Dependable $\neq$ Unaffordable

Alan L. Cox, Kartik Mohanram, and Scott Rixner
Rice University
Houston, TX 77005
{alc,kmram,rixner}@rice.edu

Abstract

This paper presents a software architecture for hardware fault tolerance based on loosely-synchronized, redundant virtual machines (LSRVM). LSRVM will provide high levels of reliability by tolerating hardware faults at all levels of the system. Historically, such hardware fault tolerance has only been achievable using custom-designed hardware and proprietary operating systems. Today, however, technological trends and economic factors are driving a reduction in the amount of custom-designed hardware. We believe that this path should be followed to its ultimate conclusion: a highly-available, fault-tolerant computing system based entirely on commodity hardware and open-source operating systems. Our revolutionary approach utilizes virtualization to efficiently provide redundancy on modern commodity hardware. When combined with existing application-level fault tolerance mechanisms, LSRVM will provide very high levels of reliability at extremely low cost.

1. Introduction

Traditionally, the use of highly-dependable, fault-tolerant computing systems has been limited to industries that can afford their considerable costs, such as the banking and financial sectors. Such systems have consisted of tightly integrated hardware and software. Specifically, they have utilized custom-designed hardware, proprietary operating systems, and modified application software in order to achieve the desired level of reliability. However, for most applications, it is too expensive to develop or acquire such customized hardware and software, even if the applications are in mission critical domains. Therefore, there is a need for highly-dependable, fault-tolerant computing systems based entirely on commodity hardware and operating systems.

There is significant interest and research into architectural and system support for improving software dependability. However, several studies and technological predictions indicate that transient faults in hardware will constitute an important—and possibly dominant—failure mode that may significantly impact the reliability of future commodity computing systems (e.g., [1–5]). We are fast approaching an era of increased hardware logic faults, similar to the era of hardware memory faults in the past. Therefore, highly-dependable computing systems must be able to tolerate both hardware and software faults. Solutions exist to protect individual components of the system from hardware faults [6–8]. However, an end-to-end solution is required in order to protect the entire system. Specifically, an application's execution must be protected from beginning to end, including all computations, memory accesses, and I/O operations.

Recently, Hewlett-Packard has described their NonStop Advanced Architecture (NSAA), which builds upon the loose-lockstep processing concepts that were originally developed at Tandem [9, 10]. The main idea behind loose-lockstep processing is that multiple redundant copies of the application and operating system execute on parallel processing resources and are only synchronized at I/O operations and external interrupts.

Loose-lockstep processing can be much more efficient than traditional lockstep processing which requires complex processors to execute instructions at exactly the same rate and compare their execution on an instruction-by-instruction basis. Furthermore, it provides a higher level of reliability than techniques that only protect the execution core from hardware faults. The only observable behaviors outside the system come from I/O operations, so they not only must be checked for correctness, but also are the only results that truly need to be checked for correctness. In order to efficiently check the I/O operations performed by the redundant copies, the I/O operations of each copy must occur in the same order. Therefore, the redundant copies must also be synchronized on interrupts, because if each copy receives the interrupt at a different time, then the I/O operations generated by each copy will potentially occur in a different order, making comparison difficult, if not impossible.

The loose-lockstep concept therefore relies on the following four capabilities for reliability:

1. **Isolation and redundancy:** Isolated redundant copies of the operating system and the application.
2. **I/O voting:** Reliable voting to ensure that all I/O operations generated by the redundant applications and operating systems are identical.
3. **Interrupt synchronization:** Synchronized interrupts to ensure that the redundant operating systems will generate the same I/O operations in the same order as each other.
4. **Recovery:** A mechanism to restart one of the copies after a failure by replicating the state of the surviving copies.

The current implementation of the NSAA utilizes commodity hardware with some additional specialized hardware. We believe the right approach to fault tolerance is to take these ideas further and provide a complete software solution that provides similar levels of tolerance to hardware faults on unmodified commodity hardware running open-source operating systems. Moreover, we argue that a virtual machine monitor that implements *loosely-synchronized, redundant virtual machines* (LSRVM) is the best

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASID'06 October 21, 2006, San Jose, California, USA.
Copyright 2006 ACM 1-59593-576-2

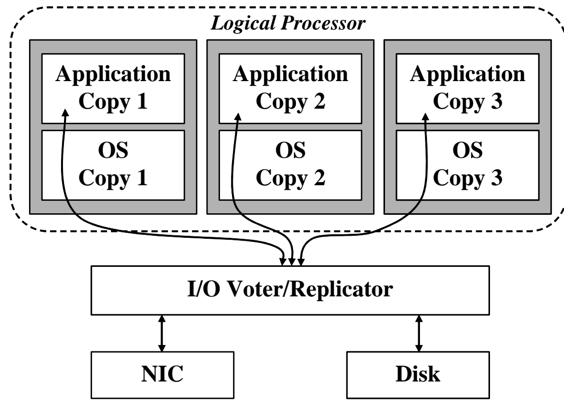

Figure 1: Loose-lockstep Processing with I/O Synchronization

means to this end. Specifically, we intend to modify the open-source Xen virtual machine monitor to provide transparent redundancy for both the operating system and the application. This will allow applications that have been written for commodity operating systems, such as Linux, to tolerate hardware faults without modification on commodity hardware.

LSRVM implements the four capabilities of loose-lockstep reliability as follows:

1. **Isolation and redundancy:** Xen provides isolation among guest operating systems as a part of virtualization, and we provide redundancy by running identical software in loose-lockstep in multiple guest domains.
2. **I/O voting:** All I/O operations go through the Xen control software, providing a natural location for a software voter/replicator.
3. **Interrupt synchronization:** The Xen hypervisor receives all interrupts, so it can synchronize the delivery of these interrupts to the redundant guest domains.
4. **Recovery:** The Xen migration features provide a natural substrate on which to build a mechanism to clone the surviving copies of the system after a failure.

By implementing loose-lockstep processing within the Xen framework, application execution will be replicated transparently within multiple guest domains providing application-level and operating system-level reliability without any additional programming overhead. However, for the system to be completely tolerant of hardware failures, both the software voter/replicator and the Xen hypervisor must be reliable as well. These two components are the only single points of failure in the LSRVM architecture.

The following section provides additional details on the NonStop Advanced Architecture. Section 3 then describes the Xen virtual machine monitor and how loose-lockstep processing can be implemented on top of Xen. Section 4 explains the additional steps that must be taken to ensure the reliability of the system, and Section 5 concludes the paper.

2. The NonStop Advanced Architecture

The current implementation of the NSAA utilizes commodity hardware with some additional specialized hardware. As described in [10], off-the-shelf Itanium servers are augmented with a hardware logical synchronization unit (LSU) and high-speed reintegration links that connect the memories of the system. These systems run the NonStop Kernel (NSK), which manages system resources and ensures that the system remains reliable.

As shown in Figure 1, the general principle is that a collection of physical processors form a *logical processor* which executes redundant copies of the application on each physical processor. In the figure, each gray box represents a physical processor, and the collection of three physical processors form a logical processor. The I/O operations of each logical processor are checked by the LSU to ensure that each physical processor has generated identical I/O operations (including addresses, values, etc.). The LSU therefore includes the functionality of the "I/O Voter/Replicator" in Figure 1. The LSU is also responsible for notifying the NSK that there has been a hardware or software failure in one of the physical processors within the logical processor when a mismatch occurs.

Upon a mismatch, the NSK can then restart the failed physical processor within the logical processor group. This is accomplished by using the high-speed reintegration links to copy the memory state of the correctly functioning processors to the failed processor. This can be done while the other processors continue to execute, as the reintegration links snoop on all memory activity and forward updates to the newly restarted processor. After reintegration, the new processor begins loose-lockstep execution as before. After multiple failures, the NSK may choose to notify the administrator that the failure is unlikely to be transient and the hardware should be replaced.

So that the LSU will receive I/O operations in the same order for all physical processors within the logical processor, each physical processor must execute the same logical control flow, even though instructions may not execute in lockstep across the processors. This means that all context switches of any type must occur at the same point in the logical control flow for each processor. Specifically, all asynchronous interrupts must occur at the same point on each processor. Since asynchronous interrupts can typically occur at any time, the NSAA provides a *rendezvous* mechanism to synchronize the physical processors within a logical processor before dispatching an interrupt.

The rendezvous mechanism depends on the notion of *voluntary rendezvous opportunities* (VRO), which are code sections embedded throughout the operating system and applications that can be used to schedule interrupts. Interrupt processing is therefore split into two phases. In the first phase, the NSK receives the interrupt and proposes a VRO at which to schedule the interrupt using the LSU to communicate the proposed VRO. When each physical processor reaches that VRO it waits to receive the interrupt. In the second phase, when all physical processors have reached the VRO, the NSK then delivers the interrupt to each processor, which then process the interrupt normally. This guarantees that the logical control flow on all processors within the logical processor will be identical.

3. Loosely Synchronized, Redundant Virtual Machines

We intend to implement loose-lockstep processing entirely in software so that commodity hardware and operating systems can be used to construct highly reliable systems. Specifically, we intend to make use of a virtual machine monitor to provide resource isolation and to regulate interrupts and I/O operations. We will then add facilities into the virtual machine monitor to enable redundant copies of the system to run reliably using loose-lockstep processing among multiple virtual machines.

3.1 The Xen Virtual Machine Monitor

A virtual machine monitor allows multiple operating systems to share a single machine safely. It provides isolation between operating systems and manages access to hardware resources.

Xen is an open source virtual machine monitor based on the Linux kernel [11]. While Xen requires minor modifications to

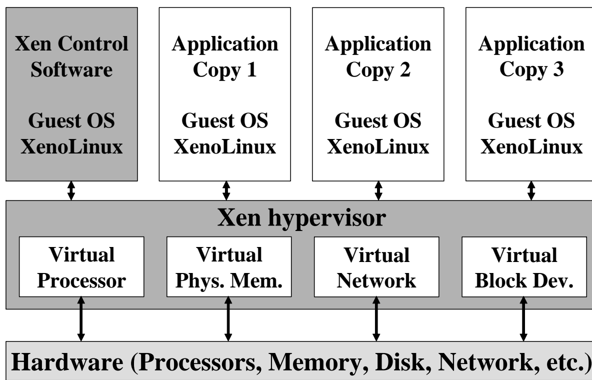

Figure 2: The Xen Virtual Machine Environment

the guest operating systems that run within it, several commodity operating systems, including Linux, have already been ported to Xen. In order to provide isolation and fair access to resources, Xen performs three key functions. First, Xen allocates the physical resources of the machine to the guest operating systems and isolates them from each other. Second, Xen receives all interrupts in the system and passes them on to the appropriate guest operating systems. Finally, all I/O operations must go through Xen in order to ensure fair and non-overlapping access to I/O devices among the guests.

Figure 2 graphically shows the organization of the Xen virtual machine monitor. Xen effectively consists of two elements: the hypervisor and additional control software. The hypervisor provides an abstraction layer between the guest operating systems and the actual hardware. One of the major functions of the control software is to provide access to the actual hardware I/O devices. The hypervisor grants the control software access to the devices and does not allow the guest operating systems to access them.

In Xen, each guest operating system, therefore, is protected from other guests that are running on the machine by the hypervisor and shares I/O resources fairly through the control software. This enables each guest to behave as if it were the only system running on the machine without worrying about protection and fairness.

Xen also allows virtual machine migration. A guest can be migrated from one system to another by pausing the guest, copying its memory, and resuming it at the destination system. Guests can also migrate while they are "live", meaning that they do not need to be paused for the entire duration of the migration.

3.2 Loosely Synchronized, Redundant Processing in Xen

We propose to use the infrastructure already developed for Xen to implement loosely synchronized, redundant virtual machines (LSRVM), a variation of the loose-lockstep processing architecture described in Section 2. In our proposed system, the redundant copies of the Linux operating system would run within Xen. The hypervisor can then ensure that each copy of Linux receives all interrupts at the same time as every other copy by using the NSK's rendezvous mechanism during calls into the Xen hypervisor. Furthermore, I/O operations from all of the guest operating systems will have to go through the Xen control software, allowing us to implement software voting schemes within Xen to ensure that all I/O operations are checked for correctness. Therefore, by augmenting the hypervisor and control software with redundancy and voting mechanisms, we will be able to support the abstraction that a logical processor is a collection of one or more guest operating systems.

Today, in order to support device virtualization, Xen intercepts all hardware interrupts and forwards them to the appropriate guest operating systems. These two steps are of necessity decoupled in Xen. As the hypervisor executes at the highest privilege level, it must receive all physical interrupts. The hypervisor is then responsible for distributing the interrupt to the appropriate guest(s). Consequently, to synchronize the delivery of interrupts among guest operating systems, only modest changes to Xen's interrupt forwarding will be required. Specifically, interrupt forwarding to a guest operating system will be delayed until the guest operating system arrives at a VRO. Initially, we will insert a VRO at each entry point into Xen.

In addition, Xen intercepts all trap instructions that are executed by applications running on guest operating systems. Consequently, system calls on the guest operating system will include a VRO without the need for any modifications to the guest operating system. Moreover, under Xen an idle guest operating system does not spin or halt the processor, but instead calls into Xen. Such an organization should provide low interrupt response latency. However, microprocessors are beginning to introduce support for virtualization that will reduce the frequency of execution of the hypervisor [12, 13]. For instance, an additional privilege level will be added specifically for hypervisor execution. The impact of this on the proposed design is that a system call trap will no longer enter the hypervisor, but rather will trap to the guest operating system directly. Therefore, without further modifications, the number of VROs will be dramatically reduced. We intend to fix this problem by modifying the trap handler in Linux to keep track of VROs and to enter the hypervisor only when a VRO has been reached. This approach will enable us to exploit the performance advantages of the virtualization support in future processors while still efficiently synchronizing the guest operating systems.

Upon a hardware fault within one guest, that guest must be stopped, and a new replica of the correctly executing guests must be created. We intend to modify the migration features of Xen to allow running guests to be cloned so that replicated execution can be resumed after a guest is terminated because of a failure.

In summary, LSRVM implements the four capabilities of loose-lockstep reliability discussed in the Introduction as follows:

1. **Isolation and redundancy:** Xen provides isolation among guest operating systems as a part of virtualization, and we provide redundancy by running identical software in loose-lockstep in multiple guest domains.
2. **I/O voting:** All I/O operations go through the Xen control software, providing a natural location for a software voter/replicator. The architecture and reliability of this voter/replicator will be discussed in Section 4.2.
3. **Interrupt synchronization:** The Xen hypervisor receives all interrupts, so it can synchronize the delivery of these interrupts to the redundant guest domains.
4. **Recovery:** The Xen migration features provide a natural substrate on which to build a mechanism to clone the surviving copies of the system after a failure.

By implementing loose-lockstep processing within the Xen framework, application execution will transparently be replicated within multiple guest domains providing application-level and operating system-level reliability without any additional programming overhead. However, for the system to be completely tolerant of hardware failures, both the software voter/replicator and the Xen hypervisor must be reliable as well. These two components are the only single points of failure in the LSRVM architecture.

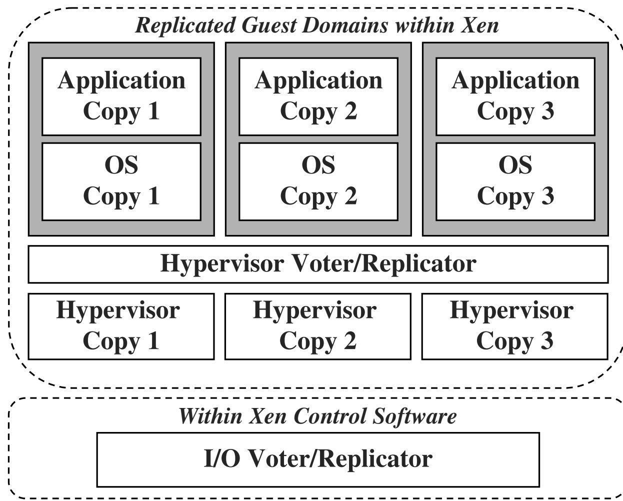

Figure 3: Loose-lockstep Processing Within Xen

4. Software reliability

In order to realize loose-lockstep processing in the NSAA, proprietary hardware mechanisms were implemented within the LSU. As described in Section 2, the LSU constitutes a single point of failure in the NSAA, so it is composed of self-checking hardware in order to make it a highly reliable component of the system. In the proposed LSRVM architecture, the responsibilities of the LSU are distributed throughout Xen (in both the hypervisor and the control software). The hypervisor and the software voter/replicator within the control software constitute single points of failure within the LSRVM. Therefore, it is important to use software-based fault tolerance techniques in order to achieve a level of reliability approaching that provided by the LSU in the NSAA architecture. The need for such software-based fault tolerance techniques is not specific to our approach, but rather would be necessary for any system architecture that uses software to implement the functionality provided by the hardware LSU.

The LSRVM organization is presented in Figure 3. The I/O voter/replicator and the triply-redundant copies of the operating system and applications are fundamental to the loose-lockstep model, as was shown in Figure 1. In addition, for a software-based system, the hypervisor must also be triplicated and include a voter/replicator, as shown in Figure 3. The replicated hypervisor ensures that hardware faults do not cause failures during the execution of the hypervisor.

4.1 Hypervisor Reliability

A failure during the execution of the hypervisor is catastrophic because such errors may potentially affect more than one guest operating system, nullifying the benefits of the redundant guests. For example, if a hardware error were to occur when the hypervisor is in the process of updating the page table for a particular guest, then that guest may have access to memory that belongs to another guest. This will potentially cause errors in both guests, which the loose-lockstep model may not be able to recover from, as it is designed to deal with the failure of a single replica at a time. Therefore, a system with a single hypervisor is vulnerable to hardware errors that can become catastrophic.

Providing redundant copies of the hypervisor eliminates this vulnerability. However, it becomes mandatory to vote every time the hypervisor either tries to change the state of the hardware or return a value to the guest operating systems. However, after the vote, a hardware fault could still corrupt the state change or return value. A corrupted replica of the return value should be caught later by the I/O voter/replicator when the guest receiving the corrupted data performs an incorrect I/O operation. The possibility of a hardware fault during a hardware state change, however, requires correctness checks within the hypervisor, backed by the hypervisor voter/replicator. After voting on a hardware state change, only a single hypervisor may actually change the hardware state (i.e., the page table base). Since a hardware fault could occur during the state change, it is necessary for all of the replicated hypervisors to check the state of the hardware and vote on its correctness after the change to ensure that it was updated correctly. If it was not, then the hypervisors must perform the update again.

4.2 Voter/replicator Reliability

Traditional research and design of majority voter/replicators for the hardware application space has focused on self-checking implementations [14–19]. Such a self-checking voter/replicator was implemented in the NSAA architecture [10]. A hardware voter/replicator is said to be self-checking if it is both fault-secure and self-testing, where the properties are defined with respect to a specified class of hardware faults *within* the voter. The most commonly used fault model for self-checking voter/replicator design is the single stuck-at fault model, which is a logical fault model for permanent faults in integrated circuits. It assumes that one of the wires in the circuit is permanently fixed at the high or low logic value, regardless of the inputs to the circuit [20]. A hardware voter/replicator is fault-secure *iff* for every single stuck-at fault in the voter/replicator, it never produces an incorrect output for any valid input. A hardware voter/replicator is self-testing *iff* for every single stuck-at fault in the voter/replicator, it produces an invalid output for some valid input.

Traditional research and design of voter/replicators for the software application space has focused on implementations from a software fault tolerance standpoint [21–25]. For software applications, the emphasis is on verifying the results of program execution. It is assumed that the voter/replicator is itself reliable, and is minimally exposed to hardware failures that may affect its execution.

The software voter/replicator for the LSRVM architecture is different from hardware-only and software-only voters in that it constitutes a single point-of-failure that is exposed to hardware failures. The exposure arises because the utilization of the voter/replicator is potentially high if the applications generate significant I/O traffic. The software voter/replicator for the LSRVM architecture must thus be the equivalent of a self-checking hardware voter/replicator both in functionality and reliability. It is proposed to use software fault tolerance techniques based on robustness in its design to make it fault-secure and self-testing. The software property of robustness is defined as “the extent to which software can continue to operate correctly despite the introduction of invalid inputs” [26]. Extending these concepts further, it is now widely accepted that robustness enhancements can be used to embed self-checking attributes in software [25].

The fault-secure property for the software voter/replicator is potentially achieved by using robustness techniques to detect and handle incorrect inputs as well as by keeping a tight check on control sequences during execution of the voter/replicator software. Such checks include monitoring for infinite loops, incorrect loop terminations, illegal branches, etc. that may occur due to failures in the hardware and raising exceptions as appropriate when failures are detected. By using exceptions to facilitate early detection and handling of failures during its execution, the voter/replicator ensures that an incorrect output is never produced for any valid input.

The self-testing property for the software voter/replicator requires that the voter/replicator be able to identify potential flaws in its design as well as hardware faults that may corrupt its execu-

tion. Although software's complexity makes enumerating all possible faults intractable, the software voter/replicator is well-defined with a simple interface. As a result, well known software fault tolerance techniques for robustness enhancement based on reasonableness checks on the outputs and keeping a tight check on control sequences during execution of the software (as described above) can be used to enhance the coverage to faults not only during development and test, but also at runtime.

5. Conclusions

In order to achieve true software dependability, the underlying execution environment must be reliable. It is impractical to expect that commodity hardware will provide such reliability without a dramatic increase in cost. LSRVM is a practical software-only approach to hardware fault tolerance that provides a reliable substrate upon which to implement software dependability techniques.

LSRVM is a novel, virtual machine-based software architecture for building highly reliable computer systems. The architecture presented in this paper protects not only the application but also the operating system from hardware errors, enabling the use of open-source operating systems. Furthermore, fault tolerance is completely implemented in software, eliminating the need for custom-designed hardware. The only single point-of-failure in the proposed architecture is confined to a well-defined, self-checking voter/replicator module. This module is simple enough that software fault tolerance techniques based on robustness enable a realization that replicates the functionality and reliability of a self-checking hardware voter/replicator.

6. References

[1] P. Rubinfeld, "Virtual roundtable on the challenges and trends in processor design: Managing problems at high speeds," *IEEE Computer*, vol. 31, no. 1, pp. 47–48, 1998.

[2] S. Borkar *et al.*, "Parameter variations and impact on circuits and microarchitecture," in *Design Automation Conference*, pp. 338–342, 2003.

[3] S. Borkar, T. Karnik, and V. De, "Design and reliability challenges in nanometer technologies," in *Design Automation Conference*, pp. 75–75, 2004.

[4] M. A. Breuer, S. K. Gupta, and T. M. Mak, "Defect and error tolerance in the presence of massive numbers of defects," *IEEE Design and Test of Computers*, vol. 21, pp. 216–227, May-June 2004.

[5] J. Tschanz, K. Bowman, and V. De, "Variation-tolerant circuits: circuit solutions and techniques," in *Design Automation Conference*, pp. 762–763, 2005.

[6] T. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Intl. Symposium on Microarchitecture*, 1999.

[7] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," in *Fault-Tolerant Computing Symposium*, pp. 84–91, 1999.

[8] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance," *SIGPLAN Notices*, vol. 35, no. 11, pp. 257–268, 2000.

[9] J. Bartlett, J. Gray, and B. Horst, "Fault tolerance in Tandem computer systems," Technical report 86.2, Tandem Computers, March 1986.

[10] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "Nonstop advanced architecture," in *Intl. Conference on Dependable Systems and Networks*, 2005.

[11] P. Barham *et al.*, "Xen and the art of virtualization," in *ACM Symposium on Operating Systems Principles*, 2003.

[12] Intel, *Intel Virtualization Technology Specification for the Intel Itanium Architecture (VT-i)*, April 2005. Revision 2.0.

[13] Advanced Micro Devices, *Secure Virtual Machine Architecture Reference Manual*, May 2005. Revision 3.01.

[14] D. K. Pradhan, ed., *Fault-tolerant computing: Theory and techniques*, vol. 1. NJ, USA: Prentice-Hall, 1986.

[15] M. Nicolaidis and B. Courtois, "Strongly code disjoint checkers," *IEEE Trans. Computers*, vol. 37, pp. 751–756, June 1988.

[16] C. E. Stroud, "Reliability of majority voting based vlsi fault-tolerant circuits," *IEEE Trans. Very Large Scale Integration Systems*, vol. 2, pp. 516–521, December 1994.

[17] D. K. Pradhan, *Fault Tolerant Computer System Design*. Prentice-Hall, 1996.

[18] S. Mitra and E. J. McCluskey, "Word-voter: A new voter for triple modular redundant systems," in *VLSI Test Symposium*, pp. 465–470, 2000.

[19] J. M. Cazeaux, D. Rossi, and C. Metra, "New high speed CMOS self-checking voter," in *Proc. Intl. On-line Testing Symposium*, pp. 58–63, 2004.

[20] M. L. Bushnell and V. D. Agrawal, eds., *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. MA, USA: Kluwer Academic Publishers, 2000.

[21] J. H. Wensley *et al.*, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," vol. 66, pp. 1240–1255, Oct. 1978.

[22] N. G. Leveson *et al.*, "The use of self checks and voting in software error detection: An empirical study," *IEEE Transactions on Software Engineering*, vol. 16, pp. 432–443, Apr. 1990.

[23] J. L. Gersting *et al.*, "A comparison of voting algorithms for n-version programming," in *Intl. Conference on System Sciences*, pp. 253–262, 1991.

[24] J. M. Bass, G. Latif-Shabgahi, and S. Bennett, "History-based weighted average voter: A novel software voting algorithm for fault-tolerant computer systems," in *Euromicro Conference*, pp. 402–409, 2001.

[25] L. L. Pullum, *Software fault tolerance techniques and implementation*. MA, USA: Artech House, Inc., 2001.

[26] IEEE Standard 729-1982, *IEEE Glossary of Software Engineering Terminology*. IEEE, 1982.