GassyFS: An In-Memory File System That Embraces Volatility

Noah Watkins, Michael Sevilla, Carlos Maltzahn
University of California, Santa Cruz
{*jayhawk,msevilla,carlosm*}@*soe.ucsc.edu*

Abstract

For many years storage systems were designed for slow storage devices. However, the average speed of these devices has been growing exponentially, making traditional storage system designs increasingly inadequate. Sufficient time must be dedicated to redesigning future storage systems or they might never adequately support fast storage devices using traditional storage semantics.

In this paper, we argue that storage systems should expose a persistence-performance trade-off to applications that are willing to explicitly take control over durability. We describe our prototype system called *GassyFS* that stores file system data in distributed remote memory and provides support for checkpointing file system state. As a consequence of this design, we explore a spectrum of mechanisms and policies for efficiently sharing data across file system boundaries.

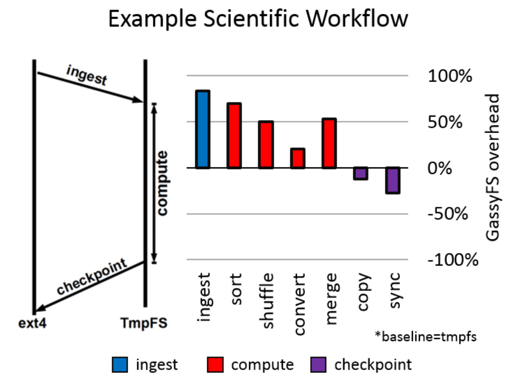

Figure 1: GassyFS is comparable to tmpfs for genomic benchmarks that fit on one node, yet it is designed to scale beyond one node. Its simplicity and well-defined workflows for its applications lets us explore new techniques for sharing data.

1 Introduction

A wide range of data-intensive applications that access large working sets managed in a file system can easily become bound by the cost of I/O. As a workaround, practitioners exploit the availability of inexpensive RAM to significantly accelerate application performance using in-memory file systems such as *tmpfs* [10]. Unfortunately, durability and scalability are major challenges that users must address on their own.

While the use of tmpfs is a quick and effective solution to accelerating application performance, in order to bound data loss inherent with memory-based storage, practitioners modify application workflows to bracket periods of high-performance I/O with explicit control over durability (e.g. checkpointing), creating a trade-off between persistence and performance. But when single-node scalability limits are reached applications are faced with a choice: either adopt a distributed algorithm through an expensive development effort, or use a transparent scale-out solution that minimizes modifications to applications. In this paper we consider the latter solution, and observe that existing scale-out storage systems do not provide the persistence-performance trade-off despite applications already using a homegrown strategy.

Figure 1 shows how our system, GassyFS, compares to tmpfs when the working set for our genomics benchmarks fit on a single node. That figure demonstrates the types of workflows we target: the user ingests BAM input files (10GB, 40MB), computes[1] results, and checkpoints the results back into durable storage (24GB) using `cp` and `sync`. The slowdowns reported by Figure 1 demonstrate that GassyFS has performance comparable to tmpfs, in our limited tests, with the added benefit that it

[1]The workload uses SAMtools, a popular Genomics tool kit. The jobs process are (input/output sizes): sort (10GB/10GB), shuffle (10GB/52MB), convert into VCF (40MB/160MB), merge (20GB/13.4GB)

is designed to scale beyond one node. Scaling the workload to volumes that do not fit into RAM without a distributed solution limits the scope of solutions to expensive, emerging byte-addressable media.

One approach is to utilize the caching features found in some distributed storage systems. These systems are often composed from commodity hardware with many cores, lots of memory, high-performance networking, and a variety of storage media with differing capabilities. Applications that wish to take advantage of the performance heterogeneity of media (e.g. HDD vs SSD) may use caching features when available, but such systems tend to use coarse-grained abstractions, impose redundancy costs at all layers, and do not expose abstractions for volatile storage—memory is only used internally. These limitations force applications that explicitly manage durability from scaling through the use of existing systems. Furthermore, existing systems originally designed for slow media may not be able to fully exploit the speed of RAM over networks that offer direct memory access. What is needed are scale-out solutions that allow applications to fine-tune their durability requirements to achieve higher performance when possible.

Existing solutions center on large-volume high-performance storage constructed from off-the-shelf hardware and software components. For instance, iSCSI extensions for RDMA can be combined with remote RAM disks to provide a high-performance volatile block store formatted with a local file system or used as a block cache. While easy to assemble, this approach requires coarse-grained data management, and I/O must be funneled through the host for it to be made persistent. Non-volatile memories with RAM-like performance are emerging, but software and capacity are a limiting factor. While easy to setup and scale, the problem of forcing all I/O related to persistence through a host is exacerbated as the volume of data managed increases. The inability of these solutions to exploit application-driven persistence forces us to look to other architectures.

In the remainder of this paper we present a prototype file system called GassyFS[1], a distributed in-memory file system with explicit checkpointing for persistence. We view GassyFS as a stop-gap solution to providing applications with access to very high-performance large volume persistent memories that are projected to be available in the coming years, allowing applications to begin exploring the implications of a changing I/O performance profile. As a result of including support for checkpointing in our system, we encountered a number of ways that we envision data to be shared without crossing file system boundaries, by providing data management features for attaching shared checkpoints and external storage systems, allowing data to reside within the context of GassyFS as long as possible.

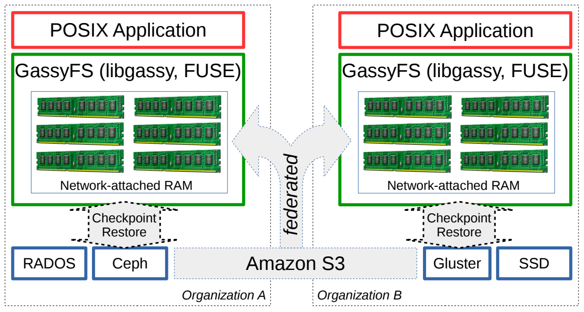

Figure 2: GassyFS has facilities for explicitly managing persistence to different storage targets. A checkpointing infrastructure gives GassyFS flexible policies for persisting namespaces and federating data.

2 Architecture

The architecture of GassyFS is illustrated in Figure 2. The core of the file system is a user-space library that implements a POSIX file interface. File system metadata is managed locally in memory, and file data is distributed across a pool of network-attached RAM managed by worker nodes and accessible over RDMA or Ethernet. Applications access GassyFS through a standard FUSE mount, or may link directly to the library to avoid any overhead that FUSE may introduce.

By default all data in GassyFS is non-persistent. That is, all metadata and file data is kept in memory, and any node failure will result in data loss. In this mode GassyFS can be thought of as a high-volume tmpfs that can be instantiated and destroyed as needed, or kept mounted and used by applications with multiple stages of execution. The differences between GassyFS and tmpfs become apparent when we consider how users deal with durability concerns.

At the bottom of Figure 2 are shown a set of storage targets that can be used for managing persistent checkpoints of GassyFS. We will discuss durability in more detail in the next section. Finally, we support a form of file system federation that allows checkpoint content to be accessed remotely to enable efficient data sharing between users over a wide-area network. Federation and data sharing are discussed in Section 6.

3 Durability

Persistence is achieved in GassyFS using a checkpoint feature that materializes a consistent view of the file system. In contrast to explicitly copying data into and out of tmpfs, users instruct GassyFS to generate a checkpoint of a configurable subset of the file system. Each checkpoint may be stored on a variety of supported backends such as local disk or SSD, a distributed storage system such as Ceph, RADOS or GlusterFS, or to a cloud-based service such as Amazon S3.

A checkpoint is saved using a copy-on-write mechanism that generates a new, persistent version of the file system. Users may forego the ability to time travel and only use checkpointing to recover from local failures. However, generating checkpoint versions allows for robust sharing of checkpoints when one file system is actively being used.

GassyFS takes advantage of its distributed design to avoid the single-node I/O bottleneck that is present when persisting data stored within tmpfs. Rather, in GassyFS each worker node performs checkpoint I/O in parallel with all other nodes, storing data to a locally attached disk, or to a networked storage system that supports parallel I/O. Next we'll discuss the modes of operation and how content in a checkpoint is controlled.

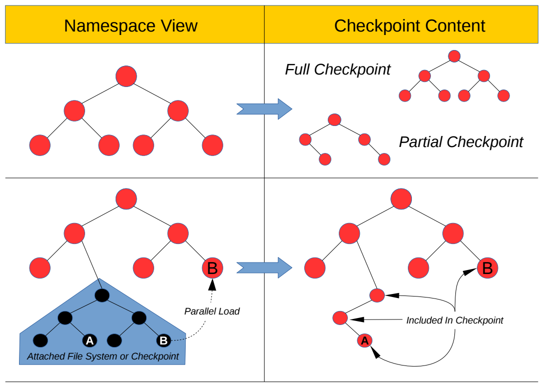

Figure 3: Namespace checkpointing modes. Top: full/partial checkpoints, Middle: checkpointing attached file systems (e.g., connected with POSIX), Bottom: checkpointing inode types (e.g., code to access remote data)

4 Extensible Namespaces

In the previous section we discussed how persistence is achieved in GassyFS using a basic versioning checkpoint scheme. In this section we discuss how the content of each checkpoint can be customized using a set of policies provided by an application or user. We consider two broad scenarios, illustrated in Figure 3. The first column in the figure represents the state of the GassyFS namespace at a particular point-in-time, and each red (light) node represents a component of the GassyFS namespace fully managed by GassyFS (i.e. metadata and data). The right column depicts which content may be included in the checkpoint.

In the first row of Figure 3 the GassyFS standard mode of operation is illustrated in which all data is fully managed by GassyFS. When a checkpoint is created a configurable subset of the namespace is persisted within the checkpoint being created.

The second row illustrates the flexibility of GassyFS to integrate with external resources. The highlighted sub-tree represents an attached external namespace constructed from either a remote file system or a mounted checkpoint. Each node in this sub-tree is black, depicting that its namespace is visible but its data is not managed by GassyFS. In this mode content contained in the attached system can be selectively included in a checkpoint.

Write-back persistence. By attaching an external file system as a sub-tree within GassyFS persistence can be achieved using a basic write-back policy or using explicit flushing. In this way GassyFS can be configured to behave like a scalable buffer-cache. Note that simply constructing an overlay mount on the host would achieve the same result, but has two drawbacks. First, since the mount is not managed by GassyFS it has no ability to integrate the external content or maintain references within a checkpoint. And second, all I/O to the external mount must flow through the host, preventing features like parallel I/O that take advantage of the GassyFS worker nodes.

5 Programmable File Types

GassyFS includes experimental support for file types that allow users to specify storage system policies programmatically. The motivation for this feature stems from single node file systems, where many policies are hard-coded into the implementation. For example, the ext4 file system uses 5 "tricks" in its block and inode allocation: (1) multi-block allocation, (2) delayed allocation, (3) co-locating data and inodes in the same block, (4) co-locating directory and its inode in the same block, and finally (5) 128MB block groups [3]. Distributed file systems serve different workloads and are significantly larger and more complicated than single node file systems; they have more resources, larger variations in latencies, and more failures.

Exposing these policies is especially important for GassyFS because the target workloads and their bottlenecks are well-understood. The user already knows that circumventing storage IO with tmpfs is a valid strategy for keeping the CPU busy, so they probably also know how to get high performance from their application. Our aim is to give these users the tools to programmatically adjust their system using domain-specific knowledge.

GassyFS embeds a Lua virtual machine wherever a policy decision is made. This effectively decouples policy from mechanism, which helps future designers explore trade-offs and isolates policy development from code-hardened systems. Decoupling policy mechanism is not a new technique but designing the storage system with programmability as a first-class citizen is. For example, McKusick credits policy/mechanism separation as the biggest reason for the success of the block allocation mechanisms in the Fast File System [6] yet there are no mechanisms for transparently exposing and modifying the underlying logic in modern file systems.

Scripting is useful in large systems for 3 reasons:

- accelerates deployment: tweaking policies, like changing a threshold or metric calculation, does not require a full recompile or system start-up.
- transparency: if the bindings carefully expose the correct metrics and functions, the user does not need to learn the system's internal functions, variables, or types.
- portability: scripts can be exchanged across platforms and system versions without compatibility issues.

We use Lua, the popular embedded scripting language, to define dynamic policies in GassyFS. Although Lua is an interpreted language, it runs well as modules in other languages and the bindings are designed to make the exchange of variables and functions amongst languages straightforward. The LuaJIT virtual machine boasts near-native performance, making it a logical choice for scriptability frameworks in systems research [13]. In storage systems, it has been effectively used both on [5, 14] and off [9] the critical path, where performance is important.

Lua is an ideal candidate for storage system programmability. It is small both in its memory footprint and its syntax has been shown to be ideal for embedding in systems [13]. It's portability allows functions to be shipped around the cluster without having to compile or reason about compatibility issues. And finally, Lua provides relatively robust sandbox capabilities for protecting the system against some malicious cases, but primarily from poor policies.

6 Federation

The use of checkpoints in GassyFS to achieve persistence encourages users to maintain data within GassyFS. Of course this is not possible to do in perpetuity, and ultimately data must be exported into other domains, archived, or shared. Driven by proliferation of container technologies such as Docker we observe that dissemination of environments through file system checkpoints is becoming an accepted practice which has proven highly beneficial for reproducibility. GassyFS draws inspiration from this trend, adding additional features for efficiency.

GassyFS supports features that allow users to collaborate by forming ad-hoc links between shared GassyFS checkpoints. Sharing can take place over private connections such as SSH, or by using a storage backend such as Amazon S3 to host checkpoints. A user may either mount an entire checkpoint, or attach a checkpoint into an existing namespace, allowing data to be selectively fetched on demand without the cost of retrieving an entire checkpoint.

7 Implementation and Evaluation

GassyFS passes nearly all of the Tuxera POSIX correctness suite, built and run unit tests for Git and PostgreSQL, IOzone configuration tests, and can successfully build the Linux kernel, and Ceph on multiple nodes. A simple block allocator and checkpointing facility are implemented and our next step is to implement extensible namespaces and programmable file types.

Most of our performance analysis has been done with the Genomics SAMtools benchmark. When scaling the number of threads on a single node, GassyFS approaches the performance of tmpfs and we observe no more than $1.6\times$ overhead for any of our configurations. Interestingly, when testing GassyFS in pseudo-distributed node (*i.e.* multiple processes on the same node) we see performance increases. We suspect that either we get unforeseen parallelism with multiple processes or Linux does not favor large `mmap()`s. Regardless, this result indicates that spreading GassyFS processes or chopping memory across nodes or within nodes (or a mix) might help performance.

8 Related Work

MemFS [12, 11] is an in-memory parallel file system that stripes file content across a set of nodes that contribute memory to a storage pool. The system is used in HPC environments to provide fast, locality-agnostic access to intermediate files generated in many-task computing (MTS) workflows. The MemFS file system is purpose-built for MTS, sacrificing POSIX semantics. Write buffering, sequential prefetch, load-balancing, and elasticity optimizations are used by MemFS to enhance performance, but the system provides no fault-tolerance capabilities.

The Network RamDisk [4] is a virtual block device in which data is distributed across remote memory, and can be formatted as a standard local file system. The system supports a fully volatile mode, or a mode in which parity or replication is used to provide resilience against one node failure. Since the block device abstraction is opaque, file system-specific features are difficult to realize. The dRamDisk [8] project is similar, but adds an adaptive prefetch policy that expands the read-ahead region.

More recently Mühleisen et. al [7] assemble a remote RAM disk using an RDMA interconnect to export remote RAM over NFS as a set of local loopback block devices that may be aggregated at the block or file system level (e.g. using mdadm or btrfs). The multiple levels of indirection involved in this approach at best introduce overhead, but make efficient data management difficult.

Chai et. al [1] examine the performance of pNFS over Infiniband using RAM-based storage, demonstrating that the storage stack can take advantage of high-performance media by showing performance gains over more traditional storage such as disk. Dahlin et. al [2] present cooperative caching, which shows that file system performance can increase when clients pool their memory resources together, further validating the benefits of remote memory caches.

Lua has been used in operating and storage system design to provide support for flexible policies. In storage systems Lua has been used as a control in metadata and data services. Sevilla et al. [9] use Lua to define metadata balancing policies in the Ceph file system in which Lua policies are periodically evaluated avoiding any hot code path. Grawinkel et al. [5] used Lua along the critical path to implement distribution policies decisions for file data in the pNFS storage cluster, giving them the flexibility to change the access semantics and functionality on the fly. In our previous work on storage programmability we use Lua in the RADOS object store to define storage interfaces, and show that these code fragments must be treated with the same durability concerns as data in order to ensure data can continue to be accessed [14]. Neto et al. [13] use Lua to make policies for network packet filtering and CPU throttling. The overheads reported are a testament to the effectiveness and practicality of using Lua in a system along the fast path.

9 Conclusion

The emergence of software-defined storage helps systems evolve to meet user needs through new abstractions and the introduction of new media. However, control over persistence is one area that is lacking. Exposing these abstractions helps applications, already making explicit persistence-performance trade-offs, utilize different resources (e.g. RAM) in storage systems to meet their needs. We have developed GassyFS and positioned it as platform for exploring these ideas. Throughout this process, we discovered that by checkpointing file system state we uncover interesting methods to share data without leaving the context of the file system.

References

[1] Chai, L., Ouyang, X., Noronha, R., and Panda, D. K. pnfs/pvfs2 over infiniband: Early experiences. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07* (New York, NY, USA, 2007), PDSW '07, ACM, pp. 5–11.

[2] Dahlin, M. D., Wang, R. Y., Anderson, T. E., and Patterson, D. A. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1994), OSDI '94, USENIX Association.

[3] Ext4. Ext4 Disk Layout.

[4] Flouris, M. D., and Markatos, E. P. The network ramdisk: Using remote memory on heterogeneous nows. *Cluster Computing 2*, 4 (1999), 281–293.

[5] Grawinkel, M., Sub, T., Best, G., Popov, I., and Brinkmann, A. Towards Dynamic Scripted pNFS Layouts. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis* (Washington, DC, USA, 2012), SCC '12, IEEE Computer Society, pp. 13–17.

[6] McKusick, M. K. Keynote Address: A Brief History of the BSD Fast Filesystem, February 2015.

[7] Mühleisen, H., Gonçalves, R., and Kersten, M. Peak performance: Remote memory revisited. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware* (2013), DaMoN '13, pp. 9:1–9:7.

[8] Roussev, V., Richard, G. G., and Tingstrom, D. dramdisk: efficient ram sharing on a commodity cluster. In *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International* (April 2006), pp. 6 pp.–198.

[9] Sevilla, M. A., Watkins, N., Maltzahn, C., Nassi, I., Brandt, S. A., Weil, S. A., Farnum, G., and Fineberg, S. Mantle: A Programmable Metadata Load Balancer for the Ceph File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2015), SC '15.

[10] Snyder, P. tmpfs: A virtual memory file system. In *Proceedings of the Autumn 1990 EUUG Conference* (1990), pp. 241–248.

[11] Uta, A., Sandu, A., Costache, S., and Kielmann, T. In *CCGrid Scale Challenge '15* (2015).

[12] Uta, A., Sandu, A., and Kielmann, T. Overcoming data locality: An in-memory runtime file system with symmetrical data distribution. *Future Generation Computer Systems 54* (2016), 144 – 158.

[13] Vieira Neto, L., Ierusalimschy, R., de Moura, A. L., and Balmer, M. Scriptable Operating Systems with Lua. In *Proceedings of the 10th ACM Symposium on Dynamic Languages* (New York, NY, USA, 2014), DLS '14, ACM, pp. 2–10.

[14] Watkins, N., Maltzahn, C., Brandt, S., Pye, I., and Manzanares, A. In-Vivo Storage System Development. In *Euro-Par 2013: Parallel Processing Workshops* (2013), Springer, pp. 23–32.

Notes

[1]http://github.com/noahdesu/gassyfs