Proactive IO Scheduling: Discriminating cache and media in the Storage Device

Jongmin Gim Taeho Whang Youjip Won
ECE Division, Hanyang Univ.
Seoul, Korea

Abstract

Modern state of art storage devices are equipped with controller cache to improve IO latency. Especially, high-end storages, e.g. RAID controller [2] and high-end PCIe connected Solid State Drive (SSD) [3], have DRAM cache whose size ranges from 512MB to 16GB. IO latency becomes order of magnitude faster when the requested data is transferred from the cache than when the requested data is transferred from storage media, e.g. disk platter or NAND flash. We carefully envision that as storage device lies closer to CPU, and becomes faster due to the adoption of new media, e.g. NAND Flash, Phase Change RAM [5], STT-MRAM [4] and etc., when IO request is served from the device cache, it may be sufficiently fast and that OS may not have to switch context to service the respective IO. What lies in the core of the problem is to effectively estimate the device caching behavior and to predict cache hit probability.

Currently, we are developing a technique which enables the host to predict whether a given request will be serviced from the device cache or not. Since this technique is embedded in the host OS, it is difficult to adopt elaborate schemes and at the same time it requires significant amount of effort to make the technique computationally cheap. We are developing cache hit prediction technique, *Anchor based Cache Hit Prediction (ACHiP)*. It extracts the IO cluster size, i.e. the size of IO which is loaded from the storage media to storage cache. *ACHiP* maintains *Anchor* and *Window Size* to denote the storage region which is loaded on the device cache. We categorize the state of caching pattern into three cases: *hit*, *partial* and *miss*. If the incoming IO request lies within the windows size from the anchor, *ACHiP* algorithm predicts it as *hit*. The *partial* means that the beginning of the block number of IO is in window size but the length of IO is greater than the window size. Otherwise, it recognizes as a *miss*. *ACHiP* examines IO response time, IO length, and cache access pattern to determine cache hit or miss occurs. *ACHiP* consists of three technical ingredients: (i) device monitor that estimates cache size and cache pattern, (ii) history manager that records log of IO requests (*Anchor*, *Window Size*), and (iii) decision module that determines whether cache hit or miss will occur for a given IO request.

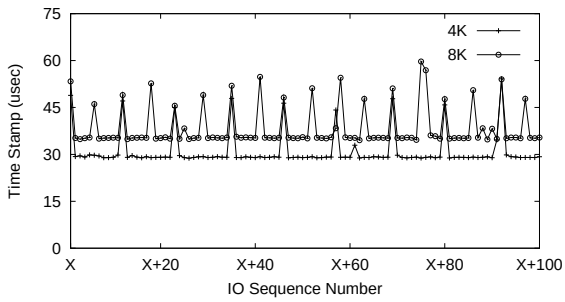

Figure 1: Device Monitor: extracting window size for fusion IO [1]

Device monitor probes the device and estimates the device cache size and caching pattern. It extracts IO latencies from 4KB to 512KB IO size for sequential workloads. It works only one time when a block device is registered. The difference of IO latencies for a given IO size informs whether IO is from device cache or not. We can observe a pattern in IO latency result. Device monitor determines that maximum values are caused by a cache miss and on the other hand cache hit gives minimum values. To extract *window size*, it measures the distance from position of i^{th} cache miss to $i + 1^{th}$. The decision module exploits information of window size to judge whether cache is hit or not.

Fig. 1 illustrates how device monitor extracts window size and threshold value for 4KB and 8KB IO with Fusion IO SSD drive where x and y axis represent i^{th} block from X position for a given IO size and response time,

respectively. It opens low device and measures latencies for 4KB and 8KB IO for sequential read. Experiment platform is Core i7 930 CPU, DDR3 8G memory and Linux 2.6.32. Device monitor measures the elapsed time between when the IO requested is sent to the device (unplugging the device queue) and when the respective storage device sends the completion signal to the host. Time Stamp Counter (TSC) is used for estimating IO latency. Since it consists of only one inline assembly, the overhead of measuring the time interval is negligible (measured time interval is 85ns on average). Note that host side IO acceleration techniques such as file system read-ahead is disabled. The result shows that IO latency exhibits periodic behavior as IO proceeds. There are two different values, 28us and 46us, for IO latency of 4KB IO. Device monitor determines the threshold value for basis of cache hit. For the 4KB IO, threshold value is 37us (28+46/2). This value is conjugated as criteria of determining cache hit/miss in decision module. Shorter IO latency is when the request is serviced from the device cache and longer IO latency occurred when the request is serviced from the storage media (NAND Flash). *Window Size* corresponds to the period of two consecutive peaks in the IO latency time series. In this case, window size is 40KB $\pm$ 4KB for 4KB. Likewise, 8KB IO size shows 40KB $\pm$ 8KB interval between two maximum values. In Fig. 1, 4KB IO has two times longer interval than 8KB IO, it is due 4KB's block size is half of 8KB's. Additionally, result from the first access (position X in x axis) shows that cache miss occurs. This is a basis of $anchor$ that blocks not registered by the history manager must make cache miss. The response times for all IO size are used determine cache hit/miss.

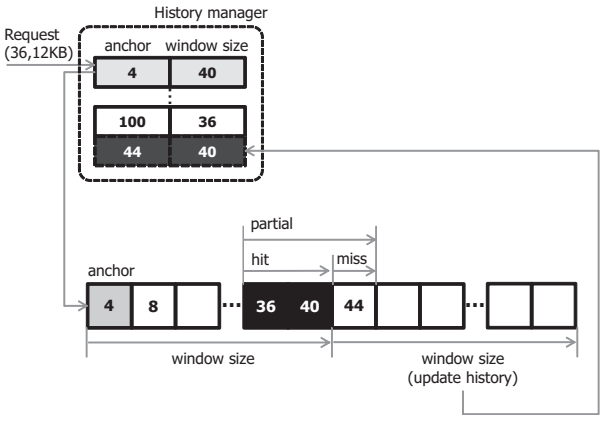

Figure 2: *ACHiP*: determining cache hit

History manager maintains the list of (*Anchor*, *Window size*) information. Decision module determines cache hit, partial, and miss for incoming IO. Currently, we are using very primitive approach in cache hit prediction. If incoming IO lies in the address range covered by one of the (*Anchor*, *Window size*) pairs, decision module determines it as cache hit. On the other hand, if IO is not covered by any of the (*Anchor*, *Window Size*) pair, decision module determines it as cache miss, and updates the history. If incoming IO spans in the boundary of *Anchor+window size*, decision module estimates the response time for the IO, and then make decision on cache hit or miss based upon the threshold value. When a given IO lies in the address range over the *Window Size* even though its start address is in *Window Size*, decision module determines it as partial. In case of partial, decision module determines portion of blocks in the history as cache hit, and excess part of blocks is determined as cache miss. Then, it updates history of (*Anchor*, *Window size*) pair for the blocks missed cache. According to cache hit or miss, history manager registers the pair to the list. Fig. 2 illustrates the *ACHiP* mechanism in case of partial.

Final issue in designing cache hit prediction module is how to implement cache replacement policy. Knowing the cache replacement policy of the storage device is important in predicting the cache miss. Cache replacement occurs when the cache is full. Currently, we assume that when the total window size which is maintained at the history database is larger than the size of storage cache, history manager deletes history in LRU manner. Further elaboration is required to properly estimate the cache replacement algorithm of the storage device.

References

[1] FUSIONIO. iodrive product family user guide - linux for driver release 2.1.0.

[2] INTEL. Intel RAID Controller RS2BL080 Hardware User's Guide.

[3] JOSEPHSON, W., BONGO, L., LI, K., AND FLYNN, D. DFS: A file system for virtualized flash storage. *ACM Transactions on Storage (TOS) 6*, 3 (2010), 1–25.

[4] LI, J., AUGUSTINE, C., SALAHUDDIN, S., AND ROY, K. Modeling of failure probability and statistical design of spin-torque transfer magnetic random access memory (stt-mram) array for yield enhancement. In *Proc. of Design Automation Conf. (DAC '08)* (Anaheim, CA, June 2008), pp. 278–283.

[5] QURESHI, M. K., SRINIVASAN, V., AND REVERS, J. A. Scalable High Performance Main Memory System Using Phase-Change Memory Technolohy. In *Proc. of the 36th ISCA* (Austin, Texas, USA, June 2009), pp. 24–33.