Dependable Computing and Online Testing in Adaptive and Configurable Systems

Nirmal R. Saxena, Santiago Fernandez-Gomez, Wei-Je Huang, Subhasish Mitra, Shu-Yi Yu, and Edward J. McCluskey
Stanford University

We describe cost-effective design techniques for very dependable high-performance adaptive computing systems. These design techniques take advantage of the particular properties of adaptive computing systems.

The growth in computing and communication infrastructure has been in part due to integrated circuit chips like microprocessors, memory, application-specific integrated circuits (ASICs), and programmable logic devices (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 configurable 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 marketplace. PLDs were initially used as design prototypes but now are increasingly used in mainstream applications (see www.xilinx.com and www.altera.com), such as communications, data processing, industry, networking, and high reliability.

Amid this spectrum of IC chips, a new concept called adaptive computing is emerging. Adaptive computing systems (ACSs) represent a new technology that is derived by combining microprocessor, memory, and configurable logic. Academia and industry[1,2,3,4] are actively pursuing research and development in the ACS area. ACS designs open up new opportunities for compilers and hardware synthesis tools[5,6] to optimally map applications into segments that run well in the processor-memory complex and into segments that run well in the configurable logic.

ACSs are well-suited for applications such as digital signal processing, image processing, control, bit manipulation, compression, and encryption algorithms. Most of the research and development in the ACS area has been dictated by the performance requirements of target applications. Performance is an important requirement; however, the use of ACS applications in areas like nuclear reactors, fly-by-wire systems, remote (space station and satellite) locations, and life-critical systems makes dependability an equally important requirement. Dependability comprises availability, reliability, testability, maintainability, and fault tolerance. Dependable computing, in general, requires the following functionality:

- Offline test methods to detect and isolate failed components. This may be necessary for initial configuration (fault avoidance) and during reconfiguration (fault location and repair).

0740-7475/00/$10.00 © 2000 IEEE

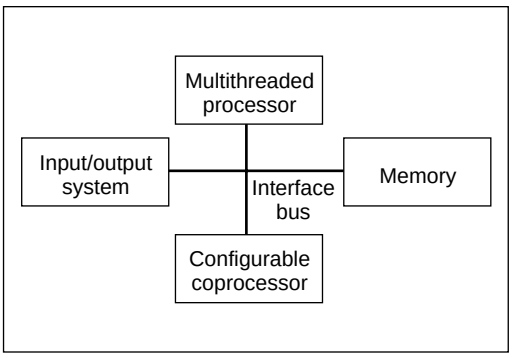

Figure 1. MT-ACS architecture.

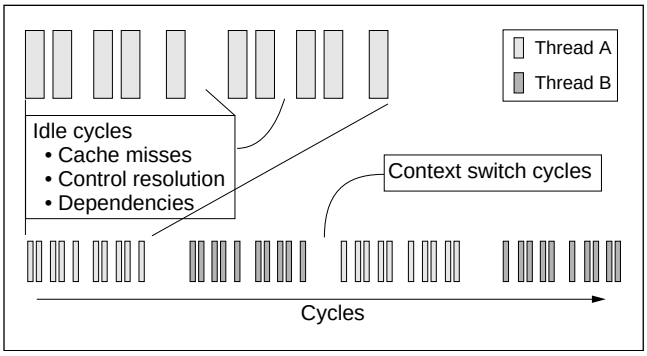

Figure 2. Multiprogramming on a traditional microprocessor.

- Online test and checking methods to detect and identify system failures during normal system operation.
- Error recovery methods to restore operation of the system after failure.

With very few exceptions,[8,13,14,47] little work has been done in the dependable ACS area.

This article describes our recent work on dependable computing in ACS platforms. This work is part of the Department of Defense Advanced Research Project Agency-funded ROAR project at Stanford University's Center for Reliable Computing. ROAR is an acronym for reliability obtained by adaptive reconfiguration. ROAR's objective is to use the adaptive and reconfiguration capabilities of ACS for high performance and fault tolerance. ROAR's proposed approach does not preclude adoption of earlier defect-tolerance and FPGA fault-tolerance work. In addition, ROAR is also addressing system-level and application-level fault tolerance for runtime errors in the ACS.

In this article, we show application of ROAR's results to an ACS architecture (see Figure 1) that uses a multithreaded processor instead of a single-threaded processor. We call this the MT-ACS architecture.

MT-ACS architecture

Rather than building a specific implementation, we are using simulation and emulation testbeds (see www.cadence.com and www.annapmicro.com) to model the MT-ACS architecture. This gives us the flexibility to study the reliability and performance benefits for a variety of implementations.

In the following sections, we will describe essential features of multithreaded processors and configurable coprocessors.

Multithreaded processors: concepts and architecture

This section introduces the concept and architecture of multithreaded processors.[36,37,38] For the sake of clarity, we present necessary definitions and background information to distinguish the concept of multithreaded processors from other "multi" terminology found in the computer architecture and operating systems literature. The concept of multiple programs taking turns in execution is known as multiprogramming. Multiprogramming is also known as multitasking. A computer system that has more than one processor is called a multiprocessor. The implementation of multiprogramming in a multiprocessor is multiprocessing. The operating system (OS) schedules multiple processes or threads. In this article, we will use the terms *thread* and *process* as synonyms.

Figure 2 illustrates multiprogramming in a traditional processor. Under the control of the OS, the processor context is switched from one thread to another thread. The change in context could be caused by the thread itself or when the OS preempts. The OS preempts the running threads at specified time quantum (typically in hundreds of milliseconds). The time quantum is usually programmed using an interval timer register. The interval timer interrupts the processor on expiration of the time quantum to signal transfer of control to the OS. The OS uses this opportunity to either continue the

existing thread or switch to another thread. The context of a thread is all of the associated state (register, memory, and control) that is necessary to restart the thread. The time taken to switch the processor's context associated with a thread is the context switch time (usually in hundreds of microseconds).

Figure 2 shows the switching of threads labeled A and B separated by the OS consumed context switch cycles.

The vertical bars in Figure 2 denote single or multiple instructions executed in a given cycle. The gaps between the instructions executed in a thread are due to events like cache misses, branch resolution latency, and other data flow dependency events. These events cause underutilization of the processor resources. Also, the context switch overhead per thread affects the system throughput. The system throughput can be improved if the processor is capable of holding multiple-thread contexts. For example, idle cycles within the context of a thread could be utilized by scheduling instructions from another thread. The OS can take advantage of the processor's capability to hold multiple-thread contexts and reduce the number of context switches per thread. 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 multithreaded or multicontext processors.[45]

Multithreading and fault tolerance. While multithreading is not a new idea, the idea of using multithreading for fault tolerance in processors and configurable logic is new and was first proposed by Saxena and McCluskey.[13,14] Fault tolerance is accomplished by using multiple threads of computations and algorithms. For example, three copies of the same thread can be run in a multithreaded processor and the results voted on by a voter thread. OS support is required to manage redundant threads and effect recovery. A key benefit of implementing fault tolerance with multithreading is the accomplishment of a level of reliability similar to that of NMR at almost the cost of simplex hardware. Another key benefit is that the implementation of fault tolerance does not require any new design feature and uses all of the architectural features that are already present in the multithreaded processors. By implementing fault-tolerant applications using multiple threads, it is possible to recover from temporary faults and from some permanent faults.

Configurable coprocessor

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 configurable coprocessor extends the functionality of the processor by providing special logic configuration functionality. The instruction set of processors in ACS architectures may allow flexible extensions by means of configurable coprocessor instructions. Coprocessor instructions are instructions in which the data movement functions are defined between the processor or the memory and the configurable coprocessor, but the data transformations are left unspecified.

The two emulation testbeds that are being used to model configurable designs in our project are the QuickTurn System Realizer M250 (http://www.cadence.com) and the Annapolis Micro Systems Inc. Wildforce Board (http://www.annapmicro.com). The System Realizer M250 is a modular system that uses FPGA-based emulation technology for emulating ASICs. The Wildforce Board is a standard-size PCI card containing four FPGAs, one Control Processing Element, a user-programmable crossbar, and provisions for add-on capabilities.

The configurable coprocessor is used in a variety of data transformation functions with the twin objectives of high performance and fault tolerance. We have developed fault-tolerant design techniques for robotics, compression, signal processing, and encryption applications. Next we describe experiments and results of porting of these designs on our testbeds. Then we describe different software models that can be used for dependable computing in the MT-ACS architecture.

Porting of applications in the configurable coprocessor

Successful porting of various applications in

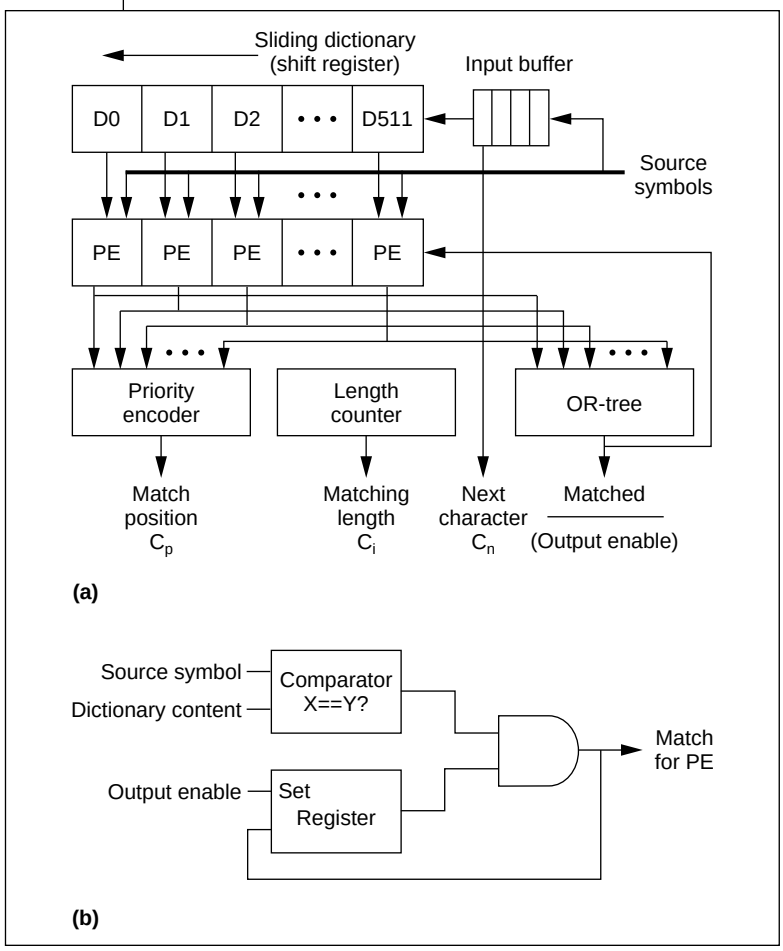

Figure 3. A systolic array implementation of LZ compress encoder.

configurable logic has been demonstrated both in academia and in industry. We have developed and are developing configurable logic microarchitectures for compression, encryption, control, and signal processing applications that meet the twin objectives of high performance and reliability. Some of these are discussed here.

Compression algorithms

Data compression provides an inexpensive solution to optimize the utilization of limited resources in communication and storage systems. Many authors[18] have proposed lossless data compression algorithms to achieve this objective. We have chosen the Lempel–Ziv (LZ)-based data compression[16,17] algorithm as an initial candidate for an ACS implementation. LZ-compress algorithm is dictionary-based. Two hardware approaches that have been used to accelerate LZ-compress are content addressable memory (CAM)-based and systolic array-based.

We used a systolic-array approach for the LZ-compress on our emulation testbeds. The idea in the systolic-array approach is to separate the comparators from the CAM-based dictionary cells and to balance the trade-off between throughput and the number of array components. A special high-performance case proposed in Jung and Burleson[20] with the same throughput as the CAM approach is shown in Figure 3a. In this architecture, the sliding dictionary is implemented in shift registers, and each processing element (PE) executes the string comparison function. The detailed PE structure is shown in Figure 3b, with the match result of each PE now activated by the output of the same PE in the previous cycle.

Fault tolerance in lossless compression techniques is crucial because these techniques are frequently applied in digital data coding with bit-precision requirements. To improve the reliability, the traditional NMR technique adds identical functional modules. Each module generates a copy of output code words, and a comparator at the output stage detects if an error occurs. This method can greatly reduce the error rate. Nevertheless, the price to duplicate all hardware resources is very high, because this would amount to replicating the entire dictionary. A more efficient method is to augment the microarchitecture in Figure 3 with concurrent error detection capability, thereby allowing the opportunity to maximize the dictionary size for performance.

For lossless data compression, the fundamental reliability requirement for the encoder is to ensure that the output code can be correctly reconstructed in the decoder. Therefore, if we emulate the reverse process on the compressed code words and compare the results with the uncompressed source data before emitting the output, the integrity of the encoder can be guaranteed. The conceptual scheme of this error-detection technique is shown in Figure 4, where a decoder decompresses the encoded code words, and a checker compares the reconstructed data with the delayed source input. We call this the inverse comparison technique. The inverse comparison technique is

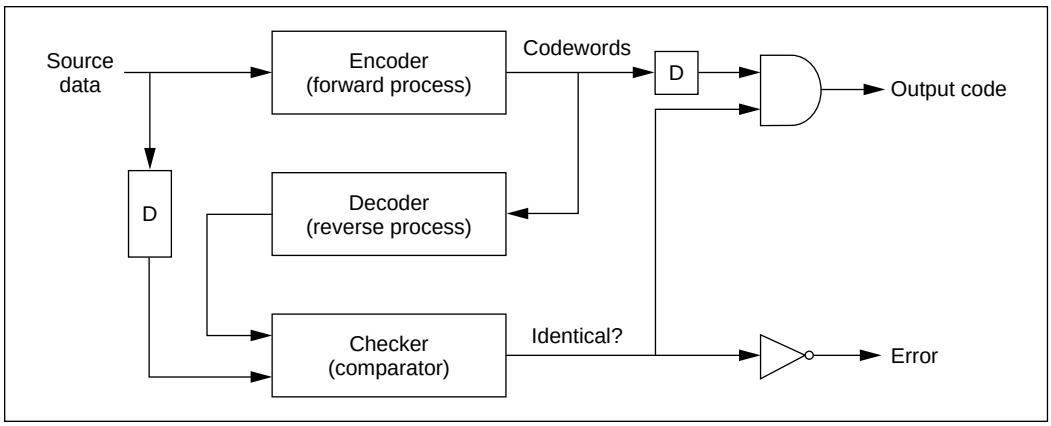

Figure 4. A concurrent error detection scheme for LZ compress.

appropriate for applications in which the complexity of the inverse operation is low. The decompression of LZ implementation in Figure 4 is simply a memory access, and thus the extra cost for error detection can be smaller than for other implementations of LZ-based algorithms.

Table 1 shows implementation results of our proposed scheme in the Quickturn and Wildforce emulation testbeds. The decoder shares the dictionary with the encoder, and the checker block is implemented by using an extra PE.

Table 1. Systolic array LZ-compress implementation on emulation platforms.

	Number of 4013LCA FPGAs	Number of 4036XLA FPGAs
Implementation	**Quickturn M250**	**Wildforce Board**
Without inverse comparison	14 chips	4 chips
With inverse comparison	18 chips	5 chips

Table 2. Performance comparison with C implementations.

Parameters	Wildforce	C program on Ultra II	C program on Xeon
Number of chips	4	1	1
Clock rate	16 MHz	300 MHz	450 MHz
Throughput	128 Mbps	3.8 Mbps	5.2 Mbps

We have been able to achieve a compression throughput of 128 Mbps/sec on the Wildforce emulation board for an LZ-77 implementation without inverse comparison. Using several benchmark files, we ran experiments on a 300-MHz Sun Ultra II workstation and a 450-MHz Xeon workstation to measure the compression throughput for a software version of LZ-compress. We obtained the source code for the LZ-compress from Nelson and Gailly.[18] We used the best available compilers to compile this source code. Table 2 shows these results.

Even at 16 MHz, the Wildforce implementation achieves more than a factor of 20 improvement in the compression throughput. We believe that two orders of magnitude throughput improvement is attainable using higher capacity FPGAs.

Robotic control algorithms

Robots are frequently utilized under hazardous or human-inaccessible circumstances, such as in nuclear power plants or in spacecrafts. Researchers have been attempting to provide an effective fault-tolerant robotic system. Traditionally, control algorithms have been instrumented on general-purpose processors. For example, a mobile Puma 560 Manipulator[15] implements a control equation to compute the force strategies for position control entirely in a compiled C program running on a dedicated processor. To demonstrate the feasibility of porting control algorithms to ACS platforms, we took a second-order linear control algorithm and developed several configurable designs.

A general structure of this control algorithm is illustrated in Equation 1. In control theory lit-

Table 3. Different implementations of DES in configurable hardware.

	DES		With parity check		2-threaded		2-threaded with parity check	
	Synth.	P&R	Synth.	P&R	Synth.	P&R	Synth.	P&R
Area (no. of CLBs)	265	452	353	514	438	586	533	647
(% of FPGA)	(21)	(34)	(28)	(39)	(34)	(45)	(42)	(49)
Speed (MHz)	33	45.5	30	31.1	24.4	30.4	21.6	30.1

erature, this is called the proportional integral differential algorithm:

$$f = m\left(\ddot{x}_d + K_p e + K_i \int e dt + K_d \dot{e}\right) + b\dot{x} + kx \quad (1)$$

where f is the force output of the controller; x_d is the desired position; x is the actual position; e is the error between the desired and the actual positions; and m, K_p, K_i, K_d, b, and k are coefficients for the control equation. A fully parallel and discrete implementation of this proportional integral differential algorithm would have taken nine adders and six multipliers. We developed several configurable logic designs to implement the control equation with multithreading capability for error detection. Unlike general-purpose microprocessors, another flexibility we had in configurable logic was to control the precision of adders and multipliers. We found that even with 16-bit arithmetic precision, the control system attained similar response time and stability as the one obtained using 32-bit arithmetic precision. The performance obtained, measured in terms of time to compute the force output, using the Wildforce implementations was comparable and in many cases better than that obtained with a C program implementation running on a 300-MHz Ultra II processor.

To test the fault-tolerance capability of multithreading in the robotics controller, we injected several faults (by modifying the look-up table bits in the FPGAs) in the Wildforce implementations. For all of the injected faults in a single-threaded implementation, the output response was significantly different from the fault-free response. The three-threaded implementation using a masking voter in the configurable logic masked most of the injected faults. However, for some faults (most likely in the shared resources between multiple threads), the faulty response was unacceptable. These results show that with multithreading, fault-tolerance capability can be accomplished. However, there is vulnerability to faults in shared resources.

Data encryption standard

Encryption has become a fundamental part of communications because of the growing need for secure transmission of information through insecure channels such as public computer networks. Encryption's origin goes back 4,000 years, but the major advances are due to the intensive research in the past four decades.[21,22,23]

For a comprehensive approach to encryption, see Schneier.[24]

One of the most used encryption methods for the past 20 years has been the data encryption standard (DES) algorithm.[25] With speed as an objective, several commercial software and hardware implementations have been announced.[24] ACS implementations with significant performance improvements over software DES have been suggested.[2,4] We have recently ported four different implementations on the Wildforce emulation testbed. These implementations were as follows:

- without error detection
- with parity-encoded S-Boxes
- two-threaded design
- two-threaded design with parity-encoded S-Boxes

Table 3 shows the areas and maximum clock rates for these four implementations of DES.

The plain text and cipher text processing was done using a C program running on a 450-MHz Pentium II processor. The hardware design of DES on the Wildforce board was

accessed using C functions. In order to compare the throughput of software and hardware implementations, we applied several input vector sets. Results are summarized in Table 4.

Some comments are in order about the results reported in Table 4. The overall 1.92 Mbytes/sec DES bandwidth (i.e., including the C program and the hardware implementation on the emulation board) did not change, even when we ran the hardware design at a higher clock rate, because the bandwidth was limited by the I/O bandwidth of the Wildforce Board. We measured the peak I/O bandwidth of the Wildforce Board at 6 Mbytes/sec. Even with the I/O bandwidth restriction, we see more than a factor of 20 speedup over the best software implementation. In an actual implementation of the MT-ACS architecture, one should expect much higher speedups. Fault-injection experiments on DES implementation demonstrated characteristics similar to the robotics experiments. While fault tolerance was accomplished for most of the injected faults, vulnerability remained for faults in common resources. This vulnerability can be mitigated by using design diversity techniques.

Table 4. DES bandwidth results.

DES implementation	Throughput
Software implementation (Phil Karn and Jim Gillogly Code)	28 Kbytes/sec
Software implementation (Cryptlib version 2.1 by Peter Gutmann)	134 Kbytes/sec
Configurable DES implementation (Wildforce Board running at 20 MHz, throughput includes C code running on CPU)	1.92 Mbytes/sec
Peak throughput for single-threaded design (without I/O bandwidth limitation of Wildforce Board)	16.38 Mbytes/sec
Peak throughput for two-threaded design (without I/O bandwidth limitation of Wildforce Board)	22.56 Mbytes/sec

Design diversity: a new ACS opportunity

Design diversity can prove to be useful in the context of dependable ACS. The programmability of FPGAs can be utilized to achieve diversity among the different modules. By utilizing the reconfiguration capability of the ACS systems, together with design diversity and a reduced number of spares, there are opportunities to improve the dependability of systems over conventional techniques.

In the context of the MT-ACS architecture, diversity is applicable to all the different components of an ACS. For example, for the processor, we could use diverse versions of the software for accomplishing a specific task and voting the results of the different versions (*N*-version programming). For designs mapped to FPGAs, we can use synthesis techniques for diverse implementations of different logic circuits. At the microarchitecture level, we can schedule the different threads of our applications (e.g., robotics, fast Fourier transforms, compression, DES) in a diverse way to different units using diverse scheduling techniques.[46,47]

Application performance and reliability

On-chip resources in current-generation microprocessors[28,29] are underutilized. On-chip idle resources or spare capacity can be utilized for error detection and fault tolerance. This observation has been the basis of other work.[30,31,32] There is some reliance on microarchitecture changes[30,31,33] or compiler support[32,33] to manage the spare capacity for fault tolerance. The microarchitecture changes[30,31,33] are needed to dynamically schedule redundant operations during idle cycles for error checking. In superscalar or very large instruction word microprocessors, spare capacity can be identified during compile time.[32,33] Therefore, with compiler support, redundant instructions can be scheduled for error checking. The advantage of a compiler-based approach is that no hardware change for fault tolerance is required. With OS support, an inexpensive solution to make an ACS application dependable is to run multiple redundant copies and use a separate process to detect and mask errors from multiple runs. This is software-based NMR and is possible because of the ability of the ACS architecture to support multiprogramming. The disadvantage of this approach in that the performance of the ACS application is degraded. For example, a software triple modular redundancy (STMR) implementation of the ACS

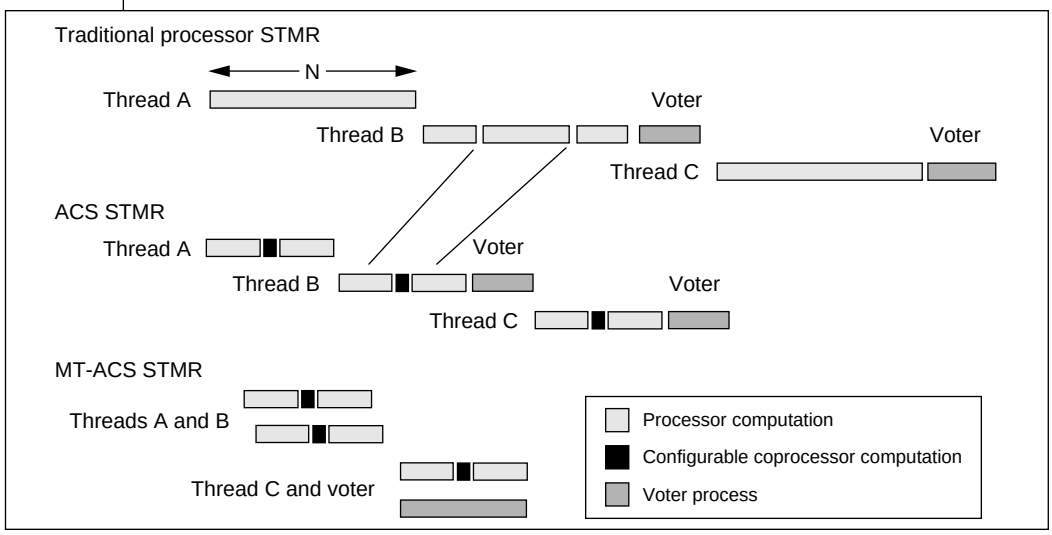

Figure 5. Software triple modular redundancy models.

application that runs entirely on the processor would slow down the performance by at least a factor of two. This performance slowdown is mitigated by using MT-ACS architecture. Figure 5 illustrates STMR execution models for an application in traditional, ACS, and MT-ACS architectures.

In general, the multithreading capability of MT-ACS architecture permits better performance than traditional ACS architectures. This performance advantage also translates to a reliability advantage. In all these models, it is assumed that the voter process looks for specified memory locations for the corresponding redundant threads to effect comparison and error masking. When the voter thread detects disagreement between redundant threads, it may not be able to isolate the failure to a processor or configurable coprocessor. In order to improve the diagnostic resolution for the voter thread, it is useful to enhance the design implemented in the configurable coprocessor with concurrent error detection capability. Concurrent error detection in the configurable coprocessor also protects against permanent faults that can corrupt the computation of all redundant threads.

We are developing analytical, simulation, and emulation methods to quantify the performance and reliability gains in the MT-ACS architecture. For example, above, using emulation, we showed potential performance improvements attained by porting applications to ACS implementations. Unlike traditional ACS architectures, another performance-enhancing aspect of the MT-ACS architecture is the use of a multithreaded processor. In this section, we will briefly cover performance aspects of multithreading and estimate reliability improvements in the ACS and MT-ACS architectures over traditional computing methods.

Multithreaded processors improve the overall instruction throughput by increasing the utilization of idle resources. The improvement in throughput can be quantified by the ratio between the total instruction throughput (all threads combined) of a multithreaded processor and the total instruction throughput in a single-threaded processor. This ratio is the speedup in instruction throughput attained by multithreaded processors over the instruction throughput in single-threaded processors. The value of speedup depends on the multithreaded architecture (i.e., the number of threads supported and the type of multithreading) and application characteristics. Figure 6 shows our analytical estimate of speedup for a Lempel–Ziv–Welch compress benchmark application running on models of a multithreaded processor supporting up to eight threads. These speedup values corroborate measured and published results.[45,48] For example, with two threads, we can see a speedup of 1.6.

The failures in computing systems can be attributed to internal or external causes. The internal causes include permanent wear-out failures and intermittent failures due to weak and marginal parts. External causes include permanent and temporary failures induced by temperature, power supply, and radiation fluctuations. In space and other high-radiation environments, it has been observed that memo-

ry logic state could switch due to external radiation emissions. This is termed single-event upset (SEU). Certain SEUs do not permanently damage the memory devices, so the correct memory state and operation can be restored after reinitialization. Actel data[10] suggest that the rate of SEU for memory and FPGA devices is around 10^{-7}/bit/day. At this rate, a 10-megabit storage device could expect one upset per day. To give a reliability perspective, an application without any fault-tolerance protection running continuously (like an OS) on a microprocessor in a satellite using a working set (WS) of 10 megabits (about one Mbyte) could fail once every day.

One of the reasons we have chosen the reliability analysis with respect to SEUs is that as a fault-avoidance technique, radiation-hardened devices[10,11,12] are being used to protect against radiation-induced upsets. However, there is generally a limited availability of these parts. Also, the available parts are several generations behind their commercial, off-the-shelf (COTS) counterparts. For example, the commercially available MIPS radiation-hardened microprocessor R3000 is at least six generations behind the commercially available MIPS R12000 microprocessor. There is increasing interest in using COTS components with fault-tolerance techniques to either replace or reduce the dependence on radiation-hardened parts. For example, a related project[49] is conducting a satellite experiment to measure error data in space environments and also compare the effectiveness of an R3000 MIPS COTS processor (using software-implemented hardware fault-tolerance techniques) with respect to an R3000 radiation-hardened processor.

In the SEU environments, increasing the application performance (through improved algorithms, new architectures, or increased clock rate) increases the reliability of computation. This is because a computation that completes in a shorter time is less likely to be hit by an SEU. This beneficial relationship between performance and reliability may not hold for failures that are due to internal causes. For example, increasing the clock rate of a processor may accelerate intermittent failure mechanisms related to circuit delays. Assuming SEU as the failure mechanism, we now present quantitative reliability estimates for STMR models.

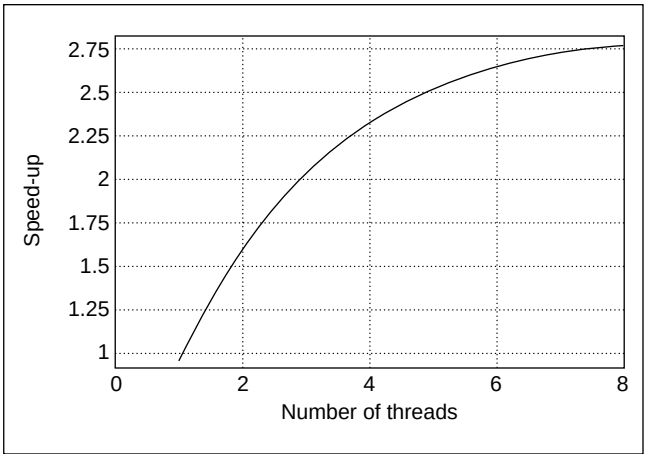

Figure 6. Instruction throughput improvement in multithreaded processors.

The variable N, in Figure 5, denotes the single-thread execution cycles (in billions). The SEU rate used for the reliability plots is 10^{-7}/bit/day. We assume that the processor is running at a 100-MHz clock rate. With this clock rate, the 10^{-7} upsets/bit/day translates to 1.15×10^{-12} upsets/bit/cycle.

Note that the x-axis is not exactly a time reference as is used in traditional reliability plots to compare hardware triple modular redundancy plots. In STMR, the completion time is more than that of a simplex thread. For a common reference, we use the simplex execution cycles (in billions) to index the x-axis; however, in terms of reliability calculation, we use the actual cycles used by a particular STMR model. To illustrate, in Figure 7, the reliability of traditional STMR at $N = 300 \times 10^9$ simplex execution cycles would approximately correspond to the 900×10^9 execution cycles (threads A, B, and C) and the voter cycles. Figure 7 presents reliability plots for several STMR models. We assumed that acceleration due to a configurable coprocessor gives ACS applications a factor of 10 improvement over the traditional simplex. ACS simplex is the simplex execution on an ACS platform (without multithreaded processor). The speedup due to multithreading in MT-ACS is assumed to be 1.65. The size of data produced by each thread is assumed to be similar to the WS size of the application.

The plots in Figure 7 clearly demonstrate sig-

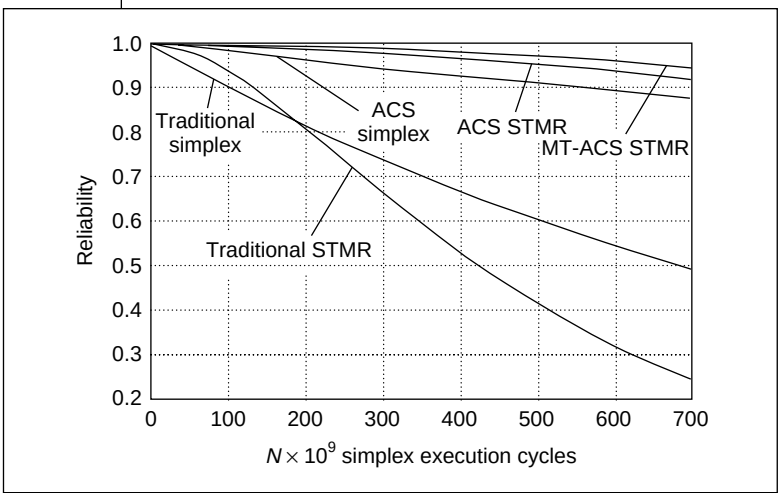

Figure 7. STMR reliability comparisons.

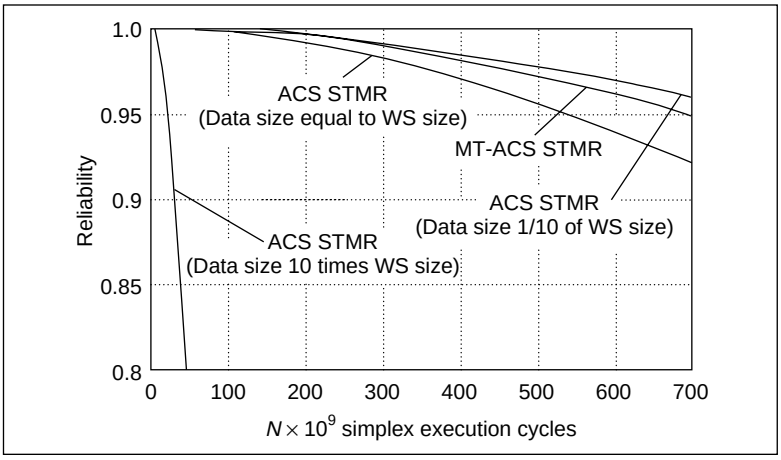

Figure 8. ACS STMR reliability sensitivity to data size.

nificant reliability improvements of ACS and MT-ACS STMR over traditional STMR. While not clear in the plot, the traditional STMR reliability is higher than that of ACS simplex for up to 5×10^9 traditional simplex cycles. The MT-ACS STMR shows higher reliability than all models because of the performance benefits derived from both multithreaded and configurable computing. However, with multithreaded computing, the single-thread performance degrades (as is indicated by the slowdown) at the expense of overall throughput improvement. This has some negative bearing on the reliability of individual threads.

Unlike MT-ACS STMR, the ACS STMR reliability is sensitive to the amount of data size produced by the redundant threads (as these data have to wait in volatile storage before voting). Figure 8 illustrates this sensitivity. In general, we can conclude that the ACS STMR reliability for large data sizes (greater than WS size) degrades significantly and for small data sizes improves over that of MT-ACS STMR. A factor that can degrade MT-ACS STMR reliability is when there is no significant improvement in instruction throughput due to multithreading. This can happen due to resource thrashing caused by multithreading. However, according to measured data,[45] thrashing does not seem to be a significant issue.

We want to point out though that the MT-ACS architecture is a more general case of traditional ACS architectures. That is, it can emulate ACS applications by running only one thread at a time for applications with small data sizes. In other words, MT-ACS architecture can adapt to application characteristics to meet reliability and performance objectives.

THERE IS NO FEATURE in the MT-ACS architecture that is explicitly for fault tolerance. Yet all of its features (e.g., configurable computing and multithreading) can be used for fault tolerance. We conjecture that the successful fault-tolerant design techniques will not be those that influence design features but will be those that are influenced by existing architectural and design features. This position might appear to reduce the fault-tolerance techniques to a subsidiary role; however, the economics of microprocessors is such that design decisions are almost always influenced by cost and performance considerations.

With a wide adoption of configurable computing, it appears that fault tolerance may have a rebirth. Reduced feature sizes and lower operating voltages will make semiconductor chips more susceptible to temporary failures (like SEUs). The need for fault tolerance will become more critical. That fault tolerance will become critical is reinforced by Hewlett-Packard and UCLA's recent announcement of a breakthrough molecular-level gate.[50] The following quote from that article is relevant to fault tolerance: "In the chemically based world of molecular-level engineering, the ability to work around failed components would be critical, as many of the chemically grown transistors would be flawed."

The ROAR project is work in progress. In this article, we have presented some of our project's accomplishments. We are continuing to work in several areas.[51,52] Some of these areas include fault-injection experiments using a multithreaded processor emulator for reliability estimation, fault location and built-in self-test algorithms for configurable logic, fast reconfiguration, and repair strategies using precompiled configurations.

Looking into the future, ACS architectures that combine multithreaded processing and configurable coprocessors will open up the marketplace for COTS components and will reduce the cost and performance gap between fault-tolerant and non-fault-tolerant designs. ■

■ References

1. C.R. Rupp, M Landguth, T. Garverick, E. Gomersall, H. Holt, J.M. Arnold, and M. Gokhale, "The NAPA Adaptive Processing Architecture," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, Apr. 1998.
2. T. Miyamori and K. Olukotun, "A Quantitative Analysis of Configurable Coprocessors for Multimedia Applications," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, Apr. 1998.
3. J.R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with Configurable Coprocessor," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, Apr. 1997, pp. 12-21.
4. J. Villasenor and W.H. Mangione-Smith, "Configurable Computing," *Scientific American*, June 1997.
5. M. Gokhale and J.M. Stone, "NAPA C: Compiling for a Hybrid RISC/FPGA Architecture," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, Apr. 1998.
6. C Level Design, Inc., "Compiling ANSI C to Synthesis-Ready HDL," Product Datasheet, 1999.
7. W.B. Culbertson, R. Amerson, R.J. Carter, P. Kuekes, and G. Snider, "Defect Tolerance on the Teramac Custom Computer," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, Apr. 1997, pp. 116-123.
8. J. Lach, W.H. Mangione-Smith, and M. Potkonjak, "Efficiently Supporting Fault-Tolerance in FPGAs," *Proc. ACM/SIGDA Int'l Symp. FPGAs*, Feb. 1998, pp. 105-115.
9. J.R. Heath, P.J. Kuekes, G. Snider, and R.S. Williams, "A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology," *Science*, vol. 280, pp. 1,716-1,721, 12 June 1998.
10. Actel Corporation, "Design Techniques for Radiation-Hardened FPGAs," Application Note, 1999.
11. G.R. Brown, L.F. Hoffmann, S.C. Leavy, and J.A. Mogensen, "Honeywell Radiation Hardened 32-Bit Processor Central Processing Unit, Floating-Point Processor, and Cache Memory Dose Rate and Single Event Effects Test Results," *IEEE Radiation Effects Data Workshop*, 1997, pp. 110-115.
12. H. Schleifer, T. van den Ropp, and W. Reczek, "Radiation Hardening of Dynamic Memories by the Use of a New Dummy Cell Concept," *IEEE Workshop on Memory Technology*, 1994, pp. 126-129.
13. N. Saxena and E.J. McCluskey, "Dependable Adaptive Computing Systems—the ROAR Project," *Int'l Conf. Systems, Man, and Cybernetcis*, Oct. 1998, pp. 2,172-2,177.
14. N. Saxena and E.J. McCluskey, "Fault-Tolerance with Multi-Threaded Computing—a New Approach," *Fast Abstracts FTCS-29*, pp. 29-30, June 1999.
15. O. Khatib, "Force Strategies for Cooperative Tasks in Multiple Mobile Manipulation Systems," *Int'l Symp. Robotics Research,* Munich, Oct. 1995.
16. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Information Theory*, vol. 23, no. 3, pp. 337-343, 1977.
17. J. Ziv and A. Lempel, "Compression of Individual Sequence via Variable-Rate Coding," *IEEE Trans. Information Theory*, vol. 24, no. 5, pp. 530-536, 1978.
18. M. Nelson, and J.-L. Gailly, *The Data Compression Book*, 2nd ed. IDG Books, 1995.
19. D.J. Craft, "A Fast Hardware Data Compression Algorithm and Some Algorithmic Extensions," *IBM J. Research and Development*, vol. 42, no. 6, Nov. 1998.
20. B. Jung and W.P. Burleson, "Efficient VLSI for Lempel-Ziv Compression in Wireless Data Communication Networks," *IEEE Trans. VLSI Systems*, vol. 6, no. 3, pp. 475-483, 1998.
21. C.E. Shannon, "Communication Theory of Secrecy Systems," *Bell System Technical J.*, vol. 28, pp. 656-715, Oct. 1949.
22. D. Kahn, *The Codebreakers: The Story of Secret Writing*. New York: Scribner, 1996.
23. A. Diffie and M.E. Hellman, "New Directions in Cryptography," *IEEE Trans. Infomation Theory*, vol. 22, no. 6, pp. 644-654, Nov. 1976.

24. B. Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C*. New York: John Wiley & Sons, Inc., 1996.
25. Federal Information Processing Standards, "Data Encryption Standard," Publication 46. Springfield, Va.: U.S. Dept. of Commerce/National Bureau of Standards, National Technical Information Service, 1977.
26. A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault-Tolerance During Program Execution," *Proc. Int'l Computer Software and Appl. Conf*., 1977, pp. 149-155.
27. A. Avizienis and J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *Computer*, pp. 67-80, Aug. 1984.
28. "Microprocessor Report," MicroDesign Resources, p. 25, 14 July 1997.
29. "Microprocessor Report," MicroDesign Resources, p. 27, 22 June 1998.
30. J.H. Patel and L.Y. Fung, "Concurrent Error Detection in ALUs by Recomputing with Shifted Operands," *IEEE Trans. Computers*, vol. 31, pp. 589-595, July 1982.
31. G.S. Sohi, M. Franklin, and K.K. Saluja, "A Study of Time-Redundant Fault Tolerance Techniques for High-Performance Pipelined Computers," *Proc. 19th Int'l Symp. Fault-Tolerant Computing*, June 1989, pp. 436-443.
32. M.A. Schuette and J.P. Shen, "Exploiting Instruction-Level Parallelism for Integrated Control-Flow Monitoring," *IEEE Trans. Computers*, vol. 41, pp. 129-140, Feb. 1994.
33. D.M. Blough and A. Nicolau, "Fault Tolerance in Super-Scalar and VLIW Processors," *IEEE Workshop on Fault-Tolerant and Parallel Distributed Systems*, Amherst, Mass., July 1992.
34. C.L. Chen et al., "Fault-Tolerance Design of the IBM Enterprise System/9000 Type 9021 Processors," *IBM J. Research and Development*, vol. 36, no. 4, pp. 765-778, July 1992.
35. T.J. Slegel, et al., "IBM S/390 G5 Microprocessors," *Hot Chips 10*, Palo Alto, Calif., 16-18 Aug. 1998.
36. J.E. Thorton, "Parallel Operation in the Control Data 660," *AFIPS Conf. Proc., Fall Joint Computer Conf*., vol. 26, pp. 33-40, 1964.
37. B.J. Smith, "A Pipeline, Shared Resource MIMD Computer," *Proc. Int'l Conf. Parallel Processing*, pp. 6-8, 1978.
38. W.J. Kaminsky and E.S. Davidson, "Developing a Multiple-Instruction-Stream Single-Chip Processor," *Computer*, pp. 66-76, Dec. 1979.
39. J.H. Lala and R.E. Harper, "Architectural Principles for Safety-Critical Real-Time Applications," *Proc. IEEE*, vol. 82, no. 1, pp. 25-40, Jan. 1994.
40. M.R. Lyu and A. Avizienis, "Assuring Design Diversity in N-Version Software: A Design Paradigm for N-Version Programming," *Proc. DCCA*, pp. 197-218, 1991.
41. R. Riter, "Modeling and Testing a Critical Fault-Tolerant Multi-Process System," *Proc. FTCS*, pp. 516-521, 1995.
42. D. Briere and P. Traverse, "Airbus A320/A330/A340 Electrical Flight Controls: A Family of Fault-Tolerant Systems," *Proc. FTCS*, pp. 616-623, 1993.
43. Y. Tohma and S. Aoyagi, "Failure-Tolerant Sequential Machines with Past Information," *IEEE Trans. Computers*, vol. 20, no. 4, pp. 392-396, Apr. 1971.
44. Y. Tamir and C.H. Sequin, "Reducing Common Mode Failures in Duplicate Modules," *Proc. ICCD*, pp. 302-307, 1984.
45. J.P. Laudon, "Architectural and Implementation Tradeoffs for Multi-Context Processors," PhD dissertation, Electrical Engineering Dept., Stanford Univ., 1994.
46. S. Mitra, N. Saxena, and E.J. McCluskey, "Design Diversity for Redundant Systems," technical report, Center for Reliable Computing, Stanford Univ., 1999.
47. S. Mitra, N. Saxena, and E.J. McCluskey, "A Design Diversity Metric and Reliability Analysis for Redundant Systems," *Proc. IEEE Int'l Test Conference*, 1999.
48. D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. Int'l Symp. Computing Architecture*, 1996, pp. 191-202.
49. ARGOS Project, Center for Reliable Computing, Stanford Univ., July 1999.
50. T. Quinlan, "New Era in Chip Design," *San Jose Mercury News,* 10 July 1999.
51. C. Zeng, N.R. Saxena, and E.J. McCluskey, "Finite State Machine Synthesis with Concurrent Error Detection," *Proc. Int'l Test Conf*., Atlantic City, N.J., 28-30 Sept. 1999, pp. 672-679.
52. D. Das and N.A. Touba, "A Low Cost Approach for Detecting, Locating, and Avoiding Interconnect Faults in FPGA-Based Reconfigurable Systems," *Proc. IEEE Int'l Conf. VLSI Design*, Goa, India, 7-10 Jan. 1999, pp. 266-269.

Nirmal R. Saxena is currently a coinvestigator for the ROAR project at Stanford University. His research interests include computer architecture, fault-tolerant computing, combinatorial mathematics, probability theory, and VLSI design/test. He received the BE degree in electronics and communication engineering from Osmania University, India, in 1982; the MS degree in electrical engineering from the University of Iowa in 1984; and the PhD degree in electrical engineering from Stanford University in 1991. He is a senior member of the IEEE.

Santiago Fernandez-Gomez has worked as a research associate at the Center for Reliable Computing at Stanford University since April 1999. He received the engineering degree in telecommunications from the Polytechnic University of Madrid in 1993 and the PhD degree from the University of Vigo (Spain) in 1999.

Wei-Je Huang is currently with the Center for Reliable Computing, Stanford University, working on the ROAR project. He received the BSEE degree from National Taiwan University, Taipei, Taiwan, in 1995, and the MSEE degree from Stanford University in 1999, where he is currently pursuing the PhD degree. His research interests include fault-tolerant computer architecture and adaptive computing systems.

Subhasish Mitra received the BE degree in computer science and engineering from Jadavpur University, Calcutta, India, in 1994 and the MTech degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, in 1996. He is currently working toward the PhD degree at the Center for Reliable Computing at Stanford University under the supervision of Edward J. McCluskey. His research interests include digital testing, logic synthesis, and fault-tolerant computing. He received gold medals for being the top student in the School of Engineering in the undergraduate and the MTech levels. He was the registration and finance chair of CRC-IEEE Pacific Northwest Test Workshop (known as BAST) in 1998 and 1999.

Shu-Yi Yu received the BS degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1997. She received the MS in electrical engineering from Stanford University in 1999. She is currently pursuing her PhD degree in electrical engineering at the Center for Reliable Computing, Stanford University. Her research interests include adaptive computing system and fault-tolerant computing.

Edward J. McCluskey is the principal investigator for the ROAR project at Stanford University. At Stanford University, he is a professor of electrical engineering and computer science, as well as director of the Center for Reliable Computing. His Stanford research focuses on logic testing, synthesis, design for testability, and fault-tolerant computing. He served as the first president of the IEEE Computer Society. He is the recipient of the 1996 IEEE Emanuel R. Piore Award. He is a Fellow of the IEEE, AAAS, and ACM and is a member of the NAE. He has published several books, including two widely used texts.

■ Send questions and comments about this article to Nirmal R. Saxena, Center for Reliable Computing, Stanford University, Room #236, MC 9020, Gates Building 2A, 353 Serra Mall, Stanford, CA 94305; saxena@crc.stanford.edu.