Increasing Relevance of Memory Hardware Errors
A Case for Recoverable Programming Models

Dejan Milojicic, Alan Messer, James Shau, Guangrui Fu, Alberto Munoz
HP Labs
1501 Page Mill Road, MS 1U-18, Palo Alto, CA 94304
[dejan, messer, jshau, guangrui, bmunoz]@hpl.hp.com

Abstract

It is a common belief that most of computer system failures nowadays stem from programming errors. Computer systems are becoming more complex and harder to maintain and administer, making software errors an even more common case, while contemporary computer architectures are optimized for price and performance and not for availability. In this paper, we raise a case for an increasing relevance of memory hardware soft-errors. In particular with the introduction of 64-bit processors, memory scaling is significantly increased, resulting in higher probability for memory errors. At the same time, due to the ubiquitous use of computers, such as at higher altitudes, environmental conditions impact errors (terrestrial cosmic rays). Finally, in shared memory systems, the failure of one node's memory can take the whole machine down. Current commodity systems do not tolerate memory errors, neither commodity hardware (processors, memories, interconnects) nor software (operating systems, applications, application environments). At the same time, users expect increased reliability. We present the problems of such errors and some solutions for memory error recovery at the processor, operating system and programming model level.

1 Introduction

Demand for increased performance and high availability of commodity computers is increasing with the ubiquitous use of computers and the Internet services which serve them. While commodity systems are tackling the performance issues, availability has received less attention. It is a common belief that software (SW) errors and administration time are, and will continue to be, the most probable cause of the loss of availability. While such failures are clearly commonplace, especially in desktop environments, it is believed that certain other hardware (HW) errors are also becoming more probable. Processors, caches, and memories are becoming larger, faster and more dense, while being increasingly used in ubiquitous and adverse environments such as at high altitudes, in space, and in industrial applications.

Ziegler et al. [21, 22] have shown that these changes will lead to increased transient errors in CMOS memory due to the effects of cosmic rays, approximately 6000 FIT (1 FIT equals 1 failure in 10^9h) for one 4Mbit DRAM. Tandem [18] indicates that such errors also apply to processor cores or on-chip caches at modern die sizes/voltage levels. They claim that processors, cache, and main memory are all susceptible to high transient error rates. A typical processor's silicon can have a soft-error rate of 4000 FIT, of which approximately 50% will affect processor logic and 50% the large on-chip cache. Due to increasing speeds, denser technology, and lower voltages, such errors are likely to become more probable than other single hardware component failures.

With the increasing evolution to larger tightly-interconnected commodity machines (such as Sun's Enterprise 10000 machines), the probability of soft-errors and error containment problems increases further. Soft-error probability increases not only due to increased system scale, but also due to an increased number of components on the memory access path. Since the machines are tightly-coupled, memory path soft-errors introduce error containment problems which without some form of software error containment can lead to complete loss of availability.

Techniques such as Error Correction Codes (ECC) and ChipKill [7] have been used in main memories and interconnects to correct some of these errors (90% for ECC [18]). Unfortunately such techniques, only help reduce visible error rates for semiconductor elements that can be covered by such codes (large storage elements). With raw error rates increasing with technological progress and more complicated interconnected memory subsystems, ECC is unable to address all the soft-error problems.

For example, a 1Gb memory system based on 64Mbit DRAMs still has a combined visible error rate of 3435 FIT when using Single Error Correct-Double Error Detect (SEC-DED) ECC [7]. This is equivalent to around 900 errors in 10000 machines in 3 years. Unfortunately, current commodity hardware and software provide little to no support for recovery from errors not covered by ECC whether detected or not.

Such problems have been considered by mainframe technology for years, but in the field of commodity hardware, it is currently not cost-effective to provide full redundancy/support in order to mask errors. Therefore, the burden falls to commodity hardware and the software using it to attempt to handle these errors for the highest availability. Most contemporary commodity computer systems, while providing good performance, pay little attention to availability issues resulting from such errors. For example, the IA-32 architecture supports only ECC on main memory rather than across the system, requiring system reboot on errors not covered by this ECC. Consequently, commodity software such as the OS, middleware and applications have been unable to deal with the problem.

Future commodity processor architectures may provide support to detect and notify the system of such probable errors. For instance, upcoming IA-64 processors, while not recoverable in the general case, do offer some support with certain limitations. In this paper, we present the problems of such errors in the context of support from IA-64-based hardware. We then propose certain software techniques that could be used at different levels (firmware, operating system, and applications) to allow availability to be maintained in software.

The rest of the paper is organized as follows. Section 2 describes related work. In Section 3 we present recovery from memory errors. We describe the IA-64 recovery model in Section 4, the OS recovery in Section 5, and recoverable programming models in Section 6. In Section 7 we summarize the paper and present future work.

2 Related Work

Availability in computer systems is determined by hardware and software reliability. Hardware reliability has traditionally existed only in proprietary servers, with specialized redundantly configured hardware and critical software components, possibly with support for processor pairs [1], e.g. IBM S/390 Parallel Sysplex [15] and Tandem NonStop Himalaya [6]. Sysplex supports hot swap execution, redundant shared disk with fault-aware system software for error detection and fail-over restart. Tandem supports redundant fail-over lock-stepped processors with a NonStop kernel and middleware, which provide improved integrity through the software stack. These systems provide full automatic support to mask the effects of data and resource loss. They rely on reliable memory and fail-over rather than direct memory error recovery.

Another approach is fault containment and recovery with “node” granularity. In these systems, each node has its own kernel. When one node fails, the others can recover and continue to provide services. Systems of this type include the early cluster systems [16], and NUMA architectures, such as Hive [3, 20].

Hardware faults are difficult to catch and repeat. Therefore, a lot of research and development is based on emulation and injection of hardware faults [11, 12]. Faults can be injected by specially designed hardware or by software. Hsueh et al. give a survey and comparison on different injection methods [9].

Software reliability has been more difficult to achieve in commodity software even with extensive testing and

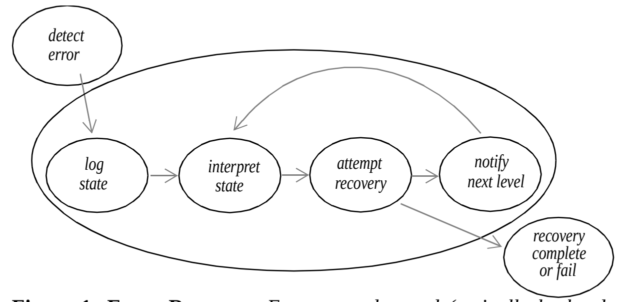

Figure 1: Error Recovery. *Errors are detected (typically by hardware), then the error state is logged, interpreted, and recovery attempted. If unsuccessful, the next level may be notified.*

quality assurance. Commodity software fault recovery has not evolved very far. Most operating systems support some form of memory protection between units of execution to detect and prevent wild read/writes. But most commodity operating systems have not tackled problems of memory errors themselves or taken up software reliability research in general. Examples include Windows 2000 and Linux. They typically rely on failover solutions, such as Wolfpack by Microsoft [16].

A lot of work has been undertaken in the fault-tolerant community regarding the problems of reliability and its recovery in software [2, 8, 13]. These include techniques such as checkpointing [8] and backward error recovery [2]. A lot of this work has been conducted in the context of distributed systems rather than in single systems. There are also techniques for efficient recoverable software components, e.g. RIO file cache [5], and Recoverable Virtual Memory (RVM) [17].

Rio [5] takes an interesting software-based approach to fault containment aimed at a fault-tolerant file cache, but with general uses. By instrumenting access to shared data structures with memory protection operations, wild access to the shared data structures becomes improbable. This containment using software techniques can be viewed as a similar concept to the one we motivate with our more general investigation into a framework for recovery in this paper.

3 Memory Errors Recovery Overview

In an ideal solution, all hardware would be replicated and processors would be able to detect, notify, and recover from hardware errors, and the affected operations could be restarted if errors are recovered from. This way all errors would be masked by hardware and higher levels of software would not need to be aware of errors.

Unfortunately, only expensive high-end servers partially provide this form of support, whereas commodity systems provide little support and thus must rely on software to recover from errors. We believe that software needs to recover from errors at different levels, as pre-

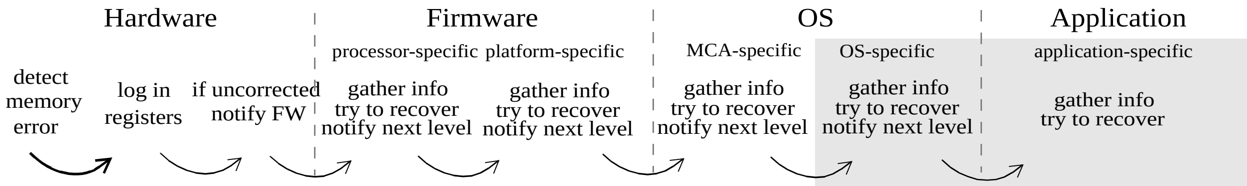

Figure 2: Memory failure recovery scenario. *Memory error is typically detected by HW (e.g. parity error) and it is logged. If the error cannot be contained, it is notified to FW. FW gathers information and attempts recovery. Recovery is performed at the processor- and at the platform-level. If recovery is possible, the state is prepared for OS and it is notified. OS attempts recovery at the MCA- and at the OS-level. In case of successful OS recovery, application is notified with relevant state. Application analyzes the state and attempts to recover. All but the first arrows are optional.*

sented in Figure 1:. An error is typically detected at the hardware level, and then it is interpreted, logged, and if needed the next level is notified. The interpret/log/notify phases are repeated at different levels until either the error is recovered or determined as non-recoverable. This order of events is presented in Figure 2:, where the firmware (FW) level is split into processor and platform specific, and the OS level into Machine Check Abort (MCA, a serious error exception) and OS-specific.

Most systems don't have support for the full chain of recovery through to application. Instead, they provide little more than error detection, as indicated by the shaded portion of Figure 2: and subsequent tables.

How these steps might lead to recovery is described in more detail in Table 1. Hardware typically detects errors, such as a timeout on memory access, parity error, or illegal machine state. Even if hardware does not detect some errors, it is possible for software to detect this inconsistent data (usually resulting in silent data corruption) typically represented in the form of invalid pointers or incorrect checksums. Hardware logs the error in registers and if the error is not corrected it either notifies firmware (e.g. by raising an MCA) or uses "erroneous" data. Erroneous data raises notification when accessed, thereby postponing recovery until data is used. Firmware determines whether it can contain the error or otherwise notifies the OS. Currently, no further software recovery is done in existing systems, but we believe the same framework should continue, allowing the OS and application to attempt to distinguish the error by type and criticality in order to determine if error is recoverable. As part of logging, the state is translated between the levels, e.g. parity error is converted to physical page enqueued on bad page list.

Table 1: How is a Failure Recovered at Different System Levels.

Level	Recovery			
	Detect	**Interpret**	**Log**	**Notify**
Hardware	*timeout, parity, illegal machine state*		*write registers*	*eager:* *raise MCA signal* *lazy:* *erroneous data*
Firmware		*distinguish failure type/criticality, e.g. thread restartable*	*translate HW state into FW state*	*raise OS MCA*
OS	*SW detected data inconsistency*	*distinguish failure type/criticality e.g.OS/app thread*	*translate FW state into OS state*	*raise app exception*
Apps	*SW detected data inconsistency*	*distinguish failure type/criticality e.g. data lost*	*translate OS state into app state*	*alert user*

The outcome of the recovery hierarchy can be full or partial recovery, or system failure (see Table 2). Full recovery effectively masks the errors from higher levels; error may be logged for statistical purposes, but no action is needed from higher levels. In case recovery is not possible, the system is halted or rebooted to prevent corrupt data from propagating to network or disk.

4 The IA-64 Recovery Model

In most processors, a machine-check notifies execution of a serious error, but leaves the processor in an undefined state requiring a system reboot due to loss of containment. The IA-64 architecture extends the recoverability of machine-checks by providing several types of well-defined error scenarios. This provides the potential for more information to allow software containment of the error.

IA-64 is a complex architecture. The current processor implementations (Merced, McKinley) support out-of-order completion of memory operations, speculative prefetching, advanced instruction retirement, and an exposed VLIW (Very Long Instruction Word) architecture. This leads to difficulties in providing full error detection/handling while remaining cost-effective due to the extra complexity. In order to isolate software from implementation dependencies, the IA-64 architecture abstracts machine-check handling. This allows implementations to support different detection and logging while maintaining the same architectural interface.

The IA-64 architecture [10] defines five flags to describe the state of the error and the processor in the

Table 2: Memory Failures Recovery Outcome (read horizontally).

Level	Recovery		
	Full Recovery	**Partial Recovery**	**System Failure**
Hardware	*mask errors*	*halt / downgrade performance/functionality*	*halt/ reboot*
Firmware	*mask error, (notify OS)*	*notify OS*	*reboot*
OS	*continue to execute OS*	*notify app* *kill user thread*	*reboot*
Application	*continue to execute application*	*notify user*	*terminate*

Table 3: Description of the IA-64 Machine-Check System State.

Flag	Description
Storage Integrity Synchronized	*All loads and stores before the machine-check occurred and those following appear to have not occurred.*
Continuable	*All in-flight operations completed or, tagged as erroneous/incomplete and are restartable on re-issue.*
Machine-Check Isolated	*The machine-check was isolated by the system and may or may not be recoverable.*
Uncontained Storage Damage	*Error contained in the storage hierarchy, but it may contain corruption.*
Hardware Damage	*Non-essential hardware has been damaged and the processor will continue to run at degraded performance.*

presence of a machine-check (see Table 3). Depending on the processor/platform implementation, a combination of these flags will be signalled to the operating system by the processor firmware.

If storage integrity is not synchronized, execution is not continuable, or storage damage is not contained, then the repercussions for software using the processor are quite severe. The current execution may not be restartable or corruption may have occurred to state in user or kernel space.

We believe that in these cases, there is scope for software recovery, especially for probable error cases. However, since containment may have been lost, corrupt state may have propagated through system hierarchy.

Table 4 gives an outline of the typical kinds of error handling in an IA-64-based system. Current commodity systems do not support software-level of recovery for certain classes of hardware faults, leaving recovery to the firmware and hardware. This enables simple recovery of some common cases where sufficient information at this level is available. However, in the cases of "non-continuable" execution, we believe that recovery is insufficient at this level and requires semantic information in the operating system and application levels. The architectural assertion has been that unless one can identify exactly the effects of the loss of containment, one cannot recover (the shaded area).

At the IA-64 firmware/OS interface there are several possible techniques which we believe can extend the recoverability in cases when processor implementations lose containment. These center around techniques for additional containment information and for recovery from the errors. Additional information for higher level recovery may be improved by techniques, such as:

1. determine if an error affected user or kernel space
2. backtrack execution point to locate erroneous execution

IA-64 recovery may be improved by techniques, such as:

1. less susceptible versions of instruction sequences
2. checkpointing execution

Table 4: Error Detection/Containment/Recovered on IA-64 System

Level	Recovery			
	Detection	**Interpretation**	**Containment**	**Recovery**
Hardware	*single bit $ ECC, memory ECC, bus errors, interconnect*	*HW signals to form error logging information*	*certain types of memory access (instruc. access exceptions, etc.)*	*those with no platform side-effects (e.g. refetch instruc.)*
Firmware	*N/A*	*implementation specific error information*	*sensitive error cases*	*mask and report nested errors*
OS	*is error critical to OS data structures*	*architectural error info. & OS error detection info.*	*error effects on units of execution (threads, processes)*	***if** app. recovery possible - notify it **else if** localized - terminate it **else** reboot*
Applications	*transaction error*	*error notification & info. from OS*	*errors within single transaction*	*restart operation*

An important part of our work will be to analyze IA-64 implementations to determine the error scenarios and their effects on the system state. We also wish to determine how probable these scenarios are under typical executions. This will allow us to focus investigation on techniques to improve availability where most probable and beneficial.

5 Memory Errors and Operating Systems

Error recovery at the OS level is an attempt to improve the availability of a kernel with containment and recovery at the process/thread granularity. Since complete error masking support at the OS level is a large and complex task we do not attempt to recover from all errors. Instead, we believe that availability can be effectively improved by OS awareness to containment of probable errors and recovery through armoring of critical data structures. This would increase availability where most effective, while either passing rare errors requiring complex recovery onto the application or rebooting the system if they are so severe.

Those machine checks that cannot be recovered at lower levels will most likely require some form of OS localization of dependencies, notification for potential application recovery, and containment by potential termination of dependent execution. Once localized, whether or not the system can survive depends on the state of system with respect to the current execution. The execution state may be classified in order of increasing complexity for recovery:

1. User threads accessing user data.
2. Kernel threads which are restartable, idempotent, or do not modify kernel state, e.g. get-time-of-day.
3. Kernel threads modifying non-critical kernel data: e.g. copy of I/O buffers, not holding critical locks.
4. Kernel threads inside critical sections armored against errors, e.g. transaction-based and replica.

We believe (and intend to investigate) that these states cover a large number of error scenarios and do not require a traditional reboot. Instead they can lead to application recovery if it exists (1), kernel-assisted retry (2-4), or finally to containment by execution termination if other recovery fails.

Armoring data structures (such as that by Taylor et al. [19]) aims to make critical data structures more resilient to asynchronous notification of errors and to errors in the data structures themselves. This support aims to enable roll-back for re-execution once errors are contained and/or replication for data corruption recovery.

Armoring critical data structures can also help reduce another important problem for memory errors, silent data corruption. Tandem [18] details that those errors not caught by ECC cause potential data corruption. While 46.4% are detected (illegal instructions, invalid computation, etc.) and 41.2% are benign (locations not accessed), at least 12.4% cause silent data corruption. Armoring of critical data structures reduces the probability of these affecting the OS and causing a reboot.

6 Recoverable Programming Models

In this section we investigate how software (firmware, OS, or application) can react to errors. The task of hardware is to detect errors, log the event, and notify errors to the firmware and operating system unless it can be masked. If hardware can mask the error, there is no action required by the system software, otherwise the software needs to be aware of the error and it needs to react accordingly. We compare four programming models with four hardware support models in an attempt to find appropriate tradeoffs and an optimal balance between the two.

The simplest model from the software perspective is when all errors are masked by hardware. In that case, the programming model requires no changes. Programs are the same as when errors are not a concern. If the hardware cannot mask the errors then it needs to expose them to the next level in hierarchy. It can do so in three ways with different repercussions to the software model. The simplest (software-wise) approach is if hardware can notify software in an exact and restartable manner, by raising a signal, such as an interrupt (by IA-64 definition, MCAs are not guaranteed to be "exact" or "restartable" unlike other interruptions). Software can react to this signal by invoking an exception handler. Handlers have minimal impact on software, since they are an extension to the existing code.

If the error cannot be notified in an exact manner (i.e. notification is delivered asynchronously, such as in the case of IA-64 MCA), then the software needs to offer even more support for recovery. This can be achieved in terms of Structured Exception Handling (SEH), as presented in the following code excerpt:

```
try {
        accessMemory;
} except {
        performRecovery();
}
```

In order to be able to restart the memory transaction, it is required that, as a part of the `try` clause, sufficient architectural state be saved, in addition to guaranteeing completion of all operations before the "`except`" clause.

If there is neither support for notification nor notification guarantees are sufficient (e.g. asynchronous notification delivery cannot guarantee recovery), then hardware logging of errors combined with software polling of errors can be used, as described below (details omitted, such as disabling notifications and enabling logging):

```
accessMemory;
if (pollErrorLog()) {
        raiseException();
}
```

Table 5: Recoverable Shared Memory Programming Models v. Hardware Support: "X" is a reasonable match, "x" "over-powered", and empty entries "under-powered".

increasing hardware support →

↓ *increasing software support*

Software Model	**Hardware Support**			
	Logging	**Async. Notification**	**Synch. Notification**	**Reliable Hardware**
No Recovery				***X***
Exception handler			***X***	*x*
SEH & Transactions/ checkpointing		***X***	*x*	*x*
Polling for Errors (each mem access)	***X***	*x*	*x*	*x*

In Table 5 we compare how different programming models match the hardware support. In the columns, we describe the increasing hardware support (logging of errors, async. notification, sync. notification, and reliable hardware). Rows describe increasing complexity of software support (no recovery case, exception handlers, structured exception handling, and polling for errors). Table entries with capital ***X*** mark a reasonable match between software and hardware. Small ***x*** marks possible but over-powered model, where hardware provides more support and will be under-utilized by the programming software model. Empty entries describe forbidden (under-powered) cases, where hardware does not provide sufficient support for the proposed recoverable programming model.

In Table 6 we compare the presented programming models from the perspective of code complexity, performance, error-proneness, application error-awareness, suitability for OS v. applications, and the need for extra hardware. Intuitively, the more hardware support is available the less software support is needed for recov-

Table 6: Recoverable Programming Models Comparison

Programming Model	Characteristic					
	Legacy v. Complexity	**Perfo-rmance**	**Impl. Error-Prone**	**Application Error-Awareness**	**OS v. App. Suited**	**Extra HW**
No recovery	*no change*	*fast*	*no*	*no*	*app*	*exten-sive*
Exception Handler	*min. change*	*expensive (save arch. state)*	*mini-mal*	*small*	*either*	*pro-cessor feature*
SEH & Transactions/checkpoint	*relatively complex*	*modest (save partial state)*	*fairly*	*yes*	*either*	*yes*
Poll for Error	*complex*	*slow*	*very*	*yes*	*OS*	*minim.*

ery, the better performance, and less error-prone is the implementation of the model. The less hardware support, the more awareness of the errors is required from software. Alternatively, the less aware software is, the more extra hardware support is required. Complex software support is potentially applicable at the OS level, but it is typically not appealing for the applications.

7 Summary and Future Work

In this position paper, we emphasized the case for recovering from memory errors in future commodity systems. There is a common belief that software errors dominate all other errors. While acknowledging this fact, we believe that with the introduction of IA-64-based systems (i.e. those with significantly larger memories), with ubiquitous use of computers (e.g. at high altitudes), as well as with the increasing complexity (e.g. multiprocessors with commodity interconnects), the probability of memory errors will significantly increase, beyond the expectations of users.

We have overviewed the current state of the IA-64 processor support for errors, the OS support, and we compared a few recoverable programming models. We believe that recovery from memory errors will be an important requirement for most commodity systems of tomorrow, and clear understanding of processor architecture, OS, and application support will be needed for systems that require high reliability.

It is important to point out that many recovery aspects discussed here have existed in mainframe technology for over twenty years. Our contribution is to address these issues for commodity hardware and software, resulting in a suggested framework of how to compose and distribute recovery functionality in future commodity hardware and software for improved availability.

Our future work will consist of further understanding the behavior of IA-64, of experimental improvements to the OS architecture's susceptibility to errors, and in continued research into the tradeoffs of recoverable programming models.

Acknowledgments

We are indebted to Valentin Anders, John Janakiraman, Dan Osecky, Todd Poynor, Mike Traynor, Tom Wylegala, and John Wilkes for reviewing the paper and otherwise contributing to the project. Their comments significantly improved the content and presentation.

References

[1] Bartlett, J., “A Nonstop Kernel”, Proceedings of the Eighth Symposium on Operating Systems Principles, Asilomar, Ca, pp 22-29, Dec. 1981.

[2] Brown, N.S. and Pradhan, D.K. “Processor- and Memory-Based Checkpoint And Rollback Recovery”, IEEE Computer, pp 22-31, Feb. 1993.

[3] Chapin, J., et al., “Hive: Fault Containment for Shared-Memory Multiprocessors,” *Proc. of the 15th SOSP*, Dec. 1995, pp 12-25.

[4] Charlsworth, A., “Starfire: Extending the SMP Envelope”, IEEE Micro, January/February 1998, pp 39-49.

[5] Chen, P.M., et al., "The Rio File Cache: Surviving Operating System Crashes", *Proc. of the 7th ASPLOS*, pp 74-83, October 1996.

[6] Compaq, Product description for Tandem Nonstop Kernel 3.0. Download Feb. 2000, http://www.tandem.com/prod_des/tdnsk3pd/tdnsk3pd.htm.

[7] Dell, T. J., “A White Paper on the benefits of Chipkill - Correct ECC for PC Server Main Memory”, IBM Microelectronics Division, Nov. 1997.

[8] Gray, J., and Reuter, A., “Transaction Processing: Concepts and Techniques,” Morgan Kaufmann, 1993.

[9] Hsueh, M-C, Tsai, T. K., and Iyer, R. K., "Fault Injection Techniques and Tools", IEEE Computer, pp 75-82, April 1997.

[10] Intel IA-64 Architecture Software Developer’s Manual, Volume 2, Intel 1999, Intel Corporation.

[11] Kanawati, G., et al., “FERRARI: A Flexible Software-based Fault and Error Injection System", IEEE Transactions on Computers, Vol 44, No2, pp 248-260, Feb, 1995.

[12] Kao, W.-l., et al., “FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults", IEEE T-SE vol 19, no 11, pp 1105-1118, November, 1993.

[13] Kermarrec, A-M., et al., “A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability”, *Proc. of the 25th FTCS*, pp 289-298, June 1995.

[14] Lampson, B., “Hints for Computer System Design”, *Proc. of the 9th SOSP*, October 1983, pp 33-48.

[15] Nick, J.M., et al., “S/390 Cluster Technology: Parallel Sysplex”, *IBM Systems Journal*, vol 36, no 2., pp 172-201, 1997.

[16] Pfister, G., “In Search of Clusters”, Prentice Hall, 1998.

[17] Satyanarayanan, et al. “Lightweight Recoverable Virtual Memory”. Proc. SOSP, pp 146-160, Dec. 1993.

[18] Tandem, Compaq Corporation, “Data Integrity for Compaq NonStop Himalaya Servers”, White Paper, 1999.

[19] Taylor, D., et al., “Redundancy in Data Structures: Improving Software Fault Tolerance”, IEEE T-SE, vol 6, 1980, pp 595-602.

[20] Teodosiu, D., et al., “Hardware Fault Containment in Scalable Shared-Memory Multiprocessors,” *Proc. of the 24th ISCA*, pp 73-84, June 1997.

[21] Ziegler, J. F., et al., “IBM Experiments in Soft Fails in Computer Electronics (1978-1994)”, IBM Journal of R&D, vol 40, no 1, pp 3-18, January 1996.

[22] Ziegler, J. F., “Terrestrial Cosmic Rays”, IBM Journal of R&D, vol 40, no 1, pp 19-40, January 1996.