The percentage of these cases is shown in the sixth row. This case can happen, for example, when the bit-flip is in the unused field of an instruction and therefore has no effect on the results. The last row denotes the total percentage of faults that produced incorrect outputs without being detected (the sum of the second, the third and the fourth rows). On average, in the programs without EDDI, 20% of the injected faults produced incorrect outputs and were not detected as shown in Table 2.
Later, EDDI is included in the benchmark programs by the compiler postprocessor were developed. A bit-flip is injected into the generated machine code, and then the corrupted machine code is executed. The results for 500 injected faults are shown in Table 3.

	FFT	Hanoi	Com-press	Quick Sort	Fibonacci	Insert Sort	Matrix Mul.	Shuffle
Incorrect Result	38,0%	6,0%	17,0%	16,0%	16,0%	13,0%	25,0%	7,0%
Infinite Erroneous Result	2,0%	3,0%	5,0%	1,0%	0,0%	0,0%	0,0%	0,0%
Processor Hung	6,0%	1,0%	2,0%	0,0%	2,0%	1,0%	2,0%	0,0%
Detected by OS	24,0%	41,0%	36,0%	42,0%	53,0%	45,0%	48,0%	31,0%
Correct Result	30,0%	49,0%	40,0%	41,0%	35,0%	41,0%	25,0%	62,0%
Total	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
Undetected Incorrect Output	46,0%	9,0%	24,0%	17,0%	18,0%	14,0%	27,0%	7,0%

Table 2- Results of fault injection into original programs.

	FFT	Hanoi	Com-press	Quick Sort	Fibonacci	Insert Sort	Matrix Mul.	Shuffle
Incorrect Result	1,8%	0,6%	1,4%	0,6%	1,0%	0,6%	1,6%	0,4%
Infinite Erroneous Result	0,4%	0,2%	0,2%	0,0%	0,0%	0,0%	0,0%	0,2%
Processor Hung	0,0%	0,6%	0,2%	0,2%	0,4%	0,4%	0,2%	0,4%
Detected by EDDI	29,0%	15,8%	17,0%	22,6%	15,2%	19,8%	24,6%	32,6%
Detected by OS	22,0%	30,6%	32,4%	29,2%	23,8%	13,2%	27,4%	25,8%
Correct Result	46,8%	52,2%	28,8%	47,4%	49,6%	52,6%	46,2%	40,6%
Total	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
Undetected Incorrect Output	2,2%	1,4%	1,8%	0,8%	1,4%	1,0%	1,8%	1,0%
Difference	-43,8%	-7,6%	-22,2%	-16,2%	-16,6%	-13,0%	-25,2%	-6,0%

Table 3- Results of fault injection into the programs with EDDI.

EDDI shows high error detection capability: approximately 98.5% of injected faults in most programs produced incorrect outputs without being detected. On the other hand, original programs without EDDI produced undetected incorrect results that average 20% of the total faults.

Since extra instructions are added to the original assembly code, the program with EDDI suffers from an increase in code size and loss of performance compared to the original one. In addition, since we use general purpose registers as shadow registers, more register spilling occurs with EDDI; more spilling causes more performance overhead since it increases the number of memory operations.

Conclusion

EDDI is a pure software technique for error detection and correction that achieves high error coverage with performance penalty due to the time redundancy introduced into the system.

Algorithms to implement better performing EDDI and performance results can be found in [6]

CFCSS

Introduction

Control Flow Checking by Software Signatures (CFCSS) is a pure software method that checks the control flow of a program using assigned signatures.

CFCSS uses an algorithm that assigns a unique signature to each node in the program graph and adds instructions for error detection. Signatures are embedded into the program during compilation time using the constant field of the instructions and compared with run-time signatures when the program is executed. There is also another algorithm used to reduce the code size and execution time overhead caused by checking instructions in CFCSS.

Branching fault injection experiment is used with benchmark programs to determine the CFCSS performance. In the benchmark programs without CFCSS, an average of 33.7% of the injected branching faults produced undetected incorrect outputs; however, in the programs with CFCSS, only 3.1% of branching faults produced undetected incorrect outputs. CFCSS increases the error detection capability by an order of magnitude without the help of extra hardware added for error detection.

In CFCSS the program is divided into basic blocks. All nodes in the program graph are assigned different arbitrary numbers (signatures), which are embedded into the program during preprocessing or compile time. During program execution, a run-time signature G is stored in one of the general purpose registers called the *global signature register* (GSR), and compared with the stored signature of the node whenever control is transferred to a new node. For multiple branching cases, a run-time adjusting signature D is combined with G. The complete algorithm and an example program are presented in [7].

Comparison of CFCSS with other software signature techniques can be found in [7].

Results

The source files were compiled and assembly codes were generated. One of the branch deletion, branch creation or branch operand change was randomly applied to the assembly code, and then a machine code was generated by compiling the faulty assembly program. This machine code was executed and the result of 500 iterations is shown in Table 4.

For the second part of the experiment, CFCSS is included in the assembly source code and the branch fault (branch deletion, branch creation, and operand change) is inserted into the code. The resulting assembly code is compiled and executed. The result of 500 iterations is shown in Table 5. In this table, the last row is the percentage of faults that result in error and are not detected by either CFCSS technique or the operating system.

	Com-press	FFT	Matrix mul	Quick sort	Insert sort	Hanoi	Shuffle
Incorrect Result	21,0%	11,8%	26,6%	15,8%	26,2%	12,0%	22,8%
Infinite Erroneous Result	3,6%	16,8%	1,0%	0,0%	1,2%	2,0%	0,0%
Processor Hung	1,6%	16,8%	6,2%	17,8%	12,2%	7,8%	12,6%
Detected by OS	57,4%	36,0%	55,0%	50,0%	50,4%	54,6%	50,4%
Correct Result	16,4%	18,6%	11,2%	16,4%	10,0%	23,6%	14,2%
Total	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
Undetected Incorrect Output	26,2%	45,4%	33,8%	33,6%	39,6%	21,8%	35,4%

Table 4- Result of branch fault injection into original programs

	Com-press	FFT	Matrix mul	Quick sort	Insert sort	Hanoi	Shuffle
Detected by CFCSS	30.8%	34.4%	41.0%	28,8%	37.2%	34.6%	40.0%
Incorrect Result	0.8%	0,0%	2.4%	0.6%	0.6%	0.0%	1.6%
Infinite Erroneous Result	0.2%	2.8%	0,0%	0,0%	0,0%	0.2%	0,0%
Processor Hung	1.0%	1.4%	0.6%	2.8%	3.4%	1.6%	1.6%
Detected by OS	12.0%	13.6%	9.4%	17.4%	11.0%	8.0%	7.2%
Correct Result	55.2%	47.8%	46.6%	50.4%	47.8%	55.6%	49.6%
Total	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
Undetected Incorrect Output	2.0%	4.2%	3.0%	3.4%	4.0%	1.8%	3.2%

Table 5- Result of branch fault injection into the programs with CFCSS

Table 6 shows the code size overhead introduced by CFCSS in the benchmark programs that were used in the fault injection simulations. This table also includes the code size overhead for the benchmark programs from the public domain. The second column indicates the number of instructions before adding checking instructions to the original program. The third column shows the number of checking instructions added and the fourth column is the number of basic blocks in the program. The last column shows the overhead of adding the checking instructions to the original program, i.e., the ratio of checking instructions to original instructions.

The calculation intensive programs, such as FFT, have larger size basic blocks than data analysis programs. Thus the overhead is smaller for these programs (around 30% for FFT). On the other hand, programs such as sorting and searching have small size basic blocks because they have frequent branch instructions. Therefore, the overhead of checking instructions in these programs is relatively high compared to calculation intensive programs.

Program	Total Instructions	Checking Instruction	Basic Blocks	Overheads
LZW	452	280	105	61.9%
FFT	411	140	47	34.0%
Matrix Mul	763	316	102	41.4%
Insert Sort	132	84	29	63.6%
Quick Sort	254	142	47	55.9%
Hanoi	179	60	27	33.5%
Shuffle	688	183	59	26.6%
Adder Check	426	138	66	32.4%
Hash Table	1149	376	199	32.7%
Binary Tree	3330	1643	559	49.3%
External Sort	1430	776	280	54.3%
Symbol Table	393	224	91	56.9%
Counter	40	19	9	47.5%
Simple Sort	191	130	41	68.1%

Table 6- Checking instruction overhead

Watchdog Timer [3]

Some errors may cause a task to hang. A watchdog timer was implemented to detect these errors. There are also errors that are detected by hardware (e.g., reserved instruction, or illegal address). For these errors, an exception is generated and the OS suspends the faulty task. Since the task does not respond to the main control program in time, the watchdog timer expires and the task is restarted. Watchdog Timer implementation requires hardware support, and is not a pure software technique therefore it will not be explained in any further detail.

Software Techniques Results

To increase the number of collected errors, the size of code or data in each test program has been expanded Table 7 shows the results of these tests. For data expansion, the tests have been re-run. The Insert Sort test with floating-point numbers shows an unusually high error rate but it is indicated that its running time was not long enough to support any conclusion. The results from the FFT programs with two large data sizes show that the number of errors scales with the amount of memory used by the program and the error rates are close to the rate derived from the memory tests. The error rate of the Quick Sort programs with two large codes sizes is lower than other tests because not all the SEUs in the code segment cause errors in the program. This is due to the fact that the SEU may occur in the initialization part of the program (that is executed once at the program start and never used again until a program restart), or in bits of an instruction that are not used in the instruction opcode, or in unused words of the code segment.

Table 8 shows the total number of errors in each test program and the detection mechanism that detected these errors. Most of the errors were detected by EDDI. However, there are errors detected by CFCSS and watchdog timer, especially for tests with larger code sizes. Overall, the combination of EDDI, CFCSS and watchdog timer detected a total of 321 errors. There is one case of undetected error (detected by assertion check explained above). These numbers yield 99.7% error detection coverage. Out of the 321 detected errors, 4 cases have unsuccessful recovery. This shows that the implemented recovery technique had a 98.8% success rate.

Summary

The results from the Hard board show that despite all the hardware fault tolerance techniques used in the board; there are cases of undetected errors. Even if single points of failure are eliminated by better design, additional fault tolerance techniques, perhaps in software, may still be required for high reliability.

As expected, on the COTS processor board, memory SEUs were the main source of errors in the ARGOS experiments. Memory tests collected 503 errors during 350 days. The geographical maps show concentration of SEUs in the *South Atlantic Anomaly* (SAA) region and the polar cusps. MBUs have also been identified which consists 1.44% of the SEU events.

Software-implemented error detection and recovery techniques that are used in ARGOS have been effective for the error rate observed in the COTS board. Software EDAC improves the availability of the COTS board by an order of magnitude and had only 3% performance overhead. Even though hardware EDAC would be preferable for main memory, software EDAC provided acceptable reliability for the experiment. The experimental results show 99.7% error detection coverage and 98.8% error recovery coverage. Results show that COTS with SIHFT are viable techniques for low-radiation environments such as the LEO orbit of the ARGOS satellite.

The major source of errors in the COTS board had been the SEUs in the main memory. No errors have been detected in the processor. The average error rate calculated has been 5.5 SEUs/MB per day. There had been 7 cases of MBU in the errors that were collected during memory tests. Therefore 1.44% of SEUs in the memory modules were actually MBUs.

It has been seen that the upsets have concentrated in the SAA region like the Hard Board. There are also indications that the SRAMs are actually affected by the low-radiation levels as well, since there are errors detected in the low-radiation areas. Therefore error detection and correction is necessary in low radiated environments too.

Program	Code Size (KB)	Data Size (KB)	Running Period (days)	Number of Errors	SEUs/MB per day
Integer Sort	4	160	178	156	5
FP Sort	4	80	26	21	10
Quick Sort - Integer	50	8	54	13	4
	75	8	132	30	3
FFT	6	160	41	42	6
	6	80	132	60	5

Table 7- Errors in the COTS board from the computation tests

Program	Number of Errors	Errors Detected by Each Technique			Undetected Errors
		EDDI	CFCSS	Watchdog	
Integer Sort	156	156	-	-	-
FP Sort	21	21	-	-	-
Quick Sort - Integer	43	31	5	6	1
FFT	102	99	1	2	-
Total	**322**	**307**	**6**	**8**	**1**

Table 8- Errors detected by each SIHFT technique and undetected errors in the computation tests on the COTS board

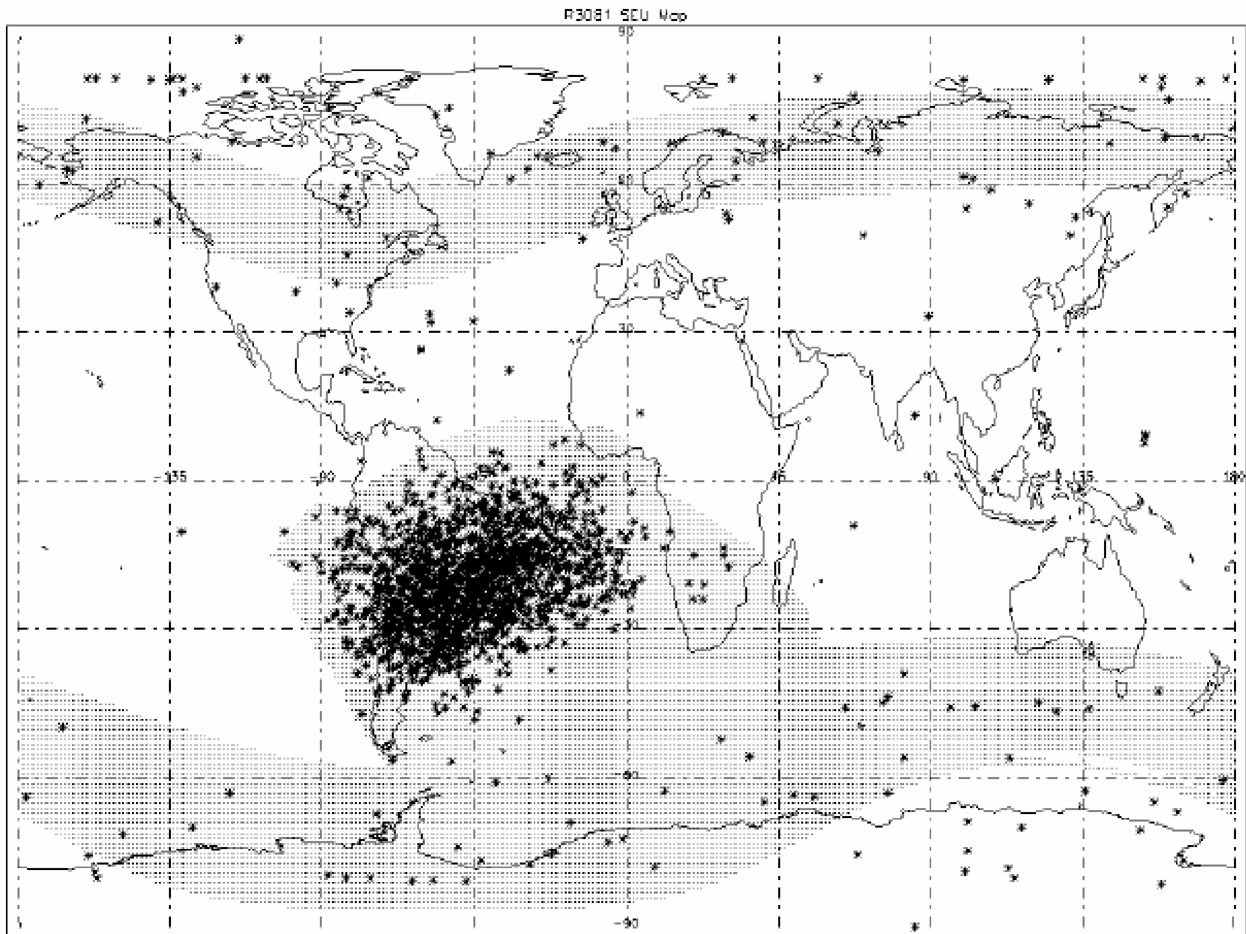

Table 9- Geographical map of the upsets in the COTS board

CONCLUSION

Fault tolerance is a must have for reliable computing. Space missions are very special purpose activities where all mission critical tasks are carried out by computers boosting the need for reliable components and computer systems. The challenge in the ARGOS project is to determine to what extend the commercial of the shelf components can be used in space environment. This goal is motivated by the availability of high performance, low-cost component in the market. To answer this question, data have been collected in an actual space environment thereby avoiding the necessity of relying on questionable fault injection methods. Same tests have been run on both COTS board and hardboard simultaneously.

The results from the hardboard shows that despite all the fault avoidance and fault tolerant techniques built into the hardware, there are cases of undetected errors. Even though single points of failure are eliminated by special designs and protection, additional fault tolerance techniques can actually improve the reliability.

COTS board where the software implemented error detection, correction and recovery techniques that require no hardware support have been implemented shows a high rate of error coverage. Experiment results show that 99.7% of all errors have been detected and 98.8% have been recovered despite the memory error rate of 5.55 SEUs/MB per day. Unfortunately it is not possible to compare the performance of to systems due to the variety of core clock frequency and existence of caches between to systems.

The conclusion of the project is that COTS with SIHFT are viable techniques to be used in low-radiation environments like LEO orbit of the ARGOS satellite.

References

[1] Shirvani, P.P., *"Fault-Tolerant Computing for Radiation Environments", CRC-TR 01-6*, Stanford University, Stanford, CA, June 2001

[2] Shirvani, P.P. and E.J. McCluskey, "Fault-Tolerant Systems in a Space Environment: The CRC ARGOS Project," *CRCTR 98-2*, Stanford University, Stanford, CA, Dec. 1998.

[3] Shirvani, P. P. and at al, "Software-Implemented Hardware Fault Tolerance Experiments: COTS in Space," *Proc. International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8),* Fast Abstracts, pp. B56-7, New York, NY, June 25-28, 2000.

[4] Nahmsuk Oh, "Software Implemented Hardware Fault Tolerance," Ph.D. thesis, Stanford University, Dec. 2000.

[5] Shirvani, P.P., "Fault-Tolerant Computing for Radiation Environments," *CRC-TR 01-6* (Ph.D. Thesis), Stanford University, Stanford, CA, June, 2001.

[6] Oh, N., P.P. Shirvani and E.J. McCluskey, "Error Detection by Duplicated Instructions In Super-scalar Processors," to appear in *IEEE Transactions on Reliability* Sep. 2001.

[7] Oh, N., P.P. Shirvani and E.J. McCluskey, "Control Flow Checking by Software Signatures," to appear in *IEEE Transactions on Reliability* Sep. 2001.

[8] Wood, K.S., et al., "The USA Experiment on the ARGOS Satellite: A Low Cost Instrument for Timing X-Ray Binaries," Published in *EUV, X-Ray, and Gamma-Ray Instrumentation for Astronomy V*, ed. O.H. Siegmund & J.V. Vellerga, SPIE Proc., Vol. 2280, pp. 19-30, 1994. 2224-9, July 1997.

[9] Shirvani, P.P., N. Saxena and E.J. McCluskey, "Software- Implemented EDAC Protection Against SEUs," to appear in *IEEE Trans. on Reliability, Special Section on Fault- Tolerant VLSI Systems*, June 2000.