Multi-Chance Adaptive Replacement Cache (MC-ARC)

Shailendra Tripathi
Architect,
Tegile Systems
Outline

- Adaptive Replacement Cache (ARC)
- Scaling Issue of ARC
- Design Goals
- MC-ARC
- Test Results
- References
- Q & A
Adaptive Replacement Cache (ARC)

- Two Separate Lists – MRU, MFU
- Two Ghosts Lists - Ghost MRU, Ghost MFU
- Track recently evicted pages, has no data
- Top Level Separation for Metadata and Data
- Referenced Pages are tracked in ANON list
- MRU + MFU = GMRU + GMFU = ARC size
- List size dynamically adaptive based on hits on the respective ghost lists.
Images sourced from the blog at: http://www.c0t0d0s0.org/archives/5329-Some-insight-into-the-read-cache-of-ZFS-or-The-ARC.html
ARC Mechanism

The diagram illustrates the ARC (Associative Release and Compaction) mechanism, which is a storage management technique used in computer systems. It consists of three main components:

1. **Cache**: This is the storage area where frequently accessed data is kept. It is divided into three parts:
   - **Ghost List**: Contains entries that have been removed from the cache but are still in use.
   - **Most Recently Used (MRU)**: Contains the least recently used pages.
   - **Most Frequently Used (MFU)**: Contains the most frequently used pages.

2. **Cache Directory**: This directory keeps track of the pages in the cache. It contains a mapping of page numbers to their cache status.

3. **Entry**: Each entry in the cache and cache directory includes a pointer to the page in memory and a tag that allows the cache to determine if the page is still in use.

In the ARC mechanism, when a new page is added to the cache, it is placed in the Ghost List. As pages are accessed and moved to the MRU list, they are also moved to the MFU list. If the cache becomes full, the least recently used page is removed from the MRU list and the least frequently used page is removed from the MFU list. This process helps maintain the balance between recent and frequent access.
ARC Mechanism Continued

[Diagram showing the ARC mechanism continued with explanations and connections between the Ghost List, most recently used, most frequently used, and Cache Directory.]
## ARC Performance – Scale View

Lockstat Output on 4K 100% random read – from 8 Windows client running IOMETER/FIO

<table>
<thead>
<tr>
<th>Count</th>
<th>indv</th>
<th>cuml</th>
<th>rcnt</th>
<th>nsec</th>
<th>Lock</th>
<th>Caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120476</td>
<td>4%</td>
<td>27%</td>
<td>0.00</td>
<td>181191</td>
<td>ARC_mru+0x58</td>
<td>remove_reference+0x63</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61496</td>
<td>2%</td>
<td>31%</td>
<td>0.00</td>
<td>3662</td>
<td>ARC_mru+0x58</td>
<td>arc_evict+0x54d</td>
</tr>
<tr>
<td>50349</td>
<td>2%</td>
<td>36%</td>
<td>0.00</td>
<td>214993</td>
<td>ARC_mru+0x58</td>
<td>add_reference+0x7e</td>
</tr>
</tbody>
</table>

Lockstat

Tool to capture locking contention on Solaris/OS platform.

Test Platform

Test was run on Tegile’s Entry Level All flash product with 96GB RAM & Intel(r) Xeon(r) CPU E5-2450 v2 @ 2.50GHz

Test Steps

Create 100 GB LUN from each client (total 800GB data)
100% random prep read for 30 minutes
Measure Results - 10 minutes 100% random read run
Scaling Issue of ARC

- LRU Lists - global Lock per list
- Each Access / Drop – Linked List movement
- Storage Service times < 75 us (SSD) and going down (NVMe/3D).
- Increasing CPU Cores & Threads
- Allocation increasingly higher priority than Caching value
- Impacts both Read and Write IOs
Million IOPS per box goal

- Read service time - end-to-end <= 62.5 us
- SSD service times similar order
- Pipeline processing & SSD servicing
- Processing <= SSD service times all the time
- Constant overhead – interrupt (at least 2, one context switch, and, copy out) ~10us

Locking & Alloc/Eviction – Single digit overhead
Ideal Page Cache / Design Goals

1. Minimizes big lock usage
2. Minimize movements (within or across)
3. Short term - Long term value differentiation
4. Scan resistant

ARC – good in 3 and 4

Items 1 and 2 are scaling bottleneck
Scalable MC-ARC – Lockless Page Cache

- Linked lists are replaced with page arrays
- Configurable Page size
- Page array points to cache buffers
- Pages are linked in FIFO order
- Pages are dynamically linked for flexibility
MC-ARC - Insertion

- Reserve Swap – Atomic Add
- Swap Pointer – Atomic Swap
MC-ARC – Remove

- Swap NULL – Atomic Swap
- Dec Ref Count – Atomic Dec

![Diagram showing the process of MC-ARC in a remove operation.](Diagram of MC-ARC process)
MC-ARC - Access

- Buf hash table lookup
- Remove from the old Page
- Re-insert in the page at the head
- Protected by buffer lock
Eviction

- Take the page out from tail
- Check if array pointer is valid
- Try buf hash lock – Locked and eligible, free buf
- Try lock fails, add page to a private list
- At the end of the eviction span, re-insert the private list at tail
Test Results

- Scaled IOPS from 275K to 1 Million IOPS
- Lockstat Contention – No ARC function in top 50
- Higher block sizes – 4x10GB ports are saturated at 100% Random Reads
- OS memory management – next level bottleneck
- CPU burns out at small block sizes (4k/8k)
References

- ARC: A Self-Tuning, Low Overhead Replacement Cache“ - by N. Megiddo & D. Modha
- https://pthree.org/2012/12/07/zfs-administration-part-iv-the-adjustable-replacement-cache
- http://www.c0t0d0s0.org/archives/5329-Some-insight-into-the-read-cache-of-ZFS-or-The-ARC.html