Why We Started Putting Unpopular Assets in Memory

03/24/2020

Yuchen Wu

Part of Cloudflare's service is a CDN that makes millions of Internet properties faster and more reliable by caching web assets closer to browsers and end users.

We make improvements to our infrastructure to make end-user experiences faster, more secure, and more reliable all the time. Here's a case study of one such engineering effort where something counterintuitive turned out to be the right approach.

Our storage layer, which serves millions of cache hits per second globally, is powered by high IOPS NVMe SSDs.

Although SSDs are fast and reliable, cache hit tail latency within our system is dominated by the IO capacity of our SSDs. Moreover, because flash memory chips wear out, a non-negligible portion of our operational cost, including the cost of new devices, shipment, labor and downtime, is spent on replacing dead SSDs.

Recently, we developed a technology that reduces our hit tail latency and reduces the wear out of SSDs. This technology is a memory-SSD hybrid storage system that puts ***unpopular*** assets in memory.

The end result: cache hits from our infrastructure are now faster for all customers.

You may have thought that was a typo in my explanation of how the technique works. In fact, a few colleagues thought the same when we proposed this project internally, "I think you meant to say ***'popular*** assets' in your document". Intuitively, we should only put popular assets in memory since memory is even faster than SSDs.

You are not wrong. But I'm also correct. This blog post explains why.

Putting popular assets in memory

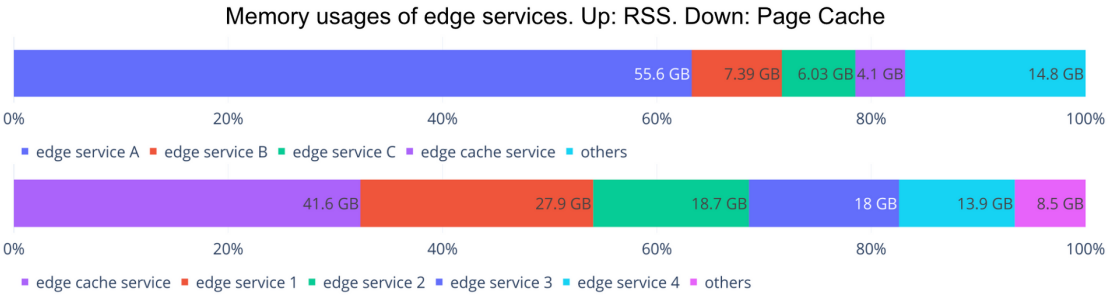
First let me explain how we already use memory to speed up the IO of popular assets.

Page cache

Since it is so obvious that memory speeds up popular assets, Linux already does much of the hard work for us. This functionality is called the "page cache". Files in Linux systems are organized into pages internally, hence the name.

The page cache uses the system's available memory to cache reads from and buffer writes to durable storage. During typical operations of a Cloudflare server, all the services and processes themselves do not consume all physical memory. The remaining memory is used by the page cache.

Below shows the memory layout of a typical Cloudflare edge server with 256GB of physical memory.



We can see the used memory (RSS) on top. RSS memory consumption for all services is about 87.71 GB, with our cache service consuming 4.1 GB. At the bottom we see overall page cache usage. Total size is 128.6 GB, with the cache service using 41.6 GB of it to serve cached web assets.

Caching reads

Pages that are frequently used are cached in memory to speed up reads. Linux uses the least recently used (LRU) algorithm among other techniques to decide what to keep in page cache. In other words, popular assets are already in memory.

Buffering writes

The page cache also buffers writes. Writes are not synchronized to disk right away. They are buffered until the page cache decides to do so at a later time, either periodically or due to memory pressure. Therefore, it also reduces repetitive writes to the same file.

However, because of the nature of the caching workload, a web asset arriving at our cache system is usually static. Unlike workloads of databases, where a single value can get repeatedly updated, in our caching workload, there are very few repetitive updates to an asset before it is completely cached. Therefore, the page cache does not help in reducing writes to disk since all pages will eventually be written to disks.

Smarter than the page cache?

Although the Linux page cache works great out of the box, our caching system can still outsmart it since our system knows more about the context of the data being accessed, such as the content type, access frequency and the time-to-live value.

Instead of improving caching popular assets, which the page cache already does, in the next section, we will focus on something the page cache cannot do well: putting unpopular assets in memory.

Putting unpopular assets in memory

Motivation: Writes are bad

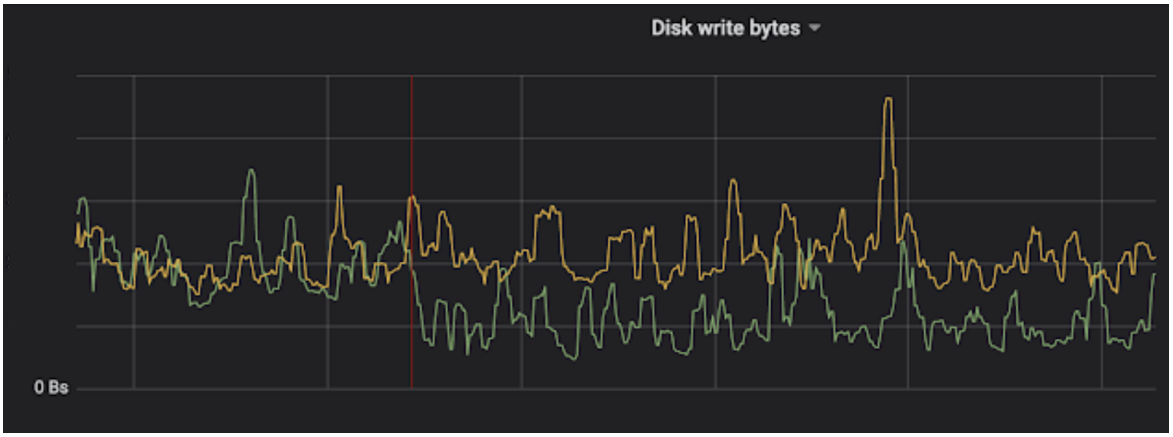
Worn out SSDs are solely caused by writes, not reads. As mentioned above, it is costly to replace SSDs. Since Cloudflare operates data centers in 200 cities across the world, the cost of shipment and replacement can be a significant proportion of the cost of the SSD itself.

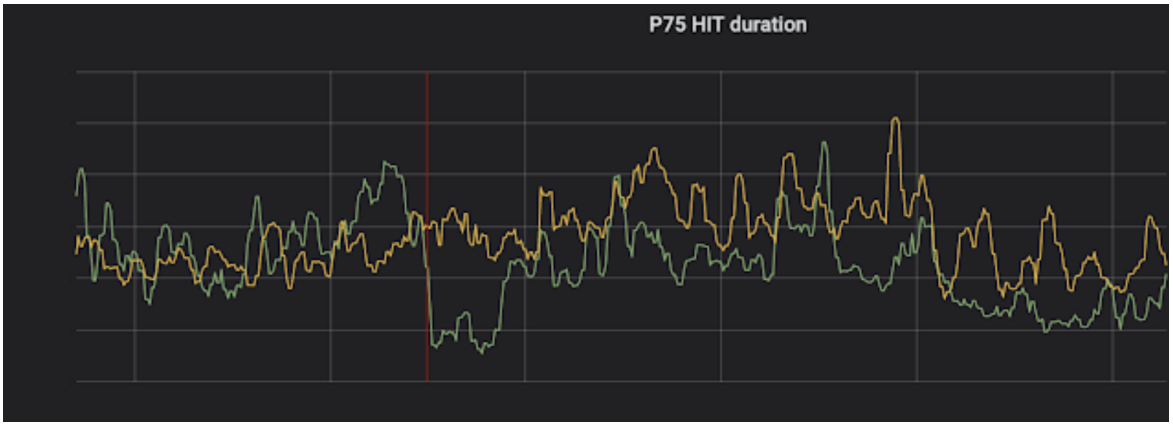
Moreover, write operations on SSDs slow down read operations. This is because SSD writes require program/erase (P/E) operations issued to the storage chips. These operations block reads to the same chips. In short, the more writes, the slower the reads. Since the page cache already cached popular assets, this effect has the highest impact on our cache hit tail latency. Previously we talked about how we dramatically reduced the impact of such latency. In the meantime, directly reducing the latency itself will also be very helpful.

Motivation: Many assets are "one-hit-wonders"

One-hit-wonders refer to cached assets that are never accessed; they get accessed once and then cached yet never read again. In addition, one-hit-wonders also include cached assets that are evicted before they are used once due to their lack of popularity. The performance of websites is not harmed even if we don't cache such assets in the first place.

To quantify how many of our assets are one-hit-wonders and to test the hypothesis that **reducing disk writes can speed up disk reads**, we conducted a simple experiment. The experiment: using a representative sample of traffic, we modified our caching logic to only cache an asset the second time our server encountered it.





The red line indicates when the experiment started. The green line represents the experimental group and the yellow line represents the control group.

The result: disk writes per second were reduced by roughly half and corresponding disk **hit tail latency was reduced by approximately five percent**.

This experiment demonstrated that if we don't cache one-hit-wonders, our disks last longer and our cache hits are faster.

One more benefit of not caching one-hit-wonders, which was not immediately obvious from the experiment, is that, it increases the effective capacity of cache because of the reduced competing pressure from the removal of the unpopular assets. This in turn increases the cache hit ratio and cache retention.

The next question: how can we replicate these results, at scale, in production, without impacting customer experience negatively?

Potential implementation approaches

Remember but don't cache

One smart way to eliminate one-hit-wonders from cache is to not cache an asset during its first few misses but to remember its appearances. When the cache system encounters the same asset repeatedly over a certain number of times, the system will start to cache the asset. This is basically what we did in the experiment above.

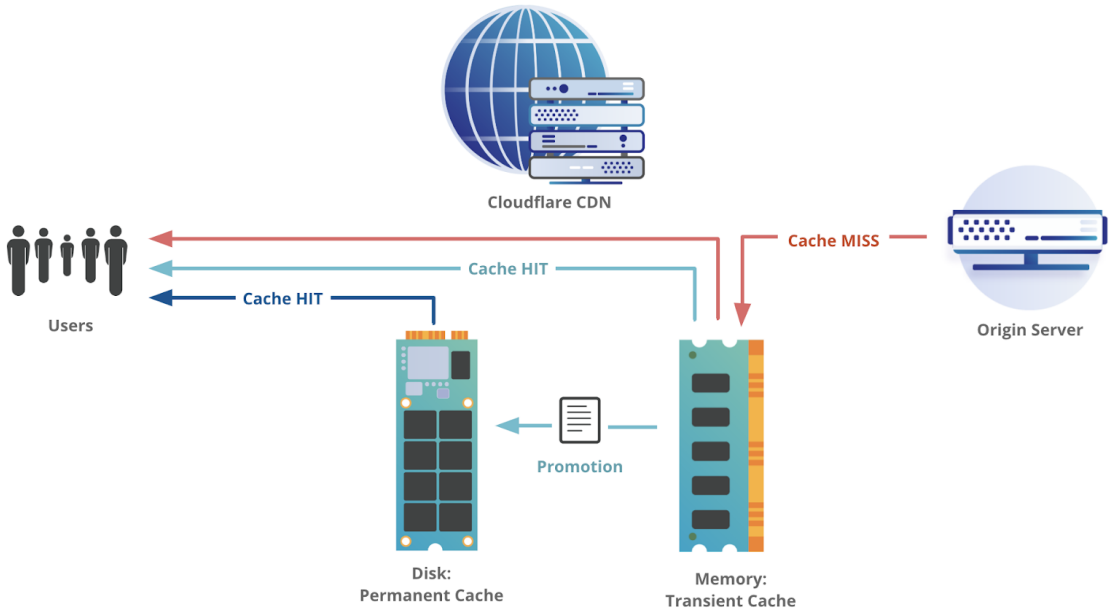
In order to remember the appearances, beside hash tables, many memory efficient data structures, such as Bloom filter, Counting Bloom filter and Count-min sketch, can be used with their specific trade-offs.

Such an approach solves one problem but introduces a new one: every asset is missed at least twice before it is cached. Because the number of cache misses is amplified compared to what we do today, this approach multiplies both the cost of bandwidth and server utilization for our customers. This is not an acceptable tradeoff in our eyes.

Transient cache in memory

A better idea we came up with: put every asset we want to cache to disk in memory *first*. The memory is called a "transient cache". Assets in transient cache are promoted to the permanent cache, backed by SSDs, after they are accessed a

certain number of times, indicating they are popular enough to be stored persistently. If they are not promoted, they are eventually evicted from transient cache because of lack of popularity.



This design makes sure that disk writes are only spent on assets that are popular, while ensuring every asset is only “missed” once.

The transient cache system on the real world Internet

We implemented this idea and deployed it to our production caching systems. Here are some learnings and data points we would like to share.

Trade-offs

Our transient cache technology does not pull a performance improvement rabbit out of a hat. It achieves improved performance for our customers by consuming other resources in our system. These resource consumption trade-offs must be carefully considered when deploying to systems at the scale at which Cloudflare operates.

Transient cache memory footprint

The amount of memory allocated for our transient cache matters. Cache size directly dictates how long a given asset will live in cache before it is evicted. If the transient cache is too small, new assets will evict old assets before the old assets receive hits that promote them to disk. From a customer's perspective, cache

retention being too short is unacceptable because it leads to more misses, commensurate higher cost, and worse eyeball performance.

Competition with page cache

As mentioned before, the page cache uses the system's available memory. The more memory we allocate to the transient cache for *unpopular* assets, the less memory we have for *popular* assets in the page cache. Finding the sweet spot for the trade-off between these two depends on traffic volume, usage patterns, and the specific hardware configuration our software is running on.

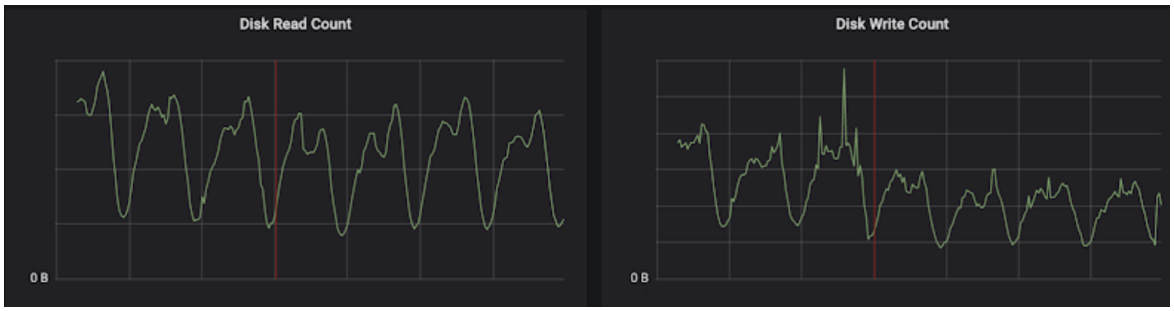
Competition with process memory usage

Another competitor to transient cache is regular memory used by our services and processes. Unlike the page cache, which is used opportunistically, process memory usage is a hard requirement for the operation of our software. An additional wrinkle is that some of our edge services increase their total RSS memory consumption when performing a "zero downtime upgrade", which runs both the old and new version of the service in parallel for a short period of time. From our experience, if there is not enough physical memory to do so, the system's overall performance will degrade to an unacceptable level due to increased IO pressure from reduced page cache space.

In production

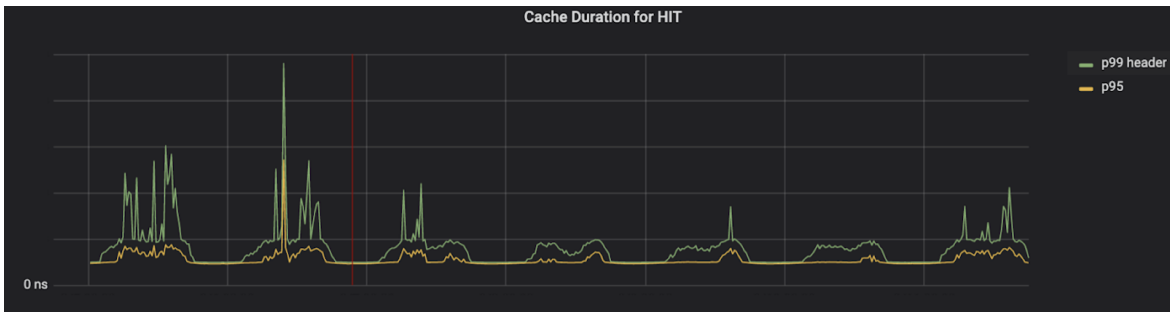
Given the considerations above, we enabled this technology conservatively to start. The new hybrid storage system is used only by our newer generations of servers that have more physical memory than older generations. Additionally, the system is only used on a subset of assets. By tuning the size of the transient cache and what percentage of requests use it at all, we are able to explore the sweet spots for the trade-offs between performance, cache retention, and memory resource consumption.

The chart below shows the IO usage of an enabled cohort before and after (red line) enabling transient cache.



We can see that disk write (in bytes per second) was reduced by 25% at peak and 20% off peak. Although it is too early to tell, the life-span of those SSDs should be extended proportionally.

More importantly for our customers: our CDN cache hit tail latency is measurably decreased!



Future work

We just made the first step towards a smart memory-SSD hybrid storage system. There is still a lot to be done.

Broader deployment

Currently, we only apply this technology to specific hardware generations and a portion of traffic. When we decommission our older generations of servers and replace them with newer generations with more physical memory, we will be able to apply the transient cache to a larger portion of traffic. Per our preliminary experiments, in certain data centers, applying the transient cache to all traffic is able to reduce disk writes up to 70% and decrease end user visible tail latency by up to 20% at peak hours.

Smarter promotion algorithms

Our implemented promotion strategy for moving assets from transient cache to durable storage is based on a simple heuristic: the number of hits. Other information can be used to make a better promotion decision. For example, the TTL (time to live) of an asset can be taken into account as well. One such strategy is to

refuse to promote an asset if the TTL of it has only a few seconds left, which will further reduce unnecessary disk writes.

Another aspect of the algorithms is demotion. We can actively or passively (during eviction) demote assets from persistent cache to transient cache based on the criteria mentioned above. The demotion itself does not directly reduce writes to persistent cache but it could help cache hit ratio and cache retention if done smartly.

Transient cache on other types of storage

We choose memory to host the transient cache because its wear out cost is low (none). This is also the case for HDDs. It is possible to build a hybrid storage system across memory, SSDs and HDDs to find the right balance between their costs and performance characteristics.

Conclusion

By putting unpopular assets in memory, we are able to trade-off between memory usage, tail hit latency and SSD lifetimes. With our current configuration, we are able to extend SSD life while reducing tail hit latency, at the expense of available system memory. Transient cache also opens possibilities for future heuristic and storage hierarchy improvements.

Whether the techniques described in this post benefit your system depends on the workload and the hardware resource constraints of your system. Hopefully us sharing this counterintuitive idea can inspire other novel ways of building faster and more efficient systems.

And finally, if doing systems work like this sounds interesting to you, come work at Cloudflare!

Discuss on Hacker News

Discuss on Reddit

Cache Product News Speed & Reliability