Battling Bad Bits with Checksums in the Loris Page Cache

David C. van Moolenbroek, Raja Appuswamy, Andrew S. Tanenbaum
Dept. of Computer Science, Vrije Universiteit, Amsterdam, Netherlands
{dcvmoole, raja, ast}@cs.vu.nl

Abstract—In this paper, we aim to improve the reliability of a central part of the operating system storage stack: the page cache. We consider two reliability threats: memory errors, where bits in DRAM are flipped due to cosmic rays, and software bugs, where programming errors may ultimately result in data corruption and crashes. We argue that by making use of checksums, we can significantly reduce the probability that either threat results in any application-visible effects. In particular, we can use checksums to detect memory corruption as well as validate the integrity of the cache's internal state for recovery after a crash. We show that in many cases, we can avoid the overhead of computing checksums especially for these purposes. We implement our ideas in the Loris storage stack. Our analysis and evaluation show that our approach improves the overall reliability of the cache at relatively little added cost.

I. Introduction

Reliability of operating systems is important because a failure in the operating system can affect all running applications on the system. The storage stack is the part of the operating system that deals with maintaining the user's data. Reliability of the storage stack is particularly important, because a failure in its components has the potential to destroy the only copy of important user data. This is especially true for the page cache: this component caches file data, but also holds file changes that have already been written by applications but have not yet made it to permanent storage. In this paper, we take a look at the page cache in the light of two reliability threats: memory errors and software bugs.

Memory errors, in particular those caused by external factors such as cosmic rays, may affect application and operating system memory anywhere at any time. Machines without error-correcting memory hardware are fully exposed to such errors. The page cache typically uses all of free memory for caching purposes, and is therefore relatively likely to get hit. Ideally, we would like the cache to detect memory corruption with a high probability before it gets the chance to spread to applications.

Software bugs are a different, well-known source of reliability problems. Bugs may cause arbitrary behavior when triggered. Ideally, we would like to recover from the effects of software bugs in the cache, in an application-transparent way. However, recovery can succeed only if the necessary internal state of the cache can be recovered, and the difficulty is assessing that this state has not been corrupted as a result of the crash.

In this work, we claim that we can address both these problems at little extra cost, by making use of specific information present in the storage stack. Specifically, we reuse the checksums already employed by the storage stack to detect disk corruption. If the page cache is brought in the loop regarding these checksums, it can reuse them for runtime verification of cached pages against memory errors, and for integrity assessment of the cache's internal state after a crash. This allows the cache to catch memory corruption with a high probability, and to recover from a crash when possible–all completely transparent to the running applications. We implement our ideas in the Loris storage stack, which we developed in previous work [1]. Our evaluation shows that the two solutions are not independent, and in fact interact in an overall beneficial way.

The rest of the paper is organized as follows. In Sec. II we describe our Loris storage stack. In Sec. III, we analyze the two reliability threats, argue that the cache is especially important in respect to the threats, and show that checksumming has the potential to help alleviate both. We then detail our approach to use checksums against memory errors (Sec. IV) and against software bugs (Sec. V). Sec. VI describes our implementation. In Sec. VII, we evaluate our solutions. We list related work in Sec. VIII, and conclude in Sec. IX.

II. Background: the Loris stack

Fig. 1a depicts the traditional operating system storage stack. Applications send requests to the Virtual File System (VFS) layer, which passes them to an actual file system. The file system operates on a single device; the RAID layer below may however transparently multiplex these operations across several devices for performance and redundancy purposes. Disk drivers are used to talk to the actual hardware.

In previous work, we have developed a new storage stack called Loris. It was formed by splitting up the traditional file system into three individual layers (a naming, a cache, and a layout layer), and swapping the layout layer with the traditional RAID layer, forming the physical and logical layers. The VFS and driver layers have been left as is. The result is depicted in Fig. 1b. This new stack has advantages in the areas of reliability, heterogeneity, and flexibility [1].

The four new layers communicate using files. Each file has a unique identifier and a set of associated attributes. The four layers support the following operations: *create*, *read*, *write*, *truncate*, *delete*, *getattr*, *setattr*, and *sync*.
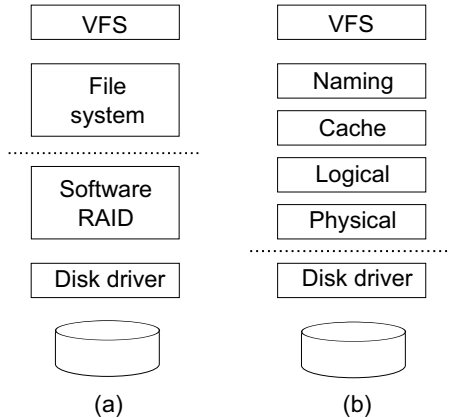
IEEE computer society

Fig. 1. The figure shows (a) the layers of the traditional stack and (b) the new arrangement in Loris. The layers above the dotted line are file aware; those below are not.

At the bottom, the **physical** layer consists of one or more **file stores**. Each file store is responsible for one underlying device, and determines the layout on that device. It manages an independent set of *physical files* on its device, converting file operations from above to block operations below. Each file store has a small cache for the metadata of its layout. All file stores are required to implement a *parental checksumming* scheme in their layout [1]. As a result, the contents of the devices are fully covered by checksums, so that all forms of disk corruption are guaranteed to be detected.

The **logical** layer is the file-based equivalent of the traditional RAID layer. It exposes a *logical file* abstraction to the layers above, multiplexing operations on files across the file stores according to per-file RAID-like policies. Thus, each logical file has an associated policy and is made up of one or more physical files on different file stores. The logical layer stores this per-file information in a special *mapping* file, which is mirrored across all devices. It also implements RAID-like recovery for when an underlying file store reports a checksum or device failure. Our prototype implements file-based equivalents of RAID 0, 1, 4, and 5.

The **cache** layer implements a page cache, caching both file data pages and file attributes. It uses a large amount of memory for caching purposes, staging and evicting parts of files according to predefined policies. For performance reasons, it does not pass on incoming *create*, *setattr*, and *write* operations directly to lower layers; instead, it delays them to reduce request costs and to aggregate changes.

The **naming** layer converts POSIX operations to Loris file operations. It manages directories, which are treated as normal files by the lower layers. It uses Loris attributes to store the POSIX attributes of files.

We have implemented a prototype of the Loris stack on the MINIX 3 microkernel operating system. On this platform, all the layers of the stack (and all the file stores) are implemented as separate userspace processes.

III. THE CASE FOR CHECKSUMMING IN THE CACHE

In this section, we discuss two reliability issues: memory errors (Sec. III-A) and software bugs (Sec. III-B). We state why it is important to address these problems specifically in the Loris page cache, and we show that in both cases, we can significantly reduce the potential risks by making use of checksums.

A. Memory errors

The problem: Various studies have shown that dynamic random-access memory (DRAM) is susceptible to errors [2], [3], [4]. Such memory errors constitute arbitrary corruption in the system's main memory at unpredictable times and locations; unsurprisingly, software does not deal well with this. Memory errors are found to be responsible for a significant fraction of system failures in the field [5].

Memory errors are categorized as either soft or hard. Soft errors are transient changes in memory state (*bitflips*) caused by external factors; in particular, cosmic rays [2]. Soft memory errors are generally assumed to be distributed randomly in both space (i.e., affected memory location) and time. Various lab and field tests have suggested that soft errors are a serious problem [3], [4], and error rates in the 200-5000 FIT (Failures In Time) per Mbit range have been cited [6], [7]. More recent field studies found lower soft-error rates [7], [8].

Hard errors are caused by physical hardware faults, and may manifest themselves intermittently or permanently (*stuck bits*). The same recent field studies found hard errors to be more common than soft errors [7], [8], although little is known about their exact cause. These studies observed that a relative minority of memory modules see the vast majority of errors, and that there is a strong correlation between the errors in both location and time [8], [9].

To counter the effects of memory errors, DRAM modules with error-correcting codes (ECC) have been developed. Typical ECC memory has single error correction and double error detection (SEC/DED) capability; the Chipkill memory family can also cope with whole-chip failures [10]. ECC memory can deal well with in particular soft errors. However, many computer systems are not equipped with ECC memory, primarily because of the added cost. This leaves them utterly exposed to memory errors. In a significant number of cases, these memory errors will end up causing serious damage to the running system [11]. Therefore, we believe that there is room for software approaches that can detect memory errors before they cause damage.

In this work, we consider only *soft* errors, on systems *without* ECC memory. All non-ECC memory is vulnerable to soft errors to some degree, and software solutions have the potential to overcome such problems. In contrast, not even SEC/DED memory can correct all hard errors [12], [9], and given that only a small subset of memory modules experiences such errors, the only real remedy for hard errors may thus be replacement of faulty modules. At the same time, we believe that any solution for soft errors can also help detect hard errors, although perhaps not as effectively.

Like hardware ECC memory, software approaches may use some form of redundancy (checksums, full copies, etc.) to detect and possibly correct memory corruption. However, such approaches are inherently imperfect: not every memory access can be guarded (e.g., device DMA), whereas any memory access can hit a bitflip. Software approaches can therefore only lower the probability of being affected by memory corruption, and more extensive forms of redundancy come at greater cost in terms of both performance and resource usage. Given the low probability of memory corruption, significant extra cost is typically not justified.

Therefore, we propose to start by significantly reducing the chance of undetected memory corruption while adding little extra performance and resource costs. We argue that the operating system page cache is the right place to start.

Why the cache: First of all, modern operating systems use all available memory (i.e., not used by applications) for caching purposes, so a large fraction of main memory is typically used by the cache for data pages. With a random spatial distribution of memory corruption, these data pages are therefore relatively likely to get hit by a soft error. Second, the operating system cache is shared by all applications. Memory corruption in the cache may affect the system integrity beyond just a single application. While this applies to all components of the operating system, measurements on our own systems revealed that the cache's pages typically use over 95% of all memory in use by the operating system. Third, in many cases we can recover from detected corruption without the overhead of keeping copies in memory: for clean data, there is already a valid copy on disk, so we can restore the data in the cache from there.

The case for checksumming: In order to detect memory corruption in the cached data pages, we propose to use checksums. As we will show in Sec. IV, we can obtain checksums for the page data essentially for free, and that only leaves the verification aspect.

B. Software bugs

The problem: Another major source of reliability problems is software bugs. Bugs may cause arbitrary behavior during software execution. Previous research suggests that the number of bugs is a linear function of the number of lines of software source code [13] and that 0.5–6 bugs per thousand lines of code can be expected even in well-written software [14].

The operating system is of particular importance in this regard because a failure in the operating system may affect all running applications on the system. Microkernel-based operating systems allow failures to be contained, since most parts of the operating system are implemented as isolated system processes running in user mode. In that case, many software bugs in the operating system will manifest themselves as observable failures in the containing system process ("process crashes"). If the cause of the failure was transient– for example, a race condition–the system then *may* be able to recover transparently.

In the MINIX 3 operating system, crashed system processes can be restarted [15]. This is necessary but not sufficient for application-transparent recovery: most system processes have internal state, and after a restart, they must recover this state. Preferably, the state would be recovered from other system processes, so that the memory image of the crashed process need not be used for state recovery. However, this is not always possible.

Why the cache: The storage stack components of the operating system are directly responsible for storing user's data. Therefore, we believe that these parts deserve extra attention. In previous work, we have outlined how the system can survive process crashes in the Loris layers above and below the cache, without requiring reuse of internal state of the crashed processes [16]. This however leaves the cache layer, for which we will show such recovery is not possible.

The strict separation of the layers of the Loris stack into separate processes, and the process isolation provided by the underlying microkernel, ensure that the cache cannot be taken down as a result of crashes in the rest of the storage stack. However, the cache itself may crash. The current implementation of the Loris cache, including supporting library routines, is well in excess of 10,000 lines of code, and makes heavy use of nonpreemptive multithreading. It is therefore likely to contain dozens of bugs.

The case for checksumming: The cache is a crucial piece of the storage stack, and typically contains the only copy of significant amounts of application-generated state: dirty file data and other delayed file operations. If the cache crashes, this state cannot be restored from external sources, and the cache can thus be recovered only if the crashed cache process's state is left in a proper condition. The main question is then how to assess this condition after a crash. To this end, we propose that the cache keep checksums of its crucial state during normal run time. As we will show in Sec. V, we can generate checksums for a large part of this state at very little added cost.

IV. DEALING WITH MEMORY ERRORS

The Loris file stores require checksums for all data in order to detect disk corruption. In this section, we discuss reusing those checksums for detecting memory errors in the cache. Since the file stores have to have a checksum for each block on disk, the cache can in principle get the checksums for all its clean pages for free. This obviates the need for *generation* of checksums specifically against memory corruption, and only leaves the *verification*.

Our goal is to reduce the window of vulnerability of undetected memory errors, while at the same time incurring little overhead. We focus exclusively on clean pages in this section, since clean pages may stay in the cache indefinitely, whereas dirty pages will be flushed to disk after (typically) at most 30 seconds, thus making them clean as well. We further discuss memory corruption in dirty pages in Sec. V.

In this section, we first analyze whether the checksums used against disk corruption are usable against memory corruption at all (Sec. IV-A), and discuss how the cache can obtain

on-disk checksums (Sec. IV-B). We then present a number of verification strategies (Sec. IV-C). Finally, we consider memory corruption in other memory of the cache and in other parts of the storage stack (Sec. IV-D).

A. Suitability of on-disk checksums

There may be significant computational overhead involved in verification of checksums. Thus, we first consider whether the on-disk checksums are the best choice for use against memory errors at all.

The Loris file stores use a checksum type from the family of Cyclic Redundancy Check (CRC) codes. Compared to simpler checksums such as exclusive OR (XOR), CRC codes are complex and traditionally implemented in software, and thus expensive in their use. However, it appears that an increasing number of platforms incorporates support for CRC in hardware [17], [18], closing the performance gap between XOR and CRC checksum computation.

At the same time, CRC codes are much stronger than XOR. XOR checksums have a Hamming Distance (HD) of 2, thus guaranteeing detection of one bit error only. The Loris file stores use the CRC-32C polynomial, which has a HD value of 4 for 5244 to 131072 bits [19], thus guaranteeing detection of up to three bit errors per block of the typical page and block size of 4096 bytes plus the checksum itself. In addition, CRC codes are guaranteed to detect burst errors in length of up to the polynomial width.

This extra strength not only helps in detecting a wider range of on-disk errors, but also helps in detecting memory errors. A single cosmic ray may affect multiple adjacent memory cells at once [12], and cells within either the same row or column are likely to fall on the same page. The on-disk checksums are very likely to detect such corruption, whereas XOR-based checksums are not.

B. Propagation of checksums

We can involve the cache in the checksumming, by propagating checksums between the cache and the file stores. The most basic approach works as follows. Immediately before issuing a *write* operation to lower layers, the cache computes the checksum of each involved data page, and sends those checksums along with the write call. The file stores involved need not compute the checksum themselves any longer, and after the *write*, the cache will have the checksum for that page until the page is changed or evicted. In addition, when the cache issues a *read* operation for data, the file store always has to obtain and verify the checksum of the data anyway, in order to be able to detect and recover from disk corruption before the data reaches the cache. However, the file store now propagates up the verified checksum along with the data, so that the cache has the checksum from that point on as well.

C. Verification strategies

We now outline several strategies that the cache can use for checksum verification. They offer different tradeoffs between coverage against memory errors and overhead from verification. We note again that we can always recover after detection:

all the pages involved are clean, and so a corrupted page can simply be read back from disk.

Background checker: On systems that see little overall storage activity, data may be held in the page cache for a long time. In general, the lifetime of data in the cache, and thus its vulnerability to errors, is potentially unbounded [20]. A single soft error may already get close to the limits of guaranteed detection of the CRC checksum; accumulation of multiple independent errors on a single page may be undetectable. To remedy this, the cache can slowly verify pages in the background. The expected time of accumulation is rather long even with high FIT/Mbit rates, but the cost of performing a slow background check is very low as well and puts a hard bound on the window of vulnerability. However, such background verification is independent from actual page accesses, and therefore ineffective at catching a single soft error before the affected page is accessed. Other strategies thus have to be considered in addition.

Check on every read: The cache could verify the checksum of a page upon every incoming *read* call involving that page. Combined with a background checker, this virtually closes the vulnerability window for clean pages. However, it comes at a steep computational cost. Even for small subpage reads, the cache can only verify the checksum of the entire containing page. An application that reads from a file in small sequential chunks will force the cache to recheck the same pages very often with little time in between, resulting in significant CPU overhead and almost no gain.

Check on read after minimum last-check time: In order to alleviate this issue, the cache can prevent the same page from being rechecked too often. The cache could recheck the page's checksum only if the last check was at least a certain minimum time ago. This eliminates duplicate checks within that time frame, while ensuring that even frequently accessed pages will occasionally be rechecked.

Check on read after minimum last-use time: One could argue that there is no point for the cache to detect a memory error, if there is a high probability that that error has already been propagated to an application. From this perspective, there is no gain in rechecking a page that has been accessed recently before. Thus, a page's checksum could be checked only if there has been at least a certain amount of time since the last access. This approach leans even further towards the performance end of the spectrum, but may be less effective in practice: the application may not have actually hit the memory error on the earlier read (for example, due to small reads); also, different applications may access the same page.

Checking memory-mapped files: The options for verification of memory-mapped pages are limited, since the operating system is only involved in the first read from each page. Depending on the applications, memory-mapped I/O may not be all that common [21]. However, operating systems typically use memory-mapped files for shared libraries, and verification can thus help protect the integrity of important parts of running applications. The cache can use more generic techniques to reduce the vulnerability window of mapped pages. For

example, it can perform more frequent background checks on these pages, or monitor them to trap after long times of no activity [22], at the extra cost and reduced accuracy of the polling-based monitoring.

D. Other memory

The cache's pool of data pages make up by far the largest fraction of memory used by the cache, and indeed by the entire Loris stack altogether. However, the cache's internal state comprises more than its data pages, and the remainder is worth at least some consideration as well.

Particularly noteworthy are the cache's data structures that describe the actual data pages, as every data page has to have such a corresponding data structure. However, this structure is much smaller than the page itself. For example, while not particularly optimized for size, the Loris page descriptor structure is currently 48 bytes, and thus about a factor 85 smaller than corresponding 4KB-page of data. The cache's data structures to track file objects and their attributes are similar in total size. Besides these, the cache has a long tail of smaller structures and global variables.

It may still be worthwhile to protect these parts of cache memory as well. Even though they are less likely to be hit by soft errors, their corruption may affect the behavior of the cache itself–comparable to the effects of software bugs. More runtime checks on internal correctness can be added (for example with `assert` statements), but it is hard to reason about the coverage of this technique.

As an alternative, we attempted to implement checksumming for the internal data structures of the cache. We found that even with the simple design of the Loris cache prototype, manually adding support for checksumming of data structures added a prohibitive amount of complexity to it. While some parts were straightforward (e.g., linked-list macros and mutex operations were good places to start), a substantial amount of ground was left to cover with manual checksum update and verification statements.

We now believe that the solution lies in compiler support, in the form of programmer-guided (annotation-based) automatic checksumming of data structures, for example using the LLVM compiler framework. This would be similar to automatic approaches proposed for use against software bugs [23], but requires manual guidance so as not to automatically double-checksum the data pages as well. We believe the cache would be an interesting use case for such a technique, and we intend to work on this in future work.

Both asserts and semi-automatic checksumming would only enable error *detection*. In Sec. V, we show that our proposed recovery system for software bugs can equally help recover from detected memory errors.

Finally, we note that the techniques presented in this entire section are also applicable to other layers of the Loris stack. For example, each file store has its own small metadata cache, and the checksum verification approaches for the page cache can be applied directly to this cache as well, although possibly with a different level of effectiveness. Any future semi-automatic checksumming solution may in fact be applied to large parts of the entire operating system, including the (small but crucial) microkernel.

V. Dealing with software bugs

In this section, we look at failures due to software bugs in the cache. Since MINIX 3 already provides several means of detecting misbehavior resulting from software bugs, we are concerned only with recovery. We propose a recovery approach that extends the checksum propagation from Sec. IV-B. In principle, it is otherwise fully independent from the memory errors solution from Sec. IV. However, we also show that there is a beneficial interaction between the two solutions if they are both used at the same time.

We first describe our assumptions regarding software bugs (Sec. V-A). We then show how any recovery procedure needs a way to verify the integrity of the crashed cache's internal records of delayed operations, how this verification can be performed, and that checksums of all dirty pages are required for it (Sec. V-B). We discuss checksumming dirty pages next (Sec. V-C), and then sketch the crash recovery procedure (Sec. V-D). Finally, we show the benefits of checksumming dirty pages for memory error detection (Sec. V-E).

A. Assumptions

Software bugs may cause arbitrary behavior. This includes scenarios where corrupted results escape detection and reach the application, in which case application-transparent recovery is impossible. Thus, we have to make assumptions about the errors we target.

First, we assume that if a software bug triggers in the cache, it will lead to a detectable failure; for example, a CPU exception, a failed assertion, or a bad interprocess call. In this case, the system can shut down the cache process, considering it to have *crashed*. Previous research on errors in operating systems suggests that a large majority of errors, if manifested in any way at all, indeed cause a detectable failure–silent failures are rare [24].

Second, we assume that the crash occurs within the execution of the requests that were active when the bug was triggered. This means that no bad results are propagated before the detection of the failure. Similarly, previous research has shown that fault propagation as the result of software bugs is relatively unlikely to occur [24], [25].

However, we do not assume that the failure was necessarily fail-stop, and want to attempt recovery even if (for example) arbitrary memory was overwritten within the cache. These assumptions are similar to those made in other contemporary work [26], [27], [23].

B. The Dirty State Store

As stated before, the cache typically has the only copy of many delayed file modification operations. This is the part of the cache state that cannot be recovered from elsewhere, and thus, the memory of the crashed cache process must be used to attempt recovery of such state. Thus, the first step for any

recovery procedure is salvaging the delayed operations present within the cache at the time of the crash.

The cache supports three different types of delayed file modifications: *create*, *setattr*, and *write* (i.e., dirty pages). In order to assess the feasibility of recovery, the recovery procedure must be able to enumerate all delayed operations in the crashed cache process's memory image, and verify that they have not been corrupted as part of the crash.

To this end, we propose that the cache use an internal store to keep track of such dirty state. We call this store the Dirty State Store (DSS). It is a small and passive part of the cache, and exposes a very narrow API to the main cache code. Using this API, delayed operations can be added to the store as new application requests arrive, and removed from it as changes are flushed to disk. The store uses its own (very basic) data structures to keep track of the operations, using a separate memory region.

However, this region is still part of the cache's address space, and any accidental overwrites from the main cache code could violate the integrity of the DSS. For this purpose, the DSS protects its data structures with checksums (XOR suffices for this). Thus, wild writes result in a checksum mismatch. The checksums need to be generated at runtime, but need never be verified unless the cache crashes.

We believe that the narrow and strictly checked API, the separate memory region, the checksumming of all parts involved, and the relative simplicity of the DSS allow us to put trust in the contents of the DSS if after a crash its checksums all match. Therefore, the recovery procedure can use the DSS memory to exhaustively enumerate and verify all delayed operations. The only requirement is that the DSS is kept up-to-date at all times, which means that its API must be used as part of each request handled by the cache.

While the file *create* and *getattr* operations can be duplicated in the DSS at little extra cost, keeping a copy of each dirty page in the DSS is not feasible: a substantial part of the cache's memory usage, and indeed of the memory in the system overall, may consist of dirty pages. Therefore, the DSS simply contains a pointer to the actual dirty page, along with a checksum of the contents of the page.

C. Checksumming dirty pages

The DSS thus requires an up-to-date checksum for each tracked (and thus, each dirty) page in cache. In terms of runtime overhead, checksumming of dirty pages dwarfs checksumming of the DSS data structures; this is where we return to interaction between in-memory and on-disk checksums.

In many cases, a page is modified only once and then flushed to disk sometime later. In such a case, performing the checksum computation upon the modification rather than the flush does not introduce any extra computational cost: the same checksum computation is simply performed a bit earlier. Thus, actual overhead is incurred only when the page is either modified again before the flush (due to another write operation), or it is discarded before the flush altogether (due to a truncate or delete operation).

Previous work suggests that cancelled writes (due to full-page overwrites, truncates, and deletes) are generally not dominant in workloads; for example, [28] reports a fraction of cancelled data bytes in the 4–27% range. This leaves subpage overwrites, to which we can apply two optimizations.

First of all, we find that a major source of subpage overwrites comes from file appends, thereby "overwriting" an unused part of the page. Many checksum types, including the CRC family, allow checksums to be computed incrementally. Therefore, instead of tracking only whole-page checksums, the cache can remember the size of the used part of each page (i.e., always the full page size, *except* for the last page of each file), and track the checksum of that part only–until the page is flushed. For appends, the checksum can then be updated using only the new part of the page, thus avoiding checksum recomputation of the existing part.

Second, for small subpage overwrites, a CRC checksum can be updated for only the modified part of the page, by computing the checksum for just the old and the new parts with precomputed zero leads and trails, and XORing these partial checksums into the original checksum. Our initial tests show that this method can be more efficient than full page checksum recomputation.

D. Recovery procedure

With the DSS in place, we can now describe the overall crash recovery procedure. The recovery procedure is assumed to have full access to the memory image of the crashed cache process. Using the DSS and its checksums, the procedure starts by assessing whether all the delayed operations can be recovered from the crashed cache process. If this is possible, then the following steps are taken.

1) The recovered file modification operations are flushed down to lower layers. The result is that the cache is completely "clean" with respect to delayed operations.
2) All of the crashed cache's state is discarded, and the new instance of the cache starts with a clean slate.
3) The naming layer is notified that it should replay all ongoing requests. The Loris operations are all idempotent, so this is always safe to do, no matter what happened before the crash.

After that, the storage stack can resume normal operation, and applications will never notice that anything went wrong. However, if the recovery procedure finds that recovery of the delayed operations is not possible, or if replaying the ongoing requests repeatedly results in a crash of the cache, then recovery is not possible, and the system halts.

E. Consequences for memory errors

Our approach for dealing with software bugs yields two important positive effects in relation to memory errors.

The availability of checksums for dirty pages allows us to check these pages for memory errors as well, using the same verification techniques described in Sec. IV-C. In fact, in practice this simply means that we no longer have to make an exception for dirty pages. Moreover, since the checksums

computed for the dirty pages are used directly as on-disk checksums, the result of an undetected memory error between the in-memory modification and flush to disk of a page, is that the memory error will be detected upon the next read from disk. In all these cases however, we can not recover the page, so whenever it is read later, the stack has to report an I/O error to the caller. At the very least, this prevents corrupted data from reaching the application.

Additionally, as we noted before, memory errors may corrupt the cache's internal data structures, and even the program code. The result is that a memory error may cause the cache to crash. The crash recovery system can and will not make any distinction regarding the cause of the cache, and thus, the system will attempt to recover the cache in this case as well. As a result, the cache's primary data structures and code will be reset, wiping out any previous memory errors. Thus, the cache is able to survive crashes resulting from not only software bugs, but memory errors as well.

VI. IMPLEMENTATION

We have implemented our ideas in the cache layer of our Loris prototype, on the MINIX 3 microkernel operating system. We briefly list what we implemented.

Memory errors: We have extended the Loris communication library to support propagation of checksums, and we have implemented basic generation and tracking of checksums in the cache layer. We have implemented the strategies described in Sec. IV-C for detecting memory errors in cached pages, except those involving memory-mapped files, as MINIX 3 currently does not support those.

Software bugs: We have modified the cache to track checksums for dirty pages as well, including the append optimization (but not the partial-rechecksum optimization) described in Sec. V-C. In addition, we have implemented the DSS. It offers a very narrow API of five calls (file create/setattr/flush, page write/flush) in under 200 lines of code, and the API calls were easy to add to the main cache code. MINIX 3 implements restarting processes after a crash, and optionally allows specified memory regions to survive process crashes. The restarted cache instance can perform the rest of the recovery procedure (as per Sec. V-D) fully by itself, by verifying the checksums of the DSS and dirty pages, flushing down all dirty operations to lower layers, freeing the surviving memory, and requesting the naming layer to reissue any ongoing calls.

VII. EVALUATION

We evaluate our work on an Intel Xeon W3565 workstation, with 4GB of RAM, and a 500GB 7200RPM Seagate Barracuda SATA hard drive.

A. Microbenchmarks

We start with microbenchmarking the cost of basic *read* and *write* calls into the cache (without checksumming). We make calls directly from within the cache layer itself, rather than from an external source program: context switching is

Operation	Size (bytes)	Time (ns)
Read	1	802
	4096	920
	8192	1463
	16386	2547
Write	1	799
	4096	822
	8192	1251
	16384	2544

Table I. Microbenchmarks for read and write calls of various sizes.

Operation	Size (bytes)	Time (ns)
CRC-32C in software [29]	4096	967
CRC-32C in hardware [30]	4096	205

Table II. Microbenchmarks for page checksum computation.

not optimized for performance on MINIX 3, and the overhead of context-switching through the other storage stack layers would otherwise reduce the relative cost of checksumming. Thus, the resulting relatively low times are worst case for us, and possibly more representative for other operating systems (we got very similar numbers in a userland benchmark on Linux). All pages involved are already cached in the cache layer, so no other layers are involved at all. The results are shown in Table I. The (nanosecond) times are averages for a million iterations.

We also measure checksum computation for a single page, using the most optimized software and hardware CRC implementations that we could find. These measurements are shown in Table II. They are representative for the additional cost of computing the checksum for a single page involved in a read and/or write operation. Thus, as can be seen, if checksumming of pages were added to the read and/or write calls, this would account for a substantial fraction of their time. We believe these overheads are not prohibitive: again, they are worst case, and applications are often not overly sensitive to such overheads [31]. However, the overheads are high enough to warrant exploring strategies that reduce the number of checksum operations.

B. Macrobenchmarks

Benchmarks: We perform further testing by means of macrobenchmarks. We use the following benchmarks in our experiments:

- PostMark (1.51), configured to perform 80K transactions on 40K files in 10 directories, with 4–28KB file sizes and

Benchmark	Err. cons.	Usage	Read call	Read page	Hit
PostMark	26%	54%	75%	75%	97%
File Server	18%	81%	37%	57%	41%
Web Server	23%	98%	93%	97%	91%
OpenSSH	10%	100%	83%	97%	98%

Table III. Macrobenchmark statistics: error consumption, total memory usage, read percentage of all read/write calls, read percentage of read/write page accesses made as part of calls, and page cache hit ratio.

	Baseline		ER		LC-tick		LC-1sec		LC-30sec		LU-tick		LU-1sec		LU-30sec	
Benchmark	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%
PostMark	0	0	21	49	4	49	2	49	2	48	4	49	2	49	2	48
File Server	0	0	46	72	28	72	27	72	12	51	28	72	27	72	8	40
Web Server	0	0	97	96	80	96	42	95	13	74	80	96	36	95	8	66
OpenSSH	0	0	99	96	13	94	2	71	0	47	13	94	0	52	0	45

Table IV. Overhead and protection against memory corruption, using various verification strategies on clean pages.

	Baseline		ER		LC-tick		LC-1sec		LC-30sec		LU-tick		LU-1sec		LU-30sec	
Benchmark	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%
PostMark	4	40	97	91	52	91	8	91	6	88	52	91	7	91	6	88
File Server	41	20	88	99	70	99	69	99	52	76	70	99	69	99	49	66
Web Server	0	3	98	99	81	99	43	99	14	77	81	99	37	98	9	70
OpenSSH	0	3	99	99	13	98	2	74	0	50	13	98	0	55	0	48

Table V. Overhead and protection against memory corruption, this time also checksumming dirty pages immediately.

512B unbuffered I/O operations.

- FileBench (1.4.8.fsl.0.8) File Server, single-threaded, but otherwise with its default configuration, run for 30 minutes at once.
- FileBench Web Server, single-threaded, with 25,000 files, directory width 50, file size 32KB, and defaults otherwise, also with 30-minute runs. For Web Server, we use a modified FileBench version which accesses the files using a Zipf distribution rather than the default round-robin approach. With this (we believe, more realistic) distribution, we avoid that each cache page sees exactly the same access interval.
- An OpenSSH build test, which unpacks, configures and builds OpenSSH in a chroot environment.

The Loris page cache is given static size of 1GB of memory, and the tests were run on the first 32GB of the disk. All tests were run at least five times; average numbers are reported. For PostMark and FileBench, we consider the run phase only.

Table III shows statistics about the benchmark runs. The first column shows the percentage of *error consumption* [11]: the probability that if an error occurs anywhere in the cache's 1GB of memory any time during the benchmark run, the error will be read back, and thus propagate to an application or to permanent storage. These are the errors that matter; other errors simply end up being discarded. While the percentages may seem low, their upper bounds are the memory usage of the benchmark runs, reported in the second column. Thus, about half of the errors occurring in memory used by PostMark would be consumed, whereas this would be a tenth with the OpenSSH benchmark. The remaining columns of the table show read/write ratios (on a per-call and per-page basis) and the overall cache hit ratio.

Overhead and protection: We measure the overhead and memory error protection of our verification strategies these macrobenchmarks. For the overhead (O%), we count the sum of all page accesses by all incoming read and/or write calls, and per verification strategy, we report the percentage of page accesses that result in an *extra* checksum computation for that page. Thus, the higher the percentage, the higher the checksumming overhead. For the protection (P%), for each page we measure the time windows in which a memory error on that page would be propagated to the application or to permanent storage, and we sum the time windows of all pages together; per verification strategy, we measure the overall fraction of this total time where a memory error occurrence would be caught by the strategy. Thus, given a random memory error in the cache that would be consumed, the resulting percentage represents the probability that this error will be detected before reaching an application.

We list the results for the following verification strategies and time intervals: check on every read (ER); check on read after minimum last-check time (LC) of a MINIX 3 clock tick (1/60th of a second), 1 second, and 30 seconds; and, check on read after minimum last-use time (LU) with the same time thresholds. We have omitted the background checker results. As stated in Sec. IV-C, the background checker cannot stand on its own as a verification strategy, and our experiments show that it indeed adds negligible protection for active workloads, unless configured to be prohibitively aggressive.

We first test our solution for memory errors from Sec. IV, which covers only clean pages. The results are shown in Table IV. The protection is below 100% even if every read is verified, because all techniques only cover clean pages, and all benchmarks involve at least some dirty data. The last-check and last-use strategies show that delaying checksumming by as little as one clock tick can significantly reduce the number of extra checksum checks, while keeping the protection at almost the same level. This is true especially for PostMark and OpenSSH, where the smaller (often subpage) I/O calls make rapid repeated page accesses common, thus resulting in overheads going down quickly. The per-tick strategies for File Server and Web Server maintain a high overhead because of the large (multipage) I/O granularity of these benchmarks. Higher time thresholds lower the overhead by much; the protection also decreases but remains substantial. There is little difference between the two strategies' results.

We repeat the same tests after adding our solution for software bugs from Sec. V on top of the memory error

solution. This shows not only the extra overhead for check-summing upon write calls, but also the extra memory error protection resulting from having checksums of dirty pages. The results are shown in Table V. We note again that we cannot recover from memory errors in dirty pages, but we do prevent corrupted data from reaching the application.

In this test, the overhead increases significantly because reads from dirty pages can now be checked as well, and the protection increases accordingly. These increases are shown by the new baseline, and reflected in the per-strategy numbers as well. Looking at the baseline, PostMark often reads back its own written data before it is flushed, and thus ends up with a high increase in protection. File Server has relatively the most writes, and thus ends up with a large increase in overhead–and, to a lesser extent, protection. Even with the every-read strategy, 100% protection is still not achieved: this is due to subpage writes causing entire pages to be rechecksummed, thereby losing the ability to detect previous memory errors in the unchanged page parts. We can solve this with the partial rechecksumming described in Sec. V-C: this method preserves checksum errors across partial updates.

Performance: The overall performance of the benchmarks is shown in Table VI. The numbers represent run times of the benchmarking phases, relative to the unmodified Loris implementation (thus, lower is better). The "clean only" columns represent Loris configurations that check only clean pages for memory errors, and correspond to the first three configurations in Table IV: the baseline (BL) that implements no memory check and thus only adds checksum propagation; a page check on every read (ER); and, a page check after at least one clock tick of not checking that page (LC-tick or LCt). The "clean and dirty" columns add checksumming on writes and the DSS keeping overhead, and thus correspond to the first three configurations of Table V.

The baselines show that the checksum propagation has no overhead at all, and checksumming pages as they are written adds only little overhead. Checking checksums on every read has a clear performance impact (up to 9% for OpenSSH). However, the LC-tick strategy reduces the overhead to at most 1%. We believe that this overhead is acceptably low, especially given that LC-tick still offers strong protection against memory errors.

C. Fault injection

We evaluated the effectiveness of our DSS solution against software bugs by means of software fault injection, using the methodology described in [32]. We ran the OpenSSH

	Clean only			Clean and dirty		
Benchmark	**BL**	**ER**	**LCt**	**BL**	**ER**	**LCt**
PostMark	1.00	1.03	1.00	1.01	1.07	1.01
File Server	1.00	1.03	1.01	1.01	1.02	1.00
Web Server	1.00	1.00	1.01	1.01	1.01	1.01
OpenSSH	1.00	1.09	1.01	1.00	1.09	1.01

Table VI. Performance relative to unmodified Loris.

benchmark 80 times, each time injecting a set of 100 random faults into the cache process at once at a random time during the benchmark. After each completed run, we checked the file system contents against those of a normal run to ensure that no corruption was propagated.

In 74 of the 80 cases, the cache crashed as a result of the fault injection, but was able to recover using the DSS. In all these cases, the benchmark completed and the final check passed. In the remaining six cases, other parts of the storage stack crashed because of deviating behavior of the cache. These cases violate our assumptions, but may still be recoverable if we harden these other layers. In no cases did the cache detect an internal checksum mismatch in the DSS or its dirty pages. Thus, the checksums had no added value in this experiment. We believe this is mainly because the OpenSSH benchmark is not write-intensive.

We then switched to the more write-intensive PostMark benchmark. We first changed it to verify the results of all its system calls, so as to detect any propagated corruption. We then ran it 80 times, each time injecting 100 faults of the *destination* type. This fault type simulates corruption by altering the destination of random instructions [32]. This time, the cache crashed 79 times. In 75 cases, the cache recovered and the benchmark completed successfully. In the other four cases, the cache detected a DSS or page checksum mismatch, and decided that it could not recover. Without the DSS and page checksums, the restarted cache would have propagated corruption in these cases. Finally, in the 80th case, one of the other layers crashed.

VIII. RELATED WORK

We list the most directly related research on memory errors and software bugs.

A. Memory errors

We are not aware of any previous research that evaluates the use of disk checksums to counter memory corruption on a single system. A Lustre design document shows how servers send ZFS disk checksums along with file data to ensure network traffic integrity, noting that with approach, the clients will detect any memory corruption in the server cache as a side effect [33]. Zhang et al [20] study disk and memory corruption effects on ZFS, and find that neither ZFS nor ext2 deal well with local memory corruption; we build on their suggestions in this work.

Generic software memory corruption detection and recovery techniques have been proposed [34], [22], [35]. However, these approaches are unable to leverage specific knowledge about the data they operate on, and thus require higher checksum generation costs, offer less effective detection strategies, and/or require more runtime resources for eventual recovery. Nevertheless, they can still be applied to other parts of system and application memory.

B. Software bugs

We are not aware of any work specifically addressing recovery from bug-induced errors in the page cache. Again, more

generic techniques can be applied. These are typically rollback based. For example, the Akeso system [27] tracks Linux kernel state changes on a per-request basis, committing the changes only at the end of the request. Compiler-based techniques are used to prevent arbitrary memory corruption. When applied to the page cache, this results in expensive copies of all dirty data; the authors show a substantial overhead in write-intensive workloads. A similar technique uses microkernels for better scalability [23], but has the same basic overheads.

CuriOS' Server State Regions (SSRs) address a similar problem by making client state available to a microkernel system server only during a request from that client [26]. If the server crashes, only the active client is killed. We believe that SSRs are not well-suited for a page cache, as it inherently shares pages between clients.

The Rio file cache [36] protects its pages by remapping them read-only when executing kernel code outside the page cache routines. This approach does not help if the cache routines themselves contain bugs, nor does it help detect memory errors. Due to space constraints, we omit a large body of other work on software bugs, but we note that many approaches (e.g., language-based ones) can not deal with memory errors.

IX. Conclusion

In this paper, we have shown that by using specific knowledge about the operation of the storage stack, we can effectively deal with certain reliability threats at a relatively low cost. We have addressed the threats of memory errors and software bugs in the page cache, and shown that there is a two-way interaction between the solutions.

Even though we have focused on the Loris storage stack in this work, we believe that the techniques presented here are sufficiently generic that they can be applied elsewhere: the techniques to detect memory errors can be applied in virtually any page cache that can be involved in on-disk checksums, and the techniques to recover from software bugs can be applied on any other microkernel.

Acknowledgment

References

[1] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum, "Loris - A Dependable, Modular File-Based Storage Stack," in *PRDC*, 2010.

[2] J. F. Ziegler and W. A. Lanford, "The Effect of Cosmic Rays on Computer Memories," *Science*, vol. 206, no. 4420, pp. 776–788, 1979.

[3] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh, "Field testing for cosmic ray soft errors in semiconductor memories," *IBM J. Res. Dev.*, vol. 40, no. 1, pp. 41–50, Jan. 1996.

[4] E. Normand, "Single event upset at ground level," *IEEE Trans. Nuclear Science*, vol. 43, no. 6, pp. 2742–2750, 1996.

[5] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *DSN*, 2006.

[6] Tezzaron Semiconductor, "Soft errors in electronic memory–a white paper," 2004.

[7] X. Li, K. Shen, M. C. Huang, and L. Chu, "A memory soft error measurement on production systems," in *USENIX ATC*, 2007.

[8] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," in *SIGMETRICS*, 2009.

[9] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design," in *ASPLOS*, 2012.

[10] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," 1997.

[11] A. Messer, P. Bernadat, G. Fu, D. Chen, Z. Dimitrijevic, D. Lie, D. D. Mannaru, A. Riska, and D. Milojicic, "Susceptibility of commodity systems and software to memory soft errors," *IEEE Trans. Comput.*, vol. 53, no. 12, pp. 1557–1568, 2004.

[12] A. Eto, M. Hidaka, Y. Okuyama, K. Kimura, and M. Hosono, "Impact of neutron flux on soft errors in MOS memories," in *IEDM*, 1998.

[13] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *ISSTA*, 2002.

[14] L. Hatton, "Reexamining the Fault Density-Component Size Connection," *IEEE Software*, vol. 14, no. 2, pp. 89–97, 1997.

[15] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Construction of a Highly Dependable Operating System," in *EDCC*, 2006.

[16] D. C. van Moolenbroek, R. Appuswamy, and A. S. Tanenbaum, "Integrated End-to-End Dependability in the Loris Storage Stack," in *HotDep*, 2011.

[17] Intel Corporation, "Intel SSE4 Programming Reference," 2007.

[18] STMicroelectronics, "STM32 Reference Manual," 2011.

[19] P. Koopman, "32-bit cyclic redundancy codes for internet applications," in *DSN*, 2002.

[20] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "End-to-end data integrity for file systems: a ZFS case study," in *FAST*, 2010.

[21] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "A file is not a file: understanding the I/O behavior of Apple desktop applications," in *SOSP*, 2011.

[22] D. Dopson, "SoftECC: A System for Software Memory Integrity Checking," Master's thesis, Massachusetts Institute of Technology, 2005.

[23] C. Giuffrida, L. Cavallaro, and A. S. Tanenbaum, "We crashed, now what?" in *HotDep*, 2010.

[24] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, "Characterization of linux kernel behavior under errors," in *DSN*, 2003.

[25] W.-L. Kao, R. K. Iyer, and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults," *IEEE Trans. Software Engineering*, vol. 19, no. 11, pp. 1105–1118, 1993.

[26] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "CuriOS: Improving Reliability through Operating System Structure," in *OSDI*, 2008.

[27] A. Lenharth, V. S. Adve, and S. T. King, "Recovery domains: an organizing principle for recoverable operating systems," in *ASPLOS*, 2009.

[28] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a distributed file system," in *SOSP*, 1991.

[29] "Slicing-by-8," http://slicing-by-8.sourceforge.net/.

[30] Intel Corporation, "Fast CRC Computation for iSCSI Polynomial Using CRC32 Instruction," 2011.

[31] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright, "A nine year study of file system and storage benchmarking," *ACM Trans. Storage*, vol. 4, no. 2, pp. 5:1–5:56, May 2008.

[32] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Fault Isolation for Device Drivers," in *DSN*, 2009.

[33] Sun Microsystems, "Lustre: End to End Data Integrity Design," 2009.

[34] P. Shirvani, N. Saxena, and E. McCluskey, "Software-implemented EDAC protection against SEUs," *IEEE Trans. Reliability*, 2000.

[35] D. Fiala, K. B. Ferreira, F. Mueller, and C. Engelmann, "A tunable, software-based DRAM error detection and correction library for HPC," in *EuroPar*, 2011.

[36] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The Rio file cache: surviving operating system crashes," in *ASPLOS*, 1996.