Checksums on Modern Filesystems, or: On the Virtuous Consumption of CPU Cycles

Alex Garnett
Research Data Management & Systems Librarian
Simon Fraser University
garnett@sfu.ca

Mike Winter
Senior Software Developer
Compute Canada
mike.winter@usask.ca

Justin Simpson
Director, Archivematica Technical Services
Artefactual Inc.
jsimpson@artefactual.com

Abstract

Computing checksums to prevent bit-rot is accepted wisdom in the digital preservation community. Yet in other domains, this wisdom is approached quite differently. New hashing algorithms continue to be developed in the cryptography community, typically with very different use cases in mind, focusing on encryption and security over integrity or identification. Checksumming is also a key feature of modern filesystems. The implementers of these filesystems concern themselves with block level integrity, rather than focus on 'files' or objects in the way digital preservation systems do. Cloud based object storage systems also compute checksums, providing integrity guarantees as part of the service. And there is the blockchain - distributed peer to peer systems where hashing is fundamental.

How do we reconcile these different approaches to bit level preservation using checksums? Can we compare the costs, in terms of compute resources or time, of the different approaches. Is there a way to verify the accepted wisdom of the digital preservation community and reconcile this with the diverse and expanding approaches to checksum validation?

This paper describes how checksumming functionality is understood and implemented in modern filesystems. A cost analysis is presented, comparing different approaches to data integrity, including pure CPU checksumming with tools such as md5sum, the block level metadata used by filesystems such as ZFS, and the contrast with data integrity done by cloud service providers object storage services. From this analysis we describe the benefits of developing a new standard for mapping the block level metadata produced by filesystem checksum reporting tools to the file-centered checksum reporting and validation required for adherence to current expectations of digital preservations accepted best practices. By better understanding different approaches to data integrity, it is possible to make better use of the computer hardware dedicated to digital preservation, taking advantage of the increased computational efficiency of file system level checksumming techniques. This work closes a gap between current best practices in digital preservation and in high-performance computing. Sample code paths for working with and validating block checksums are also demonstrated.

Introduction

"Bit rot," as it is used by digital preservation practitioners, is not a widely understood concept. The Wikipedia entry is a disambiguation page (Wikipedia, 2016), leading to 1) a short story of the same name, 2) a different page on "Software rot," which in digital preservation is generally taken to be an opposite problem, but which the Guardian (and, apparently, Vint Cerf) still include under the term Bit rot (Gibbs, 2015), and, finally, 3) to a Wikipedia page on "Data degradation" (Wikipedia, 2018), which accurately describes the problem as it is understood in digital preservation but as of this writing has a very active and contentious discussion on its internal "Talk" page about whether it ought to be merged with the article on Data corruption.

To some extent this is not surprising. End users generally have different amount of error tolerance baked into their expectations of different parts of computing ecosystems: websites may go down sometimes, and data entered into a browser may not be saved faithfully, but it is probably fair to say that most people who have never tried to load Commodore 64 software from audiocassette (or who repressed the memory of doing so) expect 100% reliability from a computer's file system[1] -- especially a filesystem which, in their perception, is functionally inert -- and would not want to even think about, let alone use, a filesystem which was less than 100% reliable.

Although research into Bit rot in consumer use cases was already being published by the mid-90s (O'Gorman et al., 1996), most of the computing industry has treated it as equivalent to hard disk failures, that is, a failure scenario which will a) always be captured through hardware monitoring, and b) be best addressed through redundancy. Meanwhile in the digital preservation community, we continue to see Bit rot monitoring introduced at the application level. The Islandora repository software includes a "Checksum Checker" module (Jordan, 2016) which is designed to periodically loop through every object in a repository and run the equivalent of a `shasum` on however many TB of files it's pointed at. Archivematica includes an analogous "fixity" tool (Artefactual, 2017) which will periodically report the result of full-collection checksums to the administrator dashboard.

The use of these tools has been enthusiastically adopted as a best practice for good reason: among the advantages provided by full-fat digital preservation, as opposed to the control group of an average IT department, is the ability to confidently guard against and systematically report Bit rot. This has become increasingly necessary as entire filesystems are virtualized and made to span multiple disks below the visible layer at which digital preservation departments typically work with them; the likelihood of insidious errors from data moving around in the backend has, if anything, increased (Addis et al., 2011). What is not necessarily appreciated by digital preservation practitioners, however, is that the hardware cost of performing these checks has increased along the same lines, and is probably nonzero even as far as they themselves are concerned.

When no one has a "bare metal" server dedicated to a single application stack anymore, and any time that one virtual server spends idle can be leased out to another, CPU cycles have a cost figure attached to them. Amazon has made this especially clear for any libraries or archives using their infrastructure, and many local

[1] A "filesystem" refers to the way that computers structure files and folders on a disk; for most of the past two decades, Microsoft has used filesystems called FAT32 and NTFS, Apple has used HFS and HFS+, and Linux has used ext2 and ext4. While older filesystems have some limitations around large files (for example, FAT32, which is still occasionally seen on SD cards and USB drives, doesn't support individual files larger than 4GB), they otherwise have relatively similar "features".

providers (and long-suffering VPS systems administrators) are following suit. Most digital preservation software has relatively modest systems requirements; it generally needs to serve a web app of some kind, plus do some amount of processing on ingest of new files, but can be safely assumed to be near-idle most of the rest of the time. Still, scalability, especially to very large individual files or very large ingested packages, is already a going concern for many of these platforms; many known issues are invisible below the terabyte scale and worsen quickly thereafter. By comparison, most solutions to calculating checksums in software scale indefinitely, but they do so linearly, using more CPU time for however many bytes are in a file. Incorporating a periodic job which churns through an entire multi-terabyte collection in this way can, in our observed cases, increase the total annual CPU consumption of a reasonably voluminous repository by a factor of ten or more. By a significant margin, checksum validation uses more CPU than any other component of a digital preservation stack.

Calculating full checksums in software is not efficient. There is also an argument to be made that it also increases wear and tear on the disks themselves, actually accelerating degradation, by repeatedly accessing large files that would normally not be read that frequently under normal use. If we agree, however, that we should be doing it, and (crucially) reporting it, we ought to examine alternative methods of monitoring checksums -- chiefly, at the hardware level -- with the goal of marrying our best practices with those of the high performance computing sector.

Implicit Checksumming

Checksums are a special class of hash functions, designed to ensure easy checking of data integrity by reducing any arbitrary-size data chunk to a fixed-length value. If the checksum of a data chunk changes it means the contents of the chunk have changed as well. Hashing is often mentioned in the context of cryptocurrencies and encryption, but it always entails the same process of running a scrambler (i.e. hash) on some input and producing predictable output. Sometimes hashing is performed in a way that makes it near-impossible to get the original input back from the output unless you have a special user-specific key that was used by the hash process -- this is the basis of encryption -- but all that differentiates our preservation checksums from other hashes is that they're designed to be run on entire files[2], and produce short strings which can then be easily compared. Because of the way these hashes work, changes on the order of a single bit between two files will produce a totally different checksum, making verification simple.

Several of the oldest and most common hashing algorithms, such as *md5* or *sha1*, are no longer considered secure or viable for use in encryption because it is too easy to brute-force your way to guessing the original input that corresponds to a hashed output on a powerful enough computer; ironically, this makes them more attractive in a digital preservation context, because we are mainly concerned with generating a hash as quickly as possible for record-keeping purposes. However, this doesn't mean that checksum monitoring is an inactive research and development domain! In fact, virtually all of the new filesystems introduced over the past decade have incorporated methods of tracking and validating checksums implicitly every time a file is accessed. These

[2] Audiovisual preservation has a practice of checksumming individual frames of video files along similar lines, to further localize any corruption errors. In this way a preservation checksum is not necessarily a full file checksum, but is always functionally similar.

filesystems -- Linux/Oracle's ZFS and BTRFS, Microsoft's ReFS, and (to a lesser degree) Apple's APFS[3] -- were all designed with the goal of making checksum monitoring near-automatic to prevent data loss.

Essentially, ZFS, BTRFS, and ReFS work by writing checksums to disk as metadata at the same time as files themselves are written, so that errors can be raised implicitly whenever the files are later accessed and the checksums do not match. There are only two aspects of this that are less than ideal for digital preservation use cases. Firstly, files would still have to be accessed individually for checksums to be "checked" in a literal sense; however, this could be tooled around -- and may not be necessary in every context, assuming proper error reporting on the disk itself. Second, these checksums are not calculated on a per-file basis; they are calculated on a *per-block* basis, usually 4KB or 64KB. These blocks are not necessarily as meaningful for preservationists as-is -- most of our tools are developed around monitoring individual files -- and would necessitate a re-orientation of checksum reporting on individual files or preservation objects toward the entire filesystem.

In a typical preservation system items can and will go unread for large periods of time, which means there will be large periods of time without implicit checksumming. This is why ZFS and ZFS-like filesystems recommend explicit filesystem ‘scrubbing’, usually run via a cron job every few weeks, in which the filesystem itself runs a low-priority process to check the integrity of each block. The scrubbing process, activated by one line in a cron file, removes the need for external fixity or integrity checking programs. Additionally, the ZFS Event Daemon for Linux can be installed to inform preservation staff when an error is detected, whether during a normal read or a scrub.

Fixing Data Errors Automatically and Additional Data Integrity Features

Beyond detecting errors automatically, a powerful feature of a checksumming filesystem such as ZFS lies in its ability to automatically restore corrupted data. If a ZFS filesystem is initialized in a RAIDZ (ZFS’ software RAID) or mirrored configuration across multiple physical disks, not only will ZFS detect bit errors but it will automatically and silently fix corrupted blocks using correct copies of the data stored in other locations in the file system. ZFS-likes are even sophisticated enough to detect when a block containing a checksum has been corrupted, rather than the checksum’s corresponding data, and will automatically fix the checksum. As with all RAID configurations, a ZFS-like mirrored or RAIDZ setup will also recover from one or more physical disk failures in a system.

This ability of ZFS-like filesystems to automatically correct corrupted data provides a huge advantage over the fixity-only systems discussed earlier in the paper. A failed fixity check only tells a preservation administrator of corruption in a file while a self-healing ZFS-like filesystem automatically restores the file to its correct state without having to resort to backups and a large amount of manual intervention by the administrator (Cook, 2005).

[3] APFS, though a contemporary of newer, checksum-aware filesystems, still largely infers data integrity from disk health rather than calculating checksums on a per-file or per-block basis the way that ReFS, ZFS, and BTRFS do; Apple justifies this on the grounds that APFS only runs on a narrow range of consumer solid state drives (i.e. those found in iPhones, iPads, and Macs) using storage drivers designed by Apple to provide equivalent monitoring (Leventhal, 2016).

Backups, Easy Data Transfer and Further Integrity Protection

As well as the advantages in data integrity provided by checksumming and self-healing, ZFS-likes also provide easy backup and transfer of data. ZFS-likes provide a filesystem ‘snapshotting’ capability that, because of the use of a copy-on-write (COW) strategy, near-instantaneously generate point-in-time duplicates of a filesystems’ state. These snapshots can be used to rollback a corrupted, damaged or mis-configured filesystem to an earlier known-good state. In addition, snapshots can be easily transferred using send/receive functionality that allows a snapshot to transferred as a single file then imported into a ZFS/BTRFS installation on a different host. This functionality can be used as a standard backup solution or as an alternative to a LOCKSS-style item distribution. The zfs-auto-snapshot tool can be used to easily configure and automate this process on Linux.

The COW implementation of snapshotting, in which snapshots only contain files that have changed from the previous snapshot, allows for incremental backups/offsite mirroring rather than a full transfer of the filesystem each time the process in run. This greatly reduces the transfer and storage overhead needed for a preservation system. Another advantage ZFS-likes offer in terms of resource usage efficiency is dynamic compression to reduce the size of data on disk. As long as enough CPU is provisioned on the storage host, the decompression of data in real-time is not noticeable. This feature is especially suitable for a preservation deployment where large amounts of data with low access rates are the norm.

In addition to being fanatical about the use and care of data checksums in persistent storage (ZFS-likes go so far as to store redundant copies of all checksums across disk areas that are separate from the data itself so combined data/checksum corruption is less likely) a common best practice is to use error-correcting code (ECC) RAM on the same machine to further reduce the possibility of data corruption. ECC memory, much like ZFS-like filesystems, checksums all data values it is storing and automatically corrects any errors that occur. RAM is vulnerable to ‘bit-flipping’, the volatile storage equivalent of bit rot. Bit flips usually occur when background radiation causes a single bit in a memory word to change state. This is of special concern for ZFS-likes, which compute checksums in memory. If the checksum calculation is corrupted in RAM before it is written to disk then all of the careful measures taken to ensure data integrity on disk are for naught.

Not yet Suitable for Cloud Installations and Other Caveats

A properly configured host with a ZFS-like filesystem, ECC memory and continuous snapshotted backups is a fantastic tool for digital preservationists to ensure very high levels of data integrity in their collections. However, there are caveats to be aware of when setting up such a system. The largest is that a high-integrity ZFS-like configuration is not yet suitable for deployment on a cloud. ZFS-likes require direct access to the physical storage assigned to a host to properly ensure that the physical state of the storage media is the same as its model of that state. This cannot be guaranteed in a cloud environment with its many levels of indirection between an instance and its underlying storage. ZFS-likes can be run on non-cloud virtualized environments such as VMWare or VirtualBox as long as their configuration is setup to allow direct access to disk (for example, enabling PCI passthrough).

In addition, careful attention to monitoring and setup is required with ZFS-likes. ZFS-likes have configurable types of write cache, RAIDZ levels, mirroring, checksumming and compression that should be planned for before a system is deployed. Scrubs, error monitoring, alerts and incremental backups must be designed and configured correctly. Due to the focus on data integrity, the large amount of overhead in checking snapshots and various caching techniques, ZFS-likes require a relatively large amount of (preferably ECC) RAM, suffer declining performance after physical drives become more than 50% full and are generally not advised to be used as swap space or boot disks.

However, for a digital preservation installation that wants to provide the highest possible data integrity over the span of years if not decades, this requirement for meticulous deployment planning is more of a feature than a bug. The administrator of a properly planned and executed ZFS-like is rewarded with a fantastically featured storage tool with data integrity guarantees that are generationally superior to standard solutions.

ZFS and BTRFS have both been in active development since the late 2000s, but have not been very widely adopted due to a combination of a) intellectual property and licensing issues (both Oracle and Red Hat have been feuding around them at various points in time), b) early instability, and c) a lack of salience around specialized filesystems, combined with higher-than-normal memory requirements[4]. Also, nobody likes migrating filesystems. ReFS is comparatively young, but is only available in Microsoft server environments which most digital preservation infrastructure does not use. Adoption is lately increasing, however -- it is only since Ubuntu 16.04 (released in April of 2016, hence 16.04) that ZFS has been available as a default, open source option when partitioning a disk -- and the popularity of Amazon's S3 object storage makes a strong case that not all filesystem types are equally suited to all use cases, encouraging more critical evaluation here. The overhead, in terms of CPU, memory, and disk usage, from using a ZFS partition as your repository storage and making use of its implicit checksum reporting in your stack is theoretically much lower than the overhead from running an md5 hashing tool like `md5sum` on 20TB of packages approximately once per month.

Cost Analysis

Some back-of-the-napkin math is useful here to actualize the cost of periodically calculating checksums in software. A typical Amazon EC2 server instance with 4 CPUs -- an "m1.xlarge" instance in this case -- currently costs about $0.35 USD per hour to run (Heaton, 2018), not including configuration time. A comparable four-core desktop CPU can compute an `md5sum` checksum on a 15GB file in just under 90 seconds (Ma, 2015). Therefore, if we multiply the average run-time upward to a repository containing 20TB of packages, assuming a good linear read speed of at least 150MB/s and not too much overhead from logging data, making API calls, seeking on the disk in between files, etc., we can conclude that it would require about 30 hours to checksum the entire 20TB, or about $10 USD at current EC2 rates. Assuming that checksums are checked in this manner about once a month, and adding some degree of server overhead to our estimate, periodic checksum monitoring with current techniques would cost approximately $200 USD per year in CPU time alone, or 15 days out of every year for that four-core machine.

While this is not a very large sum of money for many institutions, it represents a fairly idealized environment -- many institutions cannot leverage Amazon's cloud services due to different jurisdictional rules around privacy and security -- and is, in any case, helpful to put a number to. In a less idealized environment, i.e., one in which your CPU cycles cannot be priced out because they're competing with other apps on the same homegrown server, the opportunity cost of calculating checksums in software, plus I/O blocking, etc., is likely much higher.

[4] Earlier versions of ZFS were so proprietary and so questionable in terms of reliability that writing a paper which could fairly be characterized as "we should all migrate to ZFS" as recently as five years ago would effectively paint you as an Oracle shill with not-great judgment.

ZFS Pseudocode example

Adoption is probably a major problem in recommending a modern filesystem; most digital preservation implementers are more expert in software than hardware, and when provisioning (or requisitioning) generic disk, you are, as of this writing, more likely to get ext4 than ZFS due to the former's lower memory requirements and a perception that ZFS is for more specialized use cases. However, there is also the increasing adoption of dedicated non-filesystem *object storage* (such as Amazon S3) to consider; the middleware managing these volumes arguably obviates the need for manual `` `md5sum` ``-style checksumming, and our infrastructure should be responsive to this. An adaptive solution would be to start shipping preservation software like Archivematica with two separate code paths for bit-level preservation checks: the current one that runs `` `md5sum` `` on each package individually, and an alternative that simply reads the data from a checksum-aware filesystem. The barrier to doing this is small -- it only requires an agreed-upon standard for mapping filesystem checksum reporting to current expectations around `` `md5sum` ``, some of which is provided in this paper -- and more importantly, having the code paths in place would provide much more solid ground with which to negotiate with infrastructure providers or budget EC2 allocations. It also closes a gap between current best practices in digital preservation and in high-performance computing.

Per the documetation (Oracle Corporation, 2010), the best way to interact with a ZFS diagnostic engine is with the `zpool status` command; a call to `zpool status -v` (with the verbose flag) produces the following sample output:

```
# zpool status -v
  pool: tank
 state: UNAVAIL
status: One or more devices are faulted in response to IO failures.
action: Make sure the affected devices are connected, then run 'zpool clear'.
   see: http://www.sun.com/msg/ZFS-8000-HC
 scrub: scrub completed after 0h0m with 0 errors on Tue Feb  2 13:08:42 2010
config:

        NAME        STATE     READ WRITE CKSUM
        tank        UNAVAIL      0     0     0  insufficient replicas
          c1t0d0    ONLINE       0     0     0
          c1t1d0    UNAVAIL      4     1     0  cannot open

errors: Permanent errors have been detected in the following files:

/tank/data/aaa
/tank/data/bbb
/tank/data/ccc
```

Currently, Archivematica's Storage Service calls the validate function from the Python-Bagit module when a fixity scan is initiated (manually or automatically) on a given AIP:

https://github.com/artefactual/archivematica-storage-service/blob/3cada6fe040697f4aa684836e4b2eb7d3b19ab50/storage_service/locations/models/package.py#L1647. This function is looped through and called for each AIP individually when a full scan ("scan all") is requested.

In a ZFS environment, we could instead replace the "scan all" function with a single Python call to zpool like so (and alias the regular scan to the "scan all" behaviour so we only have a single call for checking the entire disk), which checks for errors:

```
error_files = subprocess.Popen(["zpool status -v | sed -n -e '/errors/,$p'"],
stdout=subprocess.PIPE)
```

We could then quickly parse the output of the fixity scan line-by-line to identify the problematic files.

Conclusions and Other Notes

Filesystems are an area of relatively active and exciting development in commodity computer hardware, and we believe that they are of greater interest to digital preservation implementers than has so far been observed. While we have discussed several problematic aspects of implementing modern, checksum-aware filesystems such as ZFS, we believe that the current state of the art for monitoring bit rot in digital preservation systems is also not necessarily ideal, and would benefit from consideration of these new developments. Bit-level presentation is generally touted as one of the most significant differentiators between our tools and those of other infrastructure providers; however, our approach to solving this problem in software is agnostic to any middleware solutions that may be more helpful, and potentially expensive relative to our other preservation infrastructure, limiting the opportunity for a mutual appreciation of value by systems administrators and other stakeholders. The digital preservation community understands better than others that not all storage is equal, and the adoption of modern filesystems could be an asset for improving communication across teams and increasing overall confidence in our services.

One final point of clarification we'd like to make is that running a full checksum in software immediately before and after files are transferred between two disks (or using the functionality built into a tool like rsync to do this automatically as part of the transfer) is always good practice, and not obsoleted by any of the other solutions here; our recommendations only pertain to scheduled checksum validation.

References

Addis, M., Wright, R., & Weerakkody, R. (2011). Digital Preservation Strategies: The Cost of Risk of Loss. SMPTE Motion Imaging Journal, 120(1), 16–23. https://doi.org/10.5594/j18003XY

Artefactual. (2017). Fixity checking and reporting - Archivematica. Retrieved April 13, 2018, from https://wiki.archivematica.org/Fixity_checking_and_reporting

Cook, T. (2005). Demonstrating ZFS Self-Healing. Retrieved April 15, 2018, from https://blogs.oracle.com/timc/demonstrating-zfs-self-healing

Gibbs, S. (2015, February 13). What is “bit rot” and is Vint Cerf right to be worried? Retrieved April 13, 2018, from http://www.theguardian.com/technology/2015/feb/13/what-is-bit-rot-and-is-vint-cerf-right-to-be-worried

Heaton, G. (2018). Amazon EC2 Instance Comparison. Retrieved April 13, 2018, from https://www.ec2instances.info/

Jordan, M. (2016). *islandora_checksum_checker: Islandora module to verify datastream checksums*. PHP, Islandora. Retrieved from https://github.com/Islandora/islandora_checksum_checker (Original work published 2014)

Leventhal, A. (2016). APFS in Detail: Data Integrity. Retrieved April 13, 2018, from http://dtrace.org/blogs/ahl/2016/06/19/apfs-part5/

Ma, E. Z. (2015, September 5). Which Checksum Tool on Linux is Faster? Retrieved April 13, 2018, from https://www.systutorials.com/136737/which-checksum-tool-on-linux-is-faster/

O’Gorman, T. J., Ross, J. M., Taber, A. H., Ziegler, J. F., Muhlfeld, H. P., Montrose, C. J., … Walsh, J. L. (1996). Field testing for cosmic ray soft errors in semiconductor memories. *IBM Journal of Research and Development*, *40*(1), 41–50. https://doi.org/10.1147/rd.401.0041

Oracle Corporation. (2010). Resolving Problems with ZFS. Retrieved April 13, 2018, from https://docs.oracle.com/cd/E19253-01/819-5461/gbbuw/index.html

Wikipedia. (2016, October 16). Bit rot. In *Wikipedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Bit_rot&oldid=744695994

Wikipedia. (2018, April 5). Data degradation. In *Wikipedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Data_degradation&oldid=834480542