WHAT'S SO BAD ABOUT POSIX I/O?

September 11, 2017 Glenn Lockwood

POSIX I/O is almost universally agreed to be one of the most significant limitations standing in the way of I/O performance exascale system designs push 100,000 client nodes.

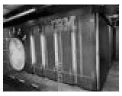
The desire to kill off POSIX I/O is a commonly beaten drum among high-performance computing experts, and a variety of new approaches—ranging from I/O forwarding layers, user-space I/O stacks, and completely new I/O interfaces—are bandied about as remedies to the impending exascale I/O crisis.

However, it is much less common to hear exactly *why* POSIX I/O is so detrimental to scalability and performance, and *what* needs to change to have a suitably high-performance, next-generation I/O model. To answer the question of why POSIX I/O is holding back I/O performance today and shed light on the design space for tomorrow's extreme-scale I/O systems, it is best to take a critical look at what POSIX I/O really means.

UNDERSTANDING THE POSIX I/O OF TODAY

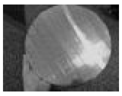
At a high level, there are two distinct components to POSIX I/O that are often conflated—the POSIX I/O API and POSIX I/O semantics. The POSIX I/O API is probably the most familiar, because it is how applications read and write data. It encompasses many of the

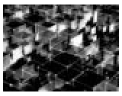
POPULAR ARTICLES


IBM MASHES UP POWERAI AND WATSON MACHINE LEARNING STACKS

SHASTA SETS UP CRAY FOR COMMERCIAL HPC AND SUPERCOMPUTING EXPANSION

WEATHER SERVICE DOUBLES UP ON CRAY SUPERCOMPUTER


IBM PUFFS UP POWER IRON ON ITS PUBLIC CLOUD


PCI-EXPRESS LENDS SERVER COMPOSABILITY A HAND

calls with which any scientific programmer should be familiar including open(), close(), read(), write(), and lseek(), and as such, are an integral part of today's applications and libraries.

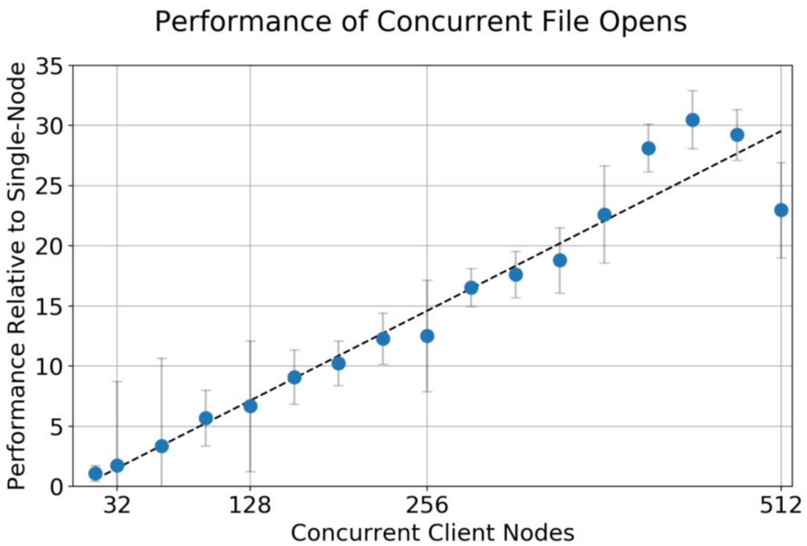
Much less obvious are the semantics of POSIX I/O, which define what is and is not guaranteed to happen when a POSIX I/O API call is made. For example, the guarantee that data has been committed to somewhere durable when a write() call returns without an error is a semantic aspect of the POSIX write() API. While this notion of write() being honest about whether it actually wrote data may seem trivial, these semantics can incur a significant performance penalty at scale despite not being strictly necessary.

The combination of the API and its semantics give rise to a few distinct features of POSIX I/O that have a measurable and adverse effect on scalability and performance.

POSIX I/O IS STATEFUL

A typical application might open() a file, then read() the data from it, then seek() to a new position, write() some data, then close() the file. File descriptors are central to this process; one cannot read or write a file without first open()ing it to get a file descriptor, and the position where the next read or write will place its data is generated by where the last read, write, or seek call ended. Thus, **POSIX I/O is stateful**; reading and writing data is governed by some persistent state that is maintained by the operating system in the form of file descriptors.

Because the operating system must keep track of the state of every file descriptor—that is, every process that wants to read or write—this stateful model of I/O provided by POSIX becomes a major scalability bottleneck as millions or billions of processes try to perform I/O on the same file system. For example, consider the time it takes to simply open a file across a large number of compute nodes:



The cost of opening a file on most parallel file systems scales linearly with the number of clients making those requests. That is to say, before one can even *begin* to actually read or write data using a POSIX interface, the **statefulness** of the POSIX I/O model incurs this performance tax that gets increasingly worse when scaling up.

POSIX I/O HAS PRESCRIPTIVE METADATA

POSIX I/O also prescribes a specific set of metadata that all files must possess. For example, programmers may be familiar with POSIX I/O calls such as chmod() and stat() or the eponymous shell commands that provide command-line interfaces for these calls. They manipulate the metadata that POSIX dictates all files must possess such as the user and group which owns the file, the permission that user and group has to read and modify the file, and attributes such as the time the file was created and last modified.

While the POSIX style of metadata certainly works, it is **very prescriptive and inflexible**; for example, the ownership and access permissions for files are often identical within directories containing scientific data (for example, file-per-process checkpoints), but POSIX file systems must track each of these files independently. At the same time, this metadata scheme is usually not descriptive enough for many data sets, often resulting in README files effectively storing the richer metadata that POSIX does not provide.

Supporting the prescriptive POSIX metadata schema at extreme scales is a difficult endeavor; anyone who has tried to ls -l on a directory containing a million files can attest to this. Despite this, the **inflexibility of this POSIX metadata** schema usually has users figuring out alternative ways to manage, ranging from README files in every directory or elaborately named files to complex metadata management systems such as **JAMO**, **iRODS**, or **Nirvana**.

POSIX I/O HAS STRONG CONSISTENCY

Perhaps the biggest limitation to scalability presented by the POSIX I/O standard is not in its API, but in its semantics. Consider the following semantic requirement taken from the **POSIX 2008 specification for the write() function**:

> *After a write() to a regular file has successfully returned:*
>
> *– Any successful read() from each byte position in the file that was modified by that write shall return the data specified by the write() for that position until such byte positions are again modified.*
>
> *– Any subsequent successful write() to the same byte position in the file shall overwrite that file data.*

That is, writes must be **strongly consistent**–that is, a write() is required to block application execution until the system can guarantee that any other read() call will see the data that was just written. While this is not too onerous to accomplish on a single workstation that is writing to a locally attached disk, ensuring such strong consistency on networked and distributed file systems is very challenging.

THE PAINS OF ENSURING POSIX CONSISTENCY

All modern operating systems utilize page cache to soften the latency penalty of ensuring POSIX consistency. Instead of blocking the application until the data is physically stored on a slow (but nonvolatile) storage device, write()s are allowed to return control back to the application only after the data is written to a memory page. The operating system then tracks "dirty pages" which contain data that hasn't been flushed to disk and writes those dirty pages from memory back into their intended files asynchronously. The POSIX strong consistency guarantee is still satisfied because the OS tracks cached pages and will serve read() calls directly out of memory if a page is already there.

Unfortunately, page cache becomes much more difficult to manage on networked file systems because different client nodes who want to read and write to the same file will not share a common page cache. For example, consider the following I/O pattern:

1. node0 writes some data to a file, and a page in node0's page cache becomes dirty
2. node1 reads some data *before* node0's dirty pages are flushed to the parallel file system storage servers
3. node1 doesn't get node0's changes from the storage servers because node0 hasn't told anyone that it's holding dirty pages
4. the read() call from node1 returns data that is *inconsistent* from that data on node0
5. POSIX consistency was just violated

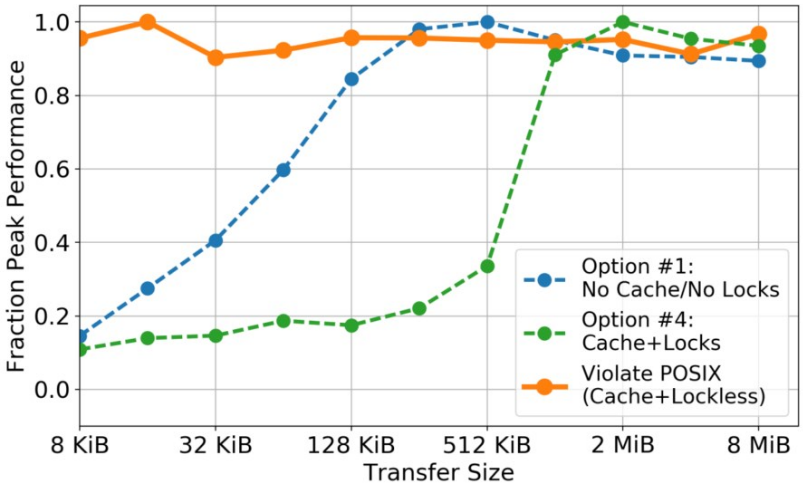
In order to adhere to POSIX's strong consistency, parallel or networked file systems must ensure that no nodes are able to read data until dirty pages have been flushed from page cache to the back-end storage servers. This means that parallel file systems must either

1. not use page cache at all, which drives up the I/O latency for small writes since applications block until their data traverses the network and is driven all the way down to a hard drive
2. violate (or "relax") POSIX consistency in a way that provides consistency for most reasonable I/O workloads (e.g., when different nodes modify non-overlapping parts of a file) but provide no consistency guarantees if two nodes try to modify the same part of the same file
3. implement complex locking mechanisms to ensure that a node cannot read from a file (or part of a file) that *may* be being modified by another node

All of these options are viable, but they come with different tradeoffs. For example,

1. Cray's DataWarp file system uses approach #1 to provide a simple, lock-free, strongly consistent I/O semantics. There is a latency penalty associated with small transaction sizes since each write() has to go over the network no matter what its size, but this performance loss is softened by DataWarp's underlying NVMe-based storage media.
2. NFS employs approach #2 and guarantees "close-to-open" consistency–that is, the data in a file is only consistent between the time a file is closed and the time it is opened. If two nodes open the same file at the same time, it is possible for them to hold dirty pages, and it is then up to client applications to ensure that no two nodes are attempting to modify the same page.
3. PVFS2 (now OrangeFS) use a combination of option #1 and #2; it does not utilize a page cache, but it also does not serialize overlapping writes to ensure consistent data. This has the benefit of ensuring consistency as long as different nodes do not attempt to modify overlapping *byte ranges* at the same time. This consistency guarantee is stronger than NFS-style consistency, which is only consistent when non-overlapping *pages* are modified.
4. Parallel file systems like Lustre and GPFS use approach #3 and implement sophisticated distributed locking mechanisms to ensure that dirty pages are always flushed before another node is allowed to perform overlapping I/O. Locking mechanisms can suffer from scalability limitations when application I/O is not judiciously crafted to explicitly avoid scenarios where multiple nodes are fighting over locks for overlapping extents.

To illustrate, consider the following basic bandwidth tests over a range of transfer sizes to file systems which take differing approaches to POSIX consistency:



The lock-free case (corresponding to option #1) suffers at small I/O sizes because it cannot benefit from write-back caching, which allows small writes to coalesce into large RPCs. Similarly, implementing locking (option #4) causes write that are smaller than the lock granularity (1 MiB in the above case) to contend, effectively undoing the benefits of write-back caching. It is only when POSIX is thrown to the wind and caching is enabled without locking that performance remains consistently high for all I/O sizes.

The POSIX consistency model also makes the other two scalability limits, statefulness and metadata prescription, even worse. For example, the stateful nature of I/O, when combined with the need to implement locking, means that parallel file systems usually have to track the state of every lock on every region of each file. As more compute nodes open files and obtain read or write locks, the burden is on the parallel file system to manage which client nodes possess which locks on which byte ranges within all files. Similarly, metadata such as mtime and atime must be updated in concert with fully consistent reads and writes. This effectively requires that every time a file is *read*, a new value for the file's last-accessed time (POSIX atime) must be *written* as well. As one can imagine, this imposes a significant burden on the part of the parallel file system that manages POSIX metadata.

MOVING BEYOND POSIX I/O FOR HPC

In the grand scheme of things, POSIX I/O's API and semantics made a great deal of sense for individual computer systems with serial processors and workloads. However, HPC poses the unique challenge of being highly concurrent (with the largest file systems having to directly serve 20,000 clients) and often multi-tenant (with large systems often running tens or hundreds of independent jobs that touch unrelated files). And yet in HPC, there are many cases where the rigidity of POSIX I/O is not strictly necessary. For example,

1. At extreme scales, applications never rely on the file system to deliver full strong consistency on parallel I/O. The performance associated with lock contention has historically been so bad that applications either ensure that no two processes are trying to write to the same part of a file at once.
2. On multi-tenant systems, it is extremely rare for two independent jobs to write to a common shared file; thus, full consistency of all data across the entire compute system is usually overkill. Thus, the *consistency domain* of any given file can often be shrunk to a small subset of compute nodes that actually manipulate the file, and other jobs that are working on completely different files need not cope with the performance penalty of ensuring global consistency.

These aspects of POSIX I/O, its API, and its semantics are the principal roadblocks that prevent POSIX-compliant file systems from efficiently scaling to tomorrow's extreme-scale supercomputing systems. In fact, these roadblocks are nothing new; the file systems used in distributed systems are already riddled with exceptions to full POSIX compliance to work around these issues:

- Lustre's **local flock option** reduces the consistency domain to a single compute node
- Every file system has a noatime option to reduce the burden of maintaining strong consistency of POSIX metadata
- **Cray's Data Virtualization Service** provides a deferopens option which relaxes the statefulness of parallel open operations

However, the industry is rapid approaching a time where the best solution may be to acknowledge that, despite its tidiness and broad adoption, POSIX I/O is simply not what HPC needs. The next step, of course, is to then define exactly what aspects of POSIX I/O are essential, convenient, and unnecessary, and then build a new set of I/O APIs, semantics, and storage systems that deliver only those features that extreme-scale computing needs.

OUTLOOK FOR A POST-POSIX I/O WORLD

Fortunately, answering these questions have been the focus of a significant amount of research and development. The hyperscale industry has made great strides to this end in defining completely new APIs for I/O, including the S3 and Swift, which optimize for scalability and throughput by stripping away most of the semantics that have made POSIX complicated to implement. Of course, these simplified semantics still remain suitable for the target write-once, read-many (WORM) workloads prolific in hyperscale, but they also prevent them from being a viable alternative for HPC applications that must interact with data in a more sophisticated manner.

There are efforts within the HPC space to find a better compromise between extreme scalability and flexible semantics, and several high-performance object stores are under heavy development with an eye towards the HPC market. Intel's **Distributed Asynchronous Object Store (DAOS)** has been detailed by ***The Next Platform*** and remains under **active and open development**. More recently, Seagate has begun developing **Mero** as its asynchronous, high-performance object store for exascale **in partnership with the European Horizon 2020 program**. Both of these systems treat both the POSIX API and its semantics as secondary features that are implemented atop a much richer API that eschew strong consistency for a combination of asynchronous and transactional semantics. By providing a lower-level I/O interface, applications that need to scale beyond POSIX have the ability to trade in the simple POSIX interface for one that was designed to enable extreme concurrency and scalability.

That said, even the most mature extreme-performance object stores remain in research and development at present. As such, these systems have not yet converged on a standard API or semantics, and although some high-value extreme-scale applications may be willing to rework their application code to most effectively utilize such new storage systems, it is difficult to envision a wholesale abandoning of the POSIX I/O API in the forseeable future.

Of course, the POSIX I/O API and the POSIX I/O semantics are not inseparable. It is entirely possible for future I/O systems to provide significant portions of the POSIX I/O API for application portability, but at the same time, dramatically change the underlying semantics to sidestep the scalability challenges it imposes. The **MarFS project, also previously detailed at *The Next Platform***, is an excellent example of this in that it defines POSIX API calls on top of S3- or Swift-like put/get semantics. While MarFS is not trying to solve a performance problem, it does demonstrate an interesting approach to moving beyond POSIX without forcing application I/O routines to be rewritten. Those applications that can adhere to a more restrictive set of POSIX I/O operations (such as WORM workloads) can run unmodified, and the reality is that **many extreme-scale HPC applications already perform well-behaved I/O** to cope with the limitations of POSIX.

Thus, the future of extreme-scale I/O is not bleak. There are many ways to move beyond the constraints imposed by POSIX I/O without rewriting applications, and this is an area of active research in both industry and academia. Advances in hardware technology (including storage-class memory media such as **3D XPoint**) and implementation (such as user-space I/O frameworks including **SPDK** and **Mercury**) continue to pave tremendous runway and opportunity for innovation in parallel I/O performance.

GLENN LOCKWOOD

Contributing Author

Glenn K. Lockwood is a member of the Advanced Technologies Group (ATG) at NERSC at Lawrence Berkeley National Lab. Lockwood specializes in I/O performance analysis, extreme-scale storage architectures, and emerging I/O technologies and

APIs. His research interests revolve around understanding I/O performance by correlating performance analysis across all levels of the I/O subsystem, from node-local page cache to back-end storage devices. To this end, he is actively involved in the **TOKIO project**, a collaboration between Lawrence Berkeley and Argonne National Labs to develop a framework for holistic I/O performance characterization, and the **Cori burst buffer project** at NERSC. Prior to joining ATG, Glenn was an analyst at the San Diego Supercomputer Center where he provided collaborative support for data-intensive computing projects in areas including ecology and genomics.

8 COMMENTS

Andrew Baptist says:

SEPTEMBER 11, 2017 AT 6:59 PM

Very good analysis about the limitations of file systems. As you note they were created at a time of direct attached storage, where strong consistency was cheap to provide.

You discuss some alternatives to POSIX, however I think you dismiss the ability of S3 or Swift to replace many of the workloads that HPC perform today. From a developers perspective, if they could have an infinite, fast sorted key-value store, many applications, would be able to transition over them quickly.

The two primary drawbacks of Amazon S3 today are 1) High latency responses (measured in 100+ ms) and 2) Eventual consistency (similar to the Cache + Lockless option you discuss for parallel file systems). However neither of these are implicit in either the S3 protocol or in the implementation of the service. Google Cloud Storage and IBM Cloud Object Storage both provide strong consistency guarantees. From a latency perspective, there is nothing intrinsic about Object Storage that makes it slower, and Pure Flash Object Storage will provide an S3 interface with low latency.

I think the days of POSIX are numbered, and the advantages you discuss with hardware and user-space implementations of I/O will make the gap between object interfaces and POSIX even more pronounced.

REPLY

Glenn K. Lockwood says:

SEPTEMBER 14, 2017 AT 1:51 PM

S3/Swift are just not designed for performance, and latency is the massive show-stopper. There's no point in buying a flash-based burst buffer if you're spending 100+ ms in software encoding REST calls and then sending it all over TCP. That data path makes sense over wide-area networks, but there are better solutions for the much more performance-oriented and small design spaces in HPC. Objects and key-values will definitely gain a lot of traction in HPC over the next decade, but performance-sensitive I/O workloads will be using more HPC-oriented interfaces. I would imagine S3 will be used to perform the transfers between the performance-optimized storage and longer-term archive tiers.

REPLY

Anonymous Coward says:

JUNE 3, 2018 AT 3:43 PM

If you need to 100+ ms to encode an http request, you're doing something wrong. (Not that S3 is unacceptable for low latency calls – the response time can take tens of ms, and that's IMO too much, for some usage scenarios. But the killer latency source is IME never the encoding itself.)

REPLY

Oren Tirosh says:

SEPTEMBER 12, 2017 AT 1:59 AM

Using nonvolatile memory the page "cache" can be the actual durable storage. I put "cache" in quotes because there are no two locations representing the same file range that can potentially be out of sync. At that point in time, it is the authoritative copy. If it is later written to some other form of cheaper storage (e.g. NVMe flash) it is not the flushing of a cache but rather logically moved to it, as in a tiered storage system.

If the nonvolatile memory is directly addressable with RDMA it is possible for nodes in a cluster to have high performance instantly consistent read-after-write using fine-grained RDMA access (less that page size).

If I understand correctly, this is exactly what the M1 filesystem from Plexistor does (now acquired by NetApp)

REPLY

John Bent says:

SEPTEMBER 13, 2017 AT 10:52 AM

Oren, I like what you say about networked RDMA removing much of the complexity. It definitely solves the caching problem. However, I fear that it doesn't solve the striping problem that I describe in the penultimate paragraph in the comment I've left below. I'm curious to hear what you think. Thanks also for the reference to M1 which I had not previously encountered.

REPLY

Suryanarayanan says:

SEPTEMBER 13, 2017 AT 1:27 AM

Great article which has good justification for why to look beyond POSIX to achieve exascale performance where I/O is the bottleneck. You had covered many upcoming development in this area and has missed out DDN storage who are already having product in this space with the advancements you had mentioned in this article. Good to have a look at it for the enthusiasts at their Infinite Memory Engine (IME) product (http://www.ddn.com/products/ime-flash-native-data-cache/)

REPLY

John Bent says:

SEPTEMBER 13, 2017 AT 10:49 AM

Glenn, great article; thanks so much for writing it. I'll add a few thoughts that I've had when I've thought about this same topic.

You've done a beautiful job explaining that there is both a POSIX API as well as POSIX semantics. Then POSIX compliance means that a file system honors POSIX semantics. I think there is simple explanation for what people expect when they say POSIX compliant: the behavior of a distributed file system serving multiple processes on multiple client nodes is the same as the behavior of a local file system serving multiple threads on a single node with direct attached storage.

From this expectation, we can think about how the behavior of a distributed system can deviate from the behavior of a local system. For this, it is useful to think of a threat model of what can go wrong. As you say, one prime candidate can be caches. So long as the same byte range can reside in multiple locations, there is the possibility for violating POSIX compliance.

The other danger comes from striping which you mention when you discuss PVFS. And, in fact, what I'm about to describe is something that I first heard from Rob Ross: POSIX compliance can be violated even in a system that doesn't have caches when data is striped across multiple server nodes. Imagine a scenario where a file's contents are 'AA' and the first byte is stored on server0 and the second on server1. Now one process overwrites with 'BB' and another overwrites with 'CC' while a third does a read. This is a race. Even on a local system this is a race but the read should return only one of three possible values: 'AA', 'BB', or 'CC.' In a distributed system, it is easy to imagine how this POSIX compliance can be violated and a read might incorrectly return something like 'AB' or 'CA' or 'BC.'

These two thread models for data consistency can then be extended to consider how metadata consistency expectations can be violated.

REPLY

James Coomer says:

SEPTEMBER 18, 2017 AT 7:09 AM

Hi Glenn,

Great explanation and I very much agree with your summary statement: "There are many ways to move beyond the constraints imposed by POSIX I/O without rewriting applications, and this is an area of active research in both industry and academia".

But it is not just research… you mentioned a lot of efforts, but not DDN's Infinite Memory Engine (IME – **http://www.ddn.com/products/ime-flash-native-data-cache/**) which instantiates, in production today an NVMe-based scale-out cache that uses another approach to get around POSIX limitations. IME runs in front of Lustre or GPFS and presents MPIIO and POSIX APIs directly to applications. It comprises an extremely high throughput distributed hash table (DHT) which resiliently stores internal metadata, co-located with an erasure coded, log-structured, bulk data cache. Data flowing into IME either from Lustre/GPFS or from the application is stored at byte granularity to avoid the need for locking and consequent scaling limitations inherent in block/page granular caches. IME also finesses Amdah's law since the layout of this data is determined dynamically and tracked by metadata stored in the DHT. This allows IME to keep all its servers fully loaded and performing at maximum bandwidth despite speed variations or even complete server failure.

REPLY

LEAVE A REPLY

Your email address will not be published.

Comment

Name *

Email *

Website

POST COMMENT

This site uses Akismet to reduce spam. Learn how your comment data is processed.

RECENT ARTICLES

IBM Mashes Up PowerAI And Watson Machine Learning Stacks

Shasta Sets Up Cray For Commercial HPC And Supercomputing Expansion

Weather Service Doubles Up on Cray Supercomputer

IBM Puffs Up Power Iron On Its Public Cloud

PCI-Express Lends Server Composability A Hand

ABOUT CONTRIBUTORS CONTACT NEWSLETTER BOOKS PODCAST THE REGISTER