The 21st International Symposium on Fault-tolerant Computing, 1991. p 2-9. IEEE Computer Society

Software Defects and their Impact on System Availability - A Study of Field Failures in Operating Systems

Mark Sullivan[1], Ram Chillarege

IBM Thomas J. Watson Research Center,
P. O. Box 704, Yorktown Heights, NY. 10598

Abstract

In recent years, software defects have become the dominant cause of customer outage, and improvements in software reliability and quality have not kept pace with those of hardware. Yet, software defects are not well enough understood to provide a clear methodology for avoiding or recovering from them. To gain the necessary insight, we study defects reported between 1986 and 1889 from a on a high-end operating system product. We compare a typical defect (*regular*) to one that corrupts a program's memory (*overlay*) given that overlays are considered by field services to be particularly hard to find and fix.

This paper:

- Shows that the impact of an *overlay* defect is, on average, much higher than that of a *regular* defect.
- Defines *error types* to classify the programming mistakes that cause software to fail.
- Defines *error trigger* to classify the events that cause latent errors in programs to surface. The error trigger distribution weights events and environments that are probably inadequately tested.
- Shows that boundary conditions and allocation management are the major causes of overlay defects, not timing.

(bu Shows that most overlays are small and not too far from their intended destination.

Further analysis are provided on defects in fixes to other defects, symptoms, and an assessment of their impact. These results provide a base line understanding useful to designers and developers. The data will also help develop realistic fault models for use in fault-injection experiments.

1. Introduction

Software failures have become a dominant cause of system unavailability. [Gray90] shows that in the the past 5 years the main cause of outage has shifted from hardware and maintenance failures to failures in software and to a lesser extent operations. In fact, improvements in hardware and maintenance shrank their contributions to outage from 50% to 10%, while those due to software grew from 33% to 60%. The trend is likely to continue given the rise in installed lines of code, dependency between vendor products, increased use of parallelism, and smaller timing windows. Clearly, the technology to detect and recover from software errors will play a critical role in increasing system availability.

Building highly available systems requires a combination of high quality software (i.e. few defects shipped), recovery mechanisms which mask any software errors that surface in the field, and the capability for non-disruptive maintenance. In order to build software for highly available systems, it is imperative that there exist a good understanding of field problems: the defects experienced in shipped code and their impact on the system. Without this understanding, design decisions and software engineering methods tend to be ad-hoc. A clear model of software errors is also necessary to test new systems effectively and to validate new designs. In this regard, fault-injection methods [Segall88] [Chillarege89] [Arlat89] have gained interest and are fast becoming a validation and evaluation methodology. Effective fault-injection requires a suite of injectable faults which accurately reflect system failure behavior.

There have been several studies of software errors; however, for the most part, they concentrate on the system development and test phases. [Endres75] studied software errors found during internal testing the DOS/VS operating system. His classification was oriented towards differentiating between high-level design faults and low level programming faults. Another early error study, [Thayer78] provides some of the same level of error analysis that our study provides, but on errors discovered during the testing and validation phases. [Glass81] provides another high level, specification-oriented picture of software errors discovered during the development process. [Basili84] studies the relationship between software errors and complexity. [Chillarege 91] provides an analysis of defects found during the test process and their impact on the growth curve.

Software errors experienced in the field have different characteristics from those detected during the system test phase. Environmental differences and workload variation often exercise defective code that the testing procedure has missed. Unfortunately, most of the published literature on field failures concentrates on reliability metrics and trends (for example, [Levendel89]) rather than error characteristics. A number of studies on field failure done at Stanford [Mourad87] [Iyer86] and CMU [Castillo81] used data from error logs to track system failures. Error log records are generated by the system after

[1] Mark was with IBM during the Summer of 90. He is currently working towards his Ph.D. at the University of California, Berkeley.

programs fail with an abnormal end (ABEND) completion code. Although error logs accurately count failure frequency, they give little semantic information about the error that caused the failure.

Understanding software errors often requires a careful inspection of the environment of failure, the traps and dumps used, and the programming changes that were required to fix the defect. This paper uses data tracked by IBM field service on software errors reported against a high-end operating system product. The text of the error report provides the extra level of detail necessary for good fault characterization.

The study analyzes two groups of software errors, **Regular** and **Overlay**, providing comparisons when relevant. The Regular software error group represents the typical software error encountered in the field. The other group, Overlay errors, is composed of software errors that resulted in a storage overlay. IBM field service uses the term ''overlay'' to describe corruption of program memory. A network protocol module, for example, could accidentally overlay a process control block with the contents of a message buffer.

The decision to single out overlays came in part from the opinion of experienced field service and development personnel. It is commonly accepted by service personnel that overlays are the hardest software errors to find and fix. It is also believed that they have a significant impact on system availability. An additional reason to better quantify overlay error characteristics is that fault-injection experiments commonly use fault models based on overlay errors.

The study shows that Overlay defects have a higher impact on average than Regular defects. Both groups of defects have been categorized by their **error type** and **trigger event**. The error type provides insight on the programming mistakes that cause field failures. The trigger illustrates the situations in a customer environment that allow the latent errors to surface. We also analyze failure symptoms and **bug fix** errors -- errors in fixes of earlier errors. Distributions are provided for each error categorization.

Section 2 describes the defect database, the data, and sampling technique used. Also described are the categories used for error type and trigger. Section 3 contains the results and a discussion of the errors. Section 4 summarizes some of the key findings of the paper.

2. Software Error Data from RETAIN

The data we use comes from an IBM internal field service database called REmote Technical Assistance Information Network (RETAIN). RETAIN serves as a central database for hardware problems, software problems, bug fixes, release information, etc. The database is available to IBM field service representatives world-wide.

Software error reports in IBM parlance are called Authorized Program Analysis Reports, or APARs for short. An APAR describes a software error recorded against a specific program product. The APAR report contains a few pages of text describing the symptoms of the problem, some context and environment information, and a description of the fix. In addition to the textual description there are fields that contain some standard attributes of the error. An error's impact on the customer can be estimated from these standard attributes:

Severity is a number between one and four. A severity one error (the highest) corresponds to a system outage and considerable impact to the customer. Severity two, also signifies damage, however, a circumvention or temporary solution to the error was readily available. Three and four correspond to lesser damage and can range from annoyance to touch and feel type problems.

HIPER Highly pervasive (HIPER) errors are flagged by the team fixing the error. HIPER software errors are those considered likely to affect many customer sites -- not just the one that first discovered the error. Flagging a error as HIPER provides a message to branch offices to encourage their customers upgrade with this fix.

IPL errors destroy the operating system's recovery mechanism and require it to initiate an Initial Program Load (IPL) or ''reboot.'' An IPL is clearly a high impact event since it can cause an outage of at least 15 minutes. This metric is probably the most objective of the impact measurements since there is little room for data inaccuracy. While labeling a error HIPER or severity one is a judgement call, the occurrence of IPL is difficult to mistake.

This paper uses error data from a high-end operating system for the period 1985-1989, representing several thousand machine years. It includes errors in the base operating system as well as a number of products that tend to be bundled with it, but does not include any major sub-systems such as database managers. Since we are primarily interested in errors that affect availability we have restricted the study to only severity 1 and 2 errors.

Software defects are similar to design errors, in that once fixed, the do not show up again - unless the fix is in error too. An APAR represents a unique software defect that escaped the testing process. It may be found several months after the product is in use and may only be triggered by a specific customer environment. Thus, APARs are different from a classicial hardware component which when replaced has yet a finite probability of failure and therfore can be associated with a failure rate. Associating a failure rate with an APAR is not meaningful, for once fixed, they do not reappear. The only variant is when the fix is in error and causes another defect.

2.1. Sampling from RETAIN

Data from APARs contain both keyed fields and free-form text. The error type and trigger event are identified by reading the text and categorizing the APAR. In order to limit the time required to complete the study, we used sampling to reduce the number of APARS to be read and categorized.

To analyze Regular and Overlay software errors, we needed to construct a group of representative APARs for each. The population of Regular errors is the complete set (approximately 3000) of severity 1 or 2 errors from 1986-1989 for the operating systems studied; thus, the Regular APAR group was a random sample from this population. To identify Overlay errors from this population, we searched the text parts of the APAR for strings containg words such as ''overlay,''

''overwrote,'' etc. which are commonly used in descriptions of overlay errors. This search yielded APARs that potentially overlays, but further reading is necessary to weed out ones that are not.

One of the problems in straight random sampling over the population of all severity 1 and 2 errors is that the severity 2 errors far out number the severity 1 errors. Thus, in any sample, there would be too few severity 1 errors to be categorized -- probably fewer than the number of categories. To overcome this problem, we pulled independent random samples from the population of severity 1 and 2. Each sample was large enough to allow the necessary categorization. We then combined the results from the severity 1 and 2 samples in the proportion they are represented in the population. We used boot-strapping [Efron86] to combine the samples rather than a simple weighted average. Boot-strapping has the advantage that when used to make confidence-interval estimates it does not depend on assumptions about the distribution of the parent population. In all, we classified 150 APARs in the Regular sample and 91 from the overlay sample. We estimate the proportion of the overlay errors to be between 15% and 25% of the severity 1 and 2 errors.

2.2. Characterizing Software Defects

There are many different ways to think about the cause of an error. Eventually, we chose the two described in the introduction as error type and error trigger. **Error type** is the low level programming mistake that led to the software failure. This classification had to describe the error well enough that it could serve as the basis for future fault injection experiments. The **error trigger** event classification was meant to give information about the environment providing insight to the testing process. When defective code survives the testing process and is released to customers, some aspect of the customer's execution environment must have caused the defective code to be executed. The trigger classification distinguishes several ways in which defective code could be executed at the customer site when the same defective code was never executed during testing.

To determine error and trigger classes, we made several passes through the sample looking for commonalities in the errors. The error classifications had to be orthogonal; each error report had to fit into only one class. The classification also had to have few enough classes so the confidence intervals on the data were acceptable. When classes were too small, we combined them into larger, more general classes.

2.3. Error Types

A few programming errors caused most of the overlays. These are:

Allocation Management: One module deallocates a region of memory before it has completely finished with it. After the region is reallocated, the original module continues to use it in its original capacity.

Copying Overrun: The program copies bytes past the end of a buffer.

Pointer Management: A variable containing the address of data was corrupted. Code using this corrupted address caused the overlay.

Register Reused: In assembly language code, a register is reused without saving and restoring its original contents.

Type Mismatch: A field is added to a message format or a structure, but not all of the code using the structure is modified to reflect the change. Type mismatch errors could also occur when the meaning of a bit in a bit field is redefined.

Uninitialized Pointer: A variable containing the address of data is not initialized.

Undefined State: The system goes into a state that the designers had not anticipated. In overlay errors, the bad state caused the program to mistake the contents of a memory region. For example, the designer may have assumed that pages are always pinned in memory when a certain routine is called. If the routine is called on an unpinned page, memory corruption can occur. For the Overlay data set, most of these undefined state errors had to do with management of page tables.

Unknown: The error report described the effects of the overlay (the part of memory that was overlaid), but not adequately for us to classify the error.

The Regular sample containing both overlay and non-overlay errors required a few additional error types:

Data Error: An arithmetic miscalculation or other error in the code makes it produce the wrong data.

PTF Compilation: Individual bug fixes are distributed together on a PTF (Program Temporary Fix) tape. Occasionally, a bug is repaired in one PTF but lost when the next one is compiled (a later bug fix is applied to an earlier version of the software).

Sequence Error: Messages were sent or received in an unexpected order. The system deferred an action, such as an acknowledgement message, but then forgot to execute the action.

Statement Logic: Statements were executed in the wrong order or were omitted. For example, a routine returns too early under some circumstances. Forgetting to check a routine's return code is also a statement logic error.

Synchronization: A error occurred in locking code or synchronization between threads of control.

Unclassified: We understood what the error was, but couldn't fit it into a category.

2.4. Software Error Triggering Events

This classification describes what allowed a latent error to surface in the customer environment. For every error in the sample, we determined the error's triggering event:

Boundary Conditions: Often software failures occur under limit conditions. Users can submit requests with unusual parameters (e.g. please process zero records). The hardware configuration may be unique (e.g. system is run with a faster disk than was available during testing). Workload or system configuration could be unique. (e.g too little memory for network message buffering).

Bug Fixes: An error was introduced when an earlier error was fixed. The fix could be an error that is triggered only in the customer environment, or the fix could uncover other latent

bugs in related parts of the code.

Client Code: A few errors were caused by errors in application code running in protected mode.

Recovery or Error Handling: Recovery code is notoriously difficult to debug and difficult to test completely. When an error is discovered, the system runs a recovery routine to repair the error. The recovery code could have errors.

Timing: Timing triggers are an important special case of boundary conditions in which an unanticipated sequence of events directly causes an error. An error that only occurs when the operating system is interrupted at an inopportune moment would be a timing-triggered error.

Unknown: The triggering event could not be determined from the available data.

2.5. Symptom Codes

When an APAR is opened a symtom code is entered into a field. This is often used by field personnel to search for other failures matching these symtpoms. Since this was a keyed field it did not require any classification effort. Failure symptoms fall into these classes:

ABEND: An abnormal program termination occurred. The currently running a application program fails and must be restarted.

Addressing Error: The operating system fails after trying to use a bad address. It should be noted that an addressing error, in this system, does not force a reboot as it does in most workstation operating systems.

Endless Wait: Processes wait for an event that will never occur.

Incorrect Output: The operating system produces incorrect output without detecting the failure.

Loop: The operating system goes into an infinite loop. IPL is required to restart the system.

Message: The operating system cannot perform the requested function but prints an error message on the screen and performs local recovery rather than ABENDing

3. Results

The earlier sections described the data source, sampling technique, and categorization of defects. Here we present the results and discuss their significance.

3.1. Impact of Overlay Errors

No single metric is best for evaluating the customer impact of different of regular and overlay defects. However, a collection of metrics provide different ways to compare the relative impact of the defect groups. Table 1 shows the percentage of APARs that caused IPLs, rated HIPER, or rated Severity 1. By each of these measures, overlay defects clearly have a higher impact than regular defects. This finding concurs with the perception of IBM's field service people.

Impact Metric	Overlay Sample	Regular Sample
IPL (reboot)	19.8	6.3
HIPER	18.7	5.2
HIPER or IPL	30.8	10.8
Severity 1	17.6	12.6

Table 1. Error Impact: Overlay vs. Regular Sample

In order to understand why overlay errors have high impact, let us first re-examine what an overlay defect is in greater detail. An *overlay* is called such because the post-mortem of the failure reveals a part of the program storage was corrupted. For example, a network protocol module can overlay a control block with the contents of a message. After the overlay, there are potentially two failures: one due to the lost message and another due to corrupted information in the control block. Often the module using the corrupted data, rather than the one containing the original error, causes the system to fail. Thus, overlays, in addition to being hard to track down, also cause propagation of errors.

Recovery is usually set up to deal with problems that could be encountered during the execution of the module and some unexpected status conditions. The recovering subsystem usually tries to reinitialize itself and re-execute the failed operation. If retry fails, a higher level subsystem attempts to recover from the error. A propagated error resulting in the loss of key control information can often defeat the established recovery mechanisms, accounting for the higher IPL counts in the Overlay sample.

Overlays caused by the operating system are often within the operating system (as we will see shortly). It has also been shown through experiments [Chillarege 89, Chillarege 87] that such errors in the operating system can remain latent for a long time. Large latency provides greater opportunity for propagation, resulting in possibly multiple error conditions that are hard to recover from. It seems that change teams recognize this possibility which accounts for the higher HIPER counts.

3.2. Characterizing Overlay Errors

Table 2 shows the breakdown of overlay error types. Each row in the table represents one of the error types defined in sections 2.2.2. The columns are the fraction of all APARs, HIPER APARs, and IPL APARs caused by each type of error. Note that a defect may be counted both under HIPER and IPL.

	Percent of		
Defect Type	all APARs	HIPERs	IPLs
Allocation Mgmt.	19	31	17
Copying Overrun	20	13	5
Pointer Mgmt.	13	16	27
Register Reused	7	6	11
Uninitialized Ptr.	5	12	0
Type Mismatch	12	10	0
Undefined State	4	0	17
Unknown	13	0	5
Synchronization	8	12	17

Table 2. Overlay Sample Error Types

The most common error types were memory allocation errors, copying overruns, and pointer management errors. Together, these three classes accounted for more than half of the total. The error types with the highest impact were memory allocation and pointer management errors, which together accounted for about half of the high impact errors.

An interesting result is that memory allocation errors were more likely than locking or synchronization errors to cause overlays in shared memory. For example, a process can request a software interrupt and then free a region of memory before the interrupt is scheduled. If the interrupt tries to use this freed memory, an overlay occurs. While the garbage collection did not work correctly, synchronization is correct; the interrupt is not scheduled while the original process is using the memory region. The most common synchronization errors occurred when interrupt handlers corrupted linked list data structures.

Mismanagement of address data (pointers) caused a significant fraction of errors and many of the high impact errors. It makes sense that a high proportion of operating system errors would affect addresses since much of what the operating system does is manipulate addresses. Also, corrupted pointer data structures are difficult to repair, making IPL necessary.

Although allocation management, pointer management and copying overrun have about the same number of error reports filed against them, copying overruns have low impact. Many of these errors appeared in the terminal I/O handling code or in code for displaying messages on the console. Copying overruns were often caused by overflows or underflows of the counter used to determine how many bytes to copy. Many others were "off-by-one" errors. In network-management code and terminal I/O handlers, buffers are processed slightly and passed from one routine to another. If the offset to the beginning of valid data or the count of valid bytes is corrupted, copying overruns occur. Most copying overruns involved only a few bytes. The few overruns which had high impact, however, caused massive corruption of memory.

The few overlay errors caused when the system went into an undefined state were fairly severe. For the most part, these errors occurred in page fault handling. When the page fault handler became confused about a process state, the process eventually corrupted so much of the system that no recovery was possible. The errors were often extremely complex. The reports usually listed a long chain of separate events and propagations that had to occur before the failure would occur.

Recovery system designers and fault-injection experimentors both need two pieces of information about overlays to develop fault models: the overlay's size and its distance from the correct destination address. Table 3 shows the average size of an overlay in bytes. Note that most overlays are small, nearly half are less than 100 bytes. Table 4 gives a rough ''distance'' between the overlaid data and the area that should have been written. For example, a copying overrun error corrupts data immediately following the buffer that the operating system is supposed to be using, hence, has distance ''Following data structure.'' An example of the distance type ''Within data structure'' is a type mismatch error in which the operating system overlays a field of the same structure it intends to update.

Overlay Size	Percent of Overlay APARS
Less than 100 bytes	48.4
100 to 256 bytes	25.3
One or more pages	4.4
Unknown size	22.0

Table 3. Average Size of an Overlay

Summarizing the size and distance tables, we find that most of the overlays are small with a vast majority of them close to their intended destination. Only about a fifth were ''wild stores'' that overwrite distant, unrelated areas of storage. This creates a challenge for fault-detection mechanisms and for recovery techniques. The smaller the granularity of overlays, the more expensive the protection mechanism, either hardware or software.

3.3. Characterizing the Regular Sample

The regular sample corresponds to defects that are representative of the typical defect in this product. Recall that the regular sample was drawn from all severity 1 and 2 defects. In all we classified 150 APARs. Table 5 summarizes the types of errors found in the regular sample. As in the table 2, each row represents one of the error types defined in sections 2.2.2 and 2.2.3. The first column shows the total percentage of APARs attributed to the error type. The second and third show the fraction of HIPERs and IPLs caused by the error type. Note that a defect may be counted both under HIPER and IPL.

The error type distribution for the Regular sample breaks errors down fairly evenly into twelve classes (plus a class for

Overlay Distance	Percent of Overlay APARS
Following data struct	30.8
Anywhere in storage	18.7
Within data struct	26.4
Unknown	24.2

Table 4. Distance Between Intended Write Address and Overlaid Address

	Percent of		
Defect Type	all APARs	HIPERs	IPLs
Allocation Mgmt.	7	4	0
Copying Overrun	2	0	4
Pointer Mgmt.	9	0	0
Uninitialized Ptr.	8	10	0
Deadlock	5	0	58
Data Error	6	5	3
PTF Compilation	8	0	0
Register Reused	3	4	3
Sequence Error	5	0	0
Statement Logic	7	4	0
Synchronization	9	0	22
Type Mismatch	1	0	0
Undefined State	12	49	6
Unknown	9	0	3
Other	10	24	0

Table 5. Regular Sample: Error Types

Unknown error types and an Other class combining several unclassifiable errors). These twelve classes can be regrouped into three larger classes: overlay and overlay-like errors, concurrency-related errors, and administrative errors. Important error types from the overlay sample -- pointer management, copying overrun, pointer initialization, and memory allocation errors -- together make a fairly large sub-group (24 percent) of the regular sample. Many of these were not identified in the text as overlay errors, but were errors which probably involved overlays.

Many of the non-overlay errors were concurrency-related. Common errors include deadlocks 5 percent, sequence errors 5 percent (programmer assumes that external events will always occur in a certain order), undefined state 12 percent (programmer assumes an external event will never occur), and synchronization errors 8 percent. Among the non-overlay errors, another large class of errors are administrative. PTF compilation errors are mistakes made in the error fix distribution rather than errors in the code itself.

3.4. Error Triggering Events

This section characterizes the events that make latent errors surface in code that has passed through system test. A defect in the field can be found months after a product has been in use. When such latent defects do surface it is usually due to stress or environmental conditions that allow them to surface. Clearly, these conditions are not exercised during system test, or they would be detected earlier. The *trigger* is meant to capture the condition that causes defective code to be executed. The trigger, thus, provides insight into areas which additional system test effort could help decrease the defect exposure. Tables 5 and 6 show the breakdown of the regular and overlay samples by error trigger. This time, the rows represent errors attributable to each trigger type defined in section 2.2.4.

Conventional wisdom about software failures in the field is that most are caused by timing-related problems. Because it is impossible to test all possible interleavings of events before the software is released, failures might occur when an untested interleaving occurs after months or years in the field. Our data partially supports this hypothesis. Bad timing triggered 12 percent of the Overlay sample and 11 percent of the Regular sample APARs.

Recovery code is also difficult to test so one would expect many of the field failures to be triggered by recovery. In fact, recovery accounted for 21 percent of the Overlay sample's error triggers and the largest fraction (35/38 percent) of its high impact errors. Recovery seemed to be less important to the Regular sample -- accounting for roughly the same fraction as timing.

	Percent of		
Trigger Event	all APARs	HIPERs	IPLs
Boundary Cond.	24	22	23
Bug Fix	20	24	5
Recovery	21	35	38
Timing	12	19	28
Unknown	17	0	0
Customer Code	6	0	6

Table 6. Overlay Sample: Error Triggering Events

	Percent of		
Defect Trigger	all APARs	HIPERs	IPLs
Boundary Cond.	34	56	4
Bug Fix	16	31	3
Customer Code	2	0	0
Recovery	13	5	31
No Trigger	12	0	0
Timing	11	8	59
Unknown	13	0	3

Table 7. Regular Sample: Error Triggering Events

A surprising result in both samples is that boundary conditions accounted for the largest fraction of RETAIN's errors (34 percent) and a high percentage of its HIPER errors (56 percent). Among overlay errors, boundary conditions had much lower impact, but still accounted for a quarter of the errors studied. Boundary conditions are the type of error that one would expect testing to detect most easily. In fact, many unanticipated boundary conditions continue to arise after the software is released.

Code reuse can partially explain the high incidence of errors triggered by boundary conditions. Programmers often use the services provided by an old module rather than write new ones with slightly different functionality. Over time, some modules are used for things the original designer never considered. While this increases productivity, it also lessens the effectiveness of module-level testing. The tests run on the old module by the original programmer do not stress aspects of the module used by newer clients.

3.5. Errors Introduced During Bug Fixes

Not every fix to a defect is perfect. Sometimes, the fixes themselves have defects, which in this paper are referred to as bug fix defects. Tables 8 and 9 show the impact of errors in bug fixes, for the overlay and regular samples respectively. For these charts, we removed all errors representing non-fix errors from the samples. The ones that remain have a much different error type distribution than the original sample.

For the Overlay sample, many errors in bug fixes were type mismatches. In these errors, a message format or data structure originally had one organization. The bug fix changed the organization, adding or changing the use of a field in the

	Percent of		
Defect Type	all APARs	HIPERs	IPLs
Allocation Mgmt.	12	21	0
Copying Overrun	17	22	0
Register Reused	17	19	100
Synchronization	6	0	0
Type Mismatch	31	17	0
Uninitialized Ptr.	18	20	0

Table 8. Overlay Sample: Bug Fix Error Types

message or structure. Later, execution of other modules turned up implicit assumptions about the message format or the data structures. For example, the programmer used an unused byte in a message header only to find after installation of the fix that another module assumed that this byte was zero under some circumstances. The Regular sample had few type mismatch errors of any kind.

Uninitialized Pointers and Register Reuse errors were also frequently caused by bad bug fixes. Some of these arose when the routine's parameters were changed without changing all of the code that called the routine. In both these cases, better tools for keeping track of cross references between routines would improve the failure rate of error repairs.

Overlay bug fixes have a fairly high HIPER rate, but are not a factor in many IPLs. There are several possible reasons for the high HIPER rate in bug fixes. One consideration is that many customers will not have installed the fix by the time the error is discovered. Flagging the second fix as HIPER will prevent other systems from ever exposing themselves to the error. Non-fix errors are probably already installed at most customer sites by the time they are discovered. If that is the case, HIPER may not be the best measure of bug fix impact. Using IPL as an impact measure, errors in bug fixes have low impact. In the Regular sample, bug fixes have little impact by either metric.

3.6. Failure Symptoms

When an APAR is opened, the symptom of the failure is recorded. Tables 10 and 11 summarize the symptoms of the failure that occurred when code containing errors was executed. An important observation from the symptom chart is that only 39 percent of overlay errors are detected as addressing violations. This suggests that the subsystem damaged by an overlay uses the corrupted data before failing, hence has an opportunity to propagate the error. The failing system does not always immediately take an address fault when it uses corrupted memory.

	Percent of		
Defect Type	all APARs	HIPERs	IPLs
Allocation Mgmt.	5	12	0
Copying Overrun	5	0	0
Pointer Mgmt.	7	0	0
Uninitialized Ptr.	7	0	0
Data Error	1	13	100
PTF Compilation	21	0	0
Register Reused	11	0	0
Sequence Error	5	0	0
Statement Logic	5	0	0
Synchronization	0	0	0
Undefined State	11	74	0
Unknown	5	0	0
Other	6	0	0

Table 9. Regular Sample: Bug Fix Error Types

	Percent of		
Failure Symptom	all APARs	HIPERs	IPLs
ABEND	33	29	22
Address Error	39	38	17
Incorrect Output	14	5	17
Infinite Loop	5	18	22
Error Message	3	5	0
Endless Wait	5	6	22

Table 10. Overlay Sample: Failure Symptoms

As expected, overlay errors are more likely to cause addressing faults than other errors. Non-overlay errors are more likely to cause the system to go into endless wait states. The common non-overlay error types -- synchronization, sequence error, and undefined state -- often appear in network and device protocols. The failures caused by these errors often cause processes to wait for events which never happen.

Non-overlay errors are also more likely to cause incorrect output than overlay errors. Incorrect output failures include jobs lost from the printer queue or garbage characters written into console messages. None of the errors we saw caused failures which corrupted user data.

4. Summary

This paper uses five years of field data on software defects to develop a taxonomy of defects, providing insight into their behaviour and impact. The data comes from IBM's field service database called RETAIN. This paper focuses on those software defects reported against a specific (unnamed) high-end operating system product.

This study is performed in the backdrop of a computer industry faced with tremendous challenges in software reliability, quality, and availability. Recent studies have demonstrated that while, in the past five years hardware reliability has made tremendous improvements, software has not. Unless software reliability improves, it will limit the total reliability and availability possible in computer systems.

Software errors found in the field are fundamentally different from classical hardware errors. Like hardware design errors, once fixed, they will not reappear. It is important to understand the type of software errors that remain undiscovered after system test and the conditions in a customer environment that allow them to surface. We have chosen to call these attributes, the *error type* and the *trigger*, respectively. This paper provides distributions of both the error type and the trigger, and provides customer impact information for each of these attributes. The paper focuses on a particular type of defect called the *overlay* by field service personnel -- errors which corrupt

	Percent of		
Failure Symptom	all APARs	HIPERs	IPLs
ABEND	21	5	3
Address Error	21	5	14
Incorrect Output	27	53	6
Infinite Loop	0	9	0
Error Message	17	0	18
Endless Wait	11	29	59

Table 11. Regular Sample: Failure Symptoms

program memory. The overlay defect is contrasted to the typical defect, herein referred to as the *regular* defect.

The study finds:

(1) Overlay defects have, on the average, a much higher impact on the system than regular defects. This is measured by its probability of causing an IPL, its probability of achieving a severity 1 rating, and its probability of being flagged as ''highly pervasive'' across the customer base.

(2) Most overlay defects are due to boundary condition and allocation problems. Contarary to popular folklore, they are less likely to result from timing or synchronization problems.

(3) Most overlays are small (order of a few bytes) and occur near the address the software was supposed to write. Less than a fifth of the overlays cause wild stores in a process address space.

(4) Non-overlay defects are dominated by *undefined state* errors in which the module implementing a network or device protocol mistakes the current protocol state and goes into a wait or deadlock state.

(5) Untested boundary conditions in the software trigger a majority of failures. Recovery and timing-triggered failures have slightly higher impact than failures triggered by boundary conditions.

(6) Among errors in fixes to other errors the causes are related to mismatch in data types and interfaces.

(7) While overlay errors are more likely to cause addressing faults than non-overlay errors, most overlay errors do not cause the system to take an address fault. That suggests that the corrupted data can actually be used before the failure occurs, making error propagation more likely.

The above list summarizes some of the salient findings. It is also the intent of this paper to provide a more structured and systematic method to classify and understand software defects. This understanding is critical to designing appropriate techniques to shield against them -- either in development or operation. Furthermore, the paper provides a framework from which fault-models for fault-injection based evaluation can be developed.

Acknowledgments

This research required the co-operation and help of numerous people and organizations. Al Garrigan, Dave Ruth, Hakan Markor, Jennifer Gors and Darius Baer have been especially helpful.

5. Bibliography

[Arlat89] J. Arlat, Y. Crouzet, and J. C. Laprie., Fault Injection for Dependability Validation of Fault-Tolerant Computing Systems., *Proc. 19th Fault Tolerant Computing Symp.*, pages 348-355, 1989

[Basili84] V. R. Basili and B. T. Errricone., Software Errors and Complexity: An Empirical Investigation. *Comm. of the ACM*, 27(1), 1984.

[Castillo81] X. Castillo and D. P. Siewiorek. Workload, Performance and Reliability of Digital Computing Systems. *Proc. 11th Fault Tolerant Computing Symp.*, 1981

[Chillarege89] R. Chillarege and N. S. Bowen. Understanding Large System Failure -- A Fault Injection Experiment. *Proc. 19th Fault Tolerant Computing Symp.*, pages 356-363, 1989

[Chillarege 91] R. Chillarege, W. Kao, R. Condit, "Defect Type and its Impact on the Growth Curve, Proceedings International Conference on Software Engineering, May 1991, Austin Tx.

[Efron86] B. Efron and R. Tibshirani. Bootstrap Methods for Standard Errors, Confidence Intervals, ... *Statistical Science*, 1(1), 1986.

[Endres75] A. Endres. An Analysis of Errors and Their Causes in Systems Programs. *IEEE Trans. on Software Engineering*, SE-1(2):140-149, 1975

[Glass81] R. Glass. Persistent Software Errors. *IEEE Trans. on Software Engineering*, SE-7, March 1981

[Gray90] J. Gray. A Census of Tandem System Availability between 1985 and 1990 *IEEE Trans. on Reliability*, 39(4):409-418, Oct 1990.

[Iyer86] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh Measurement and Modeling of Computer Reliabiltiy as Affected by System. *ACM Trans. on Computer Systems*, Aug. 1986.

[Levendel89] Y. Levendel. Defects and Reliabilty Analysis of Large Software Systems: Field. *Proc. 19th Fault Tolerant Computing Symp.*, pages 238-243, 1989

[Mourad87] S. Mourad and D. Andrews. On the Reliability of the IBM MVS/XA Operating System. *IEEE Trans. on Software Engineering*, 13(10):1135-1139, Oct 1987

[Segall88] Z. Segall, D. Vrsalovic, et. al. FIAT -- Fault Injection Based Automated Testing Environment. *Proc. 18th Fault Tolerant Computing Symp.*, pages 102-107, 1988

[Thayer78] T. Thayer, M. Lipow, and E. Nelson. Software Reliability. *TRW and North-Holland Publishing Company*, 1978