Susceptibility of Modern Systems and Software to Soft Errors

Alan Messer, Philippe Bernadat, Guangrui Fu, Deqing Chen[1], Zoran Dimitrijevic[2], David Lie[3], Durga Devi Mannaru[4], Alma Riska[5], Dejan Milojicic
Computer Systems and Technology Laboratory
HP Laboratories Palo Alto

HPL-2001-43
March 7th , 2001*

E-mail: [messer, bernadat, guangrui, dejan]@hpl.hp.com, lukechen@cs.rochester.edu, zoran@cs.ucsb.edu, davidlie@stanford.edu, durga@cc.gatech.edu, riska@cs.wm.edu

It is widely understood that most downtime is accounted for by programming errors and administration time. However, recent work has indicated an increasing cause of downtime may stem from transient hardware errors caused by external factors, such as cosmic rays. Moving to denser semiconductor technologies at lower voltages will cause an increase in transient errors. We investigate the trends in transient errors and the susceptibility of operating systems and applications to them, and we introduce ideas regarding software transient error recoverability.

We believe that if transient errors become a prominent problem, that it will be possible to improve commodity system availability with simple software recovery. Results indicate that in the Linux kernel and a Java virtual machine few errors need to be fatal. We also propose two recovery examples which we believe indicate that it is possible to increase error detection and recovery without the cost of a fail-over cluster.

* Internal Accession Date Only Approved for External Publication

[1] University of Rochester, Rochester, NY, 14627
[2] UC Santa Barbara, Santa Barbara, CA, 93106
[3] Stanford University, Stanford, CA, 94305
[4] Georgia Tech, Atlanta, GA, 30332
[5] College of William & Mary, Williamsburg, VA, 23187

Please do not distribute or reference until accepted for publication. In preparation for the International Conference on Dependable Systems and Networks, 01-04 July 2001: Göteborg, Sweden. Submission due 6th December 2000! Max. 10 letter pages at 10pt or greater.

Susceptibility of Modern Systems and Software to Soft Errors

Alan Messer, Philippe Bernadat, Guangrui Fu, Deqing Chen[1], Zoran Dimitrijevic[2], David Lie[3], Durga Devi Mannaru[4], Alma Riska[5], Dejan Milojicic

HP Labs, Uni. of Rochester[1], UC Santa Barbara[2], Stanford University[3], Georgia Tech[4], College of William & Mary[5]

[messer, bernadat, guangrui, dejan]@hpl.hp.com, lukechen@cs.rochester.edu[1], zoran@cs.ucsb.edu[2], davidlie@stanford.edu[3], durga@cc.gatech.edu[4], riska@cs.wm.edu[5]

Abstract

It is widely understood that most downtime is accounted for by programming errors and administration time. However, recent work has indicated an increasing cause of downtime may stem from transient hardware errors caused by external factors, such as cosmic rays. Moving to denser semiconductor technologies at lower voltages will cause an increase in transient errors. We investigate the trends in transient errors and the susceptibility of operating systems and applications to them, and we introduce ideas regarding software transient error recoverability.

We believe that if transient errors become a prominent problem, that it will be possible to improve commodity system availability with simple software recovery. Results indicate that in the Linux kernel and a Java virtual machine few errors need to be fatal. We also propose two recovery examples which we believe indicate that it is possible to increase error detection and recovery without the cost of a fail-over cluster.

1 Introduction

Demand for increased performance and high availability of commodity computers is increasing with the ubiquitous use of computers. While commodity systems are tackling the performance issues, availability has received less attention. It is a common belief [11] that software 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 errors are also becoming more probable [18].

Hardware errors can be classified as hard errors and transient (soft) errors. Hard errors are those that require replacement (or otherwise relinquished use) of the component. These typically happen as a consequence of physical damage of the component, e.g. by damage to connectors. Transient errors are those that result in an invalid state that can be corrected, for example, overwriting a corrupt memory location. Ziegler et al. have discovered that cosmic rays and alpha particles cause semiconductor transient errors (or soft errors) in memories [27, 28]. Initial results demonstrated approximately 6000 FIT (failures in 10^9 hours) for one 4Mbit DRAM. Tandem indicate that such errors also apply to processor cores and on-chip caches at modern die sizes/voltage levels [24]. They claim that processors, cache, and main memory are all susceptible to high transient error rates. A typical CPU can have a soft error rate of 4000 FIT, of which approximately 50% will affect processor logic and 50% the large on-chip cache.

Techniques such as Error Correction Codes (ECC) and ChipKill [10] have been used in main memories, storage media and interconnects to correct some of these errors (90% correction rate for ECC [24]). Unfortunately, such techniques only help reduce visible error rates for semiconductor elements that can be covered by such codes, i.e. large memories. Undetected errors either due to an absence of protection or escaping existing ECC protection lead to random corruption to execution state, known as silent data corruption.

We believe that due to increasing speeds, denser technology, and lower voltages which affect transient error rates, such that ECC may be unable to address all the soft error problems, and that such errors may become more probable than other single hardware component failures. 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 [10]. 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 whether detected by ECC or not.

Such problems have been considered by mainframe technology for years using expensive proprietary hardware and software [9, 20]. However, in the field of commodity systems, it is currently not cost-effective to provide full hardware redundancy/detection support to mask errors. Instead, commodity systems choose to rely on slower fail-over clusters as an economic solution to providing availability in general. Therefore, when using commodity hardware the burden of providing availability falls to using the existing hardware and software to attempt to handle these errors to provide the highest availability. For example, most contemporary commodity computer systems, while providing good performance, pay little attention to availability issues resulting from memory soft errors. The IA-32 architecture supports only ECC on main memory rather than across other components in the system, requiring system reboot on errors not covered by this ECC. Consequently, commodity software such as the OS, middleware and applications have not dealt with the problem of recovery.

Given that technology trends are causing increased prevalence of these errors, future commodity processors have improved their level of support. Most notably, future IA-64 processors [13], while not recoverable in the general case, do offer some support with certain limitations.

The goal of this paper is to investigate this processor support, the soft error rate that might be seen, and the effect on software running in future commodity systems. Based on this understanding, we illustrate the severity of these errors and outline some simple techniques that could be used to improve availability through software error handling when using future commodity processors.

The rest of the paper is organized as follows. Section 2 presents related work. In Section 3, we present the kind of support found in commodity systems and the trends in soft error rates that they are likely to see. We then describe our investigation into the influence of these errors on operating systems (Section 4) and a sample application (Section 5). In Section 6, we present the lessons learned from this work, and in Section 7, we summarize the paper and propose potential future work.

2 Related Work

Availability in computer systems is determined by hardware and software reliability. Hardware redundancy has traditionally existed only in proprietary servers, with specialized redundantly configured hardware and critical software components, possibly with support for processor pairs [1]. Examples include the IBM S/390 Parallel Sysplex [20] and Tandem NonStop Himalaya [9]. The IBM S/390 Sysplex supports internal processor checking, hot swap execution, redundant shared disk with fault-aware system software for error detection, and fail-over restart. For example, every processor has dual instruction/execution logic that validates every instruction executed. If a problem occurs, the processor automatically retries, or all necessary state is transparently moved to another processor.

Tandem supports redundant fail-over lock-stepped processors with a NonStop kernel and middleware, to provide improved integrity through the software stack. Two lockstep processors run identical copies of the same program and a checker circuit compares their outputs. On a mismatch, a software recovery sequence is initiated. These systems provide full automatic support to mask the effects of data corruption and resource loss.

Cornell's Hypervisor-based fault tolerance system provides a similar software system, providing execution in one primary virtual machine and n-1 backup virtual machines on n processors to provide an n-1 fault-tolerant system [2]. The virtual machines are static, such that once all the n virtual machines are dead, the system must be manually restarted.

Another approach is fault containment and recovery at a "node" granularity. In these systems, each node is supported by a multi-cellular kernel. When one node fails, the others can recover and continue to provide services. Systems of this type include cluster systems [21], and NUMA architectures, such as Hive [6, 25].

Hardware faults are difficult to catch and repeat. Therefore, a lot of research and development is based on emulation and injection of hardware faults [14, 15]. Faults can be injected by specially designed hardware or by software. Hsueh et al. give a survey and comparison of different injection methods [12].

Software reliability has been more difficult to achieve in commodity software even with extensive testing and quality assurance [19]. 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 or taken up software reliability research in general. They typically rely on fail-over solutions, such as Microsoft's Wolfpack [21].

A lot of work has been undertaken in the fault-tolerant community regarding the problems of reliability and recovery in software [3, 11, 16]. These include techniques such as checkpointing [11] and backward error recovery [3]. A lot of this work has been conducted in the context of distributed systems providing fail-over support rather than increasing single system availability, the focus of our work. Rio [8] takes an interesting software-based approach to fault containment for 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.

3 Commodity Soft Error Support

In most processors, a machine-check exception notifies execution of a serious error, but leaves the processor in an undefined state requiring a system reboot due to loss of containment [23, 22]. The IA-64 architecture [13]

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 are completed or tagged as erroneous/incomplete and are restartable on re-issue.*
Uncontained Storage Damage	*Error contained in the storage hierarchy, but storage may contain corruption; safe to reboot.*
Machine-Check Isolated	*The machine-check was isolated by the system and may or may not be recoverable. If no other flag is present, fatal permanent corruption has occurred.*
Hardware Damage	*Non-essential hardware has been damaged and the processor will continue to run at degraded performance.*

Table 1: Description of the IA-64 Machine-Check System State

extends support for soft errors in two ways. Additional hardware detection support has been added to the processor to provide either parity or ECC to the system bus and the three on-chip caches. These provide good coverage of most common errors while limiting cost impact. In addition, the recoverability of machine-checks has been improved providing several types of well-defined error scenarios. This provides the potential for more information to allow software containment of the error.

However, 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 [23]. This leads to difficulties in providing full error detection/handling while remaining cost-effective due to the extra complexity.

To isolate software from implementation dependencies, the IA-64 architecture abstracts machine-check handling. This allows implementations to support different detection and logging approaches while maintaining the same architectural interface [22]. The IA-64 architecture defines five flags to describe the state of the error and the processor in the presence of a machine-check (see Table 1). 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 (see Table 1), 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. But these are less severe than if the machine-check was solely isolated and not contained at all by the processor, resulting in permanent state damage. We believe that in the former cases, there is scope for software recovery for the most probable error cases [18]. However, since containment may have been lost, corrupt state may have propagated through the system hierarchy.

Level	Recovery			
	Detection	Interpretation	Containment	Recovery
Hardware	*cache 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** possible - notify it **else if** localized - terminate it **else** reboot*
Applications	*transaction error*	*error notification & info. from OS*	*errors within single transaction*	*restart operation*

Table 2: Error Detection/Containment/Recovery on IA-64 System

Table 2 outlines the typical kinds of error handling in an IA-64-based system. Current commodity systems do not support software recovery from certain classes of hardware faults, limiting recovery to the firmware and hardware only. This enables simple recovery for some common cases when sufficient information is available at this low level.

However, in the cases of “non-continuable” execution, we believe that processor-only recovery is insufficient and requires memory usage information in the operating system and application levels. In the shaded area one cannot recover because the architectural assertion has been that unless one can exactly identify the effects of the loss of containment. We believe that with some simple software modifications using the semantics of memory usage, a number of these errors need not be fatal.

3.1 Memory Soft Error Rates

Initially, we built an analytical model of a large memory subsystem to understand the trends with which soft errors would lead a Machine Check Abort (MCA) exception or silent data corruption under the IA-64 architecture. For ECC protection this equates to two-bit and multiple-bit errors; for parity, this also includes single-bit errors.

This analytical model used available research on semiconductor physics [27, 28, 30] to allow the parameterization of the memory system at an abstract level, while simplifying its manageability. Events model cosmic-ray strike impacts according to probability distributions fitted to existing research data on supply voltage, cell size, etc.). The effects of these impacts (bit flips) are inserted into a simulated memory subsystem considering DRAM placement and bit interleaving. Errors accumulate due to strikes until they are erased by a model of write traffic to the memory system. In Figures 1 and 2, we present the results of experiments with one such factor, supply voltage, and show how this affects the probability of such

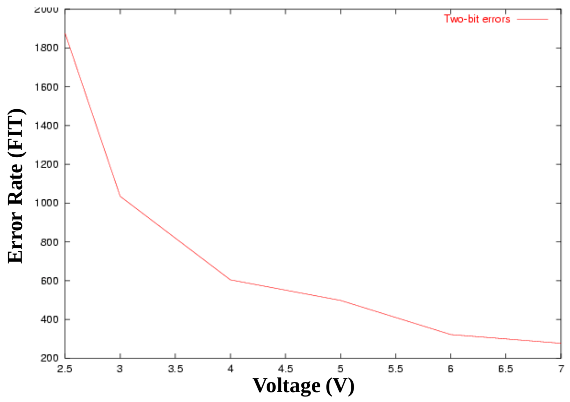

Figure 1: Soft error rates of two-bit errors as voltage is reduced

errors. As can be seen, the number of errors increases significantly as supply voltage decreases. Similar trends occur as the proximity of neighboring bits in a memory word is decreased and as cell size is decreased (not shown).

Based on Moore's law, both cache and DRAM sizes will grow significantly over the next 5 years (to around 2Gbits per DRAM) indicating the possibility for a large increase in error rates due to shrinking cell sizes and supply voltage. However, ongoing research into semiconductors indicates that changes in semiconductor fabrication techniques have so far been used to effectively prevent error rates from increasing dramatically. Baumann et al. have shown that alpha particle cascades in the semiconductor substrate cause soft-errors, leading many manufactures to adopt alternative semiconductor substrates [5]. Similar benefits are expected from Silicon-on-Insulator fabrication technologies [26].

Research has shown that DRAM manufacturers have effectively reduced their soft-error susceptibility by altering the cell construction to minimize soft-error susceptibility [4]. Ziegler's most recent research shows that initial accelerated testing and modelling continues to be truly accurate and shows that SRAM sensitivity to high energy particles has dropped as a result of manufacturing changes [29]. At the same time, hardware protection technologies have also improved, e.g. IBM's ChipKill technology [10].

It is unclear how long semiconductor techniques will be able or cost effective enough to cope with future increases in soft error rates. It is clear that the complex interactions of various semiconductor features and materials lead to a complex model for soft error susceptbility. However, as technology density increases and voltages decrease, underlying semiconductor physics certainly indicate the potential for such errors to increase. Given that these underlying causes seem to indicate continued concern, our work has focused on the

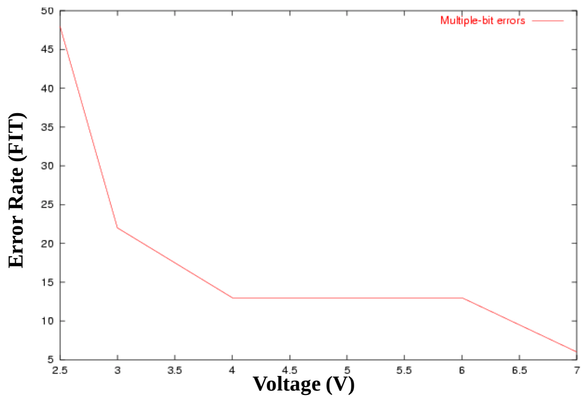

Figure 2: Soft error rates of multiple-bit errors against voltage

affect on software of such errors, if or when they become a prominent availability issue.

4 Influence on Operating Systems

Even with simple error correction techniques, it is still possible for a noticeable number of memory errors to escape that protection. Commodity operating systems can be affected in two ways when an uncorrectable soft error is detected and an exception is raised. First, a memory location will be corrupted and must be handled. Second, because execution may not be restartable and the exception may be delivered imprecisely, the current execution flow may be severely affected. Because of this, systems usually halt or reboot when encountering such exceptions.

To determine the effect of these errors on a typical system, we tried to determine the amount of execution time spent in the kernel using Linux's time accounting. We used generated Web traffic between clients and an Apache Web server for this experiment. Figure 3 shows that a substantial fraction of the Apache server processing time, as high as 70%, is spent executing within the context of the operating system (network and disk I/O). This indicates that recovery at this level could have a significant benefits for these types of applications.

4.1 Experiments on OS Consumption

In order to be affected by errors, the execution must access the memory location containing the error. We call this error consumption. If a memory location contains

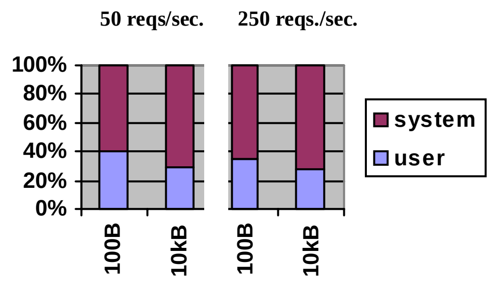

Figure 3: Apache server user/system time distribution

an error and is never accessed or if it is overwritten with new data, this will not result in a fatal exception and therefore availability will not be effected. Given high involvement of the kernel in our sample workload, we first investigated the effect of consuming errors on the operating system, Linux in this case.

Analyzing error consumption and propagation requires the use of an error injector. Given the lack of an appropriate IA-64 environment for our experiments, we performed our investigations on an IA-32 platform, simulating memory errors with watch points. A set of three debug registers allows our injector to detect data read/write accesses, or instruction fetches at any given physical location. The fault injector inserts watch points at periodic intervals, one at a time. Either the OS consumes the error (e.g. hits the watch point) or the injector deletes the watch point after some time-out.

Analyzing error casualty "a posteriori" at the time the OS halts or reboots is difficult. If the system halts and human intervention is required, it will be difficult to gather enough samples. If the system reboots, the error context will probably be lost. Instead, the kernel is modified to capture the relevant state at memory error consumption time. For our investigations, we wanted to record the consumption delay, memory type, access mode, execution mode, interrupted task and program counter.

To obtain memory type information, the OS (Linux in our case) was instrumented such that every byte of main memory can be classified. This is accomplished by modifying the memory allocators (the buddy and Slab systems for Linux) such that they register the requestor's return PC with the memory object.

Once detected by the exception handler, the state is logged, the error is cleared, and the OS is allowed to proceed. This method is non-intrusive and permits the collection of as many as 3000 samples.

For the experiments in this section, we used the following experimental setup:

- A 64 MB 500 MHz Pentium III server platform running the Linux kernel version 2.2. The working set of the application does not fit in the CPU memory cache.
- The server runs an Apache Web server and repetitively recompiles the Linux kernel.
- A single client (600 MHz Pentium III Windows NT) connected over a 10 Mbit Ethernet link. The client runs the WebStone benchmark against the Apache server, simulating 20 users. The network traffic is close to saturation.
- Across 11 experiments, 3125 error injections were performed over a 20 hour period, resulting in 1169 consumptions. To gather enough samples, we artificially injected errors at a much higher rate than a real error rate would be. We believe this does not influence the error distribution or the consumption delays, because the events are independent, and non-intrusive to the kernel (e.g. the kernel is never halted, and the tasks never killed)

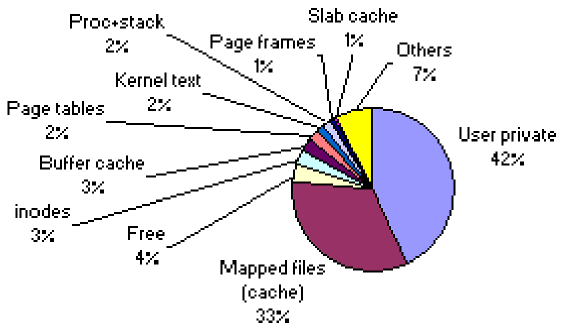

Figure 4: Linux memory objects classification by size

The first set of experiments (first 4 rows of table 3) measure the error consumption distribution over the various memory objects for a fixed period of time (4 hours). The second set characterize the consumption rates, where the elapsed time is proportional to the time-out (x 10).

Injection time-out	Elapsed	Injections	Consumptions
10 sec.	4 hours	1690	464
30 sec.	4 hours	670	278
60 sec.	4 hours	382	197
120 sec.	4 hours	228	132
10 sec.	100 sec.	12	2
30 sec.	5 min.	15	5
1 min.	10 min.	17	9
2 min.	20 min.	18	10
5 min.	50 min.	18	12
10 min.	100 min.	28	20
30 min.	5 hours	47	40
Total	**~24 hours**	**3125**	**1169**

Table 3: OS Error injection experiment sets

Figure 4 depicts the average memory usage while running the benchmark. Up to 200 distinct memory object types co-exist. The top 15 types account for 90% of the allocated memory. 75% of the memory (48 MB) is dedicated to user processes. For these benchmarks, a total of 280 processes are allocated.

Excluding the user private objects, the mapped files and the free memory, 21% of the memory belongs to kernel objects and is presumably highly sensitive to soft errors, potentially leading to kernel failures.

Given our model, the consumption rate is expected to vary with the injection time-out. The larger the time-out is, the greater the consumption rate should be. The consumption rate reaches 55% for a 120 second time-out and 85% for a 30 minute time-out (Figure 5).

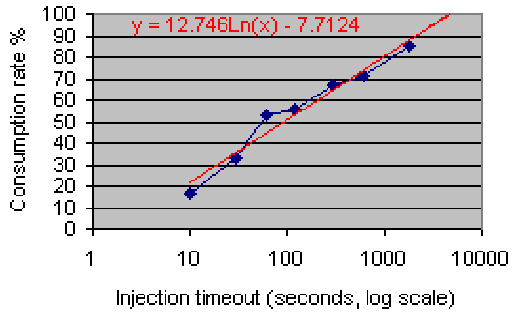

Figure 5: Fault consumption rate vs. injection time-out

The average consumption delay (Figure 6) increases insignificantly for injection time-out values greater than 5 minutes. A closer analysis reveals that 90% of the consumed errors are consumed within a minute.

These results motivates our choice for analyzing the consumption distribution for up to 2 minutes time-out values (Table 3). For a same duration experiment, larger time-outs would not significantly increase the total error distribution.

Error Distribution

Figure 7 illustrates that error injections are distributed across memory object types accordingly to their memory usage. This is no surprise, since the injector uses a uniform random generator to compute the physical addresses. The consumption (detection) distribution is not as close; in particular, the user private memory hit rate is unexpectedly high and the mapped file hit rate is unexpectedly low.

Two factors contribute to this:

- The task/thread creation rate for this workload is high and the private data pages lifetime is short. Every byte of a freshly allocated private page is cleared by the kernel (for security reasons) whether or not it will be used by a task. Our statistics confirm that 90% of the error detection for private data pages occur at page clear time, within kernel space.
- The text (here classified as a mapped file) locality is also high. The server tasks are repetitive. Only a small fraction of the text pages are referenced.

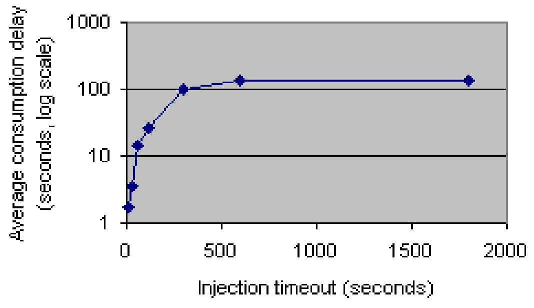

Figure 6: Consumption delay

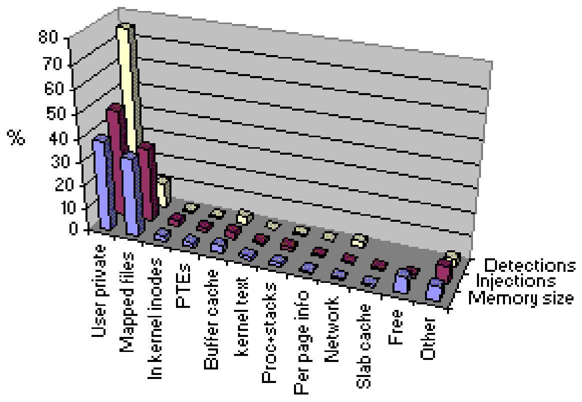

Figure 7: Affected memory by type for the Linux kernel

The error severity (Figure 8) is classified as follows:

- **Overwritten**. The memory is accessed in write mode. If the trapped instruction is re-startable the kernel can proceed. This is interesting because some processes read cache lines first on a write.
- **Signaled**. The memory is accessed in read mode, but it belongs to a user area (as opposed to kernel). In this case, the kernel may just signal the user task and proceed. This applies whether or not the processor was running in kernel or user mode.
- **Fatal**. The memory is accessed in read mode and belongs to the kernel space. In the general case, this is fatal. There may be cases where the error could be ignored or surmounted, but this would require a thorough kernel analysis.

Overall, only 8% of the detected errors are considered fatal to the system in our sample workload. 81% of the errors can simply be ignored provided that the interrupted instruction stream can be re-startable. Assuming that the hardware does not signal overwritten errors, an unmodified Linux system would be affected by 19% of the errors. The kernel already handles appropriately user data errors by signaling the relevant task. It would only panic for 8% of the errors if the processor (as opposed to the OS) is restartable after such a user data error is raised.

4.2 OS Recovery and Containment

Our results show that up to 92% of memory errors we consider should be non-fatal to the operating system.

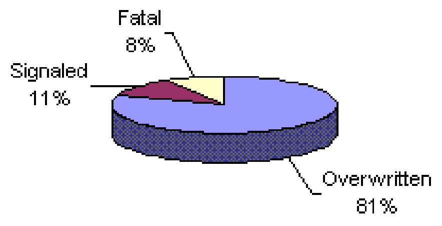

Figure 8: Error severity

This assumes that the operating system has been instrumented to capture relevant information at error detection time and is able to pinpoint the affected memory object type.

The remaining 8% is much more difficult to handle. Looking more closely at the error distribution in Figure 7, one can observe that apart from the non-kernel object types (user private and mapped files) a significant proportion of kernel objects may be altered without affecting the overall kernel availability:

- User page table entries - Some may be re-built; at worst, the task can be signaled.
- Buffer cache - Non-dirty blocks can be recovered from disk; at worst, an I/O error may be raised.
- Kernel stacks - If locks can be successfully unwound, in some circumstances the task may be destroyed.
- Network buffers - The data may be retransmitted, or an I/O error can be raised.
- Kernel text - May be reloaded if the page-in code path is not altered.
- More generally, corruptions within logs or statistical counters should not bring the system down.

We believe that decreasing the 8% fatality by half is possible, but at the cost of more complex modifications to the operating system core.

5 Influence on Application Software

System recoverability is a complex problem that involves participation at each level from the hardware through to the application software. We have seen that the operating system can be extended with simple instrumentation to increase recoverability when it receives a memory error exception.

To avoid application termination the application must consider recovery too, if the operating system finds that the error occurred in application space. The operating system can be extended to signal the application that an error occurred, but recovery for the application is not necessarily straightforward. The data corruption which caused the exception may have affected an important data structure. As we have seen for future commodity processors, if the error exception occurred during user execution, that execution is unlikely to be restartable. Therefore, the application will either need to consider recovery from such exceptions or the system will need to have mechanisms to preserve application state inorder to provide recovery for the application.

In this section, we present two initial investigations into application susceptibility to and recovery from, soft errors. First, we investigate a Java VM running Java applications and its susceptibility and recovery potential. Second, because business costs often mean that changes to applications are not economical, we present initial work into a simple mechanism for trading off performance for availability by preserving application state.

5.1 Influence on Java VM Error Consumption

To determine how the Java Virtual Machine (VM) and its Java applications can respond to soft errors and potentially detect silent data corruption, we performed several investigations instrumenting and adapting the Kaffe VM [7]. This section briefly outlines these results.

At the application level, JVMs and Java applications are of particular interest due to the large garbage collected heaps, the machine abstraction presented, and the integral exception mechanism. By presenting an abstraction between the operating system and the applications, the virtual machine simplifies application-level recovery. Because, the virtual machine has increased details of the application's status and semantics, such as memory usage, there is an improved chance of recovery when the operating system cannot transparently recover for the application.

In a Java VM, the data areas can be divided roughly into two partitions, those allocated statically for the VM and those allocated on the heap for Java objects. In a manner similar to the OS fault injector described in Section 4.1, the byte-code interpreter inserts watch points and waits for the error to be consumed. Once a byte is randomly chosen, its bits are flipped to mimic silent data corruption.

Results show that for the static data region around 5-6% of injected errors cause application errors (crashes or incorrect results), and around 2% of errors are consumed but cause no adverse result. Similar experiments on the Java object heap show that error consumption is much higher there. Between 16% and 63% of errors are consumed causing no error and between 7% and 13% are consumed causing application errors [7].

Most of this consumption comes from the Kaffe Garbage Collector (GC) using a mark and sweep strategy that touches most objects periodically. Although most of the error consumption takes place in the garbage collector, relatively few errors actually cause real problems. The main reason is that the garbage collector only uses certain data in the heap (e.g. object references) on its traversal reducing its susceptibility to the number of actual errors. On average, only 7% of the error consumption in the GC cause application errors. In comparison, 56% of static data error consumption cause application errors.

5.2 Java VM and Application Error Detection

Java VMs will be used in systems (set-top boxes, PDAs) that may not have any error protection mechanisms, such as ECC. Based on our experimental results on error consumption, we implemented a prototype solution for detecting silent data corruption for the Kaffe virtual machine.

The basic idea is that in a Java VM every Java object or array is accessed through a well-known group of byte-code operations, such as getfield and putfield. For each of these byte codes, we add code to do a checksum computation. The heap object management can be modified to store the checksum results [7].

Results show the effectiveness of the detection depends on the nature of the application. If the objects and arrays account for most of the actual errors occurring, the technique is more effective. For example in Javac, where errors in objects and arrays account for nearly 80% of all error occurrences, our prototype implementation can detect up to 38% of all errors. The limitations of this implementation are that neither large objects nor some native code references are considered.

We also compared relative slowdown of the prototype implementation with the original Kaffe implementation. The results are summarized in Table 4.

	Jess	**DB**	**Javac**	**Jack**
Slowdown	57%	43%	47%	32%

Table 4: Slowdown with silent data corruption detection enabled

5.3 Lockstep Processes

One of the biggest problems with application recovery is that execution is potentially not restartable when it receives a MCA exception. This can result in the application losing part of its execution state, making recovery complex. Ideally, the system requires a mechanism to capture the complete application state to allow recovery.

We believe that duplication of software resources (such as memory and execution state can be used to preserve state in a similar fashion to checkpointing. Resource duplication for availability can performed at different levels, such as at the hardware level [9, 20], OS level, virtual machine level [2], process level, or thread level. Each approach trades off recoverability and availability against system software complexity and resource costs. For example, duplication at the OS level offers the best availability, but at the highest complexity and resource overhead (everything must be duplicated). Mainframe machines [9, 20] provide a large amount of hardware support, trading off performance for cost while maintaining good availability.

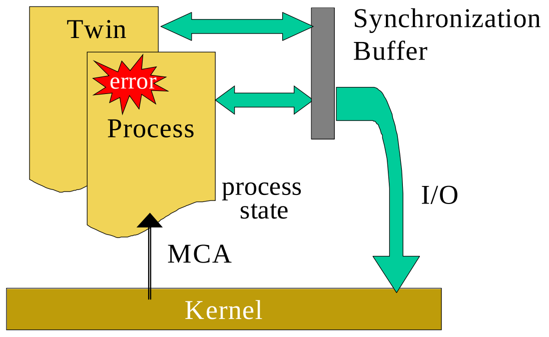

Figure 9: Twin lockstep processes duplicate software state

Our investigations have shown that soft errors need not be fatal to the operating system with the correct instrumentation and recovery. Given that there is a high business cost to adding complexity at the application level, we believe that duplicating resources on a process basis may be able to capture application state with reasonable overhead (see Figure 9).

We propose to duplicate processes and their resources so that there is a background ‘twin’ process executing simultaneously. If either process receives an MCA and dies, then it is possible that a new copy of the uncorrupted process can be duplicated from the twin. The twin processes solution is similar to lockstep processors [24] and fail-over clusters [21], but trades off performance for expensive duplicated hardware while providing appropriate availability for soft errors. In fact, our approach could allow selective hardware duplication in order to regain performance. For example, using SMP machines or increased memory size could compensate for the performance burden without using proprietary hardware solutions.

To achieve duplicate processes with separate resources, modifications are required to the existing system software. Modifications can be done either at the OS kernel level or at the application level, such as by intercepting system calls and replacing or wrapping library functions. Two important modifications are required:

Process Duplication. Process state, including its memory space and state information such as registers, open file descriptors and pending system signals need to be duplicated.

There is a trade-off in how much of the memory space to duplicate against performance. Duplicating the entire memory space including code, data, and stack will consume a lot of memory space and slow down the replication speed, but it can guarantee that the twin processes won’t be hit by the same transient errors. Another alternative is to share code pages, because code pages are never changed and can be reloaded from disk. The third approach is to use

Copy-On-Write semantics, which costs less in extra memory space and has the fastest replication, but is vulnerable if a memory error occurs on a shared page.

Synchronization. To ensure the existing system semantics, the twin processes should present a single process image to rest of the environment. To provide this abstraction, the twin processes must synchronize whenever there is an interaction with the outside world, such as I/O, signals, IPC, etc.

The first process that executes such an input operation retrieves the information from the environment leaving the second to read a buffered copy. The first process that executes a write operation is sent to a buffer, then when the twin writes its output is compared to the buffered data. If the two copies of the data are consistent, the data is written to the outside world, so the error won't be able to propagate outside the process.

Besides guaranteeing the same input and output for the two processes, synchronization also helps avoid extra I/O traffic. However, the synchronization delay is what causes most of the performance penalty. From single processor's performance point of view, this approach buys availability at the expense of performance, because the system workload is doubled and the synchronization introduces additional delay.

Lockstep processes provides a pure software approach for building high availability systems from commodity hardware. The benefits of this approach are low cost and no hardware modification or duplication. It also provides flexibility to allow users to only duplicate critical processes or applications instead of all the processes in the system. The drawback is that we cannot maintain availability when errors are fatal to the operating system. Instead it provides good coverage of application failures due to MCAs.

Using a prototype implementation under Linux, we measured the system performance under two lmbench benchmarks [17]; one memory intensive (lat_mem_rd) and one file I/O intensive (lmdd). Our preliminary results indicate that performance of our system under memory intensive workload approximately halved when its working set fits in physical memory. This is due to the worse-case situation of twice the processor workload with no idle processor resources. Our results indicate a 3-4 times slowdown with I/O intensive workloads, probably due to the use of synchronization per synchronous I/O. While these results aren't conclusive, they indicate that the worse-case overhead of this approach is poor and that intercepting I/O can be highly performance sensitive. On systems with idle/spare resources it may be possible to achieve a better performance/availability trade off.

6 Lessons Learned

From the experimental data and analysis, we believe the following interesting observations can be derived:

- Currently semiconductor techniques are managing to mask the affects of memory soft errors, but this may not continue to be true in the future as technology becomes more dense.
- Future commodity processors have begun attacking these problems, but due to the complexity of such processors the exceptions raised are not likely to be restartable.
- Our investigations show that the affect of soft errors on a modified operating system may be small, since for our sample workload we measured that 92% of memory errors need not be fatal to their execution.
- Operating system execution time can be significant on modern processors and workloads. In the presence of potentially imprecise, non-restartable exceptions, this means that consideration of kernel execution recovery could have a large potential for improved availability.
- For the Kaffe virtual machine and sample Java applications running on it, the memory errors in the object heap have a higher error consumption rate and susceptibility rate than those in the static data area.
- A large portion of heap error consumption is caused by the garbage collector (up to 75% for Jack). But this consumption causes fewer application errors than other sources of consumption (7% vs. 56%).

Given these factors, it seems appropriate to consider software recovery from these errors, because future semiconductor techniques may not continue to be able to mask them.

- By adding simple checksums to the Kaffe VM, normally undetected errors can be detected, increasing error coverage and detection by 30-40%. Using an unoptimized checksum routine, this functionality increased run-time costs by 32-57%.
- By executing twin processes to duplicate application state and resources, our results indicate that it is possible to trade-off performance for availability without the high business cost of requiring application modification.

7 Conclusions and Future Work

Semiconductor techniques are currently helping to mask soft-errors. However, given that the underlying technology trends will continue to increase the susceptibility of semiconductors to soft errors, it seems wise to consider their affect on software when hardware cannot mask them sufficiently. Future commodity processors have started considering soft-errors as more of a problem. Given this improved support, we have looked into the affect of soft errors on software if or when soft errors become a prominent problem.

Our investigations into the susceptibility of both the Linux kernel and Kaffe VM are revealing. Despite the potential data corruption that can occur, with simple instrumentation of the Linux kernel, only 8% of memory errors actually need to be fatal for our sample workload. While for the VM, a large number of errors are consumed by the garbage collector in the object heap.

Given that application modification presents a high business cost, we then investigated application recoverability from such errors. When considering silent data corruption, adding simple unoptimized checksumming to a Java VM can effectively detect 30-40% of errors at a run-time cost of between 32-57%. For more traditional applications, we also proposed that duplication of process state and resources can effectively trade-off performance for availability, while potentially using idle resources. Initial results seem to indicate that doing so improves application availability at a worst-case runtime slowdown of between 2-4 times depending on the workload for initial simple unoptimized experiments.

In the future, we would like to continue to investigate into the underlying soft error trends and the ability of hardware techniques to mask them. We would also like to perform more detailed experiments on the effects of caches to mark error consumption rates from main memory using an IA-64 simulator we are extending. Finally, we would like to extend and optimize the proposed techniques for application recovery, in order to better understand their affects and applicability.

Acknowledgements

We are indebted to John Wilkes, Ira Greenberg, Duane Dutton, George Candea, and Armando Fox for reviewing the paper and Valentin Anders, Dan Osecky, Mike Traynor, Peter Markstein, Don Wiess, and Richard Adkisson for contributing to the project. Their contributions significantly improved the project and this paper's content and presentation.

Bibliography

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

[2] Bressoud, T and Schneider F, “Hypervisor-based Fault Tolerance”, *Proc. of 15th ACM SOSP*, pp 1-11, Dec 1995.

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

[4] Baumann, R. , “Soft Error Characterization and Modeling Methodologies at TI: Past, Present and Future", 4th Annual Topical Research Conf. on Reliability, Oct. 2000.

[5] Baumann, R., et al., “Neutron-Induced Boron Fission as a Major Source of Soft Errors in Deep Submicron SRAM Devices”, 38th IEEE Intl Reliability Physics Symp., 2000.

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

[7] Chen, D., et al. “JVM Susceptibility to Memory Errors”, Submitted for the USENIX Java Virtual Machine Research and Technology Symposium, April 2001.

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

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

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

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

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

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

[14] Kanawati, G., et al., “FERRARI: A Flexible Software-based Fault and Error Injection System", IEEE Transactions on Computers, vol 44, no 2, pp 248-260, Feb. 1995.

[15] 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.

[16] Lowell, D. E. , Chen, P., “Exploring Failure Transparency and the Limits of Generic Recovery”, USENIX Operating System Design and Implementation, Oct. 2000.

[17] McVoy, L., Staelin, C., “lmbench: Portable tools for performance analysis”, Usenix Technical Conference, 1996.

[18] Milojicic, D., et al., “Increasing Relevance of memory Hardware Errors – A Case for Recoverable Programming Models”, ACM SIGOPS European Workshop, Sept. 2000

[19] Murphy, B., et al. “Windows 2000 Dependability”, IEEE Intl. Conf. on Dependable Sys. and Networks, June 2000.

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

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

[22] Quach, N., “High Availability and Reliability in the Itanium Processor”, IEEE Micro, vol.20, no.5, pp 61-69.

[23] Sharangpani, H.,Arora, H., “Itanium Processor Microarchitecture”, IEEE Micro, vol.20, no.5, pp 24-43.

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

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

[26] Tosaka, Y., "Soft Error Modeling and Simulation for SOI Circuits", 4th Annual Topical Research Conference on Reliability, October 2000.

[27] 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.

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

[29] Ziegler, J. F., “Historical Trends in the Soft Error Rate of SRAMs and DRAMs”, 4th Annual Topical Research Conference on Reliability, October 2000.

[30] Zoutendyk, J.A., et al., “Characterization of Multiple-bit Errors From Single-ion Tracks in Integrated Circuits”, IEEE Trans. on Nuclear Science vol 36, no 6, Dec. 1989.