Susceptibility of Commodity Systems and Software to Memory Soft Errors

Alan Messer, Philippe Bernadat, Guangrui Fu, Deqing Chen, Zoran Dimitrijevic, David Lie, Durga Devi Mannaru, Alma Riska, Dejan Milojicic

- *Alan Messer is with Samsung Electronic's Corporate Technology Operations, 75, W. Plumeria Dr., San Jose, CA 95134. E-mail: alan_messer@yahoo.com*
- *Philippe Bernadat is with Hewlett Packard, 5 av. Raymond Chanas, 38320 Eybens - France. E-mail: philippe_bernadat@hp.com*
- *Guangrui Fu is with the Mobile Internet Laboratory, DoCoMo Communications Lab. USA, Inc. 181 Metro Drive, Suite 300, San Jose, CA 95110. E-mail: fu@dcl.docomo-usa.com*
- *Deqing Chen is with AskJeeves at 1551 South Washington Avenue Suite 400 Piscataway, NJ 08854 Email: dchen@askjeeves.com*
- *Zoran Dimitrijevic is with Google at the University of California at Santa Barbara, Comp. Science Dept, Santa Barbara, CA 93106. E-mail: zoran@cs.ucsb.edu*
- *David Lie is with the University of Toronto, 10 King's College Road, Toronto, ON M5S 3G4 Canada email: lie@eecg.toronto.edu*
- *Durga Devi Mannaru is with IBM, Reserch Triangle Park, NC 27709. Email: durgavellanki@yahoo.com*
- *Alma Riska is with Seagate Research, 1251 Waterfront Place Pittsburgh, PA 15222 E-mail: alma.riska@seagate.com*
- *Dejan Milojicic is with Hewlett Packard Laboratories, 1501 Page Mill Rd., MS 1183, Palo Alto, CA 94304. E-mail: dejan_milojicic@hp.com*

Abstract *— It is widely understood that most system downtime is accounted for by programming errors and administration time. However, a growing body of work has indicated an increasing cause of downtime may stem from transient errors in computer system hardware due to external factors, such as cosmic rays. This work indicates that moving to denser semiconductor technologies at lower voltages has the potential to increase these transient errors. In this paper, we investigate the susceptibility of commodity operating systems and applications on commodity PC processors to these soft-errors, and we introduce ideas regarding the improved recovery from these transient errors in software.*

Our results indicate that for the Linux kernel and a Java virtual machine running sample workloads many errors are not activated mostly due to overwriting. In addition, given current and upcoming microprocessor support, our results indicate that those errors activated, which would normally lead to system reboot, need not be fatal to the system if software knowledge is used for simple software recovery. Together, they indicate the benefits of simple memory soft error recovery handling in commodity processors and software.

Index Terms — soft errors, memory errors, commodity, operating system, Java, recovery.

1 INTRODUCTION

Commodity systems such as PC systems based on the Intel IA-32 architecture running the Windows and Linux operating systems account for the bulk of computer system sales. As computers become more ubiquitous, demand for better performance and higher availability increases in cost-effective commodity systems. However because of price pressures, current commodity systems have focused on price/performance issues, giving availability less attention. It is a common belief that software errors and administration time are, and will continue to be, the most probable cause of the loss of availability [13]. While such failures are clearly commonplace, especially in desktop environments, research has shown that certain transient hardware errors, particularly in memories, are also becoming increasingly probable as technology improves [5, 32]. Since such transient errors require system reboots that can take several tens of minutes or more on large systems, these errors can affect availability considerably.

Hardware errors can be classified as hard errors (faults) or transient (soft) errors. Hard errors are those that require replacement (or otherwise relinquished use) of a component. These typically happen as a consequence of physical damage to a component, e.g., by damage to connectors. Transient (soft) errors are those that result in an invalid state in the hardware that is correctable. For example, data stored at a memory location may become corrupt, but overwriting it will remove the invalid state. Such errors may lie dormant for a significant time, since they are only detected by the system when the processor directly uses the erroneous hardware or corrupt memory location. When an error is touched by the hardware it is referred at as *Error Activation*, while activated errors which go undetected by the hardware are called *Silent Data Corruption*. Such hardware errors have been considered by mainframe technology for years using expensive proprietary hardware and software for detection and recovery [2, 26]. However, in the field of commodity systems, it has not been cost-effective to provide full hardware detection and redundancy for recovery support to mask all errors.

Results over the last 20 years have shown that soft errors due to cosmic rays and substrate alpha particles can cause semiconductor transient errors in memory hardware [32]. For example, it has been reported that a 1GB memory system based on today's 64Mbit DRAMs still has a potential combined error rate of 3435 FIT (Failures In Time -- failures in 10^9 hours) when using Single Error Correct-Double Error Detect (SEC-DED) ECC [10]. Of these errors soft-errors account for 30 times (96.7%) the errors when compared to hard errors. This is equivalent to around 300 reboots resulting from soft errors on 10000 machines in 1 year, if all the errors are activated and cause reboots. 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 reduced supply voltage.

The increasing prevalence of soft errors and their recovery has received some attention in current- and next-generation commodity processor architectures. For example, Intel's IA-32 architecture has various levels of support across processor implementations for error detection

and correction on certain buses and caches. In addition, the new Intel/HP IA-64 architecture contains increased support for the detection of, correction of, and reporting for software recovery of soft errors at the processor level.

At current error rates and memory sizes, processor-only recovery support may be sufficient. However, given the potential for a high soft error rate, we would like to understand the effect of those errors that are not masked by hardware support. In doing so, this paper aims to determine how frequently soft errors are activated by commodity PC software using commodity operating systems on commodity processors. In addition, given improved error support, we aim to determine what influence will soft errors have on commodity software workloads and what are the possibilities for recovery from those soft errors.

The rest of the paper is organized as follows. In Section 2, we present our approach to investigating error susceptibility and recovery on commodity processors. We then describe our investigation into the influence of these errors on a commodity operating system (Section 3) and a sample application platform (Section 4). In section 5, based on our understanding we analyze the probable effect of soft errors on commodity systems with and without simple, improved error handling. Section 6 presents work related to this paper. In Section 7, we present the lessons learned from this work, and in Section 8, we conclude the paper and propose potential future work.

2 APPROACH

In a commodity system based on the IA-32 or IA-64 architectures memory soft errors predominantly occur in both the cache and main memories of a system. Depending on memory size, technology sensitivity to soft errors, and price pressures, PC systems usually support at least parity detection on main memory and ECC for larger caches. Single bit errors can be effectively masked with error correction support, however as technology feature sizes shrink and voltage levels drop the probability of multiple bit errors increases. Depending on the protection used, price, and technology types the number of errors masked by this protection will vary. But

ultimately, a number of soft errors are not masked and may be activated by the software resulting in either detected errors or silent data corruption.

Determining the affect of soft-errors on a commodity system is a difficult task due to their relative infrequency and limited post-mortem information. Past work has reported the effect on availability of these errors from data acquired from in the field failures, however this information typically comes from mainframe machines where some level of post-mortem information is available. Our approach to this problem is to investigate the activation rate of emulated soft errors (rather than total availability) for off-the-shelf commodity systems and software. Based on this approach, we can gain an understanding of how soft errors are activated and with some minor kernel modifications we can classify those errors by their usage and likely effect on those commodity softwares and processors.

Based on this information, we would like to understand the processor and system status at which the memory error is activated. This is important because the error's severity on the software and thus the ability of the system software to recover from the error is directly affected by what software activates the error. We believe that the information derived from these experiments will increase the understanding of whether and now to make software execution platforms more robust to soft errors.

Soft-errors based on sources such as cosmic rays occur uniformly in memory, although the size of the error may vary due to multi-bit impacts. To mimick this occurance our approach is to insert emulated memory soft-errors uniformly distributed throughout memory and then determine their activation rate. This approach does not stress-test particular memory regions or system processing for errors to understand fully the consequences of the error on availability, other work covers this approach well. Instead, we are able to determine the effect of the uniformly inserted errors and their effect on the *total* system software (OS, application software, etc.).

In this paper we evaluate two common PC software platforms; the operating system and Java virtual machine platforms. Operating systems have the most scope for performing recovery without affecting application code because it is first to receive an error from the hardware and is in total control of the hardware. However, some errors cannot be handled by the operating system. In these cases, the Java virtual machine's abstraction may enable additional application recovery.

2.1 Existing Commodity Error Handling

The effect of a soft error on software execution depends predominantly on the processor's support for error masking, handling and reporting. In commodity processors, such as IA-32 and IA-64 architecture processors, a detected memory soft error causes the system to raise a Machine Check Architecture (MCA) exception to notify the operating system of a serious error. However, because the hardware has seen an un-correctable error and because of commodity price/performance pressures, this exception usually leaves the processor in an undefined state requiring a system reboot due to loss of containment of the error's effects at the hardware level.

A complete overview of the IA-32 or IA-64 MCA is beyond the scope of this paper. In general for IA-32 processors, while the exception leaves the processor in an undefined state, the status of the processor concerning the error is reported in a set of processor registers [17]. The IA-64 architecture extends support for soft errors in two ways [16, 28]. First, additional hardware detection is supported for processor implementation, such as providing parity or ECC protection to the system bus and the three on-chip caches. These provide good coverage of most common errors while limiting cost. Second, the recoverability of machine-check exception handling has been improved by providing several types of well-defined error scenarios. This provides more information for potential software containment of the error.

These processors report many types of errors as part of the MCA. For memory errors, three pieces of information are of interest: where the error activation occurred (both the current instruction point and the erroneous memory location), what action caused the error (e.g. read or write access) and whether it occurred in main memory or in cache. Based on these error cases,

several opportunities exist for simple error recovery. For example, a write access to a soft error in main memory need not cause a fatal exception, since the error is being overwritten rather than read. However, many architectures cause a cache read on a write, in order to read the data into cache for update. In doing so, an error is consumed and signalled needlessly. Alternatively, an error may occur in user memory during application processing. Since the error is outside the kernel its integrity has not been affected, allowing an enhanced kernel to simply kill the user process and continue.

Based on these possibilities, our approach is to understand where errors occur, and what operation was being performed, so that knowledge can be used to determine whether some form of error recovery processing could contain the error. This may lead to enhanced error recovery in the kernel or an application platform to contain the data corruption due to the error.

3 INFLUENCE OF SOFT ERRORS ON A COMMODITY OS

To understand how activated errors affect the operating system on commodity processors we need to measure and characterize soft errors with the following information:

- **Was the processor reading or writing memory?** Depending on the processor implementation, errors while writing versus reading memory may be ignored since the content is overwritten.
- **Whether the processor is executing kernel or user code?** If the error activation occurs while the processor is in user mode, there may be opportunity for terminating the current thread or task and switching to another one, thus avoiding bringing down the system
- **Whether the affected memory is in user or kernel space?** If the processor is executing kernel code, but the accessed memory is in user space, such as when reading user data in a write system call, or when reading system call arguments, it may be possible to modify the operating system to send a signal to the corresponding user task and interrupt the system call. Most operating systems are already prepared to handle an invalid memory access for such transfers.

- **What is the memory object type and what's the OS state,** if the processor is executing kernel code? Depending of the type of kernel memory and how it is accessed (read/write) the recoverability of the error can be determined from the state saved from the error insertion.

3.1 Kernel instrumentation and methodology

Simply injecting errors at random memory locations is an easy task. Determining if the error is activated and what the effects are is more difficult. The error may be activated without any visible effect, even though its future consequences can be severe. For example, an activated error may cause a reboot requiring a file system integrity check taking many minutes or hours for large systems. In the extreme case where the kernel panics or halts, analyzing error casualty "a posteriori" is complex. Restarting the system during this analysis is often a long process that may require some human intervention.

For soft-error investigation where a number of samples are required, we feel kernel panic analysis process is too slow and difficult to gather enough samples. Instead, we chose to adopt a non-intrusive approach of error activations that would give us enough information to categorize the memories usage and use this with human analysis to determine the general affect on the underlying software. To enable this hand analysis, we must modify the kernel to capture the relevant state at memory error activation time in a non-destructive and non-intrusive fashion. For each activated error our instrumentation records the activation delay and some error context, including: affected memory type at injection time, the affected memory type at activation time (since memory may be re-allocated), the access mode (read or write), the execution mode (kernel or user), the interrupted task's ID, and the program counter. This information is used under human analysis to understand how potentially fatal each error would have been to the OS.

Note, because we are using a software injection approach, we are only attempting to determine the software affect of activated errors on the system, rather than measure total availability of the systemwere. In addition, using this approach we are unable to measure activation due to no process memory activity such as DMA transfers or virtual memory lookups.

3.2 Error Injections

We performed our investigations on an IA-32 platform using watch points to simulate memory errors. Similar to break points for instructions, watch points are a means to detect any type of memory access to a given virtual memory location. A set of three debug registers in the processor allows the detection of data read/write accesses, or instruction fetches at any given memory location. Given that the watch point mechanism is virtual address driven, one limitation is that physical memory that isn't accessed through a virtual mapping is ignored. In particular simulated errors in page tables (PTE) can not be detected during page translations, nor can errors in I/O buffers during DMA operations. Most platforms use a TLB cache to minimize PTE lookups. I/O buffers usually contain user data and errors activated in such memory area are not considered fatal. This limitation shouldn't significantly impact the OS susceptibility.

The fault injector is organized as two components, a user mode program and some newly written kernel code. The user mode program is executed concurrently with the workload. The **user mode program** randomly selects physical addresses where to inject an error. Then it interfaces to our kernel injection component through a */proc* virtual file system interface to setup a watch point, called a */proc/mfi*. This interface is a convenient way to communicate the kernel without adding new system calls. If the error is activated, the kernel component returns the error context, through the same interface. It will delete the watch point once the error is activated or if some configurable time-out expires. Finally it computes the various statistics required for our analysis. The watch point facility does not allow more than one virtual address to be monitored simultaneously. Therefore, we setup a time-out to detect that the error has not been activated, and inject a new error at another random location.

The **kernel component** searches which virtual address (kernel or user) maps into the physical address provided by the user program. (We never detected multiple memory mappings while running our experiments). This virtual address is process space dependent and must be searched for each distinct task. This reverse PTE lookup is fairly expensive and can not be performed

systematically for all tasks or at each context switch. Instead it is performed when a task is first scheduled after the error was injected, then the matching virtual address is cached in a task specific data structure. Additionally the kernel intercepts all virtual mapping requests in case the physical page where the error was injected is about to be mapped. Three watch points are initialized to detect read, write, and instruction fetch on this virtual address. If a watch point exception is raised the kernel gathers the error context and returns it to the user program. See Figure 1 for a overview of this non-intrusive soft-error emulation process.

To collect the largest sample set, a new error is injected as soon as the previous one is activated or when the time-out expires, therefore injections are not strictly periodic. Overall we injected one error every 50 seconds, with minimal impact on the workload applications.

3.3 Memory objects classification

To enable human analysis of the activation point and consequences of the recorded exceptions, we need a means of classifying the usage of that memory as well as determining where is it is used. In order to classify memory usage, we chose to break down memory usage into types of memory, based on the point at which it is allocated. This allows us to determine whether the memory is used for file systems buffering, stack space, etc. and thus categorize them for further analysis. To obtain this memory type information, the Linux OS was instrumented such that every byte of main memory be classified. This is accomplished by modifying the memory allocators (the buddy and Slab memory systems [3]) so that they register the requestor's return PC within the memory object. Each distinct PC is mapped to a distinct memory type. Given any kernel virtual address, the operating system's memory type may be retrieved either from the page descriptor or the Slab header. The memory object type is determined both at injection and activation time, since the physical memory may be re-allocated in the meanwhile. This allows the program function allocating the memory (either the kernel or application) to be recorded as well as the function activating the memory and then stored for analysis on activation.

3.4 Error severity classification

We categorize each error into one of three simple error severity classifications based on whether the error was in kernel or user memory and whether the access was a read or a write:

- **Overwritten**. The memory is accessed in write mode. On many platforms, write access to an erroneous location is not detected and can be ignored. However, on some platforms, a write access may be preceded by a read when the cache loads a line, causing the processor to detect the error before it can be overwritten.
- **User Signal-able**. The memory is accessed in read mode, but it belongs to a user (as opposed to kernel) area. This applies whether or not the processor was running in kernel or user mode. In these cases the state of a particular user program has become corrupt, but the processor may allow the kernel to continue operating. As a result, the kernel can signal the user task and proceed with another one, or to interrupt the system call. Depending on the processor, some memory error exceptions indicate that processor error containment has been lost, these are not considered to be user signal-able.
- **Kernel Fatal**. The memory is accessed in read mode and the location belongs to the kernel space. In general, this is fatal, because kernel state is corrupt. There may be cases where the error could be ignored or surmounted, but this would require a more thorough kernel analysis.

3.5 Experimental Setup

For our experimentation, we used a 500 MHz Pentium III PC with 192 MB of memory running the Linux kernel version 2.2. We used two workloads:

- Workload 1: The host 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 runs the WebStone benchmark against the Apache server, simulating 20 users. The network traffic is close to saturation. For this workload the real memory was artificially reduced to 64 MB so that the memory working set be slightly larger than real memory to induce some swap activity.

• Workload 2: The host runs mySQL server 4.0.12 and iteratively executes the associated CPU bounded benchmark suite. We didn't limit memory as we did for workload 1, leaving the Linux system with 10% of memory reported free. Another different is that there is no network traffic in this workload. One characteristic of this version of MySQL is its extremely efficient memory cache, reducing the amount of I/O operations.

To collect enough performance data, we injected errors at a much higher rate than found in a real system. Unlike a real environment where errors persist in memory as long as it is not activated, our simulator cannot monitor more than one error at a time. Since the error may never be activated we need to impose a time-out. So the injection rate is not a fixed parameter, it is not uniform, and its value can only be measured a posteriori. The time-out value is the only configurable parameter.

We performed two sets of experiments. With the first one, performed with workload 1 (7 first rows of Table 1), we study how the activation and activation delay evolve as a function of the error injection time-out. In this experiment, the injection time-out varies from 10 seconds to 30 minutes. The second set of experiments, performed on both workloads (5 last rows of Table 1), allows us to characterize the error severity, and to observe the influence of the workload. Here, the errors were injected at higher rates and the experiments last longer.

Overall across 12 distinct experiments, 8624 error injections were performed over a 110-hour period, resulting in 1774 activations. In the next sections we analyze these experiments with respect to: the activation rates, activations delays, the influence of the time-out value and the workload, and the error severity.

3.6 Activation Rate and Activation Delay

Because of our need to use a time-out on memory injections, our first analysis was to determine how activation delay varies with the injection time-out. Absolute activation rates are of less interest, but logic indicates that the larger the time-out is, the greater the activation rate. With no time-out (e.g. an infinite time-out value) the activation rate should be close to 100%. Only close,

since some memory areas may never be accessed, such as unused kernel text code (initialization, unused components). Figure 2 shows that the activation rate reaches 55% for a 120 second time-out and 85% for a 30 minute time-out. This high activation rate is the result of both a memory intensive workload (as little as 4% free memory) and of minimal memory fragmentation resulting from the use of the slab allocator in the Linux kernel.

Figure 3 reports the activation delay (elapsed time between the injection and the activation). With a 2 minute time-out value, 90% of the activated errors are activated within a minute. The average activation delay increases insignificantly for injection time-out values greater than 5 minutes.

3.7 Memory Distribution and Activation Rate

Related to this analysis, we also would like to understand how activated errors are distributed by memory type to determine their recoverability. Figure 4 depicts the average memory usage distribution while running workload 1. We have categorized memory into 200 distinct memory object types. Amongst the allocated memory (96% of total real memory), the top 9 types account for 93%. 75% of the memory (48 MB) is dedicated to user processes. For this workload, a total of 280 processes are allocated. Excluding the user private objects (private as opposed to shared with other tasks), the mapped files and the free memory, 21% of the memory belongs to kernel objects.

In Figure 5 we show 3 distinct injection and activation distributions for 4 injection time-out value experiments. The first distribution is the percentage of overall memory used by each memory type. The second is the injection distribution over the various memory types and the third one is the distribution at the activation time. The figures show that error injections are distributed across memory object types according to their memory usage. This is no surprise, since the injector uses a uniform random generator to compute the physical addresses. The activation distribution is not such a close fit; 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 page lifetime is short. Every byte of a freshly allocated private page is cleared by the kernel whether or not it will be entirely used by a task.
- 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. This leads to low text hit rate.

Another important observation is that the injection time-out value has little influence over the distributions or activation rate per memory type at these time-out levels, since while the absolute activation rate varies with timeout, our activation rate per memory type is similar across timeout values. This validates our experimental non-intrusive time-out based approach mimics the activitations a real kernel would see from memory soft errors.

3.8 Error severity

Figure 6 shows the overall result of our classification across the 2 workloads. Overall, only 10% of the activated errors are considered fatal to the system for our sample workload. Most of this reduction is caused by 74% of the error activations simply overwriting an existing error. Leaving 16% of the errors which have potential for signalling the application before termination. This interesting result follows from the common operation of many software components such as stacks and virtual memory pages, both of which are generally written (a intermediate result or zeroed page respectively) before they are read.

First, let us assume that write errors are silently ignored by the hardware or that if signalled the error can be continued and the OS may be restarted. Then, we could ignore 74% of the activated errors. Given this assumption, an unmodified Linux system would be affected by only 26% of the activated errors. Second, the kernel already has support for appropriately handling existing user data errors (e.g., segmentation violations) by signaling the relevant task. The same mechanism would allow to signal applications when a read error activation occurs in user space.

With this error handling support and restartable processor error exceptions, the system would only need to panic for 10% of the errors.

Table 3 provides the activation rate and error severity distribution for the two workloads and on a per time-out basis. Our first observation is that time-out value has little influence on the severity. The activation rate is higher for the first workload since we artificially reduced the real memory so that the entire memory range is used. Despite a significant variation for the overwritten and user signal-able errors, the fatal error proportions are close: 8% for workload 1 and 13% for workload 2.

3.9 Potential OS Recovery and Containment

Our results show that up to 90% of memory errors can be considered as non-fatal to the operating system. This assumes that the operating system has been instrumented to capture relevant information at error activation time and is able to pinpoint the affected memory object type. This may allow it to discard write mode errors and signal user processes when errors occur in user memory space. While this does impose extra kernel development, we were able to apply this modification to the Linux kernel in about one man-month. Given the Linux kernel size, this seems a reasonably small implementation cost for the potential benefit.

The remaining 10% is much more difficult to handle. Looking more closely at the error distribution in Figure 5, we observe that apart from the non-kernel object types (user private and mapped files) a number 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; or an I/O error may be raised.
- Kernel stacks - If locks can be 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.

However, this decrease in fatality will come at a higher cost due to more complex modifications to the operating system core. Table 2 provides the list of kernel code routines affected by fatal read error activations with workload 1. The most frequently affected are

- *ide_output_data* - Used while writing to disk. This is mostly a consequence of the compile tasks.
- *statm_pgd_range* - Collects the memory usage statistics available through the /prof virtual file system. We were running the top program simultaneously to observe the memory usage.
- *xirc2ps_interrupt* - Processes network controller interrupts. The network is close to saturation with the Webstone benchmark.
- *filemap_nopage* - When a user task maps a shared page as private, the page must be copied. This is usually the case for initialized data sections of a program. This memory is not considered as user private. One option would consist of signaling all tasks still mapping this page and discarding it for each. The page would then be reloaded from disk when the program is next scheduled.
- *do_fork* - The kernel duplicates a significant amount of kernel data. Workload 1 induces a significant amount of task creations, since it continuously compiles small files.

4 INFLUENCE OF SOFT ERRORS ON APPLICATION SOFTWARE

System recovery is a complex problem that involves participation from the hardware through to the application software. We have seen that the operating system could be extended with simple instrumentation to increase recoverability when it receives a memory error exception. However, if the operating system determines that the error occurred in application space, in order to avoid termination, the application must consider recovery as well. The operating system could be extended to signal the application that an error occurred, but recovery for the application is not necessarily straightforward. The data corruption that caused the exception may have affected an important data structure. In addition, on commodity processors, the execution activating the error is often not continue-able after the exception. Therefore, the application will either need to

consider recovery from such exceptions or the system will need to have mechanisms to preserve application state in order to provide recovery for the application.

In this section, we present initial investigations into application susceptibility to soft errors. At the application level, Java Virtual machines (JVM) and Java applications are of particular interest to us due to the large garbage collected heaps, the machine abstraction presented, and the integrated exception mechanism. By presenting an abstraction between the operating system and the applications, the virtual machine simplifies application-level recovery by using increased knowledge of the application's status and semantics, such as whether the error is in static or heap memory.

4.1 Influence on a Java VM

To determine how the JVM and its Java applications can respond to soft errors and potentially detect silent data corruption, we performed several investigations instrumenting and adapting the open-source Kaffe VM. This allowed us to examine its memory usage, to instrument it for fault injection experiments, and to extend it to detect silent data corruption. It is also a mature system, it has reasonable performance, and it is widely used. For our experiments, we used an IA-32 RedHat Linux 6.2 platform, running Kaffe 1.0.5 in the "interpreter mode."

We instrumented the Kaffe virtual machine to inject memory errors into the data memory area and to record the memory status. In a manner similar to the OS fault injector described in Section 3.1, the interpreter loop is instrumented so that after a certain number of byte codes have been executed, the loop calls our error injection procedure to inject a memory error.

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 each test set, errors are injected into one of these data areas. When the error is activated, we determine what data area the error has hit, what type of object it is in, and we also inspect the VM status to see whether it is activated by the garbage collector. Kaffe uses the mark and sweep algorithm, which makes this inspection fairly easy because when the GC runs all of the other user threads are stopped.

To investigate the effects on some sample applications on top of the JVM, we chose four benchmark applications extracted from the SPEC JVM98 benchmark suites using the medium data configuration – ten percent [29]. To represent a range of memory uses we chose a Java expert system (SpecJVM98 name: _202_jess), a Java database (_209_db), a Java compiler (_213_javac), and a Java parser generator (_228_jack).

For our experiments, we injected 1,000 memory errors for the four benchmarks in both static and dynamic memory areas of the JVM. Figures 8 and 9 show the results of our initial investigations for the static and dynamic memory areas, respectively. These 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 activated but cause no adverse result. However, the Java object heap shows a much higher error activation rate between 16% and 63% when causing no error and between 7% and 13% when causing application errors.

The most interesting results show that there is a significant difference in the error susceptibility of the two data areas, especially that there is a large difference in the number of errors that are injected and not activated. As can be seen from figure 9, this seems to be because the Garbage Collector (GC) activates a large number of those normally latent errors. This stems from Kaffe's mark and sweep garbage collector strategy that touches most objects periodically causing latent errors to be uncovered. This may cvause an error on the GC thread. However, the GC is designed to be easily restarted to relieve load when memory is tight.

Interestingly, however, although most of the error activation takes place in the garbage collector, relatively few errors actually cause real problems (crashing the JVM, for example). We believe the main reason for this 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. By comparison, 56% of static data error activation cause application errors, whereas, only 7% of the error activation in the GC cause application errors.

4.2 Potential JVM Recovery and Containment

These results indicate that similarly to the operating system a large number of errors are latent and never detected while executing. However, it seems that applications that exhibit behavior similar to a mark and sweep garbage collector are much more susceptible to uncovering those normally latent errors. For example, an in-memory database application transverses large amount of memory in order to produce each query result. However, it is unclear whether the different search patterns of other garbage collectors are similarly affected.

Results also indicate that with a little extra application knowledge a large number of those detected errors need not be fatal. For example, the garbage collector could be modified to tolerate machine-check abort exceptions that may occur during a heap sweep. Or in the case of silent data corruption, when errors are not detected by hardware, the garbage collector could check the validity of object references before use. In fact, most garbage collectors already check object reference validity before proceeding as part of their sweep to determine which data is an object reference. This probably accounts for some of the garbage collector's existing tolerance to errors. Also, given the level of abstraction offered by the JVM, there may be opportunities for other forms of error handling, such as improved exception handling and object checksums to detect silent data corruption. Initial investigations into these ideas show several interesting approaches [7].

5 Analysis for Commodity Systems

At the beginning of this paper, we noted that it has been reported that a 1GB memory system based on today's 64Mbit DRAMs still has a potential combined unmasked error rate of 3435 FIT[1] when using ECC [10]. Given our investigation, it is interesting to consider: (1) given our activation rate evidence how many failures in time would lead to a reboot? (2) given our results for the recoverability of errors, how can this error rate be improved?

For simplicity lets assume that all activated errors are detected, which is quite common for an ECC based system. Sections 4.2 and 5.1 report similar worst case error activation rates in the

range of 11%-37%, an average of 20%. Taking the worse case activation rate and our 1GB memory system, our experimental result would indicate that the software would only need to reboot on a (3435 x 0.20) = 687 FIT[1] activated error rate.

Our analysis of the activated error reports indicates that given a small amount of modification to each piece of software not all errors need be fatal to the OS (reboot) or application (restart). We would like to convert our understanding of the various software memory susceptibilities to errors into an approximate visible error rate. The approximate combined error rate for errors that are not masked by the hardware or modified OS can be determined using the following formula:

$$\begin{aligned}
&\mathit{FatalErrorRate} = (\mathit{VHE} \times \mathit{AR}) \times (\mathit{KMS} + \mathit{AMS} + \mathit{OR}) \\
&\mathit{VHE} = \mathit{VisibleHardwareErrorRate} \\
&\mathit{AR} = \mathit{ActivationRate} \\
&\mathit{KMS} = \mathit{KernelErrorSusceptibility} \times \mathit{KernelErrorFatality} \\
&\mathit{AMS} = \mathit{SignalledApplicationErrorSusceptibility} \times \mathit{ApplicationFatality} \\
&\mathit{OR} = \mathit{OverwriteRate} \times \mathit{OverwriteFatality}
\end{aligned}$$

In section 3.9, we indicate that only 10% need be fatal to the operating system (KernelError-Susceptibility), around 74% of errors are overwritten (OverwriteRate) and 16% of errors could be signalled to the application (SignalledApplicationErrorSusceptibility). Given our design goals to minimize the operating system modifications, let’s assume KernelErrorFatality is 100%. For high-level IA-32 and IA-64 processors, most forms of data overwrite can be recovered from by the processor or exception handler, so let’s assume OverwriteFatality is 0%. Our results from Section 4.2 show that 7-13% of JVM errors would be fatal to the JVM and its application when signalled (ApplicationFatality). This indicates that with a little application knowledge the reboot error rate could be reduced to (3435 x 0.20) x ((0.1 x 1.0) + (0.16 x 0.13) + (0.74 x 0.0)) or 82.9 FIT[1], a considerably smaller rate. This drop comes predominantly from ignoring overwritten errors. However, if we assume an old IA-32 processor where overwrites are fatal, this error rate remains at 591 FIT. In these situations, when execution cannot be continued, our investigations indicate that this may be improved by focussed kernel modifications. Since these numbers are

[1] Note 97% of the referenced source error rate is accounted for by soft errors. For simplicity the remaining 3% is not taken into account in these calculations.

error rates we cannot directly calculate a machine's eventual availability without determining the downtime for each error.

6 RELATED WORK

In 1979, Ziegler et al. at IBM Research proposed that cosmic rays and alpha particles can cause semiconductor transient errors in memory hardware [32]. Since this seminal research many others in the field of semiconductors have reported other experiments verifying the result. Some semiconductor research indicates that manufacturers are managing to limit increases in soft error rates through changes to their memory design and manufacturing processes [5, 33]. Other research indicates the need for consideration of soft errors more carefully in the longer term [1, 10]. However, the general consensus is that soft errors are likely to continue to play an important role in computer system availability.

Techniques such as parity bits, Error Correction Codes (ECC) and ChipKill [10] have been used in commodity main memories, storage media, and interconnects. These technologies allow different levels of error detection and correction on locations accessed by the processor. While ECC techniques reduce the number of errors, in this paper, we are interested in the effect on software when either only parity is used or errors are not masked by ECC. We believe that as technology trends increase the probability of memory soft errors, software recovery technique may become more important. Since these errors are not masked by hardware support, they cause the severe Machine Check Architecture (MCA) exception which typically results in a processor reset. These errors are a prime candidate for increasing availability through software recovery techniques.

Numerous research has been undertaken into the influence of failures on computer systems, as well as techniques to improve hardware and software reliability. Probably closest to the work of this paper has been the work on fault injection, propagation and error handling. The systems, such as FERRARI [18], React [9], and Fine [19], have greatly improved our understanding of hardware and software faults that are difficult to catch and repeat. Hsueh et al. give a good

survey and comparison of different injection methods [15]. Some work has been undertaken into understanding errors in COTS systems, including Linux, and the PowerPC CPU [11, 14, 21]. In the operating system work [11, 14] the focus is on inserting faults in focused areas to stress test the OS to evaluate potential corruption and affect on availability. This work is complementary to work, since it focuses on availability with faults rather than soft error activation rates, both of which are required to understand total system availability. While the COTS CPU work [21] focused on inserting errors through out the processor logic using a circuit fault injector, rather than focusing on the external memory system as our work does. Again we feel this investigation is complementary to our soft error activation rate experiments.

One approach to the problem of soft errors is to use reliable hardware through the use of redundancy. Typically, this increased hardware reliability is only available in proprietary servers, with specialized redundantly configured hardware and critical software components, such as processor pairs [2]. Examples include the IBM S/390 Parallel Sysplex [26], the Tandem NonStop Himalaya [2], and the Stratus ftServer [22]. Cornell's Hypervisor-based fault tolerance system provides a software alternative using multiple virtual machines to provide an n-1 fault-tolerant system [4]. Another approach in multiple processor systems is fault containment and recovery at a "node" granularity including cluster systems, and multi-cellular NUMA architectures, such as Hive [6].

Software reliability has been more difficult to achieve in commodity software even with extensive testing and quality assurance [25]. Techniques such as recovery blocks [12], checkpoints [13], techniques for failure transparency [20], to name but a few, have greatly improved recovery. In addition, a lot of work has been conducted in the context of distributed systems providing tolerance with support such as fail-over and distributed transactions [2, 13] rather than increasing single system availability, which is the focus of our work. Rio [8] takes a novel software-based approach to fault containment for a fault-tolerant file cache, by using memory protection operations to protect against wild writes to shared data structures.

However, commodity software fault recovery has not evolved very far. Most popular 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 taken up software reliability research in general and have not tackled problems of memory errors. Instead, they typically rely on fail-over solutions, such as Microsoft's Wolfpack [27].

Part of the solution to the problem of soft errors is the more widespread use of existing availability techniques to more effectively mask errors throughout the system. However, our approach is complimentary and attempts to improve the understanding of the susceptibility and recovery of commodity software to soft errors using simple fault injection and exception handling techniques. Our goal is to contribute to the existing work on the interaction of errors with software activation and to propose simple techniques that may help reduce the effect of soft errors.

7 LESSONS LEARNED

The following observations can be derived from our experimental data and analysis:

- The effect of soft errors on a modified operating system may be small. For our sample workload we measured that 90% of memory errors need not be fatal to the operating system's execution.
- Large numbers of activations are overwritten. This stems from the write before read use of most memory locations. This is due to page (and object) clearing for security and semantic reasons.
- Kernel mode read accesses to user data only account for a small number of accesses most of which are write accesses. In addition, kernel access to kernel data only accounts for a small number of memory access. This indicates that recovery is still possible when execution cannot be continued after an MCA exception if processors ignore overwritten errors, since user processes may be signalled or terminated in all other situations.

- For the Kaffe JVM and sample Java applications running on it, the memory errors in the object heap have a higher error activation rate and susceptibility rate than those in the static data area.
- A large portion of heap error activation is caused by the garbage collector (up to 75%). But this activation causes fewer application errors than other sources of activation (7% vs. 56%).

Adding a small amount of knowledge about the operating system and application can reduce the need for reboots by a significant fraction (down to 10% for an operating system and down to 15% for Kaffe in our initial experiments). While these are only initial results, they do indicate that simple forms of error handling and software recovery can noticeably benefit system availability.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we have described how memory soft errors become an increasing cause of failures in modern systems. However, commodity recovery support from these errors is limited because of price pressures on these systems. While semiconductor researchers try to limit the causes of soft errors on systems, the consensus is that these errors will continue to effect system availability.

Current and future commodity processor implementation are begining to have increased support for soft errors signalling. Assuming this improved support, we have investigated the effect of soft errors on commodity software. In doing so, we have gained an understanding of the correlation between soft errors and the reboots they can potentially cause.

Our investigation into the susceptibility of both the Linux kernel and Kaffe Java virtual machine indicate that many errors are not necessarily activated by commodity software. In addition, despite the potential data corruption that can occur, with simple instrumentation of the Linux kernel, we believe only 10% of memory errors actually need to be fatal for our sample workload. For the virtual machine, a large number of errors are activated by the heap garbage collector that need not cause a fatal error to the Java application. Together, these results indicate that with improved processor support and a little application knowledge, few of the activated soft errors

need to be fatal to the system, especially due to overwritten errors. Recently, similar observations to those made here have lead some high-end commodity chipsets to include memory scrubbing support to take advantage of the overwritting to minimise errors are masked by hardware.

Our results are only preliminary and the interaction between hardware soft errors and the software that they affect is a complex one. Therefore, future research on other commodity software and systems would greatly benefit our work. In addition, experiments run on real-world, possibly IA-64-based hardware, would help further validate our results and perhaps improve the possibility of running with real work example workloads.

Acknowledgements

We are indebted to John Wilkes, Ira Greenberg, Duane Dutton, George Candea, and Armando Fox for reviewing early versions of this 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] Anghel L., Nicolaidis M., Alexandrescu D., "Evaluation of soft error tolerance technique based on time and/or space redundancy", XIII Symp. on Integrated Circuits and Systems Design, Manaos, Brazil, September 2000.

[2] Bartlett, J., "A Nonstop Kernel", Proc. of the Eighth Symp. on Operating Systems Principles, pp 22-29, Dec. 1981.

[3] Bonwick, J., "The Slab Allocator: An Object-Caching Kernel Memory allocator". USENIX Tech. Conf., 1994.

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

[5] Baumann, R., "Soft Error Characterization and Modeling Methodologies at TI: Past, Present and Future", 4th Annual Topical Research Conf. on Reliability, Oct. 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", USENIX JVM Symposium '01, 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] Clark, J., and Pradhan. D., "Fault Injection: A Method for Validating Computer System Dependability," with Dhiraj K. Pradhan, IEEE Computer Magazine, June 1995, pp. 47-56.

[10] Dell, T. J., "A White Paper on the benefits of Chipkill", IBM Microelectronics Division, Nov. 1997.

[11] Fabre, J.C., Salles, F., Modriguez-Moreno, M. and Arlat, J., "Assessment of COTS Microkernels by Fault Injection," in Proc. IFIP Dependable Computing for Critical Applications (DCCA'99).

[12] Goodenough, J., Exception Handling: Issues and a Proposed Notation, Comm. of the ACM 18, 683-696 (1975).

[13] Gray, J., and Reuter, A., "Transaction Processing: Concepts and Techniques," Morgan Kaufmann, 1993.

[14] Gu. W el al. "Characterization of Linux Kernel Behavior under Errors", DSN-03, 2004.

[15] Hsueh, M-C, et al., "Fault Injection Techniques and Tools", IEEE Computer, pp 75-82, April 1997.

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

[17] Intel IA-32 Architecture Software Developer's Manual, Volume 3, Intel 2002, Intel Corporation.

[18] 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.

[19] 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.

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

[21] Maderia, H., et al., "Experimental evaluation of a COTS system for space applications", DSN02, 2002.

[22] McLaughlin, B. "Evaluating Alternatives for Windows® 2000 Server Availability", White Paper, Stratus, 2001.

[23] McVoy, L., Staelin, C., "lmbench: Portable Tools for Performance Analysis", Usenix Technical Conf., 1996.

[24] Milojicic, D., et al., "Increasing Relevance of Memory Hardware Errors – A Case for Recoverable Programming Models", ACM SIGOPS European Workshop, Sept. 2000

[25] Murphy, B., et al. "Windows 2000 Dependability", IEEE Intl. Conf. on Dependable Sys. and Nets, June 2000.

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

[27] Pfister. G, "In Search of Clusters", Prentice Hall, 1998.

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

[29] Standard Performance Evaluation Corp., "SPECjvm98 Specification," Aug. 1998.

[30] Tandem, Compaq Corporation, "Data Integrity for Compaq NonStop Himalaya Servers", White Paper, 1999.

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

[32] 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.

[33] 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.

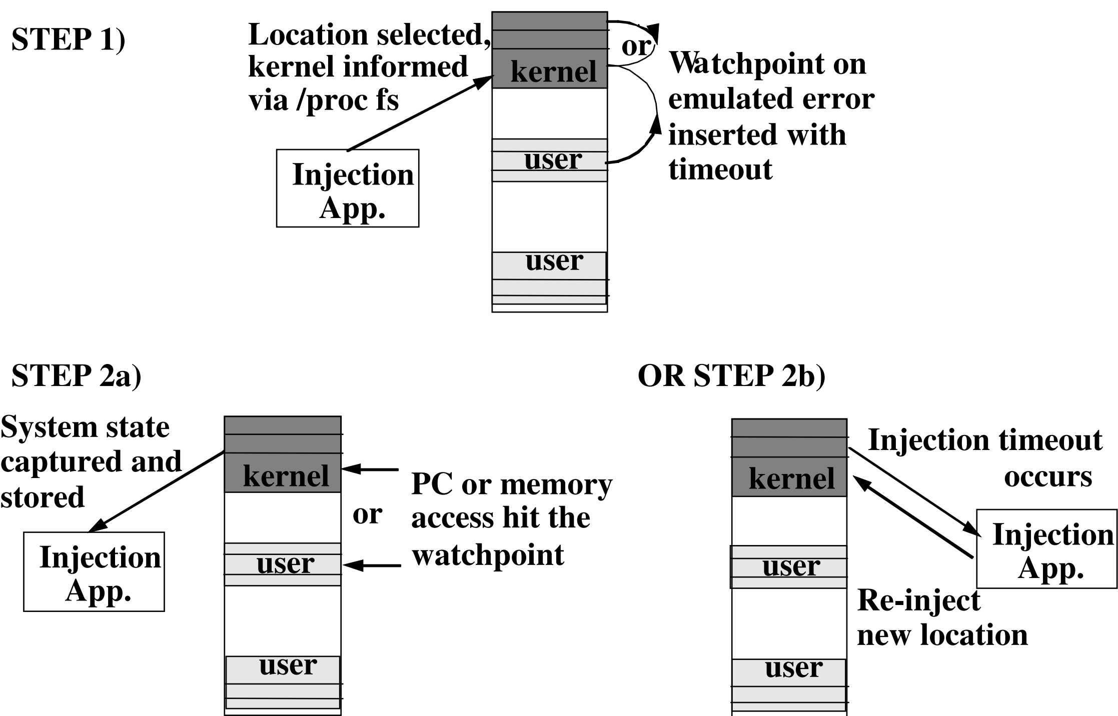


Figure 1: Non-intrusive error injection to emulate soft-errors and determine activation point/usage

Table 1: OS Error injection experiment sets

Workload	Injection Time-out	Elapsed	Injections	Activations	Activation Rate
1	10 sec.	100 sec.	12	2	17%
	30 sec.	5 min.	15	5	33%
	1 min.	10 min.	17	9	53%
	2 min.	20 min.	18	10	56%
	5 min.	50 min.	18	12	66%
	10 min.	100 min.	28	20	71%
	30 min.	5 hours	47	40	85%
1	10 sec.	4 hours	1690	464	27%
	30 sec.	4 hours	670	278	41%
	60 sec.	4 hours	382	197	52%
	120 sec.	4 hours	228	132	57%
2	60 sec.	90 hours	5499	599	11%
	Total	**~114 hours**	**8624**	**1768**	**20%**

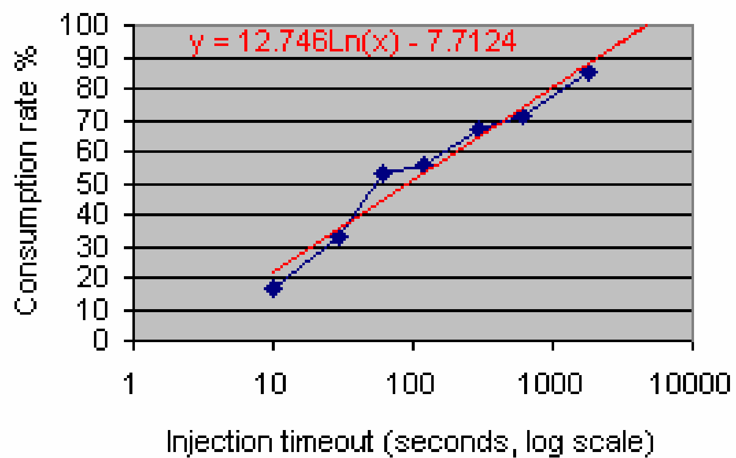


Figure 2: Fault activation rate vs. injection time-out

.

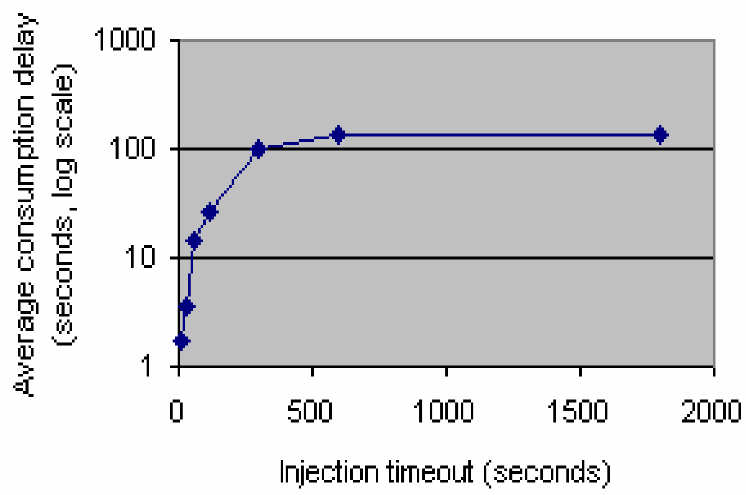


Figure 3: Activation delay

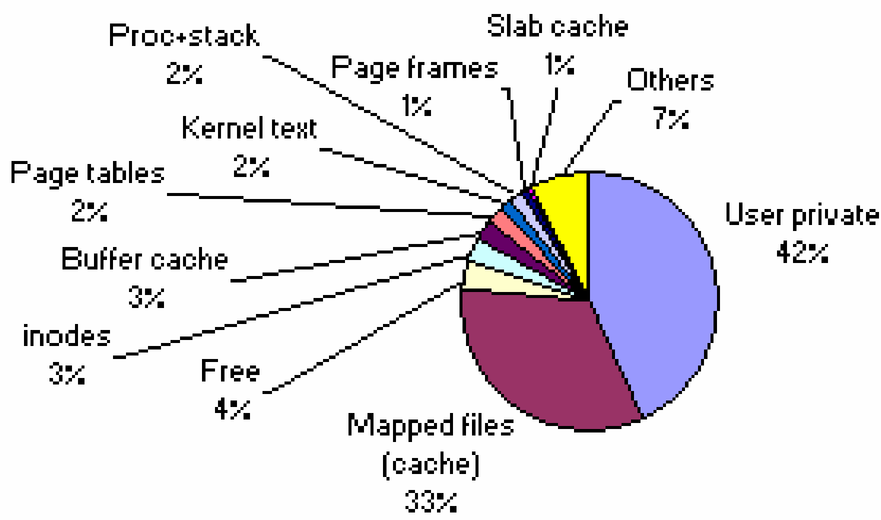
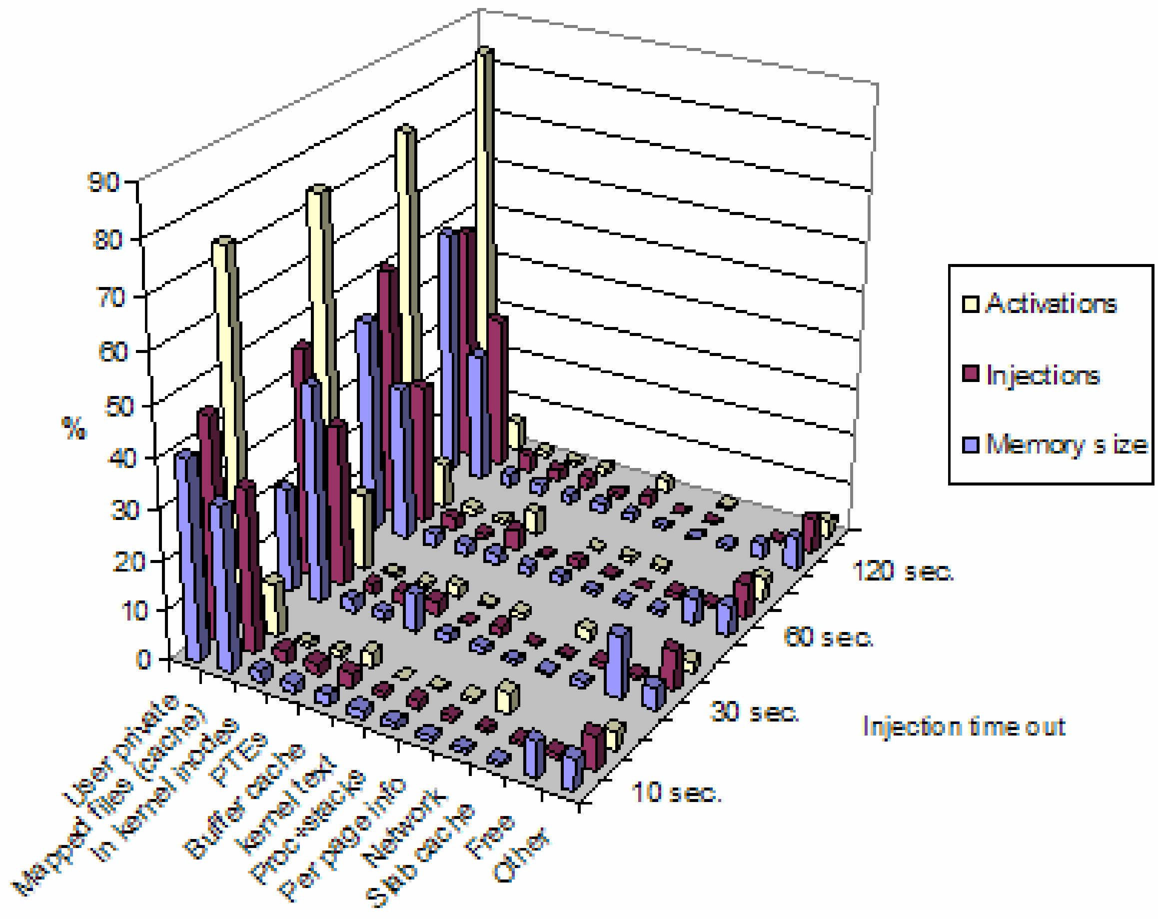


Figure 4: Linux memory objects classification by size



Figur e 5: Affected memory by type for the Linux kernel

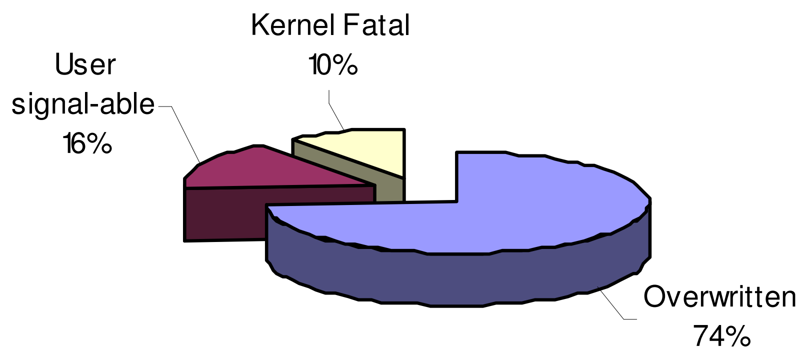


Figure 6: Error severity classification

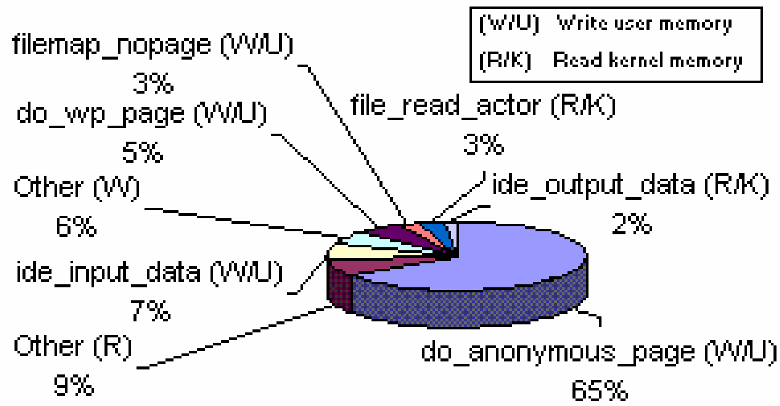


Figure 7: Function associated with program counter location when errors were activated in kernel mode

Table 2: PC locations for read access memory error activations occurring in kernel mode

Kernel location	%	Kernel location	%
ide_output_data (disk write)	20	csum_partial_copy_gene ric	2
statm_pgd_range (/proc FS)	9	math_state_restore	1
xirc2ps_interrupt (net I/O)	8	tcp_send_skb	1
filemap_nopage	8	tcp_clear_xmit_timers	1
do_fork	7	tcp_v4_rcv	1
cp_new_stat (lstat)	4	do_select	1
ext2_update_inode	4	find_buffer	1
tcp_timewait_kill	3	d_lookup	1
brw_page	3	__generic_copy_to_user	1
ext2_open_file	3	zap_page_range	1
check_tty_count	3	get_statm	1
clear_page_tables	2	vsprintf	1
get_unmapped_area	2	si_meminfo	1
lookup_dentry	2	si_swapinfo	1
skb_clone	2	collect_sigign_sigcatch	1
Total			100

Table 3: Influence of workload on error activation and severity

	Workload 1				Workload 2	Overall
Injection Time out	10 sec.	30 sec.	60 sec.	120 sec.	60 sec.	
Injections	1690	670	382	228	5499	8469
Activation rate	27%	41%	52%	58%	11%	20%
Fatal	9%	6%	10%	8%	13%	10%
Overwritten	80%	81%	78%	88%	60%	73%
User signalable	11%	13%	13%	5%	27%	17%

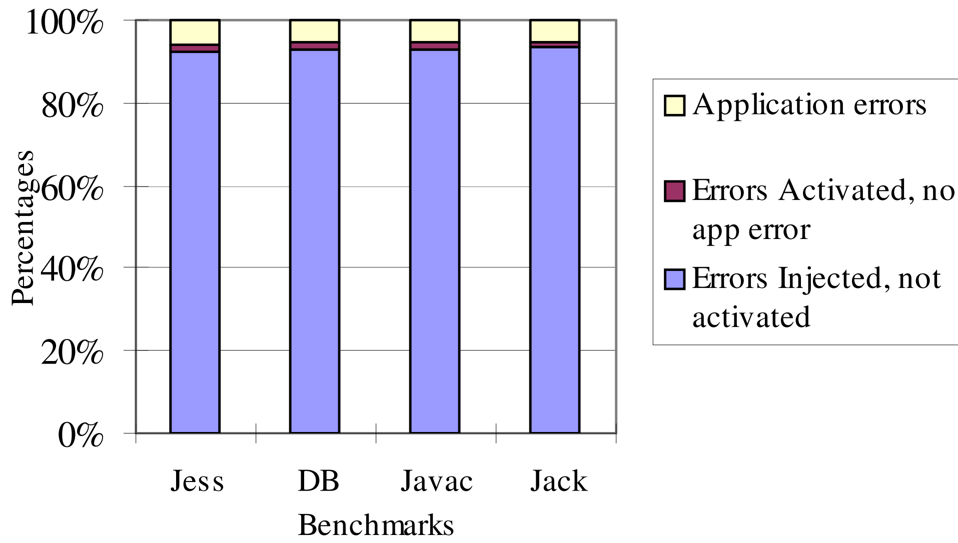


Figure 8: Error activation in the JVM's static data

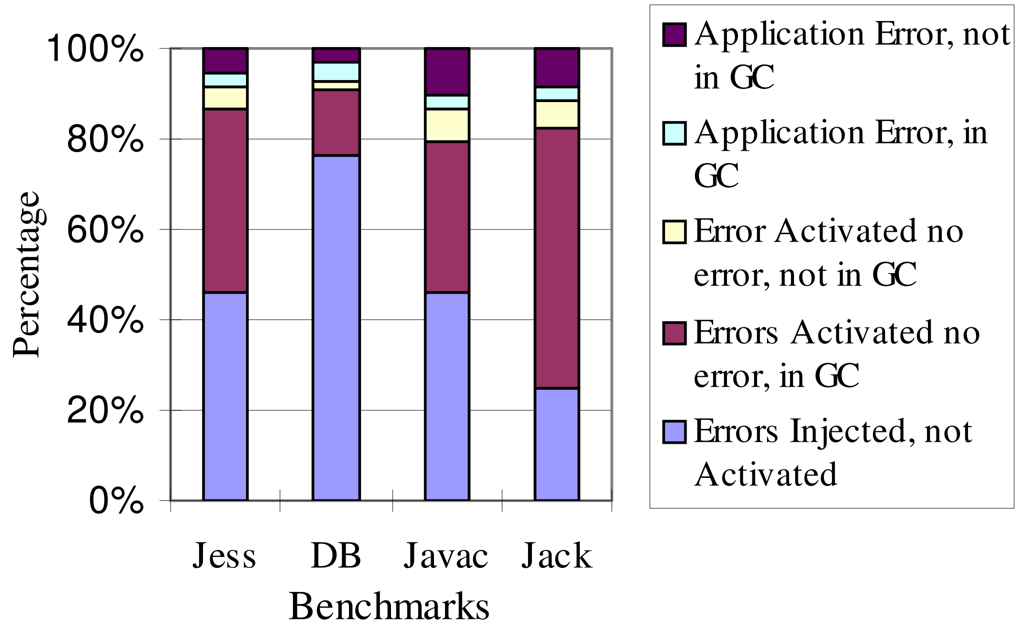


Figure 9:Error Acti vation in the JVM's heap region