Fault-Tolerance Projects at Stanford CRC

Philip P. Shirvani, Nirmal Saxena, Nahmsuk Oh, Subhasish Mitra, Shu-Yi Yu, Wei-Je Huang, Santiago Fernandez-Gomez, Nur A. Touba[*] and Edward J. McCluskey
Center for Reliable Computing, Stanford University, Stanford, CA 94305
[*]Computer Engineering Research Center, University of Texas, Austin, TX 78712

Abstract

This paper describes the fault-tolerant computing research currently active at Stanford University's Center for Reliable Computing. One focus is on tolerating hardware faults by means of software (software-implemented hardware fault tolerance). This work mainly targets faults caused by radiation induced upsets. An experiment evaluating the techniques that we have developed, is currently running on the ARGOS satellite. Another focus is on fault-tolerance techniques for adaptive computing systems implemented with field-programmable gate arrays (FPGAs).

I. INTRODUCTION

Electronic systems used in military, avionics and aerospace require high reliability and availability. Fault-tolerance has always been an essential attribute of these systems to keep them operational in harsh environments. For example, radiation —such as alpha particles and cosmic rays— can cause transient faults in electronic systems [1]. Such faults cause errors that are called single-event upsets (SEUs). SEUs are a major concern in a space environment, and have also been observed on ground levels [2]. Other sources of transient faults are electromagnetic interference and power supply glitches. An example effect is a bit-flip — an undesired change of state in the content of a storage element. The effects in combinational circuits, e.g., an arithmetic logic unit (ALU), can lead to incorrect computation results.

Fault avoidance techniques try to improve reliability by reducing the occurrence of faults. Two such techniques are shielding and radiation hardening. Shielding increases the weight and size of the system. Radiation hardening is an expensive process and, when used for a low-volume production, will lead to very costly parts. Therefore, alternative methods that do not have these drawbacks need to be explored.

Military and aerospace systems are designed for high reliability using certified components. Many of these certified components lag behind today's commercial components in terms of performance. The need for low-cost, state-of-the-art high performance computing systems in these areas has been pushing researchers to investigate new fault-tolerance techniques. Using commercial off-the-shelf (COTS) components, as opposed to military-standard or radiation-hardened components, has been suggested as one way to lower the cost and enhance the performance. The use of programmable logic devices (PLDs) instead of application specific integrated circuits (ASICs) also provides big advantages for low-volume production in terms of cost and at the same time, creates new opportunities for reconfigurable as well as fault-tolerant computing.

In the *Center for Reliable Computing* (CRC) at Stanford University, we are investigating both of these areas. In Section II, we discuss *software-implemented hardware fault tolerance* (SIHFT) techniques and the space experiment that we are involved in. Section III presents our research on *field-programmable logic devices* (FPLDs) and their use in *adaptive computing systems* (ACS).

II. SIHFT AND COTS

Commercial components are designed to function in an environment different from that of military and aerospace systems. They usually have limited fault avoidance and error detection capabilities. If commercial components are to be used for critical applications with no change in hardware, fault tolerance should be provided through software techniques. Notice that, if permanent faults have to be tolerated, typically spare components and a reconfiguration mechanism have to be available in the system. Our research in this area has led to new techniques for tolerating permanent faults in cache memories ([3]) and in FPLDs (discussed in Section III). In this section, the target faults are transients. We assume the system resources can be recovered to their correct state.

We consider two main locations in a computer system where errors can occur: the memory and the processor. Bit-flips in memory can corrupt the contents of code or data segments. A bit-flip in a location of memory that contains the instructions of a program, or in one of the registers of the processor, may cause the program to produce incorrect results. For example, a bit-flip may change an '`and`' instruction to an '`or`' instruction or change the register address that indicates an operand. Another example is a bit-flip in the location counter inside the processor, leading to an illegal jump to an undesired location in the memory. The latter is a *control-flow error* — a deviation from the correct sequence of instructions in a program. To build a fault-tolerant system, we need to detect these errors and recover from them. In sections II.A and II.B, we discuss some of the error detection and correction techniques that we are using in our project. Section II.C presents our experiment setup and its status.

A. Software-Implemented Error Detection

Transient errors that occur in a processor can be detected by executing a program multiple times, and comparing the outputs produced by each execution. This time redundancy technique is analogous to hardware redundancy techniques such as duplication and TMR (triple modular redundancy). Duplication can be done at task-level by the programmer or by the operating system (OS) (an example of the latter is presented in [4]). It can also be done at instruction level during program compilation. We have developed a technique called *error detection by duplicated instructions* (EDDI) that uses the latter approach. Figure 1 shows a sequence of instructions and how it is transformed for EDDI. Computation results from master and shadow instructions are compared before writing to memory. Upon miscomparison, the program jumps to an error handler that will cause the program to restart. Details of this technique can be found in [5].

```
ADD R3, R1, R2          ; R3 <- R1 + R2
MUL R4, R3, R5          ; R4 <- R3 * R5
ST  0(SP), R4           ; store R4
```

(a)

```
ADD R3, R1, R2          ;master instruction
ADD R23, R21, R22       ;shadow instruction
MUL R4, R3, R5          ;master instruction
MUL R24, R23, R25       ;shadow instruction
BNE R4, R24, Err        ;compare results
ST  0(SP), R4           ;master result
ST  offset(SP), R24     ;shadow result
```

(b)

Figure 1: An example of the EDDI technique: (a) original instruction; (b) instructions and data structures are duplicated (master and shadow copies).

```
ADD R3, R1, R2     ;A branchless block
MUL R4, R3, R5     ;  of instructions
ST  0(SP), R4      ;
```

(a)

```
XOR R30, R30, 0x3c  ;Gen. run-time signature
LDI R10, 0xb7       ;Load assigned signature
BNE R30, R10, Err   ;Compare the two
ADD R3, R1, R2      ;Continue normal
MUL R4, R3, R5      ;  sequence if
ST  0(SP), R4       ;  correct signature
```

(b)

Figure 2: An example of the CFCSS technique: (a) a branchless block of instructions; (b) the same block with the additional control-flow checking instructions.

EDDI can detect some of the control-flow errors. To further enhance the detection coverage for this type of error, we have developed a technique called *control-flow checking by software signatures* (CFCSS) [6]. Signature monitoring is a well-known method for control-flow checking. In this method, a signature is associated with each program block. This signature is stored in memory and checked during the execution of the program. CFCSS is an *assigned* signature method where unique signatures are associated with each block during compilation time. These signatures are embedded into the program using the immediate field of instructions that use constant operands. A run-time signature is generated and compared with the embedded signatures when instructions are executed. Figure 2 shows an example of instructions with CFCSS. In this example, `R30` (any of the general-purpose registers of the processor can be used for this purpose) holds the run-time signature and is updated as execution moves from block to block. Upon entering a block, `R30` is XORed with a constant to generate the signature of the current block. This value will be correct only if the correct sequence of blocks has been followed. The assigned signature of the current block is compared with the run-time value. Upon miscomparison, the program jumps to an error handler that will cause the program to restart. Details of this technique can be found in [6].

We have implemented a software tool to automatically generate programs with error detection capability using these techniques. Figure 3 shows the flow of this tool. First the C source code is compiled by the `cc` or `gcc` compiler and the assembly code is produced. Our post processor adds the extra instructions for EDDI and/or CFCSS (each can be enabled independently). The resultant assembly code is processed by an assembler and the executable object code with error detection capability is produced.

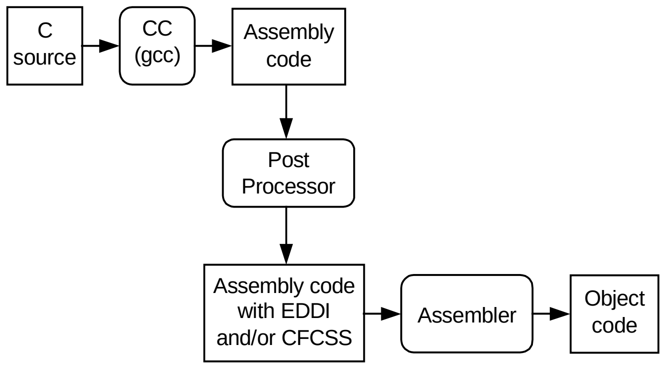

Figure 3: Flow of our software tool for adding EDDI and CFCSS.

EDDI and CFCSS are pure software techniques for detecting hardware errors. These techniques do not require any changes in hardware or any support from the OS. Therefore, they can be used in computer systems where modification to the hardware or the OS is either very expensive or impossible.

Some errors may cause infinite loops or deadlock in program execution. A well-known solution is using watchdog timers to limit the execution time of each program. If the program does not respond within its time limit, an error is indicated and the program is restarted.

To facilitate error recovery, we break a program into modules and run each module as a separate task — assuming that the system has a multitasking OS. A main module controls the execution of all the other modules. When one of the error detection mechanisms detects an error, the erroneous module is aborted and restarted, without corrupting the context of the other modules. If the error source was some transient error in the processor, the module will resume its normal execution. However, if the error source was a bit-flip in the code segment of the module, the restart will fail. At this time, we need to repair the bit-flip in memory before attempting another restart. We discuss the repair mechanism in the next section.

B. *Software-Implemented EDAC*

In many computer systems, memories are protected against SEUs by an *error detection and correction* (EDAC) code. This code is usually implemented in hardware using extra memory bits and encoding/decoding circuitry. EDAC protection can also be implemented in software. For example, a software implementation of a (255, 252) Reed-Solomon code that can do single-byte error correction is proposed in [7] for protecting RAM discs of satellite memories. This section briefly discusses our approach in this area.

B.1) Systematic Codes

A coding scheme provides a mapping of input data words to what are called codewords. Codewords contain extra check bits that are used for error detection and correction. In some codes, the check bits are appended to the data bits to form the codewords. Therefore, the data bits are not changed. These codes are called systematic (or separable) codes. In non-systematic codes, the data bits are not preserved and are mixed with check bits.

Our objective is to devise a scheme to protect the data residing in main memory. For this application, the data that is protected by software EDAC, is fetched and used by the processor in the same way as unprotected data is fetched and used. We want the EDAC software to run as a background task and be transparent to other programs running on the processor. The protected data bits have to remain in their original form if we want to make the scheme transparent to the rest of the system. This requires the use of a systematic code.

B.2) Vertical vs. Horizontal Codes

In memory systems with hardware EDAC, the memory width is extended to accommodate the check bits. Figure 4(a) shows a diagram for a 32-bit memory word that is augmented with seven check bits. Each set of check bits is calculated based on the bits of one word corresponding to one address. We refer to this type of coding as a *horizontal code*. When a horizontal code is implemented in software, each word is encoded separately and the check bits are concatenated to form a word. This check word is saved in a separate address (Fig. 4(b)).

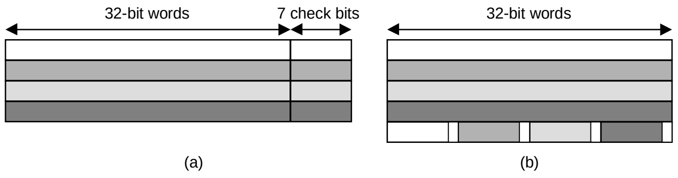

Figure 4: A horizontal code over bits of a word: (a) hardware implementation; (b) organization of bits when the code is implemented in software.

Another type of coding is shown in Fig. 5. Each set of check bits is calculated over the bits corresponding to one bit-slice of a block of words in consecutive addresses. This type of coding is used in some tape back-up systems ([8]) and we refer to it as a *vertical code*. This type of code matches well with the bitwise logical operations that are present in all common instruction set architectures (ISAs). The logical '`xor`' operation is used in the implementation of most of the error detecting codes. Many shifts and logical operations are required for encoding each word in a horizontal code. In contrast, vertical codes lend themselves into very efficient algorithms that can encode all the bit-slices in parallel — similar to the parallelism in a single-instruction multiple-data (SIMD) machine. Therefore, a vertical code is preferred for a software-implemented EDAC scheme.

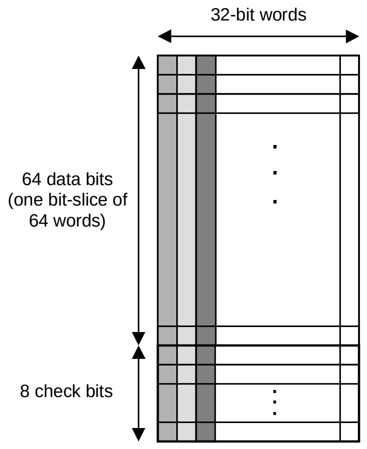

Figure 5: A vertical code over bit-slices of words.

Another aspect of these two types of codes is their handling of multiple errors. Let us assume that a *single-error correcting, double-error detecting* (SEC-DED) code is used for both types of codes. If two bit-flips occur in one word, the horizontal code cannot correct it but since each bit-flip belongs to a different bit-slice, the vertical code will be able to correct both errors. On the other hand, if two bit-flips occur in one bit-slice of a block, a horizontal code will correct both, while a vertical code will fail. In our implementation, we use a SEC-DED Hamming code in a vertical fashion. To handle multiple errors in a bit-slice, we use interleaving as shown in Fig. 6. Further details of our technique can be found in [9].

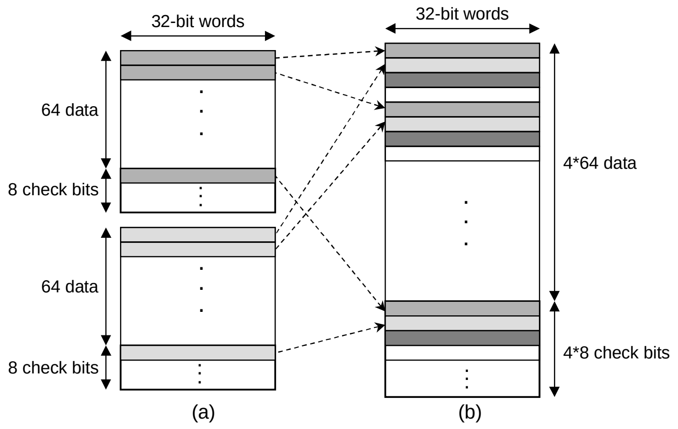

Figure 6: Logical mapping of words in a 4-way interleaving technique: (a) blocks of EDAC protected data and the corresponding check-bit words; (b) the location of these words in memory address space.

B.3) Checkpoints and Scrubbing

With hardware EDAC, the encoding is checked on each read operation and new codewords are generated on each write operation. In addition, the contents of memory are read periodically and all the correctable errors are corrected. This operation is called *scrubbing* and avoids accumulation of errors, thereby reducing the probability of multiple errors that may not be correctable.

If the same protection that is provided by hardware, is to be provided in 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. We chose to do only periodic scrubbing for software-implemented EDAC. We rely on other software-implemented error detection techniques (e.g., EDDI or CFCSS) to detect the memory bit-flip errors in code segments, if the errors are not corrected by the periodic scrubbing before the code is executed. As mentioned in Section II.A, when an error is detected after a restart, a scrub operation is enforced before a second restart is attempted.

The EDAC software is given the address and size of the memory block that needs to be protected. It requests another block from the OS to be used for the check bits. Then, it calculates the check bits (encoding) and stores them in the allocated block. Upon request, it checks for errors (decoding) and corrects them if possible. The content of the memory block may be fixed or variable. If it is fixed, the encoding is done once and the check bits remain constant. However, if the memory block is written to by the processor, the check bits have to be recalculated. There are two main types of information stored in a memory: code and data. Code segments contain instructions, and data segments contain the data that is used or produced in computations. After a program has been loaded and linked by the operating system, the contents of the code segment remain constant (with the exception of self-modifying codes that are not considered here). Therefore, a fixed set of check bits can be calculated for code segments.

Generally, the processor reads and writes to data segments, and as said earlier, it is not feasible to intercept all the write operations to update the check bits because the interceptions will incur significant performance overhead. However, for data that does not change, e.g., read-only data segments, or some calculation results that are stored for later use, the EDAC protection can be provided in software. *Application program interfaces* (APIs) can be defined so that the programmer can make function calls to the EDAC software and request protection for a specific data block. In this case, the data can also be modified through the APIs and does not have to be fixed. However, this method is not transparent to the program and the programmer needs to take control of the reads and writes to the protected data and minimize the overhead by proper design.

C. The ARGOS Project

C.1) Experiment Setup

The Stanford ARGOS project [10] is an experiment that is carried out on the computing test-bed of the NRL-801: *Unconventional Stellar Aspect* (USA) experiment on the *Advanced Research and Global Observations Satellite* (ARGOS) that was launched in February 1999. The objective of the computing test-bed in the USA Experiment on ARGOS is the comparative evaluation of approaches to reliable computing in space, including radiation hardening of processors. This goal is met by flying processors and comparing performance in orbit during the ARGOS mission. The experiment utilizes two 32-bit processors. The Hard board, built around the Harris RH3000 radiation-hardened chip set, features a self-checking processor pair configuration and has hardware EDAC for its 2MB SOI (silicon on insulator) SRAM memory. The COTS board, built around the 3081 microprocessor from IDT, uses only COTS components and has no hardware error detection mechanism. Data upload and download is possible for both boards during the mission. Therefore, it is possible to update the software in either of the processors according to the results received during the mission, and test different SIHFT techniques. The ARGOS satellite [11] has a Sun-synchronous, 800-kilometer altitude orbit with a mission life of three years. A variety of radiation environments are encountered during this mission, providing a rigorous test. SEUs are the main type of errors that we are expecting to see in ARGOS.

C.2) Status

We are currently carrying out the first example of a so-called "McCluskey test", i.e., the simultaneous operation of commercial and radiation-hardened processors of the same class in the same orbital environment. The debugging phase of our software on the satellite has finished and the boards are currently running long term tests and collecting data on the errors that occur during the mission. The programs implement many of the mentioned SIHFT techniques as well as algorithm-based fault tolerance (ABFT) [12], software duplication/TMR, and assertions (checking the validity of data at different points), to name a few. In this research, we are gathering data in an actual space environment thereby avoiding the necessity of relying on questionable fault injection.

Reconfigurable computing using FPGAs is another part of the Stanford ARGOS project. The COTS board has a Xilinx 4003 FPGA that can be reprogrammed during the mission. We will use this feature for testing the FPGA, testing other parts of the system if possible, and tolerating the faults occurring in the FPGA. FPGAs add flexibility to the system, and also, it is a good opportunity to test these devices in a space environment. The results from our research project on FPGAs and reconfigurable computing, described in the next section, can be leveraged for the Stanford ARGOS project.

III. FPLDS

The growth in computing and communication infrastructure has been in part due to the evolution of integrated circuits such as microprocessors, memory, ASICs and PLDs. Microprocessor and memory chips have been the foundation for a variety of systems ranging from embedded processors to general-purpose computers. Custom-made ASICs have catered to the needs of special-purpose applications such as graphics, signal processing, encryption and compression. PLDs are reconfigurable logic chips that allow the customer rather than the chip manufacturer to program specific functions. The key benefits of PLDs are: design flexibility and faster introduction of the product to the market. The PLDs initially started in design prototype efforts but now are increasingly used in mainstream applications like communications, data processing, industrial, networking and high reliability. Amid this spectrum of integrated circuit chips, a new concept called adaptive computing is emerging. Adaptive computing systems (ACS) represent a new technology that is derived by combining microprocessor, memory, and reconfigurable logic. ACS systems do not preclude the possibility of implementing all of the processor and memory functions in reconfigurable chips.

A. *The ROAR Project*

This work is part of the DARPA funded ROAR project at Stanford CRC. ROAR is an acronym for *Reliability Obtained by Adaptive Reconfiguration*. The objectives of the ROAR project are:

1. to develop design techniques that allow for the generation of highly dependable, adaptive computing systems with minimal loss in performance,
2. to guarantee high data integrity with no undetected errors,
3. to guarantee continuous operation by masking errors for mission critical applications,
4. to increase the availability of unattended systems by an order of magnitude through self-repair methods based on the configurability of the designs,
5. to eliminate the need for standby spares since adaptive configuration allows the use of all of the resources for performance, and
6. to develop redundancy techniques, such as diversified designs, that will protect systems against common-mode failures.

The ROAR project is addressing these objectives at various levels — ranging from the system software, architecture, and microarchitecture issues to the design, synthesis, test and diagnosis issues. The technical approach is as follows:

1. We are developing a variety of *concurrent error detection* (CED) methods for applications implemented in reconfigurable logic.
   - There is a necessity for different CED methods to meet the reliability, cost and performance goals of various target applications.
2. Techniques using design diversity are being developed to protect systems against common-mode failures.
   - Common-mode failures (CMFs) are a major reliability concern in redundant systems. CMFs have a common cause and if they affect multiple copies in an identical way then the failure is not detected and the data integrity may be compromised.
3. Instead of using stand-by spares (as is the case in traditional fault-tolerant systems), we are developing techniques that use multiple pre-compiled configurations. These configurations are loaded so as to avoid failed units in the reconfigurable hardware. Fault location algorithms help identify failed units.

B. *Application Implementations*

Multi-threading, parity-prediction, duplication, and inverse comparison are some of the CED techniques that have been developed in the ROAR project. We have successfully ported robotics control, DES encryption, and LZ-77 compress with CED capability in one of our reconfigurable test-beds [13] (Fig. 7).

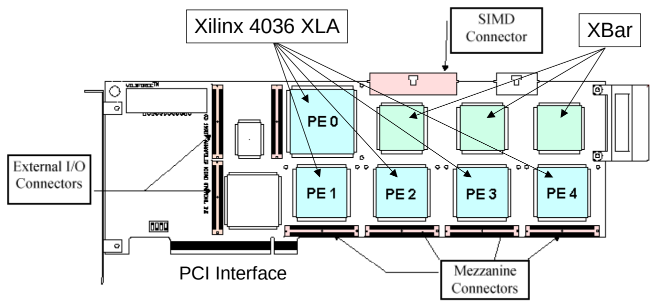

Figure 7: Annapolis Microsystem's Wildforce board.

Using this test-bed, we have also been able to demonstrate error masking and recovery for robotics control and DES algorithms by injecting faults into the look-up tables (LUTs) of the FPGAs. Both robotics control and DES implementations used multi-threading for CED. Utilization of resources by multiple threads is the idea behind multi-threading. While multi-threading is not a new idea, the idea of using multi-threading for fault-tolerance in processors and configurable logic is new and was first proposed in [14]. In contrast, for LZ-77 compression implementation, we showed that both duplication and multi-threading were not suitable CED techniques from an area efficiency point-of-view. A CED technique that uses inverse comparison was developed for LZ-77 implementation. The LZ-77 implementation (Fig. 8) comprises a sliding dictionary and processing elements (PEs). An extra PE is used to compare decompressed data with the delayed source. Details of our test-bed implementation experiments for robotics, DES, and LZ-77 compress can be found in [15].

A new synthesis technique for designing finite state machines (FSMs) with on-line parity checking has also been developed as part of the ROAR project (Fig. 9). The details of this new synthesis technique have been reported in [16]. In terms of area overhead, this synthesis technique produces better designs than duplication for a majority of benchmarks.

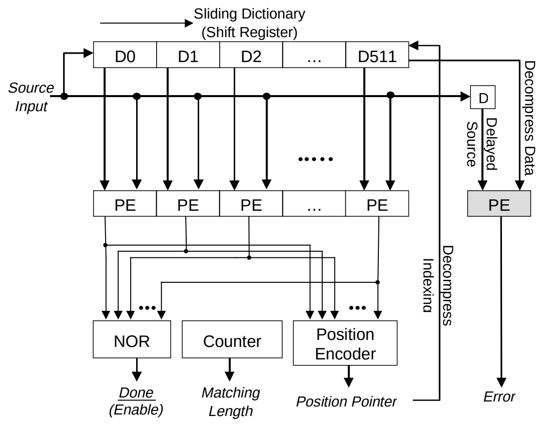

Figure 8: LZ-77 compress implementation with CED.

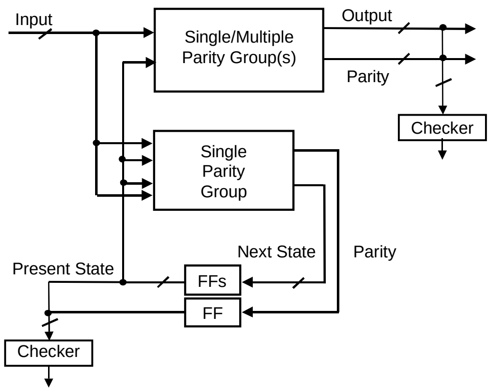

Figure 9: A finite state machine with on-line parity checking.

C. Design Diversity

For some applications, duplication is a more efficient concurrent error detection mechanism than customized CED mechanisms like parity-prediction. However, one concern about duplication is its susceptibility to CMFs. Design diversity has long been used to protect redundant systems against CMFs. Configurable logic provides an excellent opportunity to synthesize diverse designs— this was recognized in the very early stages of the ROAR project. The conventional notion of diversity relies on "independent" generation of "different" implementations. This notion is qualitative and does not provide a basis to compare the reliability of two diverse systems. We have developed a new quantitative method [17] to characterize diverse systems, and design techniques that enhance their reliability. Using analytical (derived from quantifying diversity), simulation (software modeling), and experimental (fault-injection in test-beds using diverse and non-diverse designs) methods, we have shown that for common-mode failures diverse systems improve reliability by an order of magnitude over redundant systems with identical implementations.

D. Fault Location

We have developed a pseudo-exhaustive test method to detect and locate FPGA failures [18] [19]. The FPGA hardware itself is programmed to perform the test. This method can be used to test the FPGA each time it is reconfigured. We have also developed a technique to diagnose bridging faults in the interconnect of an FPGA configuration [20]. It uses a "walking-1" approach in which only the LUTs are reprogrammed. The interconnect is tested in the way it is configured for system operation. These techniques can be used each time an ACS is reconfigured to make sure that it correctly implements the desired function.

E. Putting It All Together- An ACS Setup

We present an ACS setup that integrates some of the key contributions of the ROAR project. This setup comprises a multi-threaded processor, a configurable coprocessor, memory, and the I/O system (Fig. 10).

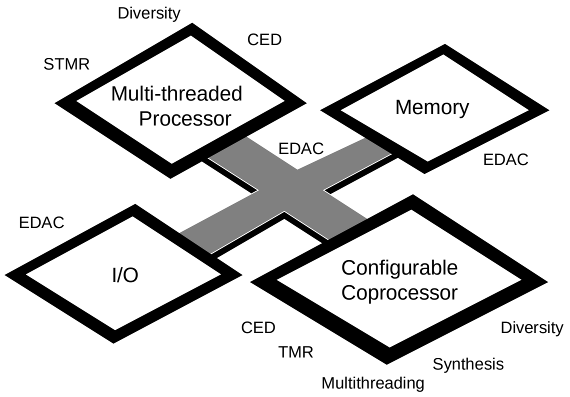

Figure 10: An ACS Setup for the ROAR Project.

The setup in Fig. 10 is different from other traditional ACS architectures in that it uses a multi-threaded processor instead of a traditional single-threaded processor. In order to support multiple contexts, processors need multiple register files and additional fetch control to manage thread switching. Processors that can support multiple contexts are called multi-threaded or multi-context [21] processors.

Fault-tolerance is accomplished by using redundant threads of computations. For example, three copies of the same thread can be run in a multi-threaded processor and the results voted upon by a voter thread. Operating system support is required to manage redundant threads and to effect recovery. A key benefit of implementing fault-tolerance with multi-threading is the accomplishment of a level of reliability and performance similar to that of TMR at almost the cost of simplex hardware. Another key benefit is that the implementation of fault-tolerance does not require any new design features and uses all of the architectural features that are already present in multi-threaded processors. If individual threads fully utilize the resources then multi-threading could degrade performance due to resource contention. By implementing fault-tolerant applications using multiple threads, it is possible to recover from temporary faults and some permanent faults.

A coprocessor is a special execution unit that extends the processing capability of processors. Coprocessors have been used in commercial processors like SGI MIPS, Hewlett-Packard Precision RISC, Sun's SPARC, and IBM PowerPC to provide special functions like floating-point and graphics operations. A reconfigurable coprocessor provides special logic configuration functionality. The instruction set of processors in ACS architectures may allow flexible extensions by means of reconfigurable coprocessor instructions. Coprocessor instructions are instructions in which the data movement functions are defined between the processor or the memory and the reconfigurable coprocessor, but the data transformations are left unspecified. Instructions that specify data movement functions are essentially coprocessor load/store instructions. These instructions must provide a generic interface for moving data to and from the coprocessor. Data transformation coprocessor instructions provide the flexibility of defining data manipulation operations for each instance of reconfigurable coprocessor logic depending upon the application. For example, the data transformation function could be a dictionary-based compression for a compress application, or a block cipher function for an encryption application.

Potential implementations of ACS are multi-board systems, multi-chip systems, or a single system-on-a-chip. Rather than building a specific implementation, we are using simulation programs and emulation test-beds [13][22] to model instances of the ROAR ACS setup. This gives us the flexibility to study the reliability and performance benefits for a variety of implementations. Figure 11 shows how an application in a traditional general-purpose processor can be accelerated using an ACS platform. The acceleration in performance comes from identifying application segments that can be implemented with fine-grain parallelism in the reconfigurable logic. Successful porting of various applications in reconfigurable logic has been demonstrated both in academia and in industry. A large body of work appears in the proceedings of IEEE FCCM and ACM FPGA conferences. The main emphasis in the reported work has been on improving the performance of the ported application. In addition to performance improvement, the emphasis in the ROAR project is to improve the reliability of reconfigurable logic designs.

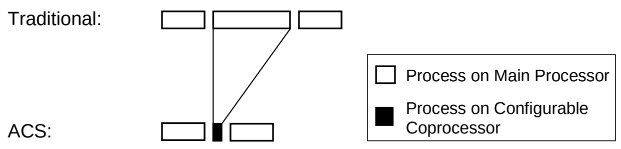
Traditional:
ACS:
Process on Main Processor
Process on Configurable Coprocessor

Figure 11: Accelerating a segment of an application on a configurable coprocessor.

For dependable computing, the integrity of computation in all stages should be immune to both temporary and permanent faults. While permanent faults can be detected by off-line diagnostic testing, on-line testing and concurrent error detection (CED) may be required to protect against temporary faults. The level of protection and recovery mechanism is, of course, determined by the dependability requirements of a particular application. The configurability of the reconfigurable coprocessors presents several opportunities to design and synthesize data transformation functions with error detection and correction capability. Figure 12 shows implementations of *software TMR* (STMR) with three identical threads (labeled A, B and C) running on a single-threaded processor, on traditional ACS, and on the ROAR ACS setup. The illustration in Fig. 12 assumes that the multi-threaded processor and the reconfigurable coprocessor can support up to two simultaneous threads.

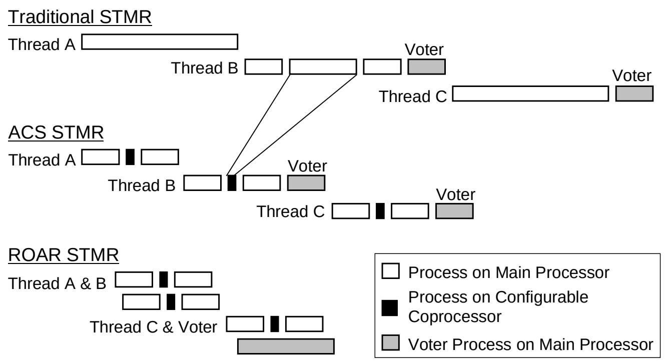

Figure 12: Three different implementations of STMR.

The concurrent error detection capability coupled with multi-threading provides protection against faults in the reconfigurable logic and the processor. We have models that demonstrate that the ROAR ACS setup improves application reliability by two orders of magnitude over simplex STMR and an order of magnitude over traditional ACS STMR.

An important aspect of the ROAR ACS setup (Fig. 10) is that it has no functional feature that has been specifically designed for fault-tolerance. The customer gets the flexibility to program fault-tolerance. This flexibility is due to the architectural features of multi-threaded processors and the programmability of reconfigurable logic. We believe that ACS architectures such as ROAR will not only enhance the effectiveness of traditional ACS architectures but will also, for the first time, make fault-tolerance viable in the COTS market.

ACKNOWLEDGMENTS

The Stanford ARGOS project is supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy, Office of Naval Research under Grant Nos. N00014-92-J-1782 and N00014-95-1-1047. ARGOS is a collaborative project with the Naval Research Laboratory (NRL) USA experiment group lead by Dr. Kent Wood as the principal investigator.

The ROAR project is supported by Defense Advanced Research Project Agency (DARPA) under Contract No. DABT63-97-C-0024. The authors would like to acknowledge contributions of Chaohuang Zeng and the emulation support provided by Quickturn Design Systems. University of Texas at Austin is a subcontractor in the ROAR project.

References

[1] Koga, R., and W.A. Kolasinski, “Heavy Ion-Induced Single Event Upsets of Microcircuits; A Summary of the Aerospace Corporation Test Data,” *IEEE Trans. on Nuclear Science*, Vol. 31, No. 6, pp. 1190-1195, Dec. 1984.

[2] Ziegler, J.F., et al., *IBM J. Res. Develop.*, Vol. 40, No. 1, (all articles), Jan. 1996.

[3] Shirvani, P.P., and E.J. McCluskey, “PADded Cache: A New Fault-Tolerance Technique for Cache Memories," *17th IEEE VLSI Test Symposium*, pp. 440-445, Dana Point, CA, Apr. 24-29, 1999. *A

[4] Bartlett, J., et al., “Fault Tolerance in Tandem Computer Systems,” *Tandem Technical Report 90.5*, Tandem Computers Inc., Cupertino, CA, May 1990.

[5] Oh, N., P.P. Shirvani and E.J. McCluskey, "Error Detection by Duplicated Instruction in Superscalar Microprocessors," *CRC-TR*, in preparation. *

[6] Oh, N., P.P. Shirvani and E.J. McCluskey, "Control-Flow Checking by Software Signatures,” *CRC-TR*, in preparation. *

[7] Hodgart, M.S., “Efficient Coding and Error Monitoring for Spacecraft Digital Memory,” *Int’l J. Electronics*, Vol. 73, No. 1, pp. 1-36, 1992.

[8] Patel, A.M. and S.J. Hong, “Optimal Rectangular Code for High Density Magnetic Tapes,” *IBM J. Res. Develop.*, Vol. 18, pp. 579-88, November 1974.

[9] Shirvani, P.P., N. Saxena and E.J. McCluskey, "Software-Implemented EDAC Protection Against SEUs," *CRC-TR*, in preparation. *

[10] Shirvani, P.P. and E.J. McCluskey, “Fault-Tolerant Systems in a Space Environment: The CRC ARGOS Project,” *CRC-TR 98-2*, Dec. 1998. *A

[11] 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.

[12] Huang, K.-H., et al., “Algorithm-Based Fault Tolerance for Matrix Operations,” *IEEE Trans. on Comp.*, Vol. C-33, No. 6, pp. 518-28, June 1984.

[13] Annapolis Micro Systems Inc., Wildforce FPGA Board *http://www.annapmicro.com/*, 1999.

[14] Saxena, N. and E.J. McCluskey, “Dependable Adaptive Computing Systems,” *IEEE Systems, Man, and Cybernetics Conf.*, San Diego, CA, pp. 2172-2177, Oct. 11-14, 1998. *R

[15] Saxena, N., S. Fernandez-Gomez, W. Huang, S. Mitra, S. Yu, and E.J. McCluskey, “Dependable Computing and On-Line Testing in Adaptive and Reconfigurable Systems,” to appear *IEEE Design and Test Magazine*, Jan-Mar 2000. *R

[16] Zeng, C., N. Saxena, and E.J. McCluskey, “Finite State Machine Synthesis with Concurrent Error Detection,” *Proc. IEEE Int’l Test Conf.*, pp. 672-679, Sep. 1999. *R

[17] Mitra, S., N. Saxena and E. J. McCluskey, "A Design Diversity Metric and Reliability Analysis for Redundant Systems," *Proc. IEEE Int’l Test Conference*, pp. 662-671, Sep. 1999. *R

[18] Mitra, S., P.P. Shirvani, and E.J. McCluskey, "Fault Location in FPGA-Based Reconfigurable Systems," *IEEE Int’l High Level Design Validation and Test Workshop*, La Jolla, CA, Nov. 12-14, pp. 143-150, 1998.*A

[19] Quddus, W., A. Jas, and N.A. Touba, "Configuration Self-Test in FPGA-Based Reconfigurable Systems", *Proc. of IEEE Int’l Symp. on Circuits and Systems*, pp. 97-100, 1999.

[20] Das, D., and N.A. Touba, "A Low Cost Approach for Detecting, Locating, and Avoiding Interconnect Faults in FPGA-Based Reconfigurable Systems," *Proc. of IEEE Int’l Conf. on VLSI Design*, pp. 266-269, Goa, India, Jan. 7-10, 1999.

[21] Laudon, J.P., *Architectural and Implementation Tradeoffs for Multi-Context Processors*, Ph.D. Dissertation, Electrical Engineering Dept., Stanford University, May 1994.

[22] Quickturn Design Systems (now part of Cadence Design), *http://*www.*quickturn.com/* or *http://*www.*cadence.com*/, 1999.

* Draft version available at: http://crc.stanford.edu/projects/argosPapers.html

*A Available at: http://crc.stanford.edu/projects/argosPapers.html

*R Available at: http://crc.stanford.edu/projects/roar/roarPapers.html