InfiniBox Flash Cache – winning performance with hybrid storage

Design considerations and lessons learned

Sivan Tal
Agenda

- InfiniBox data layers
- Basics of caching algorithms
- The inter-relations between media choice, data layout and caching algorithm
- Getting the most out of low cost HW
- Using real world production I/O traces to learn and train cache
Clarification / disclaimer

- This presentation is about:
  - Flash cache design considerations
  - Lessons learned from years of R&D in this field

- This presentation is **not** about:
  - The inner details of the InfiniBox “Neural Cache”™
  - In particular, not about the ML/DL aspects
Clarification (cont.)

- InfiniBox “Neural Cache”™ ML/AI comes into play in:
  - Making decisions about inserting/evicting logical data into the cache
  - Adapting the algorithm to the (changing, mixed) workload

- This presentation covers flash media as cache data storage
  - Cost-effective use of flash store
  - Mutual impact between caching algorithm and media layout / write patterns
  - We discuss and depict this with simple, well known algorithms
InfiniBox Data Layers

- **DRAM** – up to 3TB – hot data
  - L1 Read cache – single copy
  - Write cache – dual copy, on two nodes
- **SSD** – up to 400TB – warm-cool data
  - L2 Read cache – single copy
- **HDD** – up to 5.8PB raw – all the data
  - InfiniRaid – holds all the data, fully protected
Managing Cache – LRU

- LRU is the most common approach to caching
  - Many derivatives, notably ARC
- The basic algorithm is straight-forward:
  - On cache miss: Insert block at head, evict block at tail
  - On cache hit: Promote accessed block to head
- LRU lists in DRAM are simple and efficient
Caching on SSD

- When using SSD as cache device, LRU essentially means random writes to the device
  - Least recently accessed block can be anywhere
- Random write means high write amplification
  - Performance degradation
  - Reduced durability – up to 5x compared to sequential write → higher costs
Caching on SSD – using FIFO

- Ideal SSD write pattern: fully sequential, in a cyclic manner
- Using this pattern creates a FIFO
  - The least recently inserted block gets evicted
  - Cache hits do not promote blocks to the top
- How much do we lose vs. LRU?
  - Well, that depends…
Caching on SSD – using FIFO

Consider the FIFO cache as a track – blocks are inserted at one side, move along until they are dropped on the other side.
Using FIFO – example 1

Assume insertion rate that retains blocks for 30 min
And a certain block is accessed every 20 min

For this case: LRU – 100% hit ratio. FIFO – 50%
Using FIFO – example 2

Now assume insertion rate that retains blocks for 120 min
And that block is accessed every 20 min

For this case: LRU – 100% hit ratio. FIFO – 83%
Using FIFO – what can go wrong?

- We’ve seen that the larger the cache, the closer FIFO gets to LRU performance
- What can go wrong with this simple approach?
  - Frequent writes to the same block create holes – unused space
  - Write-only datasets waste cache space
- Next we’ll discuss some techniques to improve on that
Using FIFO – what can go wrong?

- Frequent updates to a block create invalid blocks
  - Instead of updating existing block in cache, we invalidate it and evict another block for the new version

- Possible solution: update in place
  - Breaks the sequentiality – it’s a trade-off

- Perhaps this block shouldn’t have been inserted at all? More on that later…
Using FIFO – what can go wrong?

- A hot blocks reaches the bottom of the queue…
- We can keep it in two ways:
  - Skip over it – breaking the sequentiality
  - Reinserting it at the head of the queue
Using FIFO – what can go wrong?

- Key point with large FIFO – Deciding what to exclude
- Data resides in the cache for a long time
  - Makes cost of insertion high
- Goal → minimize writing useless data to the SSD cache
Using FIFO – excluding data from cache

- Most benefit is obtained by excluding data from cache altogether. How to decide what to exclude?
- The system can learn from the I/O patterns
  - Each I/O request has: Volume, LBA, type of I/O (read vs. write), request data size
  - One can learn over time, when data is worthy of being inserted
A problem with exclusion

- When you exclude something, how do you know if it was a good decision?
  - It’s easier to know if it’s included

- Techniques for coping:
  - Ghost cache
  - Sampling-in part of excluded data
Beyond FIFO

- When using FIFO, all inserted blocks get the same amount of time in cache (more or less)
- But some blocks we want to retain for one hour, some for 4 hours, some for 24 hours…
- Possible solution: maintain multiple FIFO’s for different durations
Multiple FIFO’s

- Divide the SSD space to segments
- Create multiple FIFO’s with different duration targets
  - For example: 1, 2, 4, 8, 16, 32 hours
- Each FIFO is a linked list of segments containing cached blocks
- Assign segments to segment lists dynamically
  - Select the least useful segment for eviction
- The key decision is to which FIFO to send each block
Multiple FIFO’s – example

- Need to allocate a segment to the top list:

### Example Table

<table>
<thead>
<tr>
<th>ratio</th>
<th>Desired</th>
<th>Age of oldest</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>2</td>
<td>2.2</td>
</tr>
<tr>
<td>0.9</td>
<td>4</td>
<td>3.6</td>
</tr>
<tr>
<td>1.2</td>
<td>8</td>
<td>9.7</td>
</tr>
<tr>
<td>0.96</td>
<td>16</td>
<td>15.4</td>
</tr>
</tbody>
</table>

- Possible eviction selection: The segment which is oldest compared to desired duration
Multiple FIFO’s – example

- Segment evicted from the 3rd list and allocated to the 1st

<table>
<thead>
<tr>
<th>ratio</th>
<th>Desired</th>
<th>Age of oldest</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>2</td>
<td>2.2</td>
</tr>
<tr>
<td>0.9</td>
<td>4</td>
<td>3.6</td>
</tr>
<tr>
<td>0.85</td>
<td>8</td>
<td>6.8</td>
</tr>
<tr>
<td>0.96</td>
<td>16</td>
<td>15.4</td>
</tr>
</tbody>
</table>
Multiple FIFO’s

- Combined with “good decisions” as to which FIFO to assign for each block – this can give an advantage over single FIFO
  - At the cost of some SSD write amplification
- The choice of segment size affects WAF
  - Open-Channel (ZNS) SSD provides ideal allocation without write amplification penalty
Using I/O traces from the field

- Test different algorithms and tuning params based on real life data
- Upload selected traces via tunnel to the Infinidat cloud
  - Too much data to transfer…
    - Upload bandwidth only a few MB/s
  - Solution – sampling
    - Sample only selected LBA ranges, selected randomly
    - Sampling ratio of 1:20 is certainly good enough
    - Run simulation on scaled down cache
- We never pull user data – only i/o stats
Example: FIFO vs. Multi-FIFO vs. LRU

- Simple algorithms (without ML) simulated on real-world i/o traces

- Typical results: LRU > Multi-FIFO >> FIFO

- When SSD capacity increased – gap narrows down
Periodicity

- Most caching algorithms are optimized for exponential distribution
Periodicity

- Reality: Periodic workloads break the paradigm

Example from one of our customers’ system:
Distribution of time between accesses

Each line represent a dataset. Each dataset behaves differently, however 24h periodic trend is clear

Number of reads after x hours since last read/write to the same block

Each line is a separate dataset
Periodicity

- Periodicity can be leveraged to adjust priority of cached data
  - Priority gets higher as it approaches the period time
- Prefetching can also be applied to leverage periodicity:
  - Prefetch data shortly before period and hold it in short term cache
- The bottom line is that “linear” algorithms may suffer from periodicity, but a learning cache can leverage it
Conclusions

- Using flash as cache store involves several design considerations particular to that type of media
- Inter-dependencies and mutual impact of caching algorithms and flash write patterns
- Infinidat Neural Cache™ is about learning the data access patterns, selecting the correct algorithm for the workload and adapting to changing and mixed workloads
- Only by working with real-world workloads can you realistically develop and test your algorithmic approach
Questions?

Contact: sivant@infinidat.com