Dependability Analysis of Virtual Memory Systems

Lakshmi N. Bairavasundaram
Andrea C. Arpaci-Dusseau   Remzi H. Arpaci-Dusseau

Computer Sciences Department
University of Wisconsin, Madison
1210 West Dayton St., Madison, WI-53706
{*laksh, dusseau, remzi*}*@cs.wisc.edu*

Abstract

Recent research has shown that even modern hard disks have complex failure modes that do not conform to "fail-stop" operation. Disks exhibit partial failures like block access errors and block corruption. Commodity operating systems are required to deal with such failures as commodity hard disks are known to be failure-prone. An important operating system component that is exposed to disk failures is the virtual memory system. In this paper, we examine the failure handling policies of different virtual memory systems for different classes of partial disk errors. We use type and context aware fault injection to explore as many of the internal code paths as possible. From experiments, we find that failure handling policies in current virtual memory systems are at best simplistic, and often inconsistent or even absent. Our fault injection technique also identifies bugs in the failure handling code in these systems. The study identifies possible reasons for poor failure handling, which can help in the design of a failure-aware virtual memory system.

1. Introduction

Modern commodity operating systems cannot assume that disk drives either work perfectly or fail perfectly. Even modern hard disks are far from being perfect. They do not operate in "fail-stop" fashion: they exhibit complex partial failures, in which a set of blocks may be inaccessible [11, 20] or the data stored in some blocks may be silently corrupted [5, 15]. The complex causes of these errors are still under study [28]. Worse, such errors are not expected to diminish as disk technology improves. Increase in disk drive complexity [4], and increased use of low-cost, unreliable ATA disks mean that the incidence of such errors could increase. Therefore, commodity operating systems should be equipped to deal with partial disk failures. While high-end systems have typically employed mechanisms to deal with disk faults by using techniques like checksumming [5] and disk scrubbing [27], commodity operating systems do not have explicitly specified failure handling mechanisms and policies.

Recent work [26] has explored the failure handling policies of commodity file systems. In this paper, we explore the failure handling capabilities of virtual memory systems, an integral part of any modern operating system, and like file systems, a significant user of disk storage.

We use *type* and *context* aware fault injection techniques to elicit the failure handling policies of the virtual memory systems of two operating systems, Linux 2.6.13 and FreeBSD 6.0. We also perform a preliminary study of the Windows XP virtual memory system. We characterize the policies of these systems based on the *Internal RObustness* (IRON) taxonomy proposed in earlier work [26].

We find that these virtual memory systems are not well-equipped to deal with partial failures. Like the file systems studied earlier, the virtual memory systems use policies that are illogically inconsistent and their failure handling routines have bugs. In most cases, the failure handling policy is simplistic, and in some cases, even absent. This disregard for partial disk failures leads to many problems, ranging for loss of physical memory abstraction, to data corruption, and even to system security violations.

The paper is organized as follows. Section 2 provides a background on partial disk failures, virtual memory systems and the IRON taxonomy. Section 3 describes our fault injection and analysis methodology. Section 4 presents experimental results and analyzes the failure handling approaches of the systems. Section 5 discusses related work and Section 6 concludes.

2. Background

This section first discusses partial failures in commodity hard disks, then provides background on virtual memory systems, and finally presents the taxonomy of failure handling policies used in the paper to characterize failure handling in virtual memory systems.

2.1. Partial disk failures

This section presents different causes of partial failures in the disk subsystem and discusses a suitable failure model for disks. Almost all layers of the storage stack contribute to the partial failures exhibited by the disk subsystem. The causes range from classic problems such as media errors due to "bit rot" or head crashes, to errors in bus controllers [34], and to more recent problems arising from buggy firmware code [31]. Additionally, it has been shown that device drivers are likely to contain more bugs than the rest of the operating system [12, 10, 29]. The entire range of sources of disk failures has been documented in a recent paper [26], which also proposes a failure model for disks called the Fail-Partial Failure Model. We adopt this model for injecting disk errors in this study. Our study makes use of the following aspects of the fail-partial failure model:

- **Types of errors**: Partial disk failures can cause (a) I/O errors when blocks are read or written; that is, an error code is returned by the disk, or (b) block corruption, wherein the contents of a disk block read by the operating system is altered and no error code is returned by the disk.
- **Transience of errors**: Disk failures can be permanent ("sticky") or temporary ("transient"). In the case of a transient failure, there are no errors if the same I/O operation is performed again.

The failure model does not incorporate specific frequencies for the different error types since data on the frequency of partial disk failures is scarce. Drive manufacturers are loathe to provide such information [3, 30]. Schwarz et al. estimate that partial disk errors may occur five times more often than absolute disk failures [27]. More recent experiments by Gray and Ingen [14] with SATA disk drives uncovered 30 uncorrectable read errors from the point of view of the operating system in a 6-month experiment period.

Given that partial disk failures occur and should be dealt with, the question arises as to what component should deal with the failures. For instance, one may argue that disk mirroring can be used to deal with such errors. But, our belief is that operating system components cannot rely on mirrored disks to provide a dependable computing environment. Moreover, possible component-specific policies and optimizations cannot be employed when simple mirroring is used. Therefore, each operating system component that uses disks should include its own failure handling policy.

2.2. Virtual memory systems

A virtual memory system uses disk storage to provide applications with an address space larger than available physical memory. This helps the system execute multiple processes with large address spaces simultaneously. The disk area used by the virtual memory system is called *swap space*. The virtual memory system uses swap space to store memory pages that are not expected to be of immediate use. Typically, systems tend to remove pages that have not been accessed recently or that are not accessed frequently from memory and store them on disk (called *page-out*). When a page stored on disk is accessed again, it is brought back into physical memory (called *page-in*). The page-out/page-in process is transparent to applications (except for performance effects). Thus, the virtual memory system is responsible for handling disk errors and maintaining the illusion that the page is actually in physical memory.

Virtual memory systems make use of file systems in two situations. First, instead of directly using on-disk space, swap space can also be maintained as a file in a file system. Second, virtual memory systems allow applications to memory-map file data (*e.g.* using the *mmap* system call). When a file (or a portion of a file) is memory mapped, applications can operate on file data as if they were memory locations. User code pages are also memory-mapped from the executable file when a program is executed. In situations involving a file system, the virtual memory system depends on the file system to recover from or propagate disk errors.

The following subsections outline the features of two virtual memory systems, Linux 2.6.13 and FreeBSD 6.0, whose failure handling policies have been studied in this paper. The features of the Windows XP virtual memory system will be discussed with its evaluation in Section 4.1.4.

2.2.1. Linux 2.6.13

The Linux 2.6.13 virtual memory system has largely been derived from the previous Linux versions. It performs swapping only for user-mode pages [7]. User-mode pages are the data, stack, and code pages that form the user process. In order to keep the virtual memory system simple, pages that belong to the kernel are not paged out. This simplification is not highly restrictive as kernel pages occupy only a small portion of main memory. The page replacement algorithm used is similar to the "2Q" algorithm [18]. When paged-out pages are accessed, space is created for the pages and they are read from disk. The system also issues reads in advance (*i.e.*, *read-ahead*) based on application accesses to improve performance. The swap area can either be a separate disk partition or a file in a file system. It contains a *swap header* that has information about the swap area like number of blocks, a list of faulty blocks and so on.

Level	Technique	Comment
D_{Zero}	No detection	Assumes disk works
$D_{ErrorCode}$	Check return codes from lower levels	Assumes lower level can detect errors
D_{Sanity}	Check data structures for consistency	May require extra space per block
$D_{Redundancy}$	Redundancy over one or more blocks	Detect corruption in end-to-end way

Table 1. The Levels of the IRON Detection Taxonomy. *The table describes the different levels of Detection in the IRON taxonomy. These levels were developed in an earlier paper [26].*

Level	Technique	Comment
R_{Zero}	No recovery	Assumes disk works
$R_{Propagate}$	Propagate error	Informs user
$\mathbf{R_{Record}}$	**Record that operation did not succeed**	**Prevents dependent actions from proceeding**
R_{Stop}	Stop activity (crash, prevent writes)	Limit amount of damage
R_{Guess}	Return "guess" at block contents	Could be wrong; failure hidden
R_{Retry}	Retry read or write	Handles failures that are transient
R_{Repair}	Repair data structs	Could lose data
R_{Remap}	Remaps block or file to different locale	Assumes disk informs VM system of failures
$R_{Redundancy}$	Block replication or other forms	Enables recovery from loss/corruption

Table 2. The Levels of the IRON Recovery Taxonomy. *The table describes the different levels of Recovery in the IRON taxonomy. These levels were developed earlier [26]. The level* $\mathbf{R_{Record}}$ *has been added in this paper.*

2.2.2. FreeBSD 6.0

FreeBSD [1] is an open source operating system derived from BSD UNIX. The design of the virtual memory system in FreeBSD is originally based on the Mach 2.0 virtual memory system, with considerable updates over the years. The FreeBSD 6.0 virtual memory system allocates pages when requested from a free list of pages and it maintains sufficient free pages by paging out less frequently used (inactive) pages [23]. The FreeBSD virtual memory system also provides for paging out entire processes. This implies that in addition to user-mode pages, the kernel thread stacks of processes can be paged out and page tables can be freed when the system is under extreme memory pressure [23]. Unlike Linux, the FreeBSD virtual memory system does not perform extra read-ahead; that is, it does not issue separate block read commands, although it tries to read as many as 8 blocks as part of one read command for a block that is needed. Like in Linux, the FreeBSD swap area can either be a disk partition or a file. The FreeBSD swap area does not have any data structures like the Linux swap header.

2.3. Failure Handling Policy Taxonomy

In this paper, we extend the IRON taxonomy proposed in an earlier paper [26]. The taxonomy was originally designed to describe the failure handling policies of file systems, but we find that it is applicable to virtual memory systems as well. The IRON taxonomy consists of two axis: Detection and Recovery. Table 1 and Table 2 describe the different levels of Detection and Recovery respectively. In addition to the levels proposed previously, we include a new recovery level: R_{Record}. At this level, the system records that the I/O operation did not succeed. This level of recovery prevents the system from performing any action that assumes successful completion of the I/O operation. For example, when a write error is detected and the system recovers using R_{Record}, it does not free the "dirty" memory page assuming that it has been successfully written out to disk, thus avoiding data loss.

We also extend the IRON taxonomy by adding a third axis: Prevention. The Prevention axis encompasses techniques used to avoid loss due to partial disk failures. Table 3 describes different levels of Prevention:

• *Zero:* The system may not use any special prevention techniques. In this case, the system assumes either that the disk works or that errors can be dealt with when they occur.

• *Remember:* A basic prevention strategy that can be used is to remember that a specific block is "bad" once the system has had at least one bad experience in using the block. This strategy could prevent future data loss.

• *Reboot:* A phenomenon that has been observed for a long time is that systems are either less likely to fail or faults are cured if the systems are rebooted or reinitialized [9] (since the systems can be rid of effects of transient bugs accumulated over time). This fact can be used as a failure prevention strategy by periodically rebooting subsystems [17]. The rebooting strategy for virtual memory systems could range from disabling and then enabling a swap area periodically to even re-initializing the drivers/disk controllers.

• *LoadBalance:* This prevention technique attempts to reduce the wear on the data blocks by balancing the load on

Level	Technique	Comment
P_{Zero}	No prevention	Assumes disk works
$P_{Remember}$	Remembers disk errors	Prevents usage of blocks with errors
P_{Reboot}	Periodically re-initializes the system	Tries to avoid bugs due to excess state
$P_{LoadBalance}$	Balances the read/write load on disk blocks	Attempts to reduce the effects of "wear" on blocks
P_{Scan}	Performs read or write checks with bogus data	Detects possibly "sticky" block errors

Table 3. The Levels of the IRON Prevention Taxonomy. *The table describes the different levels of Prevention.*

them. An example of this technique is the use of wear-leveling in file systems for flash drives (like JFFS2 [2]).

• *Scan:* The final prevention technique is scanning the disk for bad blocks by performing accesses, perhaps with bogus data. This technique is used in RAID systems to weed out potential bad blocks – the process is called "disk scrubbing" [20, 27]. While this technique was classified as an *eager* Detection technique earlier [26], we feel that it can be employed as a prevention technique. Virtual memory systems can scan the swap area periodically during disk idle time or by using freeblock scheduling [21] and avoid using disk blocks found to be "bad" in the scan.

3. Methodology

In this section, we describe our fault injection and analysis methodology. Our fault injection framework consists of two components, the *Benchmark* and the *Injector*. The Benchmark layer sets the system up for exposure to disk faults. The layer consists of three types of user processes: a *coordinator* for managing the benchmarking and fault injection, *victims* that allocate a large memory region, sleep for a while and then read the memory region, and *aggressors* that allocate large memory regions to force out the victims' pages to the swap area or the file system. Disk errors are injected either when the victims' pages are paged out to disk or when they are read back by the victims.

The error injection is performed by the Injector layer, which interposes between the virtual memory system and the hard disk. Specifically, the Injector has been built as a pseudo-device driver for Linux 2.6.13, as a geom layer [23] for FreeBSD 6.0, and as an upper filter driver for Windows XP. The Injector is located above the device drivers because: (a) drivers are a significant source of errors [29] and the virtual memory system should be equipped to handle errors, and (b) the policies of the virtual memory system can be observed in isolation using this method.

The different types of errors injected by the Injector are read errors, write errors, and corruption errors. In the case of read and write errors, an error code is returned to the virtual memory system. We also ensure that valid data is not placed in memory if the read is failed with an error code. This technique is needed because the virtual memory system may ignore an error code returned; in such a case, if valid data is placed in the respective memory page, the system may seem to work just fine. For corruption errors, the block contents are altered; we zero out the block in our experiments and in case the corruption is detected, we perform a more detailed analysis, corrupting each field of the data structure with field-specific values in separate experiments.

We perform *type* and *context* aware fault injection by injecting disk errors for specific disk blocks at specific times. An example of a data type is a user-level private data segment (*user data*). Therefore, an error injected for a disk block that holds a private user data page is type-aware. A context is a basic function performed by the virtual memory system or an interface offered by the virtual memory system to applications. An example of a context is the `swapoff` system call. Therefore, an error injected for a disk block when `swapoff` is in progress is context-aware. Table 4 presents various data types for which errors are injected and indicates which virtual memory systems use them, and Table 5 presents different contexts when error injection can be performed. The different types and contexts that can be explored are dependent on the particular system under study.

In order to perform type-aware error injection, the Injector should be able to detect the type of blocks being read or written. This detection is accomplished in a variety of ways. The Benchmark layer communicates type information regarding data pages to the Injector. For example, the Benchmark allocates data pages and initializes those pages to contain specific values and conveys the values to the Injector. Thus, in such cases, the Injector uses block contents to determine the block type. Another method employed to determine the type is to use the location of the block. For example, the Linux *swap header* is always located at block 0.

The failure handling policy of the system is identified by a manual observation of the results of error injection. Specifically, we use the following sources of information:

• **The Injector**: The Injector logs all I/O operations in detail, enabling us to determine some failure handling policies; for instance, whether the virtual memory system is performing retries (read or write is repeated with the same disk block number) or remapping (disk write is repeated for a different disk block, but with the same memory page).

• **The Benchmark**: The Benchmark records all return values and signals received. This helps in determining whether an error is propagated. The Benchmark also checks (and reports) the validity of data read back. This helps in checking

Block Type	Description	Detection	Virtual Memory System
swap header	Describes the swap space	Location	Linux 2.6.13
user data	Page from private user data segment	Content	Linux 2.6.13, FreeBSD 6.0
user stack	Page from user stack segment	Content	Linux 2.6.13, FreeBSD 6.0
shared	Shared memory page used by many processes	Content	Linux 2.6.13, FreeBSD 6.0
mmapped	Memory-mapped file data	Content	Linux 2.6.13, FreeBSD 6.0
user code	Page from user code segment	Location	Linux 2.6.13, FreeBSD 6.0
kernel stack	Page from kernel thread stack of a user process	Kernel information	FreeBSD 6.0

Table 4. Data Types. *The table describes the different types of blocks that are failed and gives the detection method and applicable virtual memory system for each type. In order to detect kernel thread stack pages, we made a simple modification to the FreeBSD kernel to obtain the memory addresses of these pages.*

Context	Workload	Virtual memory system actions
`swapon`	Makes swap space available for swapping	Read swap header if any, initialize in-core structures
`swapoff`	Removes swap space from use	Page-in valid blocks and free the swap space
`pagetouch`	Page is accessed by the victim	Read page from disk
`readahead`	Workload induces readahead by reading nearby pages	Perform read-ahead by reading blocks from disk
`madvise`	Victim issues madvise (MADV_WILLNEED) to hint possible future reads	May or may not page-in the blocks specified in hint
`pageout`	Aggressors create memory pressure causing page-out	Write inactive memory pages to disk
`umount`	The file system is unmounted	May have to write of "dirty" mmaped file data
`complete`	Process scheduled again after complete page-out	Page-in essential data structures of process

Table 5. Contexts. *The table shows the workload for the different contexts that are used in the experiments and the actions performed by the virtual memory system for each context.*

whether there is data corruption.

• **System messages**: The operating system may emit error messages to the system message log.

We use these techniques to determine the failure handling policies adopted by virtual memory systems for different combinations of data type, context and error type. These techniques are primarily used to determine Detection and Recovery policies. We discuss experiments to determine Prevention policies in Section 4.1.3.

4. Analysis

In this section, we first present the results of our experiments on three virtual memory systems, then analyze the failure handling approach of the systems, and finally discuss our experience with the fault injection techniques used.

4.1. Experimental Results

We have performed a detailed analysis of the Linux 2.6.13 and FreeBSD 6.0 virtual memory systems, and a preliminary analysis of the Windows XP virtual memory system. We first focus on Detection and Recovery techniques of Linux and FreeBSD, then discuss Prevention techniques of those systems, and finally evaluate Windows XP.

We present about 30 different scenarios (combinations of data type, context and error type) for Linux and FreeBSD. All experiments involving swap space are performed using a separate disk partition as swap space (except for Windows XP), while experiments involving memory-mapped files or user code pages use the ext3 file system [33] in Linux 2.6.13 and the Unix File System (UFS2) [23] in FreeBSD 6.0. The observed failure handling policy for experiments involving a file system is a combination of the policies of the virtual memory system and the file system.

4.1.1. Linux 2.6.13

Tables 6 and 7 present the results of fault injection on the Linux 2.6.13 virtual memory system.

Detection: We find that most read errors are detected using $D_{ErrorCode}$, which is checking of return codes. The exceptions occur during `swapoff` (when the virtual memory system pages valid blocks into memory); the error is not detected (D_{Zero}) and the application to which the data belongs is given junk data on a future memory access. This could lead to application crashes or data corruption.

None of the write errors are detected (D_{Zero}). A read of the page after an ignored write error causes the virtual memory system to page-in the disk block with its previous contents. Missing these errors can lead to application crashes or application data corruption (because of bad data) or even system security problems since the application could possibly read data that belongs to another process.

		swapon	swapoff	pagetouch	readahead	madvise	pageout	umount
Read Errors	*user data*	—	**Z**	**E**	**E**	—	—	—
	user stack	—	**Z**	**E**	**E**	—	—	—
	shared	—	**Z**	**E**	**E**	—	—	—
	mmapped	—	—	**E**	**E**	**E**	—	—
	user code	—	—	**E**	—	—	—	—
	swap header	**E**	—	—	—	—	—	—
Write Errors	*user data*	—	—	—	—	—	**Z**	—
	user stack	—	—	—	—	—	**Z**	—
	shared	—	—	—	—	—	**Z**	—
	mmapped	—	—	—	—	—	**Z**	**Z**
	user code	—	—	—	—	—	—	—
	swap header	—	—	—	—	—	—	—
Corruption	*user data*	—	**Z**	**Z**	**Z**	—	—	—
	user stack	—	**Z**	**Z**	**Z**	—	—	—
	shared	—	**Z**	**Z**	**Z**	—	—	—
	mmapped	—	—	**Z**	**Z**	**Z**	—	—
	user code	—	—	**Z**	—	—	—	—
	swap header	$\mathbf{Y}^1$	—	—	—	—	—	—

Symbols **Z** Zero **E** Errorcode **Y** Sanity
— Experiment not applicable
Comments (1) Sanity checks for swap space signature, version number and bad block count

Table 6. Linux 2.6.13 Detection Techniques.
This table presents the Linux 2.6.13 detection techniques for read, write and corruption errors for combinations of data type (rows) and context (columns). Comments, if any, are provided below the tables.

		swapon	swapoff	pagetouch	readahead	madvise	pageout	umount
Read Errors	*user data*	—	**Z**	$\mathbf{P}^1$	$\mathbf{D}^5$	—	—	—
	user stack	—	**Z**	$\mathbf{P}^1$	$\mathbf{D}^5$	—	—	—
	shared	—	**Z**	$\mathbf{P}^{1,6}$	$\mathbf{D}^5$	—	—	—
	mmapped	—	—	$\mathbf{R}^2$,$\mathbf{P}^1$	$\mathbf{D}^5$	$\mathbf{D}^5$	—	—
	user code	—	—	$\mathbf{R}^2$,$\mathbf{P}^1$	—	—	—	—
	swap header	† $\mathbf{R}^{3,4}$	—	—	—	—	—	—
Write Errors	*user data*	—	—	—	—	—	**Z**	—
	user stack	—	—	—	—	—	**Z**	—
	shared	—	—	—	—	—	**Z**	—
	mmapped	—	—	—	—	—	**Z**	**Z**
	user code	—	—	—	—	—	—	—
	swap header	—	—	—	—	—	—	—
Corruption	*user data*	—	**Z**	**Z**	**Z**	—	—	—
	user stack	—	**Z**	**Z**	**Z**	—	—	—
	shared	—	**Z**	**Z**	**Z**	—	—	—
	mmapped	—	—	**Z**	**Z**	**Z**	—	—
	user code	—	—	**Z**	—	—	—	—
	swap header	**P**	—	—	—	—	—	—

Symbols **Z** Zero **P** Propagate **R** Retry **D** Record
— Experiment not applicable
Comments (1) `SIGBUS` signal (2) One separate retry for every block needed in the original request (3) Retry is not actually used (4) Operation fails but success is returned (error is not propagated) (5) This operation is remembered when page is actually touched (6) Error propagates to all processes that touch the page after the read error occurs

Table 7. Linux 2.6.13 Recovery Techniques.
This table presents the Linux 2.6.13 recovery techniques for read, write and corruption errors for combinations of data type (rows) and context (columns). † indicates a possible bug in the implementation. Comments, if any, are provided below the tables.

Almost all corruption errors are not detected and the corrupted data is returned to the application. One exception is the use of D_{Sanity} for the swap header during `swapon`. The sanity checks are for (a) the correct swap space signature (b) the correct version number, and (c) the number of bad blocks being less than the maximum allowable.

Recovery: For cases where the disk error is detected, Linux uses basic recovery mechanisms. When there is a read error for an application-accessed page, the `SIGBUS` signal is used to inform the application of an error ($R_{Propagate}$). In the case of a shared memory page, all processes that touch the page *after* the read error occurs receive the `SIGBUS` signal – in other words, the virtual memory system does not retry the read when each process accesses the page. Another use of $R_{Propagate}$ is when the swap header is corrupted, in which case an error is returned for the `swapon` call.

In the experiments with memory-mapped file data and user code, a retry is observed (R_{Retry}) for the specific disk block that the system actually needs; even if the original operation involved many disk blocks, the retry is performed for only one block. This retry may have been initiated by the file system and not the virtual memory system. When a read to the swap header fails during `swapon`, a retry is performed (R_{Retry}), but perhaps due to implementation bugs, the results of the retry are not actually used. Also, `swapon` returns success during read errors even though the call fails internally (*i.e.* it does not propagate the error).

R_{Record} is used to handle read errors for `readahead` and `madvise`. By using R_{Record}, the system records the failure of the read for future reference. In both `readahead` and `madvise`, the data is not required immediately – read-ahead is only an optimization by the virtual memory system and madvise is only a hint that the block will likely be accessed. In the `readahead` case, the error is propagated when the page is actually touched and for `madvise`, a retry is performed when the page is touched – both actions use the fact that the first read was unsuccessful.

4.1.2. FreeBSD 6.0

Tables 8 and 9 present the results of fault injection on the FreeBSD 6.0 virtual memory system.

Detection: $D_{ErrorCode}$ is used in every single case for detecting both read and write errors – the FreeBSD 6.0 virtual memory system always checks the error code returned. FreeBSD does not detect block corruption (D_{Zero}). While this leads to application crash or data corruption in most cases, it leads to a kernel crash when corruption of kernel

		swapon	swapoff	pagetouch	madvise	complete	pageout	umount
Read Errors	*user data*	—	**E**	**E**	—	—	—	—
	user stack	—	**E**	**E**	—	—	—	—
	shared	—	**E**	**E**	—	—	—	—
	mmapped	—	—	**E**	—	—	—	—
	user code	—	—	**E**	—	—	—	—
	kernel stack	—	**E**	—	—	**E**	—	—
Write Errors	*user data*	—	—	—	—	—	**E**	—
	user stack	—	—	—	—	—	**E**	—
	shared	—	—	—	—	—	**E**	—
	mmapped	—	—	—	—	—	**E**	**E**
	user code	—	—	—	—	—	—	—
	kernel stack	—	—	—	—	—	**E**	—
Corruption	*user data*	—	**Z**	**Z**	—	—	—	—
	user stack	—	**Z**	**Z**	—	—	—	—
	shared	—	**Z**	**Z**	—	—	—	—
	mmapped	—	—	**Z**	—	—	—	—
	user code	—	—	**Z**	—	—	—	—
	kernel stack	—	**Z**	—	—	**Z**	—	—

Symbols **Z** Zero **E** Errorcode
— Experiment not applicable

Table 8. FreeBSD 6.0 Detection Techniques.
This table presents the FreeBSD 6.0 detection techniques for read, write and corruption errors for combinations of data type (rows) and context (columns). FreeBSD does not read any block during `swapon` *and does not read pages in for* `madvise` *(— in the table).*

		swapon	swapoff	pagetouch	madvise	complete	pageout	umount
Read Errors	*user data*	—	**S**[3]	**P**[2]	—	—	—	—
	user stack	—	**S**[3]	**P**[2]	—	—	—	—
	shared	—	**S**[3]	**P**[2]	—	—	—	—
	mmapped	—	—	**P**[2]	—	—	—	—
	user code	—	—	**P**[2]	—	—	—	—
	kernel stack	—	**S**[3]	—	—	**S**[3]	—	—
Write Errors	*user data*	—	—	—	—	—	**D**[4]	—
	user stack	—	—	—	—	—	**D**[4]	—
	shared	—	—	—	—	—	**D**[4]	—
	mmapped	—	—	—	—	—	**D**[4]	**R**[5],**P**[6]
	user code	—	—	—	—	—	—	—
	kernel stack	—	—	—	—	—	**D**[4]	—
Corruption	*user data*	—	**Z**	**Z**	—	—	—	—
	user stack	—	**Z**	**Z**	—	—	—	—
	shared	—	**Z**	**Z**	—	—	—	—
	mmapped	—	—	**Z**	—	—	—	—
	user code	—	—	**Z**	—	—	—	—
	kernel stack	—	**Z**[1]	—	—	**Z**[1]	—	—

Symbols **Z** Zero **P** Propagate **R** Retry **D** Record
S Stop — Experiment not applicable
Comments (1) Kernel crashes when the stack is used (2) `SIGSEGV` signal (3) kernel panic (4) Memory page is not freed; alternate victim chosen for page-out (5) Upto six retries of the write operation (for all blocks) (6) I/O error returned

Table 9. FreeBSD 6.0 Recovery Techniques.
This table presents the FreeBSD 6.0 recovery techniques for read, write and corruption errors for combinations of data type (rows) and context (columns). Comments, if any, are provided below the tables. FreeBSD does not read any block during `swapon` *and does not read pages in for* `madvise` *(— in the table).*

thread stack blocks is not detected; in this case serious errors like system becoming unbootable are also possible.

Recovery: Various recovery mechanisms are used in FreeBSD 6.0 to deal with detected errors. R_{Retry} is used when memory-mapped data is written during a file system unmount. In fact, the system retries as many as 6 times for each `umount` call. We believe that these retries are performed by the file system and not the virtual memory system (we still document the behavior here since it is the behavior observed by an application using memory-mapped file data, a feature supported by the virtual memory system).

Read errors during page accesses cause the virtual memory system to deliver a `SIGSEGV` (segmentation fault) to the application, an instance of $R_{Propagate}$. Experiments showed that in the case of shared memory, unlike in Linux, processes sharing the memory region operate independently; that is, even if the error has been propagated to one of the processes that accessed the page, the disk access is retried when a second process accesses the page. $R_{Propagate}$ is also used when all write retries are failed during `umount`; an I/O error is returned to the application.

R_{Stop} is used for read errors during `swapoff` and for read errors during a page-in of the kernel thread stack. In both cases, the result is a kernel `panic`, a conservative action. During `pageout`, the virtual memory system attempts to free memory pages by writing them to swap space. If write errors occur during this page-out process, the FreeBSD virtual memory system recovers using R_{Record}. In this case, the virtual memory system remembers that the write operation has not been performed successfully, so that the memory page is not freed. Since the virtual memory system is not able to successfully free the memory page, it proceeds to select an alternate victim for page-out.

4.1.3. Prevention Techniques

Determining Prevention policies is more difficult than determining Detection and Recovery policies since the Prevention policy may not be triggered by a particular disk fault. Therefore, our methodology for uncovering the Prevention policy is to use a specific test for each Prevention technique.

$P_{Remember}$ is the only technique triggered by faults. We test for $P_{Remember}$ by injecting a "sticky" error repeatedly for the same disk block and checking whether the virtual memory system stops using the disk block. The workload performs 10 iterations of a page-out/page-in of victim pages. For both Linux 2.6.13 and FreeBSD 6.0 we find that the "bad" disk block is used repeatedly, in spite of returning an error each time. The same results are obtained for both read and write errors. This indicates that Linux and FreeBSD likely do not keep track of bad blocks (*i.e.* $P_{Remember}$ is not used).

We test for $P_{LoadBalance}$ by causing the virtual memory system to page-out many pages numerous times and checking whether all blocks in the swap area are used fairly evenly. This workload performs 10 iterations of a page-out/page-in of victim pages. In both Linux and FreeBSD, the same disk blocks are reused repeatedly, even though many other blocks in the swap area have not been written to even once. This indicates that the systems likely do not perform wear-leveling (*i.e.* $P_{LoadBalance}$ is not used).

Finally, to detect P_{Reboot} and P_{Scan} we simply observe whether these activities occur over an interval of using the virtual memory system. Given that we did not observe any instance of P_{Reboot} or P_{Scan} during any of our experiments, we infer that it is likely that neither Linux nor FreeBSD use these techniques. In summary, our experiments indicate that neither Linux 2.6.13 nor FreeBSD 6.0 appear to use Prevention techniques.

4.1.4. Windows XP

This section first outlines particular features of the Windows XP virtual memory system, then discusses its failure handling policies. Windows XP uses a file in an NTFS partition to store memory pages that get paged-out. Therefore, the failure handling policy we extract is a combination of policies of NTFS and the virtual memory system. Windows XP allows for paging out of both user and kernel memory. We inject faults only for user data pages. Read and corruption errors are injected during `pagetouch` and write errors are injected during `pageout`. We use the error code STATUS_DEVICE_DATA_ERROR for read and write errors.
Detection: Windows XP uses the error code returned by the disk to detect both read and write errors ($D_{ErrorCode}$). Corruption errors are not detected (D_{Zero}).
Recovery: Recovery from read errors is by terminating the user application, reporting the error *InPageError* ($R_{Propagate}$). Recovery from write errors is more involved. It primarily uses R_{Record}: the memory pages for which the error occurs are written elsewhere when they are selected for paging out again. As for the disk block with the error, it is first read back. If this read succeeds, a half-block write is performed. If the read fails, a half-block read is performed. Irrespective of the success or failure of the half-block operations, the block is used for future writes, using R_{Record} to deal with any errors to these writes. We have so far not been able to identify the purpose of the half-block operations. Also, after a transient write error, although the disk blocks are subsequently successfully written, they are not read back even when the application accesses the data, thereby leading to the application receiving junk data. This indicates a possible bug in handling write errors. Further investigation is required to ascertain this behavior.
Prevention: Error injection experiments demonstrated that a given disk block is not re-used after about 6 errors for the block ($P_{Remember}$). The block is added to a *bad cluster file* and is never used again unless the disk is re-formatted.

4.2. Failure Handling Approaches

In this section, we discuss the approaches that current virtual memory systems adopt to handle disk failures, contrasting the techniques used and identifying the deficiencies of the systems. We also compare the approach of virtual memory systems to that of file systems (explored in [26]). We start by summarizing the different approaches of the virtual memory systems:
• **Linux:** Linux fails to detect many disk errors (even ones where error codes are returned) and follows simple recovery schemes to deal with detected errors. With respect to corruption, only *swap header* corruption is detected.
• **FreeBSD:** FreeBSD correctly detects all disk errors with error codes, but ignores corruption errors. It uses simple recovery schemes to deal with errors, although it is more conservative than Linux for some cases – the kernel calls `panic` to stop the entire system when a read fails during `swapoff`, even if the read affects only a single application.
• **Windows XP:** Windows XP detects disk errors with error codes but ignores corruption errors. It uses simple recovery schemes. It is the only system for which we observed a prevention technique ($P_{Remember}$).

In general, the systems suffer from the following deficiencies:
• **Simple recovery techniques.** The virtual memory systems studied use only simple recovery techniques to deal with disk errors. There is no attempt to use techniques like redundancy to completely recover from the disk errors.
• **Ignoring data corruption.** Of all our data corruption experiments, only one case (Linux *swap header*) is detected. Virtual memory systems assume that disks store data reliably, which may not be true for commodity hardware.
• **Under-developed mechanisms.** A prime example of an under-developed mechanism is remembering bad blocks. The Linux swap header has a provision to store a list of bad blocks. This list can be used effectively to prevent data loss ($P_{Remember}$). However, the list is initialized during `mkswap` and not updated afterward when new errors occur (on the other hand, Windows XP actively uses and updates a bad cluster file to avoid using error-prone blocks).
• **Memory abstraction mismatch.** Applications expect all their pages to behave as if they are always in memory. The virtual memory system should maintain this memory abstraction even when there are disk errors. An important part

of maintaining the abstraction is error reporting. If an error cannot be handled by the system, it should be propagated in a manner that fits the memory abstraction. For example, Linux uses the `SIGBUS` signal to propagate page read errors (by definition, hardware failures can cause `SIGBUS` to be generated). However, FreeBSD uses the `SIGSEGV` signal (which almost always is intended to indicate a programming error) to propagate read errors, which is not appropriate.

• **Very few retries.** There were very few instances of retrying an operation when an error occurs. Retrying can solve the problem in the case of a transient error and systems would benefit greatly by employing retries [14].

• **Illogical inconsistency.** The error recovery techniques employed are inconsistent for cases which are not very different. For example, in FreeBSD, a read error for a user data page may result in a propagate in one case (`pagetouch`), while it results in kernel panic in another (`swapoff`).

• **Buggy implementation.** It is observed in Linux that failure handling code is buggy. For example, the result of a retry is ignored, making it useless. We suspect that failure handling code is rarely tested and is thus likely to have bugs, as seen elsewhere [24].

• **Security issues.** A system that is fairly secure during normal operation could become insecure when there is a partial failure. In Linux, when data is read back after a failed write, the disk block's previous contents are returned to the application, possibly delivering data that the application is not authorized to read. Such failures need to be dealt with given that there is an increasing awareness towards exploiting even transient hardware errors to attack systems [13].

• **Kernel exposure.** Systems should take special care when kernel-mode data is stored on disk. In FreeBSD, corruption of the kernel thread stack is not detected. This may result in undesirable crashes or severe data corruption.

When the policies of virtual memory systems are compared to that of file systems, we observe the following:

• Both kinds of systems share problems like illogical inconsistency and implementation bugs in failure handling code. This points to a general disregard for partial disk errors, thus exposing commodity computer systems to data loss, data corruption and inexplicable crashes.

• The Linux virtual memory system, like some file systems [26], misses a large number of write errors.

• Both virtual memory systems and file systems do not deal with corruption errors in an elegant manner. We see that file systems perform some sanity checking to deal with corruption to file system data structures, but there is no protection for user data (which is the only data handled by virtual memory systems for the most part).

• FreeBSD and Windows XP leverage an important difference between file systems and virtual memory systems in that writes are required to succeed in file systems, while virtual memory systems have alternatives like choosing another page as victim and writing it elsewhere on disk.

4.3. Experience

Experimenting with multiple systems not only helps us compare these systems, but also provides an insight into the advantages and limitations of our methodology. Our experience is that the techniques are simple to use and can be applied to many different systems. While the tool has to be rewritten for each environment, we find that the task is not onerous. We observed one particular limitation: there is no easy way to identify the source of disk accesses and the accesses may be attributed to error recovery while it may be unrelated. An example of this problem occurs when a read error is injected for a user data page in FreeBSD or Windows XP. We observe what seems to be a "retry" of the read. Even if this "retry" succeeds, the application is terminated, indicating a possible bug in the retry code. Only closer examination revealed that the second read is performed to create a core dump and not to recover from the error. It would be interesting to explore techniques to identify the exact source of disk accesses in future work.

5. Related Work

Many techniques have been developed over the years to inject faults in various systems [6, 19, 22, 32]. These and other techniques have been used in fault injection studies that explore operating system behavior under errors [16, 22]. However, these studies do not explore partial disk failures in detail, or bring out techniques and policies used by the operating system to deal with such failures.

Our study is similar in spirit to Brown and Patterson's study of failure policies of different software RAID systems [8]. While software RAIDs are type and context agnostic, the behavior of a virtual memory system differs considerably for different data types and contexts and therefore requires more complex fault injection and analysis.

This study is most related to our earlier studies on how file systems handle partial disk failures [25, 26]. Along with file systems, the virtual memory system is one of the most important operating system components that uses disks significantly. While applications are aware that a disk is used to store files, the use of disks by the virtual memory system is completely transparent to applications, requiring the virtual memory system to be robust to disk failures.

6. Conclusions

Commodity hardware is becoming increasingly unreliable due to escalating complexity and cost pressures. Operating systems can no longer assume that hardware components, especially hard disks, work or fail as a whole. The virtual memory system is an important subsystem in every modern operating system. Therefore, virtual memory systems should be designed to deal with partial disk failures. From our fault injection experiments, we find that current virtual memory systems do not employ consistent failure policies that provide complete recovery from partial failures. Improving the failure-awareness of these systems would enable them to truly virtualize memory, providing applications with a robust memory abstraction.

Acknowledgments

We would like to thank Meenali Rungta for helping with experimental setup for Windows, and Nitin Agrawal for his very useful comments on the paper. We also thank the anonymous reviewers for their thoughtful suggestions. Finally, we thank the Computer Systems Lab (CSL) for providing a great environment for research. This work is sponsored by NSF CCR-0133456, ITR-0325267, CNS-0509474, Network Appliance, and EMC.

References

[1] The FreeBSD operating system. http://www.freebsd.org.

[2] The Journalling Flash File System, version 2. http://sourceware.org/jffs2/.

[3] D. Anderson. "Drive manufacturers typically don't talk about disk failures". Personal Communication from Dave Anderson of Seagate, 2005.

[4] D. Anderson, J. Dykes, and E. Riedel. More Than an Interface: SCSI vs. ATA. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[5] W. Bartlett and L. Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, January 2004.

[6] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek. Fault Injection Experiments Using FIAT. *IEEE Transactions on Computers*, 39(4):1105–1118, April 1990.

[7] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel (Second Edition)*. O'Reilly, December 2002.

[8] A. Brown and D. A. Patterson. Towards Maintainability, Availability, and Growth Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 263–276, San Diego, California, June 2000.

[9] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 31–44, San Francisco, California, December 2004.

[10] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Banff, Canada, October 2001.

[11] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 1–14, San Francisco, California, April 2004.

[12] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Banff, Canada, October 2001.

[13] S. Govindavajhala and A. W. Appel. Using Memory Errors to Attack a Virtual Machine. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 154, Washington, DC, USA, 2003.

[14] J. Gray and C. van Ingen. Empirical Measurements of Disk Failure Rates and Error Rates. Microsoft Research Technical Report MSR-TR-2005-166, December 2005.

[15] R. Green. EIDE Controller Flaws Version 24. http://mindprod.com/eideflaw.html, February 2005.

[16] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of Linux Kernel Behavior Under Error. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2003)*, pages 459–468, San Francisco, California, June 2003.

[17] H. Ishikawa, T. Nakajima, S. Oikawa, and T. Hirotsu. Proactive Operating System Recovery. In *Poster Session of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

[18] T. Johnson and D. Shasha. 2Q: A Low-Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB 20)*, pages 439–450, Santiago, Chile, September 1994.

[19] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Transactions on Computing*, 44(2):248–260, 1995.

[20] H. H. Kari, H. Saikkonen, and F. Lombardi. Detection of Defective Media in Disks. In *The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 49–55, Venice, Italy, October 1993.

[21] C. Lumb, J. Schindler, G. Ganger, D. Nagle, and E. Riedel. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth From Busy Disk Drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, pages 87–102, San Diego, California, October 2000.

[22] W. lun Kao, R. K. Iyer, and D. Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults. In *IEEE Transactions on Software Engineering*, pages 1105–1118, 1993.

[23] M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, August 2004.

[24] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[25] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2005)*, pages 802–811, Yokohama, Japan, June 2005.

[26] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[27] T. J. Schwarz, Q. Xin, E. L. Miller, D. D. Long, A. Hospodor, and S. Ng. Disk Scrubbing in Large Archival Storage Systems. In *Proceedings of the 12th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Volendam, Netherlands, October 2004.

[28] S. Shah and J. G. Elerath. Reliability Analysis of Disk Drive Failure Mechanisms. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 226–231, Alexandria, VA, January 2005.

[29] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[30] N. Talagala and D. Patterson. An Analysis of Error Behaviour in a Large Storage System. In *The IEEE Workshop on Fault Tolerance in Parallel and Distributed Systems*, San Juan, Puerto Rico, April 1999.

[31] The Data Clinic. Hard Disk Failure. http://www.dataclinic.co.uk/hard-disk-failures.htm, 2004.

[32] T. K. Tsai and R. K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *The 8th International Conference On Modeling Techniques and Tools for Computer Performance Evaluation*, pages 26–40, September 1995.

[33] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[34] J. Wehman and P. den Haan. The Enhanced IDE/Fast-ATA FAQ. http://thef-nym.sci.kun.nl/cgi-pieterh/atazip/atafq.html, 1998.