

Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors

Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer

Center for Reliable and High-Performance Computing

Coordinated Science Laboratory

University of Illinois at Urbana-Champaign

1308 West Main Street, Urbana, IL 61801

{wngu, kalbar, iyer}@crhc.uiuc.edu

Abstract: *The goals of this study are: (i) to compare Linux kernel (2.4.22) behavior under a broad range of errors on two target processors—the Intel Pentium 4 (P4) running RedHat Linux 9.0 and the Motorola PowerPC (G4) running YellowDog Linux 3.0—and (ii) to understand how architectural characteristics of the target processors impact the error sensitivity of the operating system. Extensive error injection experiments involving over 115,000 faults/errors are conducted targeting the kernel code, data, stack, and CPU system registers. Analysis of the obtained data indicates significant differences between the two platforms in how errors manifest and how they are detected in the hardware and the operating system. In addition to quantifying the observed differences and similarities, the paper provides several examples to support the insights gained from this research.*

1 Introduction

The dependability of a computing system (and hence of the services it provides to the end user) depends to large extent on the failure-resilience of the underlying hardware (processor) and operating system. Understanding a system's sensitivity to errors and identifying its error propagation patterns and single points of failure are thus of primary importance in selecting a computing platform and in assessing tradeoffs involving cost, reliability, and performance. This study investigates two related aspects of this issue: (i) how the Linux kernel responds to errors that impact kernel code, kernel data, kernel stack, and processor system registers, and (ii) how processor hardware architecture (instruction set architecture and register set) impacts kernel behavior in the presence of errors. The goal is to compare Linux behavior under a broad range of errors on two different platforms and to understand how architectural characteristics of the target processors impact the error sensitivity of the operating system. Two target Linux-2.4.22 systems are used: the Intel Pentium 4 (P4) running RedHat Linux 9.0 and the Motorola PowerPC (G4) running YellowDog Linux 3.0.

Fault/error injection is used to stress the Linux system while running workload programs. The UnixBench [23] benchmark suite is used to profile kernel behavior and to identify the most frequently used functions representing at least 95% of kernel usage. Over 115,000 faults/errors are injected into the kernel code, data, stack, and CPU registers. The response of the kernel (e.g., crash, hang, fail silence violation, not-manifested) on each target system is automatically monitored and logged. While our previous study focused on errors in the code segment of the Linux kernel [9], the current work investigates the effect of the architecture on kernel sensitivity to a broad range of errors. Major findings are:

- While the activation¹ of errors is generally similar for both processors, the manifestation percentages for the Pentium 4 are about twice as high.
- For stack errors, there is a significant difference between the two processors in manifestation percentages (56% for P4 versus 21% for G4). A similar trend is observed in kernel data errors, where 66% (for P4) and 21% (for G4) of injected errors manifest as crashes.
- The variable-length instruction format of the P4 makes it possible for a bit error to alter a single instruction into a sequence of multiple valid (but incorrect from the application semantic point of view) instructions. On one hand, this leads to poorer diagnosability: executing an incorrect instruction sequence may crash the system and generate an exception, which does not isolate the actual cause of the problem and obstructs diagnosis. On the other hand, the same feature has the potential to reduce crash latency: executing an incorrect instruction sequence is likely to make the system fail fast.
- Less compact fixed 32-bit data and stack access makes the G4 platform less sensitive to errors. The sparseness of the data can mask errors, e.g., a larger presence of unused bits in data items means that altering any unused bit is inconsequential, even if the corrupted data is used. The more optimized access patterns on the P4 increase the chances that accessing a corrupted memory location will lead to problems.

2 Related Work

Failure behavior/characterization of operating systems has been the focus of several studies. This section briefly reviews representative examples.

User-level testing by executing API/system calls with erroneous arguments. Ballista [14] provides a comprehensive assessment of 15 POSIX-compliant operating systems and libraries as well as the Microsoft Win32 API. Fuzz [17] tests system calls for responses to randomized input streams; the study addresses the reliability of a large collection of UNIX utility programs and X-Window applications, servers, and network services. The *Crashme* benchmark [6] uses random input response analysis to test the robustness of an operating environment in terms of exceptional conditions under failures.

¹ Error activation cannot be determined while injecting into system registers, as we do not have the ability to monitor kernel access to these registers.

Error injection into both kernel and user space. Several studies have directly injected faults into the kernel space and monitored and quantified the responses. FIAT [2], an early fault injection and monitoring environment, experiments on SunOS 4.1.2 to study fault/error propagation in the UNIX kernel. FINE [12] injects hardware-induced software errors and software faults into UNIX and traces the execution flow and key variables of the kernel.

Xception [5] uses the advanced debugging and performance monitoring features existing in most modern processors to inject faults and monitor their activation and impact on target system behavior. Xception targets PowerPC and Pentium processors and operating systems ranging from Windows NT to proprietary, real-time kernels (e.g., SMX) and parallel operating systems (e.g., Parix).

MAFALDA [1] analyzes the behavior of Chorus and LynxOS microkernels in the presence of faults. In [4], User Mode Linux (equivalent to a virtual machine, representing a kernel) executing on top of the real Linux kernel is used to perform Linux kernel fault injection via the *ptrace* interface.

In [19], SWIFI is employed to guide the design and implementation of the Rio File Cache system on top of FreeBSD operating system. Edwards and Matassa in [8] use fault injection for hardening kernel device drivers. A state machine is constructed to track the state of hardware and to support the injection of faults at a specific run-time state.

Other methods to evaluate the operating system. Operating systems have been evaluated by studying the source code, collecting memory dumps, and inspecting error logs. For example, Chou et al. [7] present a study of Linux and OpenBSD kernel errors found by automatic, static, compiler analysis at the source code level. Lee et al. [15] use a collection of memory dump analyses of field software failures in the Tandem GUARDIAN90 operating system to identify the effects of software faults. Xu et al. [24] examine Windows NT cluster reboot logs to measure dependability. Sullivant et al. [22] study MVS operating system failures using 250 randomly sampled reports.

3 Methodology

Software-implemented error injection is employed to experimentally assess the error sensitivity of the Linux kernel executing on PowerPC G4 and Pentium 4 processors. Single-bit errors are injected into kernel stacks, kernel code sections, kernel data structures, and CPU system registers while running benchmark programs. NFTAPE [21], a software framework for conducting fault/error injection experiments, is used to conduct the tests.

3.1 Error Injection Environment

A driver-based Linux kernel error injector is developed to enable error injection campaigns. The injection driver (a kernel module) attached to the kernel exploits the CPU's debugging and performance-monitoring features to (i) automatically inject errors, (ii) monitor error activation, error propagation, and crash latency, and (iii) reliably log the data to remote persistent storage (*crash data storage*).

The error injection environment, shown in Figure 1, consists of (i) kernel-embedded components (injectors, crash handlers, and data deposit module) for P4 and G4 architectures, (ii) a user-level NFTAPE control host, which prepares the target addresses/registers (to be injected), starts the workload using benchmark (UnixBench [23]), and logs injection data for analysis, (iii) a hardware monitor (watchdog card) to detect system hangs/crashes in order to provide auto reboot if needed, and (iv) a remote crash data collector that resides on the control host computer to receive crash data by UDP connections. The latter capability is an important extension to the existing NFTAPE framework and allows reliably collecting the crash data even if the underlying file system (on the target node) is not accessible. The instrumentation to collect data on error latency (cycles-to-crash) is also added.

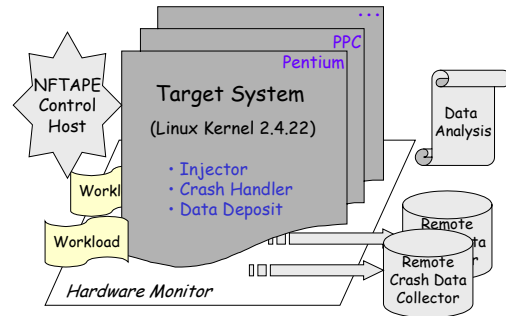


Figure 1: Error Injection Environment

3.2 Approach

An automated error injection process, illustrated in Figure 2, includes the following three major steps:

STEP 1: Generate injection targets. The *target address/register generator* provides error injection targets for the following: (i) *code injection*—an instruction breakpoint location based on selected kernel functions, kernel subsystems, and kernel code address ranges; (ii) *stack injection*—the bit patterns to inject at randomly chosen (at runtime) kernel process; (iii) *system register injection*—system registers to inject; and (iv) *data injection*—random locations in the kernel data (both initialized and uninitialized) section. The error injection targets are generated and stored before an error injection campaign is initiated. As a result, the activation rate may not be 100%, as some of the pre-generated errors are never injected because a corresponding breakpoint is never reached.

STEP 2: Inject errors. The kernel injector obtains pre-generated information on the injection target and performs the error injections. This process includes starting the benchmark, enabling performance registers to measure crash latency, and injecting errors.

STEP 3: Collect data. Depending on the outcome of the error injection, one of the following actions is taken:

1. *Error Not Activated.* Go to STEP 1 and proceed to the next injection without rebooting the target machine.
2. *Error Activated.* (a) (*Not Manifested, Fail Silence Violation, System Hang*) Log the results, reboot the target system, and proceed to the next injection; (b) (*Crash*) Collect data on crash, reboot the target system, and proceed

to the next injection. Crash causes, cycles to crash, and frame pointers before and after injections are collected using crash handlers embedded in the kernel. This information is packaged as a UDP-like packet and sent to a remote crash data collector (reliable storage) through UDP connection. Since the *file system* may not behave properly when the kernel crashes, the crash handler bypasses the kernel’s underlying *file system* and supplies these packets directly to the network card’s packet-sending function.

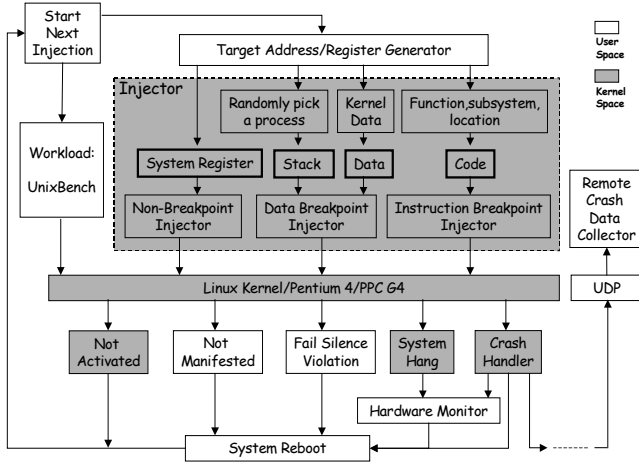


Figure 2: Automated Process of Injecting Errors

3.3 Error Activation

The CPU’s debugging registers are used to determine error activation. As shown in Figure 2, two types of breakpoint—*instruction breakpoint* and *data breakpoint*—are used in the injection campaigns. For code injections, one of the *Debug Address Registers* is used to store a 32-bit linear address to be monitored in the kernel code. When the breakpoint is reached but before the instruction located at this address is executed, an error is injected². In addition, the performance registers are used to record the latency in terms of CPU cycles.

Kernel data/stack injections use *data memory breakpoints* instead. These breakpoints stop the processor on data reads/writes but not on instruction fetches. The processor reports access to a data memory breakpoint after the target memory location is accessed (i.e., data read/write). Note that an *instruction breakpoint* is reported before executing the target instruction. Consequently, our injector inserts an error before a data memory breakpoint is set. If this breakpoint is not reached, the original value at the location is restored, and the error marked as *not activated*. If the data memory breakpoint is triggered, one of the following actions is taken: (i) *data write access*, in which the injected error is overwritten by the write operation and, thus, needs to be re-injected and is marked as *activated* or (ii) *data read access*, in which the injected error is not overwritten and is marked as *activated*.

3.4 Target System Configuration

Table 1 summarizes the experimental setup. To speed up the experiments, three P4 and two G4 machines are used in the

injection campaigns. Watchdog cards driven by Linux drivers are embedded in those machines to enable automated system reboot after a hang.

Processor	Hardware		System Software		
	CPU Clock [GHz]	Memory [MB]	Distribution	Linux Kernel	Compiler
Intel Pentium 4	1.5	256	RedHat 9.0	2.4.22	GCC 3.2.2
Motorola MPC 7455	1.0	256	Yellow-Dog 3.0	2.4.22	

Table 1: Experiment Setup Summary

3.5 Error Model

The error model assumed in this study is not contingent upon the error origin, i.e., an error could have occurred anywhere in the system—the disk, network, bus, memory, or CPU. Single-bit errors are injected into the instructions of the target kernel functions, the stack of the corresponding kernel process, the kernel data structures, and the corresponding CPU’s system registers. Previous research on microprocessors [20] has shown that most (90-99%) of device-level transients can be modeled as logic-level, single-bit errors. Data on operational errors also show that many errors in the field are single-bit errors [11].

While in a well-designed system multiple mechanisms for protecting against errors maybe available (e.g., parity, ECC³, or memory scrubbing), errors still exist. Errors could, for example, be timing issues due to hardware/software problems, to a noise source such as undershoot or overshoot, or to noise on the address bus that results in the wrong data being written to/read from the memory [16]. In the latter case, the data maybe unaltered due to the address bus noise, but the wrong location is accessed. Our error injection experiments employ memory errors and system register errors to emulate the diverse origins and impact of actual errors. Four attributes characterize each error injected:

Trigger (when?). An error is injected when (i) a target datum is read/written for stack/data injection, (ii) a target instruction in a given kernel function is reached, or (iii) a system register is used. The kernel activity is invoked by executing a user-level workload (UnixBench) program.

Location (where?). (i) This can be a randomly selected kernel stack location, a location within the initiated/uninitiated kernel data, or a system register. (ii) For kernel code injection, the location is pre-selected based on the profiling [13] of kernel functions, i.e., kernel functions most frequently used by the workload are selected for injections.

Type (what?). (i) This is a single-bit error per data word in the case of stack, data, and system register injections. (ii) For kernel code injections, it is a single-bit error per instruction.

Duration (how long?). A bit is flipped in the target to emulate the impact of a transient event, which result in the corruption of data, code, or CPU registers. (i) For data, stack, and system

² While this discussion is in the context of the Pentium 4 processor, the PowerPC G4 injection scheme is similar.

³ As indicated by manufactures, logic failure rates may erode the efficacy of ECC in designs. Hardened logic libraries or schemes to mask logic sensitivity (redundancy on critical paths, spatial and/or temporal) may be needed to account for this deficiency [3].

register injections, errors may last as short a time as it takes the corrupted data item to be overwritten due to normal system activities. (ii) For code injections, an error may persist throughout the execution time of the benchmark.

3.6 Outcome Categories

Outcomes from error injection experiments are classified according to the categories given in Table 2. In addition, the crash category is further divided into subcategories. Table 3 and Table 4 provide crash subcategories for the Pentium (P4) and PPC (G4) processors, respectively.

Outcome Category	Description
Activated	The corrupted instruction/data is executed/used.
Not Manifested	The corrupted instruction/data is executed/used, but it does not cause a visible abnormal impact on the system.
Fail Silence Violation	Either operating system or application erroneously detects the presence of an error or allows incorrect data/response to propagate out. Benchmark programs are instrumented to detect errors.
Crash	Operating system stops working, e.g., bad trap or system panic. Crash handlers embedded into the operating system are enhanced to enable dump of failure data (processor and memory state). Off-line analysis of this data allows determining causes for most of the observed crashes.
Hang	System resources are exhausted, resulting in a non-operational system, e.g., deadlock.

Table 2: Outcome Categories

Crash Category (P4)	Description
NULL Pointer	Unable to handle kernel NULL pointer de-reference.
Bad Paging	A page fault: the kernel tries to access some other bad page except NULL pointer.
Invalid Instruction	An illegal instruction that is not defined in the instruction set is executed.
General Protection Fault	Exceeding segment limit, writing to a read-only code or data segment, loading a selector with a system descriptor, reading an execution-only code segment.
Kernel Panic	Operating system detects an error.
Invalid TSS (Task State Segment)	The selector, code segment, or stack segment is outside the limit, or stack is not writeable.
Divide Error	Math error.
Bounds Trap	Bounds checking error.

Table 3: Crash Cause Categories – Pentium (P4)

Crash Category (G4)	Description
Bad Area	A page fault: the kernel tries to access a bad page including NULL pointer, or bad memory access.
Illegal Instruction	An illegal instruction that is not defined in the instruction set is executed.
Stack Overflow	Stack pointer of a kernel process is out of range.
Machine Check	An error is detected on the processor-local bus including instruction machine-check errors and data machine-check errors.
Alignment	Load/Store or other specific instructions' operands are not word-aligned.
Panic	Operating system detects an error.
Bus Error	Protection fault.
Bad Trap	Unknown exception.

Table 4: Crash Cause Categories – PPC (G4)

Latency (Cycles-to-Crash) is defined as the number of CPU cycles between error activation and the actual crash. (Note that, for system register injections, crash latency represents the time between error injection and the observed crash). Typically, latency includes three stages, as shown in Figure 3.

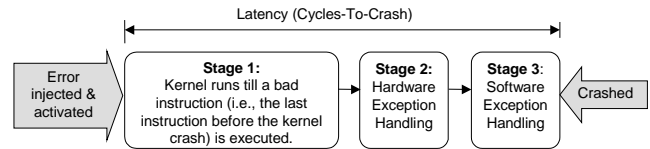


Figure 3: Definition of Cycles-to-Crash

Stage 1. A target location is reached, an error is injected, and a performance register is started to measure time to potential crash; the kernel keeps running until a bad instruction is executed, e.g., a NULL pointer is referenced; this may take from zero (immediate crash after executing/accessing the injected instruction/data) to millions of CPU cycles.

Stage 2. The CPU's hardware exception handling takes over to determine exception vector, evaluate correctness, and save and load necessary data; this may consume more than 1000 CPU cycles.

Stage 3. The hardware transfers control to the software exception handler, which typically executes about 150 to 200 instructions.

4 Overview of Experimental Results

Table 5 and Table 6 summarize the results of all injection campaigns conducted on P4 and G4 platforms⁴. The major conclusions from these tables (over 115,000 error injections) can be summarized as follows:

- While the error activation rates are generally similar for both processors, the manifestation rates for the Pentium 4 are about twice as high.
- For stack errors, there is a significant difference between the two processors in the manifestation rates (56% for P4 versus 21% for G4). A similar trend is observed in the case of an error in the kernel data (66% for the P4 versus 21% for the G4). The observed difference between the two platforms can be explained by the disparity in the way they use memory. The G4 processor always operates on 32-bit wide data items, while the P4 allows 8-bit, 16-bit, and 32-bit data transfers. As a result, it is possible for many stack and data errors on the G4 platform to corrupt unused data bits in a target data item.
- For register errors, the trend in manifestation rates is similar to the above, although the manifestation rates for both platforms are lower (5% for G4, and over 11% for P4)⁵.
- While percentages of FSVs are small (1.3% for P4 to 2.3% for G4) for code segment injection, they have significant error propagation potential and hence are a cause for concern. Observe that the P4 data injection does not cause any FSVs, while on the G4, 1% of activated data errors manifest as FSVs.

⁴ Percentage figures in the third column of the two tables are calculated with respect to all injected errors; all other percentages are given with respect to activated errors.

⁵ Register injections target the system registers, e.g., flag register, stack registers, and memory management registers. While we cannot determine the exact percentage of activated errors, the way the system is using these registers creates a high chance that errors will be activated.

Intel Pentium 4 Campaign	Injected	Error Activated	Activated			
			Not Manifested	Fail Silence Violation	Known Crash	Hang/Unknown Crash
Stack	10143	2973(29.3%)	1305(43.9%)	0(0%)	1136(38.2%)	532(17.9%)
System Registers	3866	N/A	3459(89.5%)	0(0%)	305(7.9%)	102(2.6%)
Data	46000	226(0.5%)	77(34.1%)	0(0%)	96(42.5%)	53(23.4%)
Code	1790	982(54.9%)	308(31.4%)	13(1.3%)	455(46.3%)	206(21.0%)
Total	61799					

Table 5: Statistics on Error Activation and Failure Distribution on P4 Processor

Motorola PPC G4 Campaign	Injected	Error Activated	Activated			
			Not Manifested	Fail Silence Violation	Known Crash	Hang/Unknown Crash
Stack	3017	1203(39.9%)	949(78.9%)	0(0%)	172(14.3%)	84(7.0%)
System Register	3967	N/A	3774(95.1%)	0(0%)	69(1.7%)	124(3.1%)
Data	46000	704(1.5%)	551(78.3%)	7(1.0%)	55(7.8%)	91(12.9%)
Code	2188	1415(64.7%)	580(41.0%)	33(2.3%)	576(40.7%)	226(16.0%)
Total	55172					

Table 6: Statistics on Error Activation and Failure Distribution on G4 Processor

5 Crash Cause Analysis

Figure 4 and Figure 5 show the distribution of the causes of known crashes (i.e., those for which crash dump information was logged). The crash cause distributions for the two platforms have some similarities:

- About 67% (*Bad Area*) of crashes on the G4 and 71% (sum of *Bad Paging* and *NULL Pointer*) of crashes on the P4 are due to invalid memory access; illegal instructions contribute to 16% of crashes on both platforms.
- As will be discussed in the following section, the P4 does not explicitly report stack overflow (this category is not present in the pie-chart in Figure 4). On the P4, these stack errors propagate, and most of them manifest as *Bad Paging* or *General Protection Fault*. The impact of the propagation is discussed in Section 6 in connection with an analysis of error latency.
- On both platforms, about 0.1% of crashes are due to system *Panic* (i.e., the OS detects an internal error).

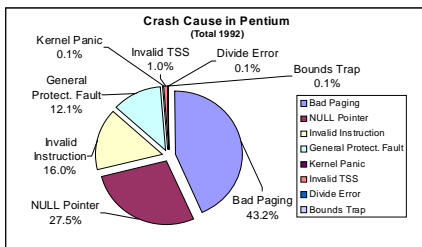


Figure 4: Overall Distribution of Crash Causes (Known Crash Category on P4)

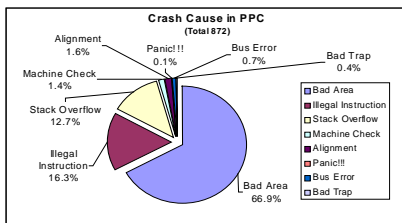


Figure 5: Overall Distribution of Crash Causes (Known Crash Category G4)

5.1 Stack Injection

In this section we analyze the manifested stack errors. Stack injections on the G4 primarily result in *Stack Overflow* (41.9%) and *Bad Area* (53.5%), while *Bad Paging* (45%) and *Null Pointer* (31%) dominate P4 results (see Figure 6).

Stack Overflow is caused by corruption of frame pointers stored on the stack. As a result, the kernel attempts to access memory beyond the currently allocated space for the stack, causing an exception to be generated. While the *Stack Overflow* category constitutes a significant percentage of crashes (41.9%) on the G4 processor, it does not appear to occur on the P4 platform. One could look for the explanation in the relative sizes of the runtime stack used by the two processors. While the average size of the runtime kernel stack on the G4 is twice that of the P4 stack, the average number of function frame pointers concurrently kept on each of the two stacks is very much the same (5 to 7 frames).

But the main reason for stack overflows not being reported on the P4 is that the Linux kernel on the P4 platform does not provide an exception to explicitly indicate stack overflow. An analysis of crash data on the P4 platform shows that it is possible to detect error patterns that correspond to a stack overflow (see Figure 7). This analysis shows that a fraction of the crashes that manifest as *Bad Paging* are actually due to stack overflow. Moreover, some of the crashes categorized as *Invalid Opcode* and *NULL Pointer* are also due to stack overflow. In addition, often on the P4 an actual stack overflow leads directly to a *General Protection* exception. As will be seen in Section 6, the fact that the P4 does not indicate stack overflows has implications for error latency (the time between an error injection and the occurrence of the crash or system detection) as well.

The above observations and analysis also explain the difference in the number of crashes attributed to *invalid memory access* for the two processors: 53% and 76% (sum of *Bad Paging* and *Null Pointer* cases) for G4 and P4, respectively. Our analysis shows that some invalid memory accesses on the P4 are actually due to a stack overflow and could have

been detected as such if the P4 (or the OS) could explicitly capture such events.

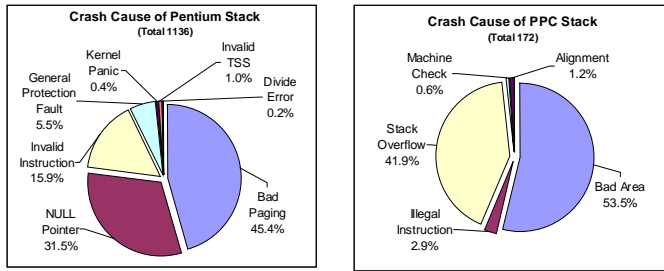


Figure 6: Crash Causes for Kernel Stack Injection

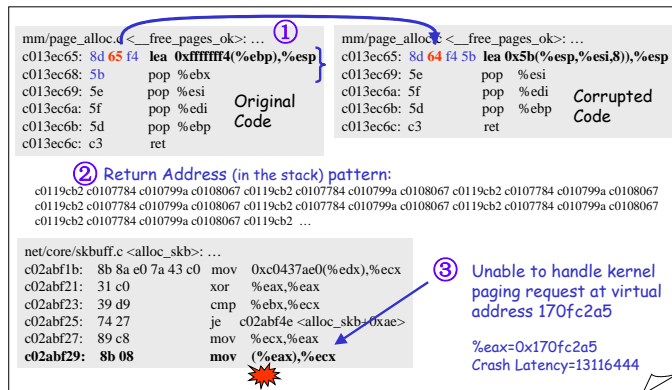


Figure 7: Error Propagation between Kernel Subsystems Due to Undetected Stack Overflow

Error Propagation Due to Stack Overflow. Figure 7 depicts an example of error propagation (between the *memory management* (*mm*) and the *network* (*net*) subsystems) due to an undetected stack overflow on the P4. A bit error in the function *free_pages_ok()* in the *mm* subsystem causes the P4 to execute the valid but incorrect instruction *lea 0x5b(esp,esi,8),esp* instead of two original instructions *lea 0xffffffff4(ebp),esp* and *pop ebx*. As a result, the stack pointer (*ESP*) gets an incorrect value ①. Corruption of the stack pointer is not detected by the P4 kernel, which continues executing in the presence of an error. Eventually, the system crashes in the presence of an error. The kernel crashes in the function *alloc_skb()* in the network subsystem③, when *mov* reads a value from *eax* register that contains an illegal kernel address (0x170fc2a5). The kernel raises the exception *Unable to handle kernel paging request* at virtual address 0x170fc2a5 and crashes. The measured crash latency is 13116444 cycles. The crash dump data show a typical stack overflow pattern ②.

P4 Stack Error. Figure 8 depicts a typical example of the consequences of an error in the kernel stack. The sample source code in the figure shows the function *kupdate()* from the kernel file subsystem. This function is used to flush to disk the *dirty data buffers* kept in RAM. The function uses a kernel data structure (*tsk*) stored on the kernel stack. In particular, *tsk->state* is at location *0xffffffffe0(%ebp)*, where *ebp* is the stack pointer ①. A bit error inserted and activated at location *0xffffffffe0(%ebp)* alters the content of *eax* so that the state of *tsk->state* is corrupted (i.e., is not set to *TASK_STOPPED* ②). Thus, incorrect parameters are passed to the scheduler (*schedule()*). When *schedule()* returns, the content of

0xffffffffe0(%ebp) is 0, which is saved in *edx*. After executing *mov 0x8(%edx),%ecx*, the NULL pointer exception is raised, and the system crashes within the next 12864 cycles ③.

G4 Stack Error. Figure 9 illustrates the impact of stack errors on the G4 platform. The sample source code is extracted from *kjournald()* in the kernel file system subsystem. This function manages a logging device journal. At the address *c008d798*, *lwz* loads the content to the register *r11* from the stack location pointed to by *40(r31)*. Register *r31* is used as a temporary stack pointer. A single-bit error causes an invalid kernel address “1” to be loaded to *r11* instead of a legal address. The kernel crashes (*Bad area* exception) trying to access memory at *0x0000004d* (the content of *r11* is used as an address: *76(r11)=0x4d*). In comparison with the P4, the crash latency here is much shorter: 1592 cycles or 210 instructions, clearly a potential advantage in attempting to limit error propagation.

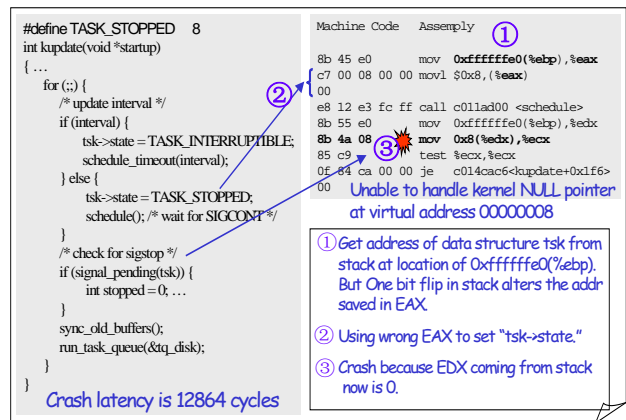


Figure 8: Consequences of Kernel Stack Error (P4)

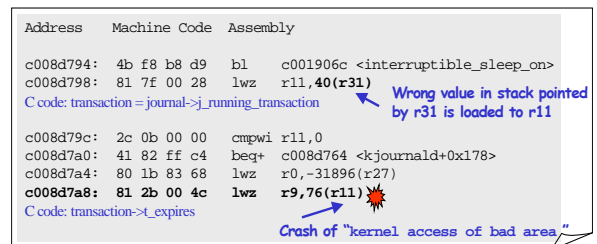


Figure 9: Consequences of the Kernel Stack Error (G4)

5.2 System Register Injection

Register error-injection campaigns on both processors target system registers. In the P4, the system registers “assist in initializing the processor and controlling system operations” [10]. System-level registers targeted on the P4 processor include the flag register (system flags only, e.g., nested task flag), control registers, debug registers, stack pointer, segment registers (*fs* and *gs* only)⁶, and memory-management registers. On the G4, error injection campaigns focus on the registers, which belong to the supervisor model in the PowerPC family (as opposed to the user model) [18]; these include memory management registers, configuration registers, performance monitor registers, exception-handling registers, and cache/memory subsystem registers.

⁶ These two segment registers are stored for each context switch as part of TSS (task state segment).

Figure 10 shows the distribution of crash causes for system register injections. Out of 99 system registers in the G4 and approximately 20 in the P4, only 15 G4 registers and 7 P4 registers contribute to the crashes and hangs observed in our experiments. The key errors observed and their associated registers are as follows:

- *General Protection* errors observed on the P4 platform are due to corruption of the control register *CR0* and the segment registers *FS* and *GS*. *CR0* contains system control flags that control operating mode and states of the processors (11 bits are used as flags, while the remaining are reserved). In an error scenario, a single-bit error disables protected mode operation and causes a general protection exception. Injections to the P4 stack pointer (*ESP*) cause either *NULL Pointer* or *Bad Paging* errors.
- Few crashes on the P4 are due to *Invalid TSS (Task-State Segment)*. All these events are caused by the corruption of the *NT* (nested task) bit in the *EFLAGS Register*. The *NT* bit controls the chaining of interrupted and called tasks. Changing the state of this flag results in an attempt to return to an invalid task after an interrupt.
- On both platforms, injections to the system registers can cause crashes due to *Invalid Instructions*: 6% and 12% on the P4 and G4, respectively. On the P4 platform, invalid instructions occur because of injections to the kernel process instruction pointer (*EIP*), which may force the processor to access memory corresponding to a random place in an instruction binary.
- On the G4, most crashes due to *Invalid Instruction* are caused by errors in the register *SPR274*, which is one of several registers dedicated to general operating system use and used by the stack switch during exceptions. Corrupting the contents of this register can force the operating system to try executing from a random memory location that does not contain a valid instruction and to generate an *Invalid Instruction* exception. A small percentage of *Invalid instructions* are due to errors in the register *SPR1008*, which is used to control the state of several functions within the CPU, e.g., enabling the instruction cache or branch target instruction cache. In an error scenario (from our experiments), a single-bit error enables the branch target instruction cache when the content of the cache is invalid. As a result, the system crashes due to an invalid instruction exception.
- A small percentage of crashes on the G4 are due to a *Machine Check* exception being raised. These crashes are caused by corruption of the *Machine State Register (MSR)*, which defines the state of the processor. In particular, the two bits responsible for enabling/disabling the instruction address translation (*IR*) and data address translation (*DR*) are error-sensitive.

5.3 Code Injection

The results of the code section injections are shown in Figure 11. *Invalid Memory Access* is a characteristic of code injections on both platforms, although for the P4 they are nearly 20% higher: 50% (*Bad Area*) on the G4 versus 70% (*Bad Paging + Null Pointers*) on the P4. A plausible explanation is

that a bit error on the P4 can convert a single instruction into a sequence of multiple valid (but incorrect from the application standpoint) instructions. This is due to the variable length of instruction binaries on the P4 (CISC architecture). On the P4 platform, this error may subsequently lead to an invalid memory access (often a *NULL Pointer*); hence, more invalid memory accesses are observed. All instructions on the G4 have the same length (32-bit RISC architecture); hence, a single-bit error (in addition to a invalid memory access) is more likely to result in an invalid instruction.

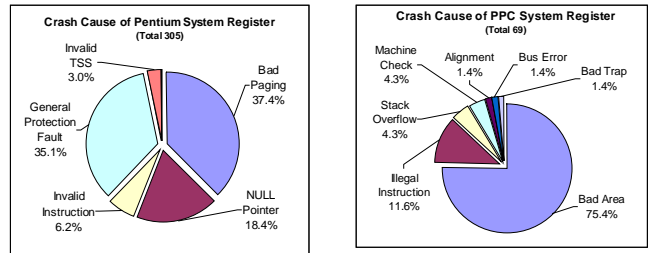


Figure 10: Crash Causes for System Register Injection

Stack Overflows resulting from code errors are a small fraction (about 5%) on the G4; compare this with the results of stack injections resulting in stack overflow (41.9%). This is most likely because the G4 has a large number of general-purpose registers (GPRs). However, very few of them (often *GPR1* and *GPR31*) are used to operate on the stack (e.g., to keep the stack pointer). Crashes due to stack overflow would require altering a register name used by a given instruction to operate on the stack. Keeping in mind the number of GPRs, the likelihood of stack overflow is low.

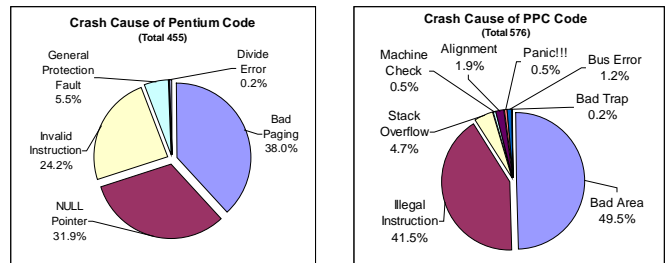


Figure 11: Crash Causes for Code Injection

The *Illegal Instruction* exceptions detected on the two platforms are also quite different: 24% on the P4 versus 41% on the G4. The difference is due in part to the inability of the P4 to diagnose many of instruction-level faults because of variable-length instructions (as explained earlier).

The preceding sections highlight the role of the hardware in detecting errors. Data injections, discussed next, highlight the operating system's detection capabilities, since the impact of errors in data must be detected at the operating system level.

5.4 Data Injection

Recall that only a small percentage of kernel data errors are activated. On both processors, the majority of manifested data errors (see Figure 12) result in invalid memory accesses: 89% (*Bad Area*) and 80% (*Bad Paging + Null Pointers*) for the G4 and P4, respectively. There is also a significant percentage of *Invalid/Illegal Instruction* cases: 9% and 18% for the G4 and P4, respectively.

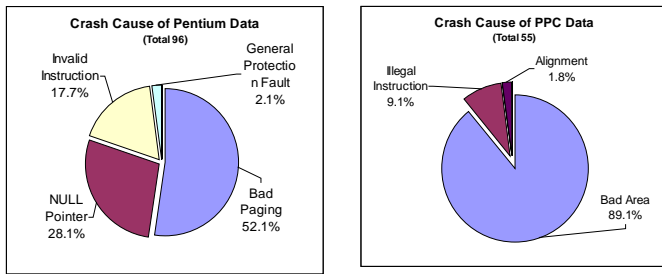


Figure 12: Crash Causes for Kernel Data Injection

The key reason that twice as many invalid instruction cases are observed on the P4 platform is that the Linux kernel (for both P4 and G4 architectures) raises the *Invalid Instruction* exception to handle a variety of error cases, some of which have nothing to do with invalid instructions. Figure 13 depicts how the kernel-checking scheme works. The *spin_lock/spin_unlock* function is frequently used to inspect kernel flags and decide whether to set a lock/unlock on kernel objects (e.g., *page_table_lock*, *journal_tatalist_lock*, *bdev_lock*, *arbitration_lock*). The flags are located in the kernel data section (*kernel_flag_cacheline*). In Figure 13, a bit error at address *0xc0375bc5* changes *4E* to *0E* ①. As a result, the compare (*cmpl*) instruction detects incorrect data ②, and an invalid instruction (*ud2a*) exception is raised ③. Since *spin_lock/spin_unlock* has a high rate of recurrence, this checking mechanism provides quick error detection.

The original cause of the errors, however, is not an invalid instruction, and the type of the exception raised may mislead the user. The results highlight the need for an operating-system-based checking scheme to provide better detection.

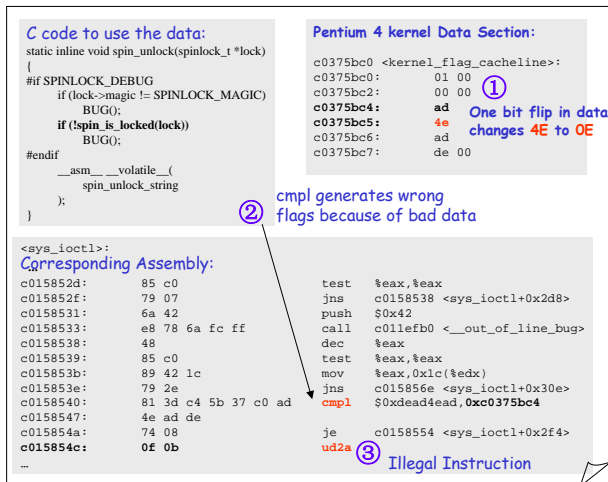


Figure 13: Illegal Instruction Crash (Due to Data Error) on the P4 Architecture

5.5 Summary

Several more generic conclusions can be drawn based on the analysis of crash data:

- The variable-length instruction format on the P4 leads to poorer diagnosability because a bit error can alter a single instruction into a sequence of multiple valid (but incorrect in the context of the application semantic) instructions. This error may subsequently lead to an invalid memory access and cause, e.g., a *NULL Pointer* exception, which can make locating the actual crash reason

difficult. All instructions on the G4 have the same length (32 bits); hence, a single-bit error is more likely to result in an invalid instruction.

- Though less compact, fixed 32-bit data and stack access makes the G4 platform less sensitive to errors. The sparseness of the data can mask errors. For example, the larger presence of unused bits in data items means altering any unused bit is inconsequential, even if the corrupted data instruction is used. The more optimized access patterns on P4 increase the chances that accessing a corrupted memory location will lead to problems.
- The data from stack overflow and the lazy detection of instruction corruption indicate that the P4 kernel (and possibly the hardware strategy) reduce the performance overhead that runtime error checks might incur but increases the risk of error propagation and the potential for serious damage to the system. In contrast, the G4 kernel is more likely to perform checking closer to an error's origin. While this approach may cost performance, it provides early detection and hence reduces the chance that errors will propagate.

6 Crash Latency (Cycles-to-Crash) Analysis

To explore further differences/similarities in Linux behavior on the two platforms, this section discusses crash latency. Figure 16 depicts the crash latency for all conducted error injection campaigns and provides details on crash latency distributions. The major findings can be summarized as follows:

- The majority (80%) of stack errors on the G4 platform are short-lived (less than 3,000 cycles). On the P4 platform, the majority (80%) of stack errors result in longer crash latency (3,000 to 100,000 cycles). The primary reason for this disparity is the way the two platforms handle exceptions. For example, the kernel on the G4 platform provides quick detection of stack overflow errors, while the kernel on the P4 architecture converts stack overflow events into other types of exceptions (e.g., *Bad Paging*), resulting in inherently slower detection.
- Errors impacting the kernel code show an opposite trend in crash latency (compared to stack errors). Here, 45% of the errors on the P4 platform are short-lived (less than 3,000 cycles), while about 50% of the errors on the G4 have latency between 10,000 and 100,000 cycles. This dissimilarity is due to the differences in the number of general-purpose registers provided by the two processors (32 on the G4; 8 on the P4).

Most system register errors are relatively long-lived (more than 10,000 cycles). This can be attributed to the fact that most of them are not visible to users and typically are not modified often.

Stack Injection. The crash latency for stack injections is given in Figure 16(A). About 80% of the crashes observed on the G4 platform are within 3,000 CPU cycles, whereas about 80% of the crashes on the P4 are in the range of 3,000 to 100,000 cycles. Crash cause analysis in Section 5.1 indicates that 40% of the G4 crashes are due to stack overflow. The G4

kernel employs a checking wrapper before executing a specific exception handler. This wrapper examines the correctness of the current stack pointer. If the stack pointer is out of kernel stack range (8Kb), the kernel raises a *Stack Overflow* exception and forces itself to stop as soon as possible. The wrapper executes before the exception handler is invoked, and as a result, the detection of the corrupted stack pointers is relatively fast. The kernel executing on the P4 does not support this wrapper mechanism. It allows an error to propagate and be captured by other exceptions, e.g., *Bad Paging* or *Illegal Instruction*, which may lengthen the time before the error is detected.

System Register Injection. As shown in Figure 16(B), 35% of the crashes on the G4 platform are within 3000 cycles. This is due mainly to errors in the *Machine State Register (MSR)*, which immediately crash the system. Crashes due to errors in the G4 kernel stack pointer and *SPR274* registers have latencies in the range between 10M and 100M cycles. On the P4 platform, 70% of crashes are within 10K cycles. These crashes are caused by errors in the *ESP*, *EIP* registers corresponding to the kernel process targeted by the injection. The longest crash latency (larger than 1G cycles) is due to errors in *FS* and *GS* registers.

Code Injection. Figure 16(C) summarizes the cycle-to-crash in kernel code section injections. The P4 architecture indicates a shorter latency (70% within 10,000 cycles) compared with the G4 architecture (almost 90% above 10,000 cycles). There are several reasons for these differences.

Figure 14 shows that a group of instructions in the P4 architecture may be transformed to another, totally different instruction group. Originally, five instructions starting with *mov* are altered to another five instructions starting with *xorl* through a bit-flip at address *0xc011e2a6*, which leads the kernel to crash at *0xc011e2b1* (“*crash here*”). Thus, errors in the code section of the P4 architecture may crash faster than those in the G4 architecture. Interestingly, poor diagnosability seems to lead to shorter error latencies in the code section.

The G4 architecture offers a much larger number of GPR registers (32) than the P4 processor (8). Function arguments are often passed using general-purpose registers, e.g., *GPR18* to *GPR29*. Registers *GPR3* through *GPR12* are volatile (caller-save) registers, which (if necessary) must be saved before calling a subroutine and restored after returning. Register *GPR1* is used as the stack frame pointer. As a result, values kept in a G4 register can potentially live longer before being used or overwritten. Hence it may take longer for an error to manifest and crash the system.

Figure 15 illustrates a bit error in the *sys_read()* function taken from the kernel code on the G4 platform. The error transforms the *mflr* instruction to *lhax*. The system crashes (kernel access of a bad area) due to an illegal address generated by *lhax: r0, r8, r0 (gpr8 + gpr0)*. Depending on the workload (and hence the values in *gpr8* and *gpr0*), crash latency may vary (see Figure 15).

Data Injection. Figure 16(D) shows that crash latency due to data errors is similar on both platforms. The error activation is small (about 1%) due to the large data space allocated for

the kernel despite its rather sparse usage. As a result, the latency distribution has a long tail, indicating that errors can stay in the system for seconds before being activated. Moreover, it is likely that many errors that did not manifest for the duration of the benchmark execution stay latent in memory and may impact the system later. To prevent crashes due to data corruption and to reduce error latency, assertions can be added to protect critical data structures.

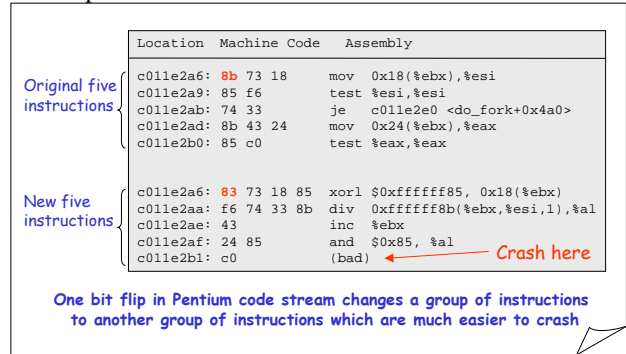


Figure 14: Bit Error Causing Change in Instruction Group (P4)

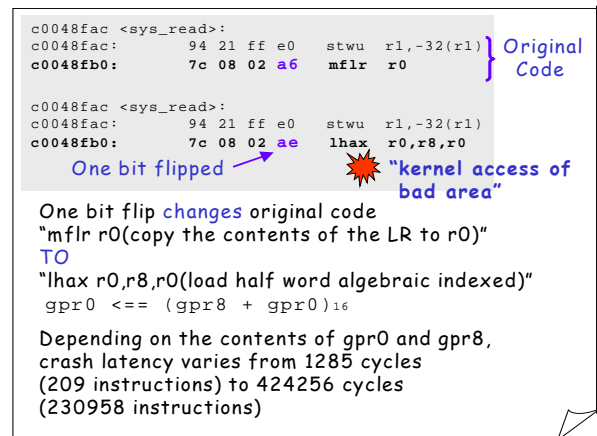
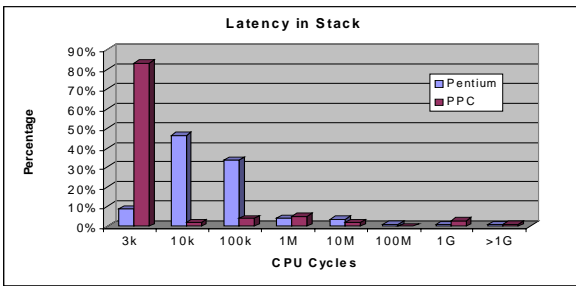


Figure 15: Variation in Crash Latency (G4)

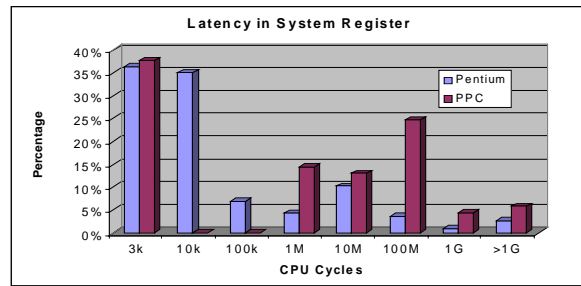
7 Conclusions

This paper describes a fault/error injection study conducted to analyze Linux kernel (2.4.22) behavior under errors on two target processors: the Intel Pentium 4 (P4) running RedHat Linux 9.0 and the Motorola PowerPC (G4) running Yellow-Dog Linux 3.0. The study’s findings indicate the following:

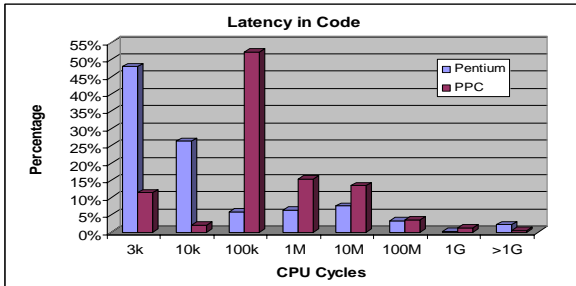
- Less dense data access of the stack and data segment makes the G4 platform less sensitive to errors. Although a similar number of errors are activated on both platforms, a much smaller number manifest on the G4 than on the P4. In comparison, the more optimized access patterns on the P4 ensure that if an error location is accessed, it is more likely to lead to problems.
- The variable-length instruction format of the P4 makes it possible for a bit error to alter a single instruction into a sequence of multiple valid (but incorrect from the application semantic point of view) instructions. While this may lead to poorer error diagnosability, it has the potential to reduce crash latency: executing an incorrect instruction sequence is likely to make the system fail quickly.



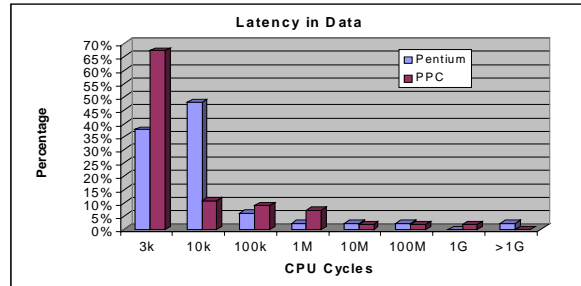
(A) Stack Error Injection



(B) System Register Error Injection



(C) Code Error Injection



(D) Data Error Injection

Figure 16: Distribution of Cycles-to-Crash

- Close analysis of the injections and their effects show that better interaction between the hardware and the operating system can improve error detectability and diagnosability. For example, stack overflow detection, not available on P4 kernel, could be added by extending the semantics of *PUSH* and *POP* instructions (already present in the IA-32 architecture) to enable checking for a memory access beyond the currently allocated stack.

Acknowledgments

This work was supported in part by Gigascale Systems Research Center (GSRC/MARCO) and in part by NSF grant CCR 99-02026. We thank Fran Baker for her insightful editing of our manuscript.

References

- J. Arlat, et al., Dependability of COTS Microkernel-Based Systems. *IEEE Transactions on Computers*, 51(2), 2002.
- J. Barton, et al., Fault Injection Experiments Using FIAT, *IEEE Transactions on Computers*, 39(4), 1990.
- R. Baumann, SEU Trends in Advanced Commercial Technology, *Workshop on Single Event Upsets in Future Computing Systems*, NASA-JPL, 2003.
- K. Buchacker, V. Sieh, Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects, *Proc. High-Assurance Systems Engineering Symposium*, 2001.
- J. Carreira, H. Madeira, J. G. Silva, Xception: A Technique for the Evaluation of Dependability in Modern Computers, *IEEE Transactions on Software Engineering*, 24(2), 1998.
- G. Carrette, CRASHME: Random Input Testing, 1996. <http://people.delphiforums.com/gjc/crashme.html>.
- A. Chou, et al., An Empirical Study of Operating Systems Errors, *Proc. Eighteenth ACM Symposium on Operating Systems Principles (SOSP'01)*, 2001.
- D. Edwards, L. Matassa, An Approach to Injecting Faults into Hardened Software, *Proc. of Ottawa Linux Symposium*, 2002.
- W. Gu, et al., Characterization of Linux Kernel Behavior under Errors, in *Proc. DSN-2003*, 2003.
- IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide, 2004.
- R. Iyer, D. Rossetti, M. Hsueh, Measurement and Modeling of Computer Reliability as Affected by System Activity, *ACM Transactions on Computer Systems*, Vol.4, No.3, 1986.
- W. Kao, et al, FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults, *IEEE Transactions on Software Engineering*, 19(11), 1993.
- Kernel Profiling, <http://oss.sgi.com/projects/kernprof>.
- P. Koopman, J. DeVale, The Exception Handling Effectiveness of POSIX Operating Systems, *IEEE Transactions on Software Engineering*, 26(9), 2000.
- I. Lee and R. K. Iyer, Faults, Symptoms, and Software Fault Tolerance in Tandem GUARDIAN90 Operating System, *Proc. FTCS-23*, 1993.
- Micron Tech Note: DRAM Soft Error Rate Calculations, TN-04-28, 1999. <http://download.micron.com/pdf/technotes/DT28.pdf>.
- B. Miller, et al., A Re-examination of the Reliability of UNIX Utilities and Services. *TR*, University of Wisconsin, 2000. <http://www.suffritti.it/informatica/tco/fuzz-revisited.pdf>.
- Motorola MPC7450 RISC Microprocessor Family User's Manual, 2004.
- W. Ng and P. M. Chen, The Systematic Improvement of Fault Tolerance in the Rio File Cache. *Proc. FTCS-29*, 1999.
- M. Rimen, J. Ohlsson, and J. Torin, On Microprocessor Error Behavior Modeling, *Proc. FTCS-24*, 1994.
- D. Stott, et al., Dependability Assessment in Distributed Systems with Lightweight Fault Injectors in NFTAPE, *Proc. 4th Computer Performance and Dependability Symposium*, 2000.
- M. Sullivan and R. Chillarege, Software Defects and Their Impact on System 118 Availability – A Study of Field Failures in Operating Systems. *Proc. FTCS-21*, 1991.
- UnixBench, <http://www.tux.org/pub/tux/benchmarks/System/unixbench>.
- J. Xu, Z. Kalbarczyk and R. K. Iyer. Networked Windows NT System Field Failure Data Analysis, *Proc. Pacific Rim International Symposium on Dependable Computing*, 1999.