

An Efficient SAN-Level Caching Method Based on Chunk-Aging

Jiwu Shu, Yang Wang, Wei Xue, Yifeng Luo

Department of Computer Science and Technology, Tsinghua University, Beijing 100084
{shujw,xuewei}@tsinghua.edu.cn { iodine01, luo-yf04}@mails.tsinghua.edu.cn

Abstract:

SAN-level caching can manage caching within a global view so that global hot data can be identified and cached. However, two problems may be encountered in the existing SAN-level caching methods: first, too many migrations from disks to cache resulting from cache thrashing; second, too few cache hits resulting from insufficient caching. This paper proposes a SAN-level caching method employing chunk-aging to control the migrations of chunks of data from disks to cache, which takes the temporal distribution of chunk accesses into account and solves the two problems above. Then by employing two LRU lists to manage the cache replacement, this method separates and manages two types of hot data chunks, one of which have burst accesses and the other have long-term frequent accesses, thus preventing them from interfering with each other in cache. The simulation results demonstrate that our method can achieve a high hit ratio while maintaining a relatively low migration rate.

1. Introduction

Ever-growing amounts of data will make the storage capacity within a storage area network (SAN) increase exponentially. Different workloads running on a storage system combine to result in a wide spectrum of access patterns at the same time. In normal array caches, cache resources are allocated approximately

according to their relative I/O rates [5]. Then different access streams meet at the array cache and hot chunks of data of one stream may very likely be evicted by those of other access streams, which results in interference between different access streams and makes the array cache much less efficient.

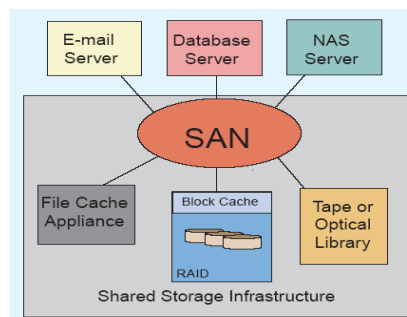


Figure 1. Topology of a typical system with SAN-level caching

Figure 1 shows a SAN environment with SAN-level caching. We envision back-end storage composed of large data chunks (the chunk size can range from 128 kilobytes to several megabytes). Caching based on front-end hosts or back-end disk arrays lacks the intelligence to identify hot data chunks within a global view, while SAN-level caching can intelligently identify such hot ones and then migrate them to cache appliances [2, 3, 7], so that the overall performance of the system can be improved. What is important is that the interference between different access streams at the array cache can be reduced greatly, and thus the array cache

This research was supported by the National Natural Science Foundation of China under Grant No. 60473101, and the National Grand Fundamental Research 973 Program of China under Grant No. 2004CB318205, and Supported by Program for New Century Excellent Talents in University.

Authors' addresses: Department of Computer Science and Technology, Tsinghua University, 100084 Beijing, China;

becomes more efficient [5]. The benefits that SAN-level global caching brings to a system can be two-fold: first, host I/Os served from the cache will have shorter response times; second, I/Os that arrive at disks will have a shorter queuing time on average because of the reduction of the I/O queue length.

In this paper, a data chunk is defined as temporally hot if it gets burst accesses in some short periods of time but its total access number is less than a predefined value, and a data chunk is defined as long-term hot if it gets long-term frequent accesses and its total access number is larger than the predefined value. It has been found that in a storage system, there are many data chunks which only get burst accesses and remain temporally hot [11]. Although they normally do not have a high priority in caching, they usually have impacts on hit ratios. There are two almost equally important objectives that we aim to achieve to improve the system performance. One is to identify and cache both temporally hot and long-term hot data chunks as early as possible to improve the cache hit ratio. The other objective is to reduce data migrations from disks to cache as much as possible.

In order to fulfill these goals, we propose chunk-aging to control the migrations of data chunks and employ two LRU lists to manage cache replacement. Chunk-aging is put forward to identify and cache hot data chunks as early as possible. It associates a weight with each chunk to take the intervals between two consecutive accesses to the chunk into account: a formula about the access interval is used to calculate the weight of a chunk that if the interval is relatively short, its weight increases and if the interval is relatively long, its weight decreases. As for cache replacement, two LRU lists are employed to separately manage temporally hot data chunks and long-term ones, which work to avoid the interference in the cache between these two types of data chunks, thus reducing migrations of data chunks to cache.

The rest of this paper is organized as follows.

Section 2 describes previous works. In Section 3, our caching method is presented. In Section 4, simulation results are presented to demonstrate the effectiveness and efficiency of our method. In Section 5, we conclude this paper and discuss our future work.

2. Related Work

In SAN environments, many strategies have been used to improve the performance of storage systems. Ergastulum [4] reported a way to automatically find the appropriate RAID levels for different streams. Hippodrome [6] automated the assignment of workload streams to different Logical Units (LUs) and reevaluated the performance results to reconfigure storage resources based on SAN virtualization. However, SAN-level caching has not yet been fully exploited to bridge the speed gap between front-end processors and storage subsystems.

Caching on demand is too aggressive to be feasible in a SAN environment [5], because the intrinsic cache thrashing will result in a huge amount of data chunk migrations from disks to cache, which may counteract the benefits that cache hits bring to the system and thus may even make system performance worse. SANBoost [5] adopted a SAN-level caching method to alleviate cache thrashing, and it can improve the performance of a SAN system by caching data chunks whose access counts exceed a predefined migration threshold. However, this frequency-based caching method only takes into account the access counts of data chunks, without considering the temporal distribution of the accesses, and counting the number of accesses does not provide enough information to determine the recent hotness of the data chunks. It also sacrifices the hit ratio from temporally hot data chunks to control the migration rate, which causes insufficient caching.

It is necessary to filter out sparsely-accessed chunks to avoid cache thrashing and to cache the

temporally hot data chunks to further improve the hit ratio. This paper propose chunk-aging as a novel efficient caching method, which not only can identify and cache hot data chunks as early as possible but also can filter out the interference of sparsely-accessed chunks. Dahlin [8] introduced the concept of file-aging to improve network file cache performance. File-aging was also studied by Smith and Seltzer [9] in the context of file system benchmarks, and Miller [14] used file-aging to predict file usage for file migration.

As for cache replacement, much research has been conducted on this issue. Recently proposed algorithms include LRU-k [10], LRFU [11], 2Q [12], MQ [13], and ARC [15]. Some of them also use multiple LRU lists to manage the cache, such as 2Q and ARC. However, these methods are designed for on-demand caches and lack the migration control process.

In this paper, we propose a cache replacement policy that is simple to implement. It uses chunk-aging to control the data migration process and uses two LRU lists to manage cache replacement. The cache replacement policy we adopt is thus as simple as the traditional classic LRU is.

3. Our SAN-level Caching Method

Our caching method logically consists of Data Chunk Stage-in and Data Chunk Stage-out. Stage-in makes use of chunk-aging to update the weight associated with each data chunk when it is accessed, and then caches this data chunk if it is qualified for caching. Stage-out employs two LRU lists to manage cache replacement.

3.1. Data Chunk Stage-in

Stage-in employs chunk-aging to control the migration process of data chunks from disks to cache. The main idea of chunk-aging is to take into account not only the access count for a data chunk but also the

temporal distribution of its accesses. It associates a weight with each data chunk to determine the hotness of the data chunk. A data chunk is assigned a large weight if it is frequently accessed during the most recent period of time, and is assigned a small weight if it is sparsely accessed. By checking the weight, the hotness of a data chunk can be determined. Chunk-aging defines a data chunk to be hot if its weight is greater than a predefined migration threshold, and this data chunk is qualified for caching.

The following formula is presented to update the weight of each data chunk when it is accessed, based on its historical weight:

$$weight(t_{n+1}) = weight(t_n) \times e^{-\alpha(t_{n+1}-t_n)} + 1$$

where t_n and t_{n+1} denote the time of two consecutive accesses to a data chunk with t_n prior to t_{n+1} , $weight(t_n)$ is the weight computed for the data chunk when it is accessed at t_n . If a data chunk is accessed for the first time, its weight is initialized as 1. The term α is defined as the sensitivity coefficient of weight versus the passage of time. The larger α is, the more drastically the weight changes with the passage of time. When $\alpha = 0$, chunk-aging degenerates into a frequency-based caching method just as SANBoost.

Chunk-aging updates the weight of a data chunk only when it is accessed, and the time and the weight of its very last access have to be recorded. Also, the access count of a data chunk is recorded for the convenience of cache replacement. In the rest of this section, we discuss how chunk-aging can solve the problems of insufficient caching and cache thrashing.

Let $weight_0$, $weight_1$, $weight_2$, and $weight_3$ respectively be the weights computed for a data chunk when it is accessed at t_0 , t_1 , t_2 , and t_3 , and let Δt_i ($i=1,2,3$) be the interval between t_i and t_{i-1} . We also assume the data chunk is accessed for the first time at t_0 and thus $weight_0 = 1$, so we have:

$$weight_1 = e^{-\alpha \Delta t_1} + 1$$

$$\begin{aligned} \text{weight}_2 &= \text{weight}_1 \times e^{-\alpha\Delta t_2} + 1 \\ &= e^{-\alpha(\Delta t_1+\Delta t_2)} + e^{-\alpha\Delta t_2} + 1 \end{aligned}$$

$$\begin{aligned} \text{weight}_3 &= \text{weight}_2 \times e^{-\alpha\Delta t_3} + 1 \\ &= e^{-\alpha(\Delta t_1+\Delta t_2+\Delta t_3)} + e^{-\alpha(\Delta t_2+\Delta t_3)} + e^{-\alpha\Delta t_3} + 1. \end{aligned}$$

The computation of weight_3 is used as an example to analyze chunk-aging. If the interval between t_0 and t_3 is long, namely $(\Delta t_1+\Delta t_2+\Delta t_3)$ is large, then the effect that the access at t_0 has on weight_3 , $e^{-\alpha(\Delta t_1+\Delta t_2+\Delta t_3)}$, is minute; if the accesses at t_1 , t_2 and t_3 converge in a short period of time, namely Δt_2 and Δt_3 are small, then the effect that the accesses at t_1 and t_2 have on weight_3 , $e^{-\alpha(\Delta t_2+\Delta t_3)} + e^{-\alpha\Delta t_3}$, is huge and accordingly weight_3 is large. Furthermore, if weight_3 is larger than the predefined migration threshold, the data chunk will be cached. So, once a data chunk gets burst accesses, its weight will soon become greater than the migration threshold and it will soon get cached, and then subsequent accesses to the chunk will get data from the cache. So the problem of insufficient caching can be solved so that cache chances can be fully exploited.

In order to show how chunk-aging avoids cache thrashing, we assume that a data chunk has been evicted from cache because it has received no access for a relatively long period of time since its last access, but it has a pretty large weight that has not been updated, since it has received no new access. Then if the chunk is accessed again Δt time later, then its historical weight is multiplied by the factor $e^{-\alpha\Delta t}$ and its weight is very likely to become less than the migration threshold, and thus the data chunk will not get cached right away. The data chunk can only get cached when it again receives dense accesses, and thus cache thrashing chances are greatly reduced.

Chunk-aging detects and caches hot data chunks

as early as possible. Just as analyzed above, by using chunk-aging, we succeed in solving the problems of insufficient caching and cache thrashing.

3.2. Data Chunk Stage-out

Stage-out provides a priority ordering of migrated chunks in the cache and evicts the least preferable chunks from cache to create spaces for incoming chunks. Once a long-term hot data chunk is cached, it should not soon be evicted from cache by temporally hot data chunks, so we partition a part of the cache resource to cache temporally hot data chunks and reserve the rest for long-term hot data chunks. And two LRU lists for cache replacement are maintained to separately manage these two parts to avoid interference between the two types of data chunks. One of the two LRU lists, referred to as LRU_1 , is used to manage temporally hot data chunks, and the other, LRU_2 , is used to manage long-term hot data chunks.

When a non-cached data chunk receives burst accesses but its access count is less than a predefined number of times L (L equals 30 in our simulation), then the data chunk is identified as temporally hot, and if it is qualified for caching, it will be linked into LRU_1 . If there is no free space, one cached chunk in LRU_1 will get evicted to create space for it; when a chunk in LRU_1 gets accesses for more than the predefined number of times L , then the chunk is assumed to have become long-term hot and it will be removed from LRU_1 and linked into LRU_2 . If there is no free space, one chunk in LRU_2 will be evicted from the cache to create space for it; if a non-cached data chunk needs caching and it has already gotten accesses for more than the predefined number of times L , then the chunk will be directly linked into LRU_2 .

By using these two LRU lists to separately manage temporally hot data chunks and long-term hot data chunks, it is ensured that long-term hot data chunks in the cache only get evicted to provide spaces

for long-term hot data chunks that need caching, and so do temporally hot data chunks. By using this two-LRU-list replacement policy, our method succeeds in avoiding interference between the two types of data chunks and thus greatly decrease data migrations.

3.3. Memory Cost and CPU Overhead

The time and the weight of the very last access and the access count of a chunk have to be recorded. However, the chunk size (more than 128K) of a SAN cache is usually much larger than that a normal local cache (4K or 8K), which means that its proportional memory cost is still low.

By using two LRU lists, Stage-out introduces only constant computing complexity. Since modern CPU has more powerful computing capability, this overhead will not introduce a large latency.

4. Experimental Results and Evaluation

4.1. Simulation Issues

We built a primitive simulator and used a SPC [1]-1 web search trace for the simulation to examine the benefits obtained from using our caching method. This trace is collected within 2 hours, and is composed of 3,000,000 access records.. We measured the benefits achieved by caching from two aspects: the cache hit ratio and the migrations of data chunks from disks to cache. We implemented the SANBoost policy and compared our caching method with the on-demand caching method and SANBoost, in which 30 is used as the migration threshold. And we also investigated how the chunk-aging caching method behaves with the one-LRU-list replacement policy compared with our caching method, which adopts the two-LRU-list replacement policy.

A 2-GB cache was simulated in our experiment.

The 2-GB cache was envisioned to be composed of 256-KB chunks (disks were assumed to be composed of 256-KB chunks too). 1/8 of all the cache chunks were used to cache temporally hot data chunks, and the rest were reserved for long-term hot data chunks. When using the trace for simulation, we did not discriminate whether an access record was a read or write access, and it was assumed that cache consistency was guaranteed, so consistency is not discussed here.

Figure 2 shows the access histogram of the SPC-1 trace, obtained by mapping block requests to the closest 256KB chunk boundaries. It can be seen from Figure 2 that many blocks are accessed less than 30 times (the typical threshold used in SANBoost), and these chunks are skipped from caching by SANBoost, while our method fully exploits these opportunities to improve the hit ratio. Different from SANBoost, our method also caches long-term hot data chunks even when they are accessed for the first few times, but not till they are accessed for at least 30 times.

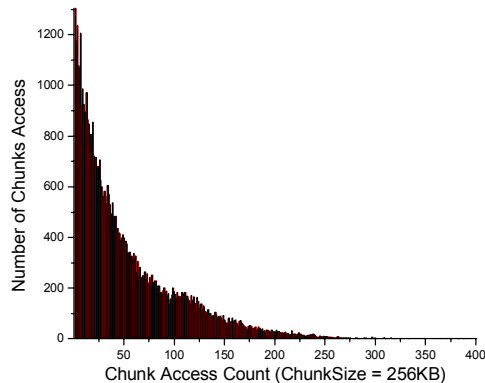


Figure 2. Histogram of Chunk Access Counts

4.2 Experimental Results

Each of the following figures shows a comparison of the hit ratio and migration number of different caching methods. The X-axis denotes the progress of the trace simulation, for example the point '8' means that 80% of the whole trace has been played. As shown

in Figure 3, the contradiction between the hit ratio and migration number is obvious: the higher the hit ratio is, the larger the migration number is. The hit ratio achieved by chunk-aging caching is respectively 29.3%, 32.1% and 39.5% higher than SANBoost

caching, while the migration number is 391,164, 433,510 and 623,955 respectively larger than that of the SANBoost, 58,408. Although the hit ratio achieved by on-demand caching is 44.7% higher than SANBoost caching, it migrates 1,100,000 more data chunks.

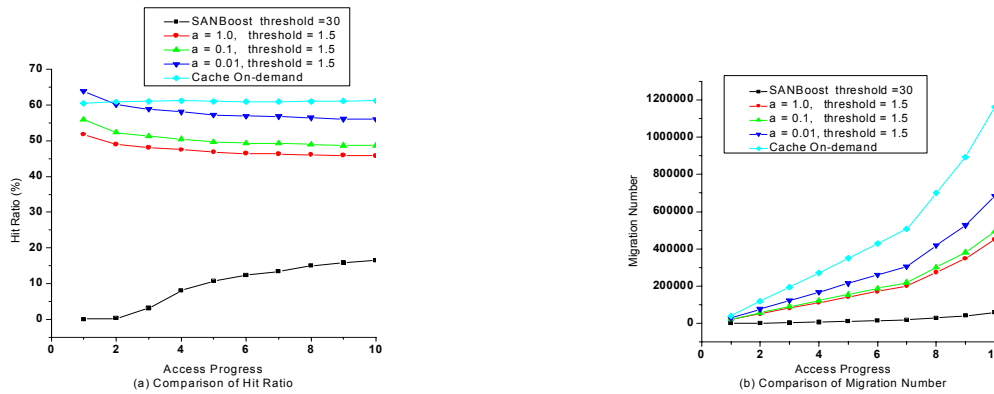


Figure 3. Comparison between on-demand caching, SANBoost caching and chunk-aging based caching, using one LRU list to manage cache replacement, and the migration threshold is 1.5 and α is 1, 0.1 and 0.01 respectively.

When the migration threshold remains unchanged, the hit ratio and migration number increase as α decreases. This is because a small α value makes the weight less sensitive to the passage of time. It will increase as long as the data chunk is accessed, even if the interval between two consecutive accesses is long.

We can see from Figure 3 that although the hit ratio achieved by chunk-aging is higher than that of the SANBoost, the migration number increases too fast to

be bearable. The reason for this is that only one LRU list is used to manage all the cache resources, and when the cache is full, temporally hot data chunks may replace the least recent long-term hot data chunk in the cache, which results in interference between the two types of data chunks and makes the migration number explode. This phenomenon also appears in Figure 4, which shows that the smaller the migration threshold is, the larger the hit ratio and migration number are.

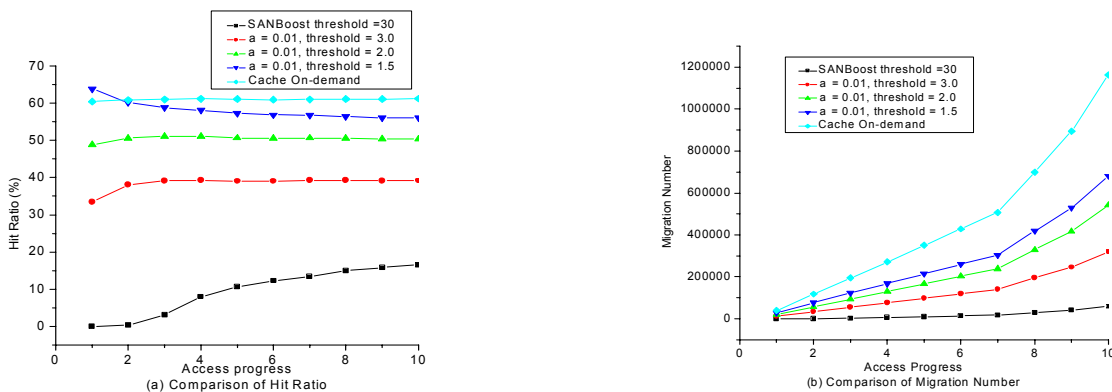


Figure 4. Comparison between on-demand caching, SANBoost caching and chunk-aging based caching, using one single LRU list to manage cache replacement. α is 0.01 and the migration threshold is 3.0, 2.0 and 1.5 respectively.

The benefits obtained from our method, which is based on chunk-aging and uses two LRU lists to manage cache replacement, are shown in Figure 5. And because two LRU lists are used to prevent interference between temporally hot data chunks and long-term hot data chunks, the contradiction between hit ratio and migration number is largely mitigated. When the

migration threshold is 4.0, 3.0 and 2.0, the hit ratio achieved by our method is 27.2%, 38.5% and 49.8% respectively higher than that of SANBoost, and the migration number is 0.91, 1.39 and 2.82 times that of SANBoost. The simulation results demonstrate that our method provides better performance with respect to both the hit ratio and the migration number.

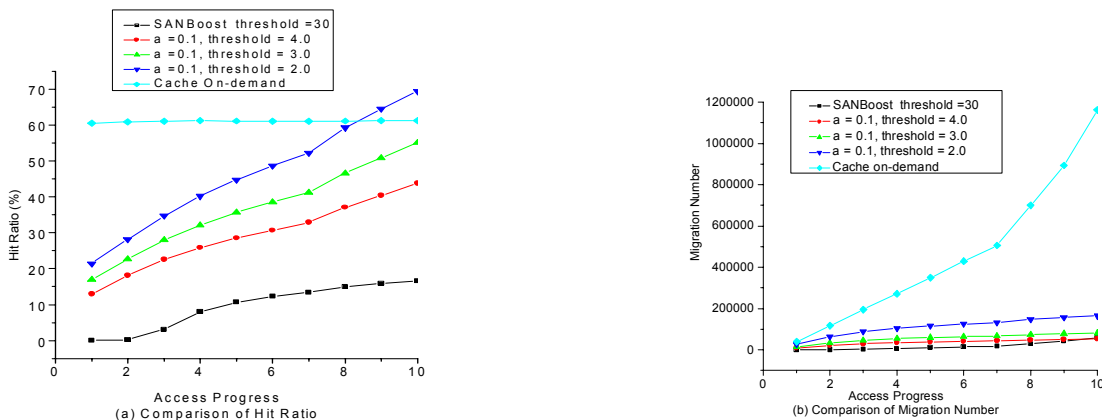


Figure 5. Comparison between on-demand caching, SANBoost caching and our caching method, which uses two LRU lists to manage cache replacement, where α is 0.1 and the migration threshold is 4.0, 3.0 and 2.0.

5. Conclusion and Future Work

This paper presents an efficient SAN-level caching method in which chunk-aging is proposed to control the migrations of hot data chunks, and by considering the access counts of data chunks, hot data chunks are categorized into temporally hot data chunks and long-term hot data chunks. Two LRU lists are used to separately manage cache resources for these two types of data chunks. As the simulation results show, unlike either the on-demand caching or SANBoost, our method not only achieves a high hit ratio but also ensures a relatively low migration rate. When the migration threshold equals 3.0 and α equals 0.1 in our method, the hit ratio reaches 55.1%, which is 3.32 times that of SANBoost caching, and its migration number is only 7.1% that of on-demand caching (the migration number of SANBoost caching is 5.1% of

on-demand caching). Our method gets much closer to the goal of achieving a high hit ratio while minimizing data chunk migrations.

Since different sensitivity coefficients of weight versus the passage of time α and different migration thresholds in our method will result in different benefits for cache hits and migrations, which will further influence the response time and throughput of systems, our future work will be focused on by on-line observing the average response time of the storage subsystem to develop a dynamic policy to tune α and the migration threshold.

6. References

- [1] SPC benchmark-1 specification, <http://www.storageperformance.org/specification.html>

- [2] SolidData, SD3000 Solid-State Disk and White Papers, <http://www.soliddata.com>.
- [3] Texas Memory Systems, RamSan Solid-State Disk, <http://www.texmemsys.com>.
- [4] E.Anderson, R.Swaminathan, A.Veitch, G.A.Alvarez and J.Wilkes, "Selecting RAID levels for disk arrays," *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, Jan 2002.
- [5] Ismail Ari, Melanie Gottwals and Dick Henze, "SANBoost: Automated SAN-Level Caching in Storage Area Networks," *Proceedings of the 13th IEEE International Conference on Autonomic Computing*, May 2004.
- [6] E.Anderson, M.Hobbs, K.Keeton, S.Spence, M.Uysal, and A.Veitch, "Hippodrome: running circles around storage administration," *Proceedings of the 2002 Conference on File and Storage Technologies*, Jan 2002.
- [7] Shu Jiwu, Yu Bing and Yan Rui, "A Design and Implementation of a Non-volatile RAM Disk in SAN environment," *Proceedings of the 3rd International Conference on Grid and Cooperative Computing Workshop on Storage Grid and Technologies*, Oct 2004.
- [8] Michael D. Dahlin, Clifford J. Mather, Randolph Y. Wang, Thomas E.Anderson, and David A. Patterson, "A quantitative analysis of cache policies for scalable network file systems," *Proceedings of the SIGMETRICS '94 Annual Conference on Measurement and Modeling of Computer Systems*, May 1994.
- [9] Keith A.Smith and Margo I.Seltzer, "File System Aging-Increasing Relevance of File System Benchmarks," *Proceedings of the 1997 SIGMETRICS Conference*, Jun 1997.
- [10] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Trans. Computers*, vol. 50, no. 12, 2001.
- [11] E. O'Neil, P. O'Neil and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1993.
- [12] T.Johnson and D.Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," *Proceedings of the 20th Conference on File and Storage Technologies (FAST)*, Jan 1995.
- [13] Y. Y. Zhou and J.F.Philbin, "The multi-queue replacement algorithm for second level buffer caches," *Proceedings of USENIX Annual Tech.Conf. (USENIX 2001)*, Jun 2001.
- [14] Timothy J. Gibson and Ethan L.Miller, "An improved long-term file-usage prediction algorithm," *Proceedings of the 25th Annual Conference on Computer Measurement and Performance (CMG'99)*, Dec 1999.
- [15] N. Megiddo, D. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache", *Proc. of FAST '03*, March 2003, pp. 115-130.