Friday, February 20. 2009

Some insight into the read cache of ZFS - or: The ARC

The Adjustable Replacement Cache (ARC) of Solaris ZFS is really an interesting piece of software Interestingly it´s based on a development of IBM, the Adaptive Replacement Cache as described by Megiddo and Modha, but the developers of ZFS extended it for their own usage. The original implementation was presented on the FAST 2003 and described in this article in ;login:.

It´s a really elegant design. In the following description i´ve simplified some mechanisms to make things easier to understand. The authoritative source of information about the ARC in ZFS is http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/fs/zfs/arc.c. But in the next paragraphs i will try to give you some insight into the inner workings of the read cache of ZFS. I will concentrate on the part, how data gets into the cache and how the cache adjust itselfs to the load and thus earns its moniker "Adjustable Replacement Cache".
CachesWell ... the standard LRU mechanisms of other file system caches have some shortfalls. For example, they are not scan-resistant. When you read a large amount of blocks sequential, they tend to fill up the cache, even when they are read just once. When the cache is full and you want to place new data in the cache, the least recently used page is evicted from the cache (thrown out of it). In the case of such large sequential reads, the cache would contain only those reads and not really frequently used data. When the large sequential reads are just used once, the cache is filled with worthless data from the perspective of a cache.

There is another challenge: A cache can optimize for recency (bycaching the most recently used pages) or for frequency (by caching the most frequently used pages). Neither way is optimal for every workload. Thus a good cache design is able to optimize itself

The inner workings of ARCThe ARC solves this challenges either in it´s original implementation as in the extended implementation in ZFS. I will describe the fundamental concept of the Adaptice Replacement Cache as described by Megiddo and Modha, as the additional mechanisms of the ZFS implementation hide a little bit the simplicity but effectiveness of this mechanism. Both implementations (the original Adaptive Replacement Cache and the ZFS Adjustable Replacement Cache) share their basic theory of operation thus i think this simplification is a viable one to explain ZFS ARC.

Let´s assume that you have a certain amount of pages in your cache. For simplicity, we will assume a 8 pages sized cache. For it´s operating the ARC needs a directory table as large as two times the size of of the cache.

This table seperates in 4 lists. The first two ones are obvious ones.
a list for the most recently used pagesa list for the most frequently used pagesThe other two are a little bit stanger in their role. They are called ghost lists. They are filled with recently evicted pages from both lists: a list for the recently evicted pages from the list of most recently used pagesa list of the recently eviced pages from the list of the most frequently used pages

The both ghost lists doesn´t cache data, but a hit on them has an important effect to the behaviour of the cache as i will explain later on. So what happens in the cache. Let´s assume we read a page from the disk. We place it in the cache. The page is referenced in the recently used list.

We read another different one. Its placed in the cache as well. And obviously it´s put into the recently used list at the most recently used position (position 1).

Okay, now we read the first page again. Now the reference is moved over to the frequently used list. It has to be used at least two times to get into this list. Whenever a page is accessed again on the frequently used page list, it is put to the beginning of the list again. By doing so really frequently accessed pages would stay in cache, but not to frequently accessed pages wanders to the end of the list and will get evicted at the end ...

Over the time both list fills and the cache is filled accordingly. But then one of the cache areas is full but you read an uncached page. One page has to be evicted from the cache to place a new one into it . Pages can just evicted from

cache, when the page in cache isn´t referenced by any of the non-ghost lists.

Let´s assume that we filled up the list for the recently used pages

So it evicts the least recently used page from the cache. This page is put onto the list of recently evicted pages.

Now the page in the cache isn´t referenced any longer and thus you can evict it from the cache. The new page to cache gets now referenced by the cache directory.

With every further eviction this page wanders to the end of the list. At a later point in time the reference to this evicted page is at the end of the list, and after the next eviction from the last recently used list it´s removed from this list, and there is no reference in the lists anymore.

Okay, but what happens when we read a page again, that´s on the list of already evicted pages. Such an attempt to read leads to a phantom cache hit. As the data of the page has already been evicted from cache, the system has to read it from the media, but by this phantom cache hit, the system knows, that this was a recently evicted page and not a page read just the first time or read a long time ago. The ARC can use this information to adjust itself to the load.

Obviously this is a sign that our cache is to small. In this case the length of lists of the recently used pages in cache is increased by one. Obviously this reduces the place for frequently used pages by one.

But there is the same mechanism on the other side. If you get a hit on the list of recently evicted pages of the frequently used pages, it decreased the the list for currently cached recently used pages by one. This obviously increased the available space for frequently used pages by one.

With this behaviour the ARC adapts itself to the load. If the load is more like accessing recently accessed files, it would have more hits in the ghost list of the recently used files and thus increase the the part of the cache for such files. And vice versa, if the load on the disks is more like accessing frequently accessed files, it would have more hits on the ghost lists for frequent accessed pages and thus increase the cache in favour of frequently used pages.

Furthermore this enables a neat feature: Let´s assume you read through a large amount of files for log file processing. You need every page only once.A normal least recently used based cache would load all the pages in the cache, thus evicting all frequently accessed pages. But as you access them only once they just fill up the cache without bringing you any advantages.

A cache based on ARC behaved differently. Obviously such a load would start to fill up the cache assigned for recently used pages really fast and such the pages are evicted soon. But as every of this page are just accessed once, it´s highly unprobable that they will be hits in the ghost list for recently accessed files. Thus the part of the cache for recently accessed files doesn´t increase by such pages read only once. Let´s assume you you correlate the data in the logfiles with a large database (for simplification we assume, it doesn´t have an own caching mechanism). It accessed pages in the database files quite frequently. The probability of accessing pages referenced in the ghost list for frequently used files is much larger on the ghost list for recently accessed files. Thus the cache for the frequently accessed database pages would increase. In essence, the cache would optimize itself for the database pages instead of polluting the cache with log files pages.

The modified Solaris ZFS ARCAs i wrote before, the implementation in ZFS isn´t the pure ARC as described by both IBM researchers. It was extended in several ways:

the ZFS ARC is variable in size and can react to the available memory. It can grow in size when memory is available or it can decrease in size when memory is needed for other things
the ZFS ARC can work with multiple block sized. The original implementation assumes an equal size for every block
You can lock pages in the cache to excempt them from eviction. This prevents the cache to evict pages, that are currently in use. The original implementation doesn´t have this feature, thus the algorithm to choose pages for eviction is lightly more complex in the ZFS ARC. It chooses the oldest evictable page for eviction.

There are some other changes, but i will leave them for the lecture of the well commented source code of arc.c

The L2ARCThe L2ARC keeps the model stated in the paragraphs above untouched. The ARC doesn´t move the pages automatically to L2ARC instead of evictim them. Albeit this would seem the logical way, this would have severe impacts. At first a sequential read burst would overwrite large amounts of the L2ARC cache as such a burst would evict many pages in a short schedule. This is an unwanted behaviour.

The other problem: Let´s assume, your application needs a large heap of memory. The modified Solaris ARC is able to resize itself to the available memory. Thus when the applications request more memory, the ARC has to gets smaller. You have to evict a large amount of memory pages at once. If every page is written to L2ARC before eviction from ARC, this would add subtantial latency until your system can provide more memory, as you would have to wait for the L2ARC media before eviction.

The L2ARC mechanism tackles the problem a little bit differently: There is a thread called l2arc_feed_thread that walks over the soon to be evicted ends of both lists - the recently used list and the frequently used list - and puts them into a buffer (8 MB). From this place another thread (the write_hand) writes them into the L2ARC in a single write operation.

This algorithm has many advantages: The latency of freeing memory isn´t increased by the eviction. In situation of mass eviction by a sequential read burst, the blocks are evicted before the l2arc_feed_thread traverses the end of the the lists. So the effect of polluting the L2ARC by such read bursts is reduced (albeit not completly ruled out).

ConclusionThe design of Adjustable Replacement Cache is much more efficient than ordinary LRU cache design. Megiddo and Modha were able to show much better hit rates with their original Adaptive Replacement Cache (a list is at the end of the ;login:-article). The ZFS ARC shares the basic theory of operation thus the hit rate advantages should be similar to the one of the original design. More important: The idea of using large caches in form of SSD get even more viable, if the cache algorithm helps them to yield an even better hit rate.

Want to learn more?The theory of ARC operation in One Up on LRU, written by Megiddo and Modha, IBM Almaden Research Center
The implementation in ZFS is descibed in the source code. The file http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/fs/zfs/arc.c contains many comments. So use the source, Luke.

Posted by Joerg Moellenkamp in English, Solaris at 16:43

Wow very interesting post!
Do you know if it's possible to have details on how the cache is currently used ? (ie: amount of cache used for MRU, MFU and ghost pages?)

There are probably DTrace probes for this?

I haven't seen that information available on our 7410 unit.
Anonymous on Feb 20 2009, 23:00

I have found this perl script that uses using kstat and shows current ARC usage summary:
http://tinyurl.com/5dwrxg

Also there seenms to be a DTrace probe called monitors sdt:::arc on which you can monitor hits, misses, deletes and evicts
Anonymous on Feb 20 2009, 23:06

Pretty sure it's "Adaptive Replacement Cache".
Anonymous on Feb 23 2009, 06:13

Thought that too a while ago, but the arc.c source code states "DVA-based Adjustable Replacement Cache" in line 27. And who i am to question the code The name would make sense, as contrary to the original size, the ZFS ARC is adjustable in size.
Anonymous on Feb 23 2009, 07:25