Dependable Adaptive Computing Systems- The ROAR Project

Nirmal R. Saxena
Center for Reliable Computing
Stanford University,
Stanford, CA 94305

Edward J. McCluskey
Center for Reliable Computing
Stanford University,
Stanford, CA 94305

ABSTRACT

This paper describes the ROAR project. ROAR is an acronym for *Reliability Obtained By Adaptive Reconfiguration*. *Adaptive Computing System* (ACS) environments provide opportunities for dependable computing solutions. We identify some of the issues in traditional dependable computing solutions. We show that adaptive and reconfigurable environments address these issues and also enable new innovative techniques that enhance the dependability of systems. Natural redundancy in applications and new synthesis algorithms are two other opportunities we use in enhancing the dependability.

1. INTRODUCTION

ROAR[1] is a joint project with participation from the Center for Reliable Computing (CRC), Stanford University, the University of Texas, Austin, and Quickturn Design Systems, San Jose. The objective of the ROAR project is to provide dependable computing solutions in an ACS framework.

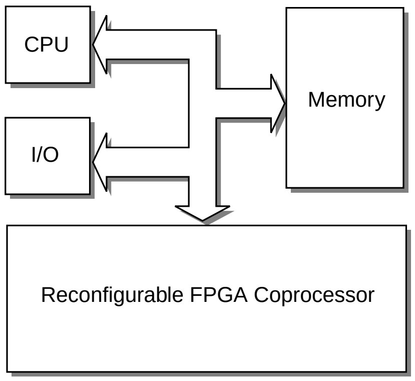

Figure 1. Reconfigurable ACS- An Architecture Model

Figure 1 illustrates one of the architecture models for adaptive and reconfigurable computing systems. An example implementation of this architecture model is the NAPA adaptive processor [1]. This ACS architecture model is useful in that it effectively implements most of the target ACS applications. This is because it not only preserves the benefits of the traditional Von Neumann programmable memory model but also augments it with a programmable reconfigurable FPGA coprocessor. Software models for operating systems and compilers are also leveragable. For example, a compiler [2] can partition and map applications into parts that run well on traditional fixed instruction processors and into other parts that run well on the reconfigurable coprocessor.

Applications that are well suited for ACS are: 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) locations, and life critical systems makes dependability also an important requirement. With very few exceptions [3] [4], little work has been done in the dependable ACS area.

Dependability comprises availability, reliability, testability, maintainability, and fault-tolerance. *Availability* is the probability that the system is operational at a specified time. *Reliability* is the probability that the system is operational up to a specified time. *Testability* is the ease with which the system can be tested. *Maintainability* is the probability that a failed system is repaired within a specified time. *Fault-tolerance* is the ability of the system to operate correctly in spite of some specified faults. Dependable computing solutions, in general, require the following support:

- Off-line test methods to detect and isolate failed components. This is necessary both for initial configuration (fault-avoidance) and during reconfiguration (fault-location and repair).
- On-line test and checking methods to detect system errors during normal system operation.
- Error recovery methods to restore operation of the system after failure.

The selection of testing and error recovery methods is influenced by the dependability requirements. The

1. This project is supported by Defence Advanced Research Projects Agency (DARPA) under the Adaptive Computing Systems (ACS) program.

dependability requirements can be broadly classified by the system's responsiveness to errors. In other words, the dependability requirement stipulates the time the system may take to restore correct operation in response to errors. Not meeting this requirement could result in loss of life or could have a significant economic consequence. The system's responsiveness to errors, for example, can be classified into the following three categories:

Corrective Action in Zero Seconds to Milliseconds
This requirement is important for applications like: nuclear reactors, life critical support equipment, and fly-by-wire systems.

Corrective Action in Seconds to Minutes
On-line transaction processing, electronic commerce, navigation and tracking, and computer servers are some of the application areas where this requirement is important.

Corrective Action in Hours to Days
Here system outage for hours to days can be tolerated without signifiant consequence. Workstations and personal computers are some examples that fit this category.

In addition to the reconfigurable FPGAs, the dependability solutions for the ACS environments must also include the CPU, memory, and I/O components. This is very important because the ACS applications use all of these components. Also, CPU and memory dependability is very important because they are used during compilation and reconfiguration. Robotics control, Fast Fourier Transforms (FFT), compression, and encryption are some of the target ACS applications that would be the focus of ROAR's research. Since ROAR is a project in progress, with two more years to complete, the material presented in this paper is preliminary and covers only some of its key ideas. As the project nears completion, a more detailed report will be forthcoming. Following is the outline of the material covered in this paper:

- Section 2 presents the reliability analysis for compensating faults and common mode failures. It is shown how diversified designs with active reconfiguration address the reliability issues with common mode failures.
- Section 3 identifies some opportunities to exploit under-utilized hardware resources both in the CPU and reconfigurable FPGAs.
- Section 4 introduces the concept of using multithreading for both general purpose computing as well as reconfigurable computing.
- A summary of other activities in ROAR and conclusions are presented in Section 5.

2. DESIGN DIVERSITY AND RECONFIGURATION

Design diversity has long been known to address common mode failures in fault-tolerant systems. Design diversity is obtained by implementing the same function in different ways. For example, a combinational function implemented in *and-or* and *or-and* form. We recognized very early in our project the opportunity to compile diverse designs in reconfigurable hardware. Use of diverse designs in *Triple Modular Redundancy* (TMR) can increase the probability of compensating faults in the case of two or three module failures. Compensating faults are such that at any given cycle a majority of the modules have the correct output. It has been shown that the classical TMR reliability model is pessimistic [5]. In the classical TMR reliability analysis, the following assumption is made: When a module fails it starts producing incorrect outputs and therefore if two or more modules fail the output of the TMR voter will be incorrect. In the compensating TMR reliability analysis the following assumptions are made:

- When a module fails it does not produce incorrect outputs for all of the inputs. Therefore there is a non-zero probability that a failed module produces correct outputs.
- When two modules fail, there is a non-zero probability that at least one of the outputs is correct. In an event like this, the TMR voter output is still correct.

The reliability analysis for both classical and compensating models is done based on the probability of the voter output being correct; however, in the compensating TMR model the voter output can be correct in spite of two module failures. This was the basis of the claim that classical TMR reliability model is pessimistic [5]. Reliability modeling of compensating faults for some combinational networks was presented in [6]. Figure 2 compares the classical TMR reliability model with a TMR reliability model assuming compensating faults based on an analytical analysis done in our project. The actual details of this analysis will appear in a separate report.

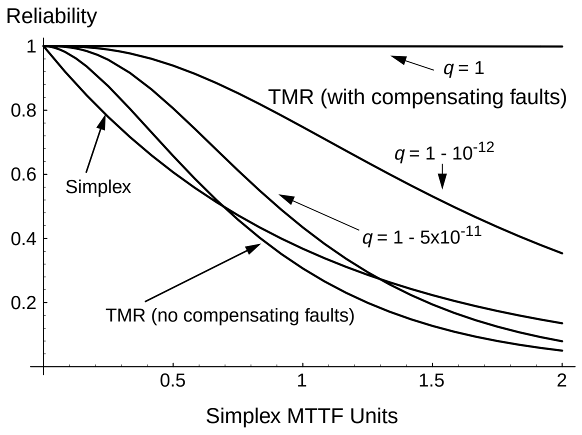

Figure 2. Diversified TMR Reliability for Stuck-At Faults

The plot is against the normalized simplex *mean time to failure* (MTTF) units. The MTTF units are in cycles. The failure rate λ is defined in terms of failures per cycle. The parameter *q* is the probability that a majority of the modules have correct output. Clearly, in this reliability model when *q*=1, the TMR reliability with compensating faults is unity and when *q*=0 the reliability is same as the classical TMR reliability. What is interesting to note is that modest departures of q from 1 pushes the reliability curves very close to the classical TMR reliability. So we have the following new observation here:

- For diversified designs to appreciably improve the TMR reliability over the classical TMR reliability, the value of q should be of the order of $1-\lambda$, where λ is the failure rate per cycle. Since λ is typically of order 10^{-12}, obtaining a value of q close to $1-10^{-12}$ is almost impossible. Therefore, diversified designs in masking redundancy do not significantly improve its reliability over the classical reliability model. This is a less optimistic conclusion than that in [6].

Next, by way of an example we demonstrate the importance of diversified designs in addressing common-mode failures. Figure 3 shows a duplex-spare system. Upon detection of error the system switches to a standby spare. For independent failures, the reliability of this system is better than simplex; however, for common-mode failures (identical modules failing the same way) the reliability is no better than simplex. Figure 4 shows the reliability plots for this system.

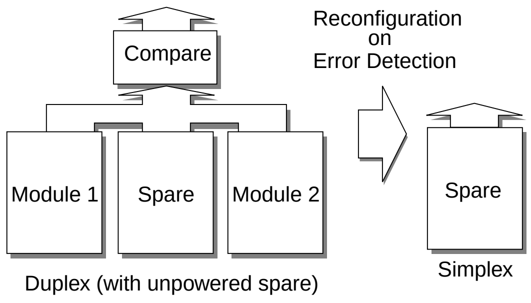
Compare
Reconfiguration on Error Detection
Module 1
Spare
Module 2
Spare
Duplex (with unpowered spare)
Simplex

Figure 3. A Duplex Spare System

Now consider the same duplex-spare system with diverse designs for the two modules. While there is a possibility of a common mode failure (for example, power supply failure) affecting both modules, due to the fact that diverse designs are used it is less likely that both failed modules will produce an identical incorrect response. In this scenario, we use the same parameter q which is the probability that a majority of modules produce correct outputs. In Figure 4 we can see that even for q=0.25 the reliability of the diversified-duplex system is much better than that of a simplex system. Even for modest values of q, if diversified designs are used in the context of error detection and reconfiguration then the reliability is improved significantly for common-mode failures.

Traditional dependable computing solutions with standby spares (as in Figure 3) achieve fault-tolerance by deactivating or activating portions of existing structure. That is, the redundant structures and spares are configured as part of the system structure. They are activated or deactivated upon detection of errors. For lack of a better name, we call the traditional dependable computing solutions as one based on *passive reconfiguration*. In dependable ACS environments, upon error detection, portions of or all of the system can be replaced with a new structure. We call ACS dependable computing solutions as one based on *active reconfiguration*. Note that in active reconfiguration there is no need to carry standby spares, new structures can be derived by only discarding failed portions of modules and reconfiguring new modules. In other words, ACS environments provide opportunities to recycle unaffected portions of failed modules into new configurations thereby economizing on the usage of resources.

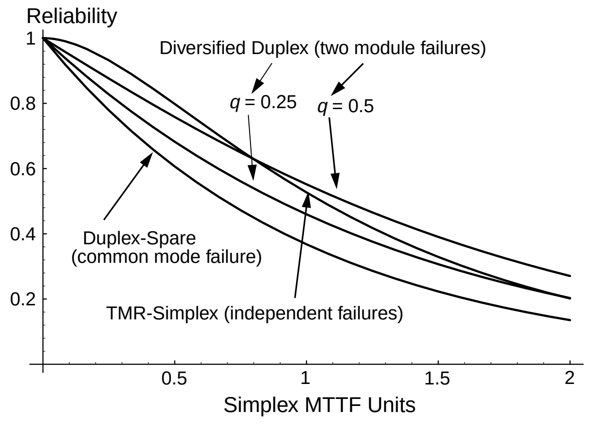

Figure 4. Diversified-Duplex with Reconfiguration- Reliability Plots

This example demonstrates that by using active reconfiguration, with design diversity and fewer resources, there are opportunities to improve the dependability of systems over passive reconfiguration solutions. Investigation of design methods to synthesize diverse designs in reconfigurable hardware is a work in progress for the ROAR project.

3. UNDER-UTILIZED OPPORTUNITIES- A DEPENDABLE COMPUTING PERSPECTIVE

In the ACS architecture model, shown in Figure 1, the reconfigurable coprocessor can be adapted either for dependable computation or for high performance computation. This adaptability exists, as was discussed in the foregoing section, due to the programmablity of FPGAs. However, the opportunity for this kind of adaptability is not apparent in fixed instruction and fixed structure CPU, which is also an important part of the ACS model. In our project we are exploring some hidden opportunities in modern fixed instruction CPUs to provide adaptive usage of resources based on the dependability requirements. We examine trends in the resource usage and efficiency in modern microprocessors. The following discussion illustrates our observations.

The graph shown in Figure 5 is derived from the integer performance, transistor count, and clock rate data provided in various issues of *Microprocessor Report*, 1994-1996. It illustrates an interesting trend in the general purpose microprocessors. The y-axis is the normalized performance over the number of logic transistors and the clock rate in Mhz of the microprocessor. The *normalized performance* is derived by taking the SpecInt92 (an industry standard integer performance benchmark for microprocessors) and dividing it by the number of logic transistors and the clock rate in Mhz of the measured microprocessor. The *x*-axis is the number of logic transistors. The trend suggests that by increasing the number of logic transistors the incremental increase in performance for a fixed clock rate is diminishing. We believe that the trend is caused by algorithmic, application, and architectural characteristics of the processor that don't make effective use of the available transistors. For example, tripling the number of adders in a processor to speed up certain computations will not triple the overall performance of the processor.

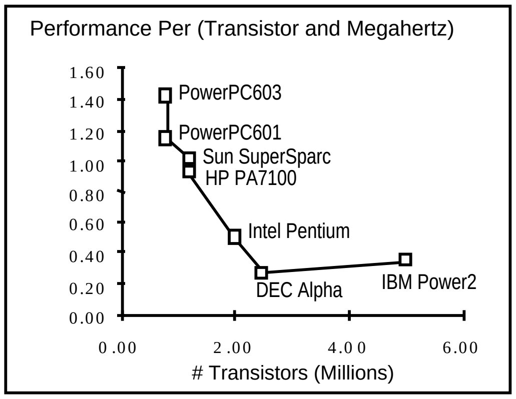

Figure 5. Normalized performance versus Number of Logic Transistors.

We assert that a trend similar to the one shown in Figure 5 exists for applications implemented with reconfigurable hardware components. When a portion of an algorithm is mapped to an FPGA, not all available gates are used. While technology trends suggest that there will be further increases in transistor density and gate counts, not all of these increases can be effectively used to improve performance. We believe, however, that these under-utilized resources can be used to improve dependability without significantly impacting the performance of applications.

4. MULTI-THREADING AND FAULT-TOLERANCE

The under-utilized resources in microprocessors can be utilized if we allow multiple independent threads to be alive at the same time. A *thread* is an independent computational entity like a process. In modern operating systems, a process can be composed of several independent threads. For example, if one thread stalls due to a cache miss or waiting for branch resolution the other thread can be scheduled in the microprocessor thereby keeping the resources busy. Multithreaded processors have the architectural and implementation support to allow more than one process context or thread to be alive at any given time. Research and development of multithreaded processors [7] [8] [9] [10] has been motivated by the recognition that single process threads lack sufficient parallelism to fully utilize the available resources in a processor. For example, commercial processors, like DEC Alpha 21164, PowerPC 604, MIPS R10000, Sun UltraSparc, and HP PA-8000 have implementation features that allow execution of up to four instructions per cycle. However, application performances reported in *Microprocessor Report* suggest that the attainable instruction throughput is up to 1.5 instructions per cycle - this is less than 40% utilization of the available resources. Threads from different processes can use these under-utilized resources. The Tera computer [7] and the Mediaprocessor [10] are implementations of multithreaded processors.

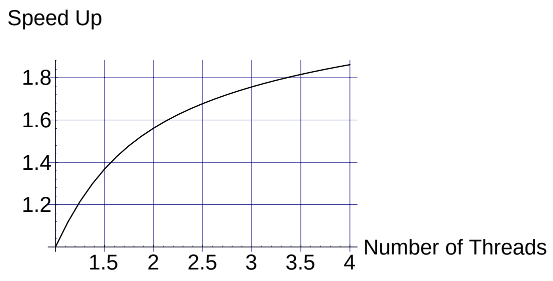

Figure 6. Typical Speedup for Scientific Code

For scientific codes, we derived an analytic estimate of instruction throughput speedup (shown in Figure 6) as a function of the number of threads for commercial microprocessors. The actual details of this analysis will appear in a separate report. Studies described in [9] [10] show instruction throughput improvements that concur with these analytic estimates. Since performance enhancement was the major motivation behind the proposal of multithreaded processors, an obvious connection to fault-tolerance went unnoticed by previous research and development efforts. In ROAR we propose to investigate, for the first time, cost-effective deployment of fault-tolerance technology in multithreaded processors by focusing on the following features:

- Several copies of a single process can run concurrently on a multithreaded processor, and the process results can be compared with each other at well-defined checkpoints. This type of fault tolerance requires significantly less hardware overhead than traditional fault tolerance techniques such as duplex or triple modular redundancy.
- The inter-process communication paradigm supported by multithreaded operating systems can be re-used for voting and result comparison of multiple copies of a process, further reducing the fault tolerance hardware overhead.
- The performance overhead using multithreading for dependable computing is around 20% for dual threads and around 60% for three threads. This estimate is based on the speedup in instruction throughput achievable by multithreaded processors.

Table 1 shows the execution times for a multithread experiment emulated on three commercial super-scalar processors. The multithread emulation was done by hand-coding two threads in the source code of the LZW compress algorithm. The source code for the LZW algorithm was obtained from [11]. The input and output files were not duplicated; only the core algorithm for directory creation and update was duplicated. The low performance overhead corroborates the value of multithreading in increasing the dependability of applications running on general purpose processors with very little performance impact.

Table 1. LZW Compress Multithread Experiment

Processor	One Thread (secs)	Two Threads (secs)	% Over head	Speed up
PowerPC 604e	9.37	11.33	22	1.65
AMD-K6	7.10	8.69	20	1.63
Sun Ultra-II	8.00	9.00	12	1.77

Note that the same level of fault-tolerance can be accomplished in a traditional microprocessor using multitasking. However, the performance overhead of multitasking is more than 100% for dual tasks and more than 200% for three tasks. Multithreaded fault-tolerance has the same flexibility as multitasking without its performance overhead. However, multithreading requires new processor architectures. In environments where multithreaded processors are not available, we could emulate multithreaded computation on traditional processors by explicitly coding multiple threads in the same program. This was illustrated in Table 1.

4.1 Multithreading in Reconfigurable Hardware

The concept of using multithreading to improve system dependability is not limited to general purpose computing. It can also be applied to special-purpose reconfigurable implementations of algorithms. We demonstrate our multithreading on-line error checking ideas using the RSA encryption algorithms given in [12]. RSA encryption involves converting a message, *A*, by computing its modular exponential as shown in Figure 7. *B*[*j*] and *P*[*j*] denote the values of *B* and *P*, respectively, at the *j*th time step and *e[j*] denotes the bit value at position *j* of the binary word *E*. A scheduled data flow graph for the inner loop of the *L* modular exponential algorithm in Figure 7 is given in Figure 8(a), where C_i is the *i*-th control step. A hardware implementation of the data flow graph is given in Figure 8(b), where *MUL* is a multiplier, *MOD* is a modulo-*M* unit, *MUX* is a multiplexer, and *REG* is a register.

```
B[0] = 1; P[0] = A;
for j < k-1 do
{
        P[j+1] = P[j]†P[j] (mod M);
        if (e[j] == 1}
                B[j+1] = P[j]†B[j] (mod M);
        else
                B[j+1] = B[j];
}
B = P[k-1]†B[k-1] (mod M)
```

Figure 7. L Modular Exponential Algorithm for computing B = A^E (mod M)

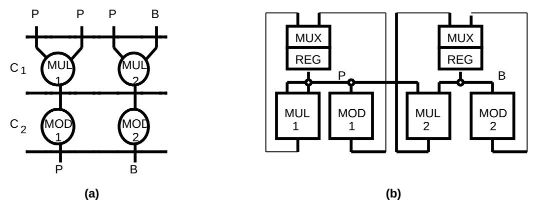

Figure 8. Implementation of L modular exponential algorithm: (a) scheduled data flow graph; (b) hardware implementation

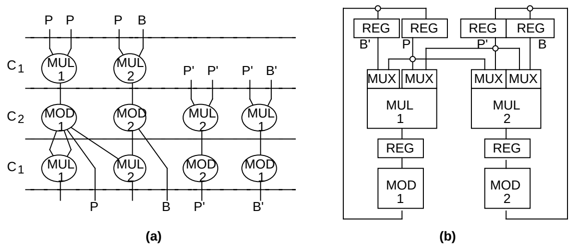

Figure 9. Dual-threaded implementation of L modular exponential algorithm: (a) scheduled data flow graph; (b) hardware implementation.

When the algorithm in Figure 8 is scheduled as shown in Figure 9, the multiplier units are idle during the first control step, and the modulo units are idle during the second control step. We can use these idle cycles and the two instances of the multiplier and modulo units to implement low-overhead, on-line error checking for this design. We do this by scheduling a redundant thread of the *L* modular exponential algorithm on the idle computation units, producing *B'* and *P'*. The scheduled data flow graph for this fault-tolerant implementation of the algorithm is shown in Figure 9(a), and the hardware implementation is shown in Figure 9(b). On-line error checking is performed, with a latency of one control step, by comparing *B* with *B'* and *P* with *P'* at the end of control step C_1. The hardware overhead for checking in this case is only four registers, two multiplexers, and comparison logic. Note that, in order to detect permanent faults, the *MUL*, *MOD* pair that computes *P* must be used to compute *B*′, and the *MUL*, *MOD* pair that computes *B* must be used to compute *P*′. An added advantage of this technique is that a single hardware implementation can be used when either high performance or high dependability is required. For example, if high-performance is required, the hardware implementation shown in Figure 9(b) can be used to execute two independent, concurrent threads of the algorithm, thereby doubling the bandwidth, but sacrificing the on-line checking capability.

Multithreaded on-line checking is a universal concept in that any hardware implementation of an algorithm can be multithreaded using extra registers, multiplexers, and control logic, and reusing some or all of the computational units. One disadvantage of this multithreading approach is that, depending on the algorithm, there might be some performance overhead if re-using computational hardware requires cycle stealing from busy units. When implementing a fault-tolerant adaptive system, it is important to have multiple hardware implementations that can execute a given function, where each implementation has different hardware resource requirements. Situations may arise where the system reconfiguration software has a shortage of available defect-free hardware units. In this case, a minimal implementation of an algorithm can result in a successful reconfiguration that accomplishes the primary function of finishing the execution of the algorithm although with lower performance (degraded mode) and without any on-line checks. The *H* modular exponential algorithm given in [12] uses only one multiplier and one modulo unit, but requires four control steps to execute each iteration of the loop. An adaptive system can use this implementation rather than the implementation in Figure 8(b), to gracefully degrade the system performance in response to previously detected and isolated defects.

5. SUMMARY AND CONCLUSIONS

In this paper we propose a dependable ACS model that comprises: a multithreaded CPU, memory, and reconfigurable coprocessor. This model provides a common architecture that can be adaptively used for either high performance computing or dependable computing. Error detection using multithreading allows for low-cost deployment of fault-tolerant technology without significantly sacrificing the performance of applications. In the event of scarcity of resources, reconfiguration can be used to enable graceful degradation.

Instead of implementing our proposed dependable ACS model in an actual chip implementation, like the NAPA Adaptive processor [1], we will use the Quickturn emulation platform. Quickturn emulation platform will allow us the flexibility to evaluate variants of the dependable ACS model for the target applications. In addition to the emulation system support, Quickturn design systems will contribute in instrumenting rapid reconfiguration algorithms using ACS hardware. The contribution of University of Austin in this effort would be in terms of providing fault-location and self-testing algorithms for the reconfigurable hardware.

Acknowledgments

The authors would like to thank Mehmet Apaydin, Subhasish Mitra, Philip Shirvani and Nahmsuk Oh for critically reviewing this paper. This research was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract Number DABT63-97-C-0024.

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. FCCM’98, Apr. 1998.

[2] M. Gokhale and Janice M. Stone, “NAPA C: Compiling for a Hybrid RISC/FPGA Architecture,” Proc. IEEE Symp. FCCM’98, Apr. 1998.

[3] W. B. Culbertson, R. Amerson, R. J. Carter, P. Kuekes, and Greg Snider, “Defect Tolerance on the Teramac Custom Computer,” Proc. IEEE Symp. FCCM’97, pp. 116-123, Apr. 1997.

[4] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, “Efficiently Supporting Fault-Tolerance in FPGAs,” Proc. ACM/SIGDA Intl. Symp. on FPGAs, pp. 105-115, Feb. 1998.

[5] Daniel P. Siewiorek and Robert S. Swarz, *Reliable Computer Systems*: *Design and Evaluation*, 2nd Edition, Digital Press, 1992.

[6] D. P. Siewiorek, “Relibility Modeling of Compensating Module Failures in Majority Voted Redundancy,” IEEE Trans. on Comput., Vol. C-24, No. 5, pp. 525-533, May 1975.

[7] R. Alverson, D. Callhan, D. Cummings, B. Koblenz, A Porterfield, and B. Smith, “The Tera Computer System,” International Conference on Supercomputing, pp. 1-6, June 1990.

[8] C. Hansen, “Architecture of a Broadband Mediaprocessor,” Proceedings COMPCON96, Feb. 1996.

[9] D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism,” 22nd Intl. Symposium on Computer Architecture, pp. 392-403, June 1995.

[10] D. M. Tullsen, S. J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm, “Exploiting Choice: Instruction Fetch & Issue on an Implementable Simultaneous Multithreading Processor,” 23rd Intl. Symposium on Computer Architecture, May 1996.

[11] M. Nelson, and J-L. Gailly, *The Data Compression Book*, 2nd. Edition, IDG Books, 1995.

[12] M. Shand, and J. Vuillemin, “Fast Implementations of RSA Cryptography,” 12th IEEE Symp. on Computer Arithmetic, pp. 252-259, 1993.