EVA Cache

Why a small size works so well

Ken Bates
Storage Performance Engineering
ken.bates@hp.com

Introduction

In order to increase performance, storage controllers often employ cache memory to reduce the effect of mechanical disk accesses. The amount of cache that is offered is a decision that is made during the initial design stages, and varies widely between different controllers. It also depends greatly on the operating system environment. That is, mainframe IO applications are typically cache-hit intensive and will benefit from larger caches in the storage subsystem. Open operating system applications, on the other hand, typically cache the IO requests in the server memory, resulting in cache-miss intensive IO requests to the storage subsystem. As such, open system applications do not benefit from large caches (where the accesses must wait to be satisfied by actual reading from the physical back-end disks).

An oft-asked question is how the EVA can perform so well with only 2 GB of cache memory. The answer to this question is that EVA is designed exclusively for open systems environments, having roots in the previous industry-leading generation of storage controllers. The HSG-80 controller, with only 1 GB of cache memory, routinely outperformed competitive storage controllers that had 8 or 16 GB of cache in environments with open systems workloads.

The reason behind this surprising fact is a combination of how typical applications access storage, coupled with the caching algorithms embedded in the HSG80. These algorithms have been substantially enhanced in the EVA HSV-1x0 controllers, providing even more performance with minimal amounts of cache memory. The result of these algorithms is the common statement that "***it's not size that counts; it's how you use it!***"

The remainder of this paper discusses in slightly more detail how applications access the underlying storage, and how the EVA can use this information to maximize cache efficiency while minimizing the size of cache.

Cache functions

The primary purpose of controller cache is to mask the relatively long service times associated with the mechanical nature of disk accesses. Today's disk drives have an average access time measured in milliseconds, while cache access times for high performance controllers are typically less than 200 microseconds. Since a disk access can take 30 to 40 times longer than a cache access, efficient cache algorithms can have a dramatic affect on overall storage performance.

Controller cache algorithms can be divided into four main areas of functionality based on I/O workload: random reads, sequential reads, random writes, and sequential writes. The amount of cache necessary for good performance depends on whether or not there are algorithms that are specific to each of these areas, and how well these algorithms are designed and implemented.

To understand why these four workloads differ so much and why the need for separate cache algorithms, some time will be spent delving into the individual I/O workloads and the specific EVA algorithms developed for those workloads.

Random reads

Read cache is used to cut down on disk accesses that repeatedly access the same data. When a disk block is accessed, the data is placed in cache on the assumption (or hope) that it will be accessed again. If the host accesses that data again, it can be returned directly from cache, thereby avoiding a disk access. Although this seems like a good approach, there are a few problems from a realistic standpoint:

1. Random access to a disk drive implies a very low probability of a cache hit. If the application randomly accesses a 1 TB file, for example, the controller would need 1 TB of cache. Most interactive database applications are random access, so read cache does very little to help, since nearly all accesses are cache misses. As a result, large amounts of read cache provide little or no benefit for a random access workload.
2. As data is read from disk, room must be made in cache for that new data, so older data will be removed. With the high cache miss probability associated with random accesses and high I/O rates of databases, this means extra overhead for the controller, as well as additional latency. Additionally, any data in cache that might have been accessed will probably be removed due to the other random read miss activity.
3. Modern databases have their own internal caches (the SGA in Oracle, for example). Since the database knows best what data should be cached, any data that is accessed repeatedly is kept within the host memory (the SGA), and the storage will never see the I/O. This is yet another reason that large amounts of cache provide very little benefit at the controller level.

In spite of this, there are times when the random access cache can be beneficial. The most common case is when there is a "hot spot" on the disk. This may be due to a specific file or portion thereof that that is repeatedly accessed, but is not cached by the application or operating system. A good example would be the index area of a database. Although this access pattern could benefit from a random access read cache, there is a problem in realistic environments, which we will now discuss.

In addition to the I/O that repeatedly accesses data within a small area, there are usually other ongoing I/O operations on other disks that are completely random. If these other accesses occur at a high rate and are read into cache, there is a possibility that they will displace the data from the hot spot area. The result will be cache misses for the hot spot data. One common way to overcome this, which is both costly and inefficient, is to simply increase the amount of cache memory in the hope that having a sufficiently large amount of cache will allow the hot spot data to be retained in cache. The problem with this is, of course, that it is mainly a matter of luck, depending on the relationship between the random access stream and the hot spot I/O. The end result can be tens of GB of cache memory required just to ensure cache hits on tens of MB of hot data.

A second approach is to disable read cache on the LUNs that have no cache hits, thereby preventing their data from polluting cache. Although this will certainly help the hot spot data to remain in cache, it suffers from the disadvantage that it requires manual monitoring and intervention. Not only that, if the workload to a LUN that has read caching disabled suddenly develops hot spots in the data, the potential for improved performance via caching is lost, since cache has been disabled on that LUN.

Rather than spending unnecessarily large amounts of cache memory in an inefficient manner or requiring continuous manual monitoring, the EVA has an advanced algorithm that can handle cases such as this with minimum cache memory. Within the EVA, there is code that monitors cache hits, and if the hit rate for a LUN drops below a certain level, will automatically disable read cache for that LUN. As a result, the random access stream will not be read into cache, and will not displace other data. The EVA will continue to monitor accesses for that LUN via data structures however, and if the access pattern is such that cache would help, the EVA will re-enable read cache for that LUN. This dynamic enabling and disabling of read cache based on changing I/O workloads is one of the many reasons the EVA doesn't need large amounts of read cache (it uses existing cache extremely effectively).

Sequential reads

Sequential reads are quite common with many applications, but are usually mixed with the I/O streams from other applications. Additionally, many applications issue sequential I/O requests with very small transfer sizes, which is extremely inefficient. A simplistic way of handling this is to simply fetch large amounts of data with every read request. The thinking (or lack thereof) behind this is that by reading in more data than is requested, the next sequential request for data will already be in cache. The fallacy with this thinking is twofold. First, only a very small percentage of requests are sequential in nature, so the additional time taken to read in the "extra" data significantly reduces performance on all accesses, sequential or not. Second, this unneeded data is stored in cache, once again causing other data to be purged from cache. This unnecessary cache purging is typically handled by once again adding massive amounts of (unnecessary) cache.

A much more advanced (and patented) algorithm is used by the EVA for prefetching data. The EVA continuously monitors the I/O stream, searching for sequential access patterns. A sequential I/O stream can be detected even if there is intervening, non-sequential I/O. As an example, if a request stream requested logical blocks 100, 2367, 17621, 48, 101, 17, 2, 15, and 102, the EVA would recognize that there is a single sequential request stream present (blocks 100, 101, and 102). Since the EVA was designed to handle large numbers of LUNs, the prefetch algorithm has been designed to recognize up to 512 simultaneous sequential streams for different areas of different LUNs. Because of this, the EVA can not only handle simultaneous I/O from disparate applications in a heterogeneous operating system environment, but can also initiate parallel prefetch operations on numerous LUNs as required.

Once a sequential stream has been detected and the originally requested data returned to the host, the EVA will request additional data from the LUN. By waiting until the original data has been returned before prefetching more data, no additional latencies are imposed on the host I/O. If the EVA prefetches the data before the host requests it, the algorithm concludes that the prefetch is working fast enough to keep up with the host, and the process continues (another prefetch after the host asks for the data). If, on the other hand, the host requests the data before the EVA has prefetched it, it is an indication that either the EVA is too slow, or the host is too fast. To handle this, the EVA will increase the size of the prefetch and continue. As the sequential read from the host proceeds, the EVA will dynamically increase the size of the prefetch as needed to stay ahead of the host requests.

After returning the data to the host, the EVA will purge the data from cache, since sequentially accessed data is rarely, if ever, requested again. Because this data is purged from cache, it takes up almost no room. As a result, a host can sequentially read an entire LUN, get 100% cache hits from the EVA, and use only a few hundred KB of cache. This result of this algorithm is that there is no penalty for randomly accessed data (no prefetch is triggered), prefetch will only occur when needed and only at the size that is needed (the size is dynamically adjusted), and minimal cache will be used (prefetch data is flushed after being returned to the host). Thus, EVA's pre-allocated 1GB of read cache is more than sufficient to sustain high performance simultaneous random and sequential read workloads.

Random writes

Write cache is used primarily as a speed-matching buffer. For random writes, completion is signaled to the host as soon as the data is in cache, resulting in very low latency. As the cache buffers start to fill, the EVA will initiate a background flush operation, writing that data to disk. With the large number of disks in the EVA, there is a very low probability of ongoing host reads colliding with a flush operation, so there is no noticeable performance loss as a result of the flush.

Interestingly enough, the size of write cache has very little impact on random workloads beyond a few hundred MB. As long as the incoming host write rate is lower than the rate at which cache flushes to disk, the cache will never fill, and the application will always see microsecond write response time. If, however, the host write rate is greater than the rate at which the data can be flushed to disk, then cache will eventually fill, no matter how large the cache is, and the host write

rate will be reduced to that of a disk. This is very similar to a bucket with a hole in it being filled with a hose. Increase the inflow rate past the outflow rate, and the bucket fills completely and overflows.

If the controller flush algorithms are poorly implemented, then it is quite possible that there will be times when cache becomes full and the application I/O must slow down to allow time for the cache to flush. This behavior is relatively easy to notice, since the application response time will have a tendency to fluctuate as the write speeds are based on cache or disk times.

The amount of write cache therefore only needs to be sufficiently large to accommodate transient bursts of write I/O activity without completely filling. Cache sizes larger than this offer no advantage, since they will never fill. By the same token, if the sustained write rate is greater than the write cache flush rate, no amount of cache memory will prevent cache completely filling and becoming ineffective.

Sequential writes

Sequential write streams are typically associated with log files, and are usually one of the more critical I/O streams from a performance standpoint. These writes usually involve small transfer sizes, such as 8 or 16 KB, and as such, are very inefficient from a storage standpoint. Furthermore, they are quite often associated with database "checkpoint" or "commit" operations; so other application I/O is gated on completion of these writes.

In addition to detecting sequential read streams, the EVA will also detect sequential write streams. Multiple sequential write requests will be aggregated in cache and, when time comes to flush the data, will be sent to the underlying disk storage as a single, large request instead of many small requests. A clear advantage of this approach is that a much higher data rate can be obtained at the disk level, increasing the efficiency of the transfer, as well as minimizing the impact on any other I/O requests that may have been directed to that particular disk drive.

When the size of the write data reaches what is known as a "strip", the EVA will flush this data to the disks. A "strip" is 512 KB in size, and is associated with the underlying virtualization technology. Although a discussion of virtualization technology is beyond the scope of this paper, writing a strip at a time means that for VRaid 5 LUNs, a single large write will encompass all members of a VRaid 5 redundancy group, and parity can be calculated and written on the fly, thereby eliminating the need for the typical read/modify/write operation. This action means that VRaid 5 sequential writes can achieve performance approaching that of VRaid 1. Flushes for VRaid 1 LUNs also occur at a size of 512 KB, so all members of a VRaid 1 redundancy group are involved in parallel transfers, making the flush extremely efficient.

Larger I/O sizes to disk would not increase performance, so the strip flush not only optimizes data rate performance, it also ensures that massive amounts of write data are not kept in cache, once again eliminating the needs for huge amounts of cache memory. As with reads, the EVA's pre-allocated 1GB of write cache (512 MB mirrored) is more than sufficient to sustain high IO performance simultaneously for both random and sequential write workloads.

Reasons for large caches

With all the above, a reasonable question to ask is if there is ever a good reason to have substantially larger caches than the EVA currently offers. There are several reasons, listed below:

1. Large caches make a customer feel good. This is strictly a matter of perception, but if the price of extended cache is justified by the "warm and fuzzy" feeling, it may be desirable to have large caches.
2. Cache memory has a very good profit margin, so large caches contribute substantially to the vendor's bottom line. Clearly not in the best interests of the customer, and a reason why the EVA only implements enough cache to meet the performance goals.

3. The controller has very poor caching algorithms, and masks this deficiency with large amounts of cache. Unfortunately, this is a very common occurrence, and results in unnecessarily high costs to the end customer. A good example of this is the “prefetch” algorithm that translates each and every read request, no matter how small, into a 64 KB read to disk. This not only wastes large amounts of cache memory, it also results in much slower disk performance since so much unneeded data has to be read.
4. Data structures and code are stored in cache memory. Although this does result in fast access from a controller firmware standpoint, the cost of cache memory is much higher than that of program memory, and gives a false sense of what can be expected from cache. The EVA contains sufficient “private” memory for the purposes of storing the code and data structures, so additional cache memory is not needed for this purpose.
5. Creating a LUN that is entirely resident in cache (often called cache or LUN binding intelligence). This is perhaps the only good reason for large caches. There are instances where the speed of an electronic disk is desired, and having the ability to carve up a segment of cache and present it to the host as a LUN is worth the high cost.

With the exception of item 5, a cache that is sized appropriately to the underlying storage and implements advanced caching algorithms does not need to be excessively large to perform well. In fact, larger caches impose additional restrictions in the form of increased battery size (for write data retention) and cache search time (which may lower performance).