Buffered I/O without page-cache thrashing

By **Jonathan Corbet**
December 12, 2019

Linux offers two modes for file I/O: buffered and direct. Buffered I/O passes through the kernel's page cache; it is relatively easy to use and can yield significant performance benefits for data that is accessed multiple times. Direct I/O, instead, goes straight between a user-space buffer and the storage device. It can be much faster for situations where caching by the operating system isn't necessary, but it is complex to use and contains traps for the unwary. Now, it seems, Jens Axboe has come up with a way to get many of the benefits of direct I/O with a lot less bother.

Direct I/O can give better performance than buffered I/O in a couple of ways. One of those is simply avoiding the cost of copying the data between user space and the page cache; that cost can be significant, but in many cases it is not the biggest problem. The real issue may be the effect of buffered I/O on the page cache.

A process that performs large amounts of buffered I/O spread out over one or more large (relative to available memory) files will quickly fill the page cache (and thus memory) with cached file data. If the process in question does not access those pages after performing I/O, there is no benefit to keeping the data in memory, but it's there anyway. To be able to allocate memory for other uses, the kernel will have to reclaim some pages from somewhere. That can be expensive for the system as a whole, even if "somewhere" is the data associated with this I/O activity.

The memory-management subsystem tries to do the right thing in this situation. Pages added to the cache via buffered I/O go onto the inactive list; unless they are accessed a second time in the near future, they will be the first pages to be kicked back out. But there is still a fair amount of overhead associated with implementing this behavior; Axboe ran a simple test and described the results this way:

> The test case is pretty basic, random reads over a dataset that's 10x the size of RAM. Performance starts out fine, and then the page cache fills up and we hit a throughput cliff. CPU usage of the IO threads go up, and we have kswapd spending 100% of a core trying to keep up.

This kind of problem can be avoided by switching to direct I/O, but that brings challenges and problems of its own. Axboe has concluded that there may be a third way that can provide the best of both worlds.

That third way is a new flag, `RWF_UNCACHED`, which is provided to the `preadv2()` and `pwritev2()` system calls. If present, this flag changes the requested I/O operation in two ways, depending on whether the affected file pages are currently in the page cache or not. When the data is present in the page cache, the operation proceeds as if the `RWF_UNCACHED` flag were not present; data is copied to or from the pages in the cache. If the pages are absent, instead, they will be added to the page cache, but only for the duration of the operation; those pages will be removed from the page cache once the operation completes.

The result, in other words, is buffered I/O that does not change the state of the page cache; whatever was present there before will still be there afterward, but nothing new will be added. I/O performed in this way will gain most of the benefits of buffered I/O, including ease of use and access to any data that is already cached, but without filling memory with unneeded cached data. The result, Axboe says, is readily observable:

> With this, I can do 100% smooth buffered reads or writes without pushing the kernel to the state where kswapd is sweating bullets. In fact it doesn't even register.

This new flag thus seems like a significant improvement for a variety of workloads. In particular, workloads where it is known that the data will only be used once, or where the application performs its own caching in user space, may well benefit from running with the `RWF_UNCACHED` flag.

The implementation of this new behavior is not complicated; the entire patch set (which also adds support to io_uring) involves just over 200 lines of code. Of course, as Dave Chinner pointed out, there is something missing: all of the testing infrastructure needed to ensure that `RWF_UNCACHED` behaves as expected and does not corrupt data. Chinner also noted some performance issues in the write implementation, suggesting that an entire I/O operation should be flushed out at a time rather than the page-at-a-time approach taken in the original patch set. Axboe has already reworked the code to address that problem; the boring work of writing tests and precisely documenting semantics will follow at some future point.

If `RWF_UNCACHED` proves to work as well in real-world workloads, it may eventually be seen as one of those things that somebody should have thought of many years ago. Things often turn out this way. Solving the problem isn't hard; the hard part is figuring out which problem needs to be solved in the first place. That, and writing tests and documentation, of course.

Buffered I/O without page-cache thrashing
Posted Dec 12, 2019 15:09 UTC (Thu) by **epa** (subscriber, #39769) [Link]

But if the data gets added to the cache, even temporarily, that could itself force other things out of the cache or even force swapping to disk. What's the reason for adding to the page cache at all, if you are only going to remove the data immediately afterwards?

Buffered I/O without page-cache thrashing
Posted Dec 12, 2019 15:11 UTC (Thu) by **corbet** (editor, #1) [Link]

I would expect the impact on the cache to be small; a matter of kilobytes rather than gigabytes.

The data is added to the cache because that way the entire buffered I/O setup — both user space and in the kernel — Just Works without additional trouble.

Buffered I/O without page-cache thrashing
Posted Dec 13, 2019 7:34 UTC (Fri) by **epa** (subscriber, #39769) [Link]

But why only kilobytes? What if the request was for much more than that? The article seems to imply the whole request gets into the cache, then once the request is completed it moves out again. Even if it was in fact for multiple pages:

> those pages will be removed from the page cache once the operation completes

And the API for readv() and writev() doesn't have a maximum size. So if there are a million pages to read, do they all go into the cache and get removed afterwards, or can they be removed piecemeal even before the operation is finished?

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 13, 2019 9:19 UTC (Fri) by **tlamp** (subscriber, #108540) [Link]

> But why only kilobytes?

Because a page is normally 4k, and only one page is allocated for a request, even if the request itself is much bigger (e.g., many GB) - IIUC.

With hugepages you may get a bit bigger but still less than the whole thing at once.

https://lwn.net/Articles/807193/

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 13, 2019 10:51 UTC (Fri) by **epa** (subscriber, #39769) [Link]

Ah, so the slowdown and thrashing of the page cache with current kernels is not caused by a few big requests, but by lots of small ones which add one page each. That's why the test case made random small reads, rather than (for example) running grep over a file bigger than memory.

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 12, 2019 15:20 UTC (Thu) by **axboe** (subscriber, #904) [Link]

It'll only stay in the cache briefly. For the latest patchset, the read side doesn't touch the page cache at all, so it's not an issue anymore for that. On the write side we still currently do use the page cache for synchronization purposes. If we can get rid of that too, then we will. But not for reasons of page cache pressure (it doesn't really matter), however on the performance side there's a nice win to not having to muck with the page cache at all.

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 12, 2019 16:29 UTC (Thu) by **nix** (subscriber, #2304) [Link]

I presume this works better than the existing approach of calling posix_fadvise(..., POSIX_FADV_DONTNEED) on the pages in question after you're done with them? (That's what bup does right now -- obviously it can only uncache data pages that way, not metadata...)

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 12, 2019 16:34 UTC (Thu) by **axboe** (subscriber, #904) [Link]

For the read side, this is a LOT more efficient than having to do FADV_DONTNEED afterwards. It's also worth noting that if the data is previously cached, a RWF_UNCACHED read will _not_ drop the pages. Basically the logic is:

1) Lookup page cache page for the read. This is a very cheap operation.
2) If page is there, lock and copy data, done.
3) If page is not there, do IO to private page, copy data, free page, done.

Writes aren't (yet) as optimal, they will always use the page cache. If _any_ page in the written range was not in page cache to begin with, the range is dropped from cache.

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 12, 2019 23:59 UTC (Thu) by **Paf** (subscriber, #91811) [Link]

Jens,

It's not really clear to me where you're piling up here when you're spinning - is this the per file stuff (the old mapping->tree_lock, now an xarray of course) or is this the global provisioning of pages? (Would expect pileup there for a multi-file workload, but not for a single file workload.)

I suppose this could help with either, but it's not 100% clear to me from the patch set description which case is being discussed. (it seems to imply single file)

Presumably write performance improvements are the multi-file case, since AFAIK all the main Linux file systems are single writer per file at the inode mutex. (A limitation it would be fun to see lifted - the currently-out-of-tree Lustre file system has had that for a few years. But there's nothing there to share, the hard part wasn't replacing the inode mutex, it's fixing all the bits of the fs level write path to work with concurrency.)

Anyway, this is very nice!

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 13, 2019 0:48 UTC (Fri) by **axboe** (subscriber, #904) [Link]

It's the page reclaim. Even at moderate throughputs, kswapd spends more time than it should. In some of the various threads off the posted versions of the patchset, I mention that I can get 100% kswapd CPU usage for higher throughputs. Still just about 2GB/sec of IO, nothing major. For ~400MB/sec of IO, I still see 10% kswapd. My test case was 10 files, and 10x RAM size for the data set. For a single file things are much (MUCH) worse. I did 10 files to try and be nice.

But the kswapd usage is just part of it, it's not the main thing for me. With a full page cache, anyone attempting to do buffered IO will suffer. If I can do 2GB/sec of RWF_UNCACHED IO with a basically empty page cache, others doing IO are hardly disturbed.

And agree, buffered writes suck due to the inode mutex. My patchset does nothing to fix that... io_uring has hashed (by inode) items serialized to work around that issue.

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 13, 2019 3:03 UTC (Fri) by **Paf** (subscriber, #91811) [Link]

Hmm, it’s not entirely relevant for this patchset, but in case you’re not already aware, the single file and many files cases are (almost certainly) contending on different locks.

The single file case will be thrashing on the lock protecting that specific mapping, and the many file case will be thrashing on the global stuff, which is much higher throughput. (Mostly because it’s lists rather than an Xarray, so insertion/removal is much faster)

I’m overly familiar with this stuff because I picked up improving page cache throughput for Lustre. (Being out of tree sucks.)

But for reads you avoid putting pages in cache and for writes it seems you just remove them immediately. I’m curious why you get benefits there (presumably in the many files case) vs just rapid flushing of pages by kswapd - if you still have to put pages in the cache and remove them, it’s not immediately obvious where the benefit comes from. Are you skipping something? Are you able to batch? Is kswapd inefficient for some reason? (eg, navigating the LRU list where you can just work through your array)

Sorry for all the questions, and doubly sorry if they’re addressed in parts of the thread I haven’t read :)

“ io_uring has hashed (by inode) items serialized to work around that issue.”
I don’t quite follow that statement (mostly because I have minimal knowledge of io_uring :)) - is this saying that io_uring won’t do parallel submission of writes to the same inode because they’d just block?

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 13, 2019 4:01 UTC (Fri) by **axboe** (subscriber, #904) [Link]

It's not lock contention. I could go into detail about this, but if you go to lore.kernel.org/linux-block and find the discussion (I forget which version if the patchset it was off, probably v1) between Linus, myself, willy you'll get a lot more details than I can convey here in a short comment.

> is this saying that io_uring won’t do parallel submission of writes to the same inode because they’d just block?

Yes, that's exactly right. It's pointless and just causes tons of contention. If this gets fixed at some point we can just lift this restriction, or even do it per fs if it's flagged somehow. As it stands, all the important file systems suffer from this, which is why io_uring behaves that way.

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 13, 2019 14:14 UTC (Fri) by **Paf** (subscriber, #91811) [Link]

Ahh, thanks. I was very confused as to why you’re seeing xarray create, but then I found this comment from epa here:
“ Ah, so the slowdown and thrashing of the page cache with current kernels is not caused by a few big requests, but by lots of small ones which add one page each. That's why the test case made random small reads, rather than (for example) running grep over a file bigger than memory.”

> And now I get it. I thought you were looking at large streaming, which is the case I've worked with in the past. Never mind what I said, then, many small reads would be different (and I've never really benchmarked it). Sorry for my confusion.

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 13, 2019 13:44 UTC (Fri) by **smooth1x** (subscriber, #25322) [Link]

Hi,

"all the main Linux file systems are single writer per file at the inode mutex"

Does that still apply to XFS with O_DIRECT?

Regards,
David.

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 13, 2019 13:54 UTC (Fri) by **axboe** (subscriber, #904) [Link]

It's only for buffered IO, XFS w/O_DIRECT is not serialized.

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 13, 2019 21:28 UTC (Fri) by **dgc** (subscriber, #6611) [Link]

> all the main Linux file systems are single writer per file at the inode mutex"
>
> Does that still apply to XFS with O_DIRECT?

1. there is no "inode mutex" anymore - it's a rwsem. :)
2. buffered writes on every filesystem take the rwsem in exclusive mode, so yes it's still single writer.
3. Filesystems can do what they like with O_DIRECT, so XFS still uses shared writer locking for O_DIRECT.

and...

4. I'm slowly working on range locking for IO in XFS, such that we can do concurrent buffered writes that still have exclusive access guarantees against other buffered/direct IO, truncate, hole punching, etc....

Basically, IO range locking is what io_uring really needs for concurrent buffered IO without needing nasty hacks to avoid exclusive writer serialisation...

-Dave.

-Dave.

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 14, 2019 18:41 UTC (Sat) by **axboe** (subscriber, #904) [Link]

Well, rwsem still means writer exclusive, which was the point :-). But range locking sounds great, when that's done I can get rid of the hashed inode serialization for io_uring on regular file (buffered) writes that it currently does. How will I be able to detect if the fs supports this or not?

Another thing that buffered writes really needs (for io_uring) is RWF_NOWAIT support. Are you (or anyone you're aware of) working on that? I might just take a stab at it if not.

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 16, 2019 3:51 UTC (Mon) by **dgc** (subscriber, #6611) [Link]

I have no plans to make range locking externally detectable - just like you can't tell if a filesystem uses the rwsem as a mutex rather than a rwsem, external code should have no awareness that the filesystem is actually doing things concurrently internally via range locking. Perhaps it could be flagged via a superblock feature flag, but that's something I've not really cared about at this point...

AFAIA no one is working on RWF_NOWAIT for buffered writes. I'm not sure it's worth the trouble at this point in time because any amount of other IO (e.g. other buffered writes) will result in buffered writes always returning -EAGAIN to the caller so you'll just end up punting them all to an async thread, anyway...

-Dave.

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 12, 2019 17:50 UTC (Thu) by **thoughtpolice** (subscriber, #87455) [Link]

Direct I/O puts other unfortunate requirements on you if you wish to use it properly, such as alignment requirements for good performance (the buffer must align with an underlying logical sector size.) Buffered I/O has no such restrictions, so it is much easier to program with, without introducing a lot of logic to handle devices correctly. However, the "pollution" of the page cache, like the test case Jens gave, is a natural consequence of this -- because, by design, buffered I/O must use the page cache.

Therefore, the RWF_UNCACHED flag allows you to keep using the (simpler) buffered I/O interface, but simply fixes the pollution issue. It makes the simpler API less troublesome to use, effectively. The fact that the page is quickly added to the cache and then thrown away is really just a natural consequence of the design. End goal (easier to program) vs means of achieving that goal (quick add/remove), and all that.

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 12, 2019 17:50 UTC (Thu) by **Sesse** (subscriber, #53779) [Link]

I thought this part was interesting:

“Pages added to the cache via buffered I/O go onto the inactive list; unless they are accessed a second time in the near future, they will be the first pages to be kicked back out. But there is still a fair amount of overhead

associated with implementing this behavior”

I didn't know at all that this was the behavior, but it sounds very reasonable to me, so I'm surprised that it's still bad. What is all the overhead about? Simply traversing the inactive list? (It can't be splitting of huge pages, since they are not supported in the page caching yet…)

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 12, 2019 19:28 UTC (Thu) by **hnaz** (subscriber, #67104) [Link]

The main costs are traversing the inactive list and removing the pages from the cache tree. Why is that so expensive? It's actually not, when you look at it on a page-by-page basis. But when you have a low cache hit rate (file set several times larger than RAM) and do IO at 500MB/s or upwards, you are doing hundreds of thousands of these reclaim transactions per second. That adds up. It's the sheer number of operations that is killing us here - all to populate and depopulate a cache tree that we're not really benefiting from...

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 12, 2019 20:00 UTC (Thu) by **Sesse** (subscriber, #53779) [Link]

I guess what confuses me is that I would assume these are cheap operations compared to actually doing the I/O (even though the I/O involves DMA).

In a theoretical world with a hugepage-backed buffer cache, would the equation be any different?

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 12, 2019 20:39 UTC (Thu) by **hnaz** (subscriber, #67104) [Link]

> In a theoretical world with a hugepage-backed buffer cache, would the equation be any different?

It would be, because we'd instantiate and reclaim cache entries in units of 2M (or whatever the huge page size on the architecture) instead of 4k. That's a 512x reduction of list and tree operations for the same amount of data going through.

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 13, 2019 4:17 UTC (Fri) by **champtar** (subscriber, #128673) [Link]

Maybe it's already the case, but would it make sense to reclaim x page each time (say 1024) when you need one and none are available ?

Reply to this comment

Buffered I/O without page-cache thrashing
Posted Dec 13, 2019 6:43 UTC (Fri) by **magnus** (subscriber, #34778) [Link]

How does this work if you're reading blocks of data sequentially through a file in this new mode, does it require each read call to end on a page boundary to not read in the page data at the read block boundaries

twice? Or is the current page at the current file position considered active and not reclaimed from the cache?

Reply to this comment

Buffered I/O without page-cache thrashing

Posted Dec 13, 2019 14:29 UTC (Fri) by **axboe** (subscriber, #904) [Link]

If you're doing sub-page reads continually, you're going to have a bad time. It'll work fine for random access, and for streamed access you really want to ensure you use IOs that are large enough to get full bandwidth. There's no automatic read-ahead with this interface.

Reply to this comment

Buffered I/O without page-cache thrashing

Posted Dec 13, 2019 10:37 UTC (Fri) by **dgm** (subscriber, #49227) [Link]

Very interesting approach. I guess the (obvious?) option of just limiting the size/rate at which a process can add data to the page cache has been tried and discarded. It would be interesting to know why.

Reply to this comment

Buffered I/O without page-cache thrashing

Posted Dec 19, 2019 20:30 UTC (Thu) by **mklwn** (subscriber, #121081) [Link]

Hi,

What kernel is this in or what rev will it go in ?

thx,
-m

Reply to this comment