

SOPA: Selecting the Optimal Policy Adaptively for a cache system

YANG WANG, JIWU SHU, GUANGYAN ZHANG, WEI XUE, WEIMIN ZHENG
Tsinghua University

With the development of storage technique, new caching policies are continuously being introduced, which makes it increasingly important for cache systems to select the optimal caching policy dynamically under varying workloads. This paper proposes SOPA, a mechanism to adaptively select the optimal policy and perform policy switching. First, SOPA encapsulates the functions of a caching policy into a module, and enables online policy switching by policy reconstruction. Second, SOPA selects the optimal policy dynamically by collecting and analyzing access traces, and reduces the decision-making cost by asynchronous decision. The simulation evaluation showed that no single caching policy could perform well under all of the different workloads, but SOPA could select the appropriate policy for each workload. The real-system evaluation showed that SOPA could reduce the average response time by up to 20.3% compared with LRU and up to 11.9% compared with ARC.

Categories and Subject Descriptors: D.4.2 [Operating Systems]: Storage Management; H5.2 [Information Storage and Retrieval]: Information Storage

General Terms: Algorithms, Performance, Design

Additional Key Words and Phrases: caching policies, policy adaptation, policy switch

1. INTRODUCTION

1.1 Motivation

Caches can improve the overall performance of storage systems greatly. They are widely used in file systems [Nelson et al. 1998], database systems [Li et al. 2005; Chou and Dewitt 1985], RAID controllers [Menon 1994], disks [Menon and Hartung 1988], Web servers [Cao and Irani 1997] and so on. With the development of storage architecture and applications, new caching policies are continuously being designed, such as MQ [Zhou and Philbin 2001], ARC [Megiddo and Modha 2003], SANBoost [Ari et al. 2004], DULO [Jiang et al. 2005] and WOW [Gill and Modha 2005] and. Some of them are self-adaptive, such as MQ and ARC; some of them make optimizations for specific workloads.

This research was supported by the National Natural Science Foundation of China under Grant Nos. 10576018, 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 (NCET-05-0067).

Authors' addresses: Department of Computer Science and Technology, Tsinghua University, 100084 Beijing, P. R. China; email: {iodine01, zhang-gy04}@mails.tsinghua.edu.cn, {shujw, xuwei, zwm-dcs}@tsinghua.edu.cn, Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permission@acm.org.

© 2007 ACM 1553-3077/2007/03-ART3 \$5.00 DOI = 10.1145/1227835.1227838 <http://doi.acm.org/10.1145/1227835.1227838>

For example, DULO can achieve a high hit ratio under workloads with many sequential accesses. SANBoost is designed for the caches in the SAN environment. Self-adaptive caching policies can perform well in many cases, but they are not optimal under some specific workloads.

In fact, as a result of the evolving and changing access patterns, no single policy is able to adapt to all workloads [Ari et al. 2004] and new policies will also appear definitely. This makes it increasingly important for cache systems to select the optimal caching policy dynamically. In most existing systems, the caching policies are tightly coupled with the cache systems, making it hard to perform policy switching.

As the CPU technology evolves, modern machines have a more powerful computing capability. In this paper, we make use of the abundant computing ability of CPUs to select the optimal caching policies dynamically for the cache systems.

1.2 Our Contributions

To enable caching policy selection and switching, we propose SOPA, which evaluates the existing policies and performs policy switching online. SOPA mainly uses two techniques to achieve its goals:

—The modularization of caching policies and policy switching: The cache systems call the functions of the caching policy modules through a fixed interface, and they do not depend on the implementation details of these policies. By policy reconstruction, SOPA enables the online switching between any two caching policies and allows the online deployment of new policies.

—The online policy decision-making: SOPA can select the optimal caching policy dynamically under a varying workload. It collects access trace and selects the optimal policy through trace analyzing. It reduces the decision-making cost by asynchronous decision.

SOPA was evaluated by both simulation and real system experiments. The simulation results showed that a single caching policy could not perform well under all of the different workloads, but SOPA could select the appropriate policy for each workload and achieved satisfactory hit ratios. The real system evaluation on an IPSAN system showed that SOPA could reduce the average response time by up to 20.3% compared with LRU and up to 11.9% compared with ARC.

1.3 Outline of this Paper

In Section 2, we briefly review relevant work. In Section 3, we introduce the architecture of SOPA. In Section 4 and 5, we present the results of simulation and real system evaluations. Finally we present conclusions and future work in Section 6.

2. RELATED WORKS

First, we have a brief view about the existing caching policies. Then, we discuss two kinds of caching policy adaptation: 1) Intra-policy Adaptation: to adapt to more workloads by tuning the parameters of the caching policies. 2) Inter-policy Adaptation: to suggest the optimal caching policy by comparison among several policies.

2.1 Existing Caching policies

With the development of storage systems, various caching policies have been introduced. LRU and LFU are the earliest caching policies. Later policies take into account both recency and frequency factors, such as LRU-k [O'Neal et al. 1993], 2Q [Johnson and Shasha 1994], FBR [Robinson and Devarakonda 1990], LRFU [Lee et al. 2001], LIRS [Jiang And Zhang 2002], MQ [Zhou and Philbin 2001], ARC [Megiddo and Modha 2003] and CAR [Bansal and Modha 2004]. These policies mainly try to raise the hit ratio and require no additional information from the cache system. Some policies also take into account other effects, or try to use some additional information from the cache system. SANBoost [Ari et al. 2004], WOW [Gill and Modha 2005] and Karma [Yadgar and Factor 2007] are such examples.

In fact, as a result of the evolving and changing access patterns, no single policy is able to adapt to all workloads [Ari et al. 2002] and new policies are appearing almost every year.

2.2 Intra-policy Adaptation

Some caching policies have tunable parameters, and by tuning these parameters, they can adapt to more workloads. For part of them, the tuning of parameters is done manually by the administrator. For example, the *Correlated Reference Period* in LRU-k, K_{in} and K_{out} in 2Q, L_{hir} in LIRS, λ in LRFU, and F_{new} and F_{old} in FBR are all manually tunable parameters. The designers of these policies give some suggestions about the tuning of these parameters, but it requires the administrators to have abundant experience and it is difficult to change these parameters online.

Self-adaptive caching policies can tune the parameters online without the interference of the administrators. MQ and ARC are two typical self-adaptive caching policies. The *lifetime* parameter of MQ is self-adaptive. MQ uses several queues to identify the priority of a cache block and lifetime is used to determine how long a cache block can stay in a queue. MQ recommends using the peak temporal distance as the value of lifetime, and it can be dynamically estimated for each workload. ARC uses two queues, T_1 and T_2 , to balance recency and frequency. The sizes of T_1 and T_2 are self-adaptive. T_1 contains blocks that have been accessed only once recently, while T_2 contains blocks that have

been accessed at least twice recently. Two history lists, B_1 and B_2 , are kept for T_1 and T_2 respectively. When ARC finds that an access hits B_1 , it will increase the size of T_1 and decrease the size of T_2 . Conversely, when an access hits B_2 , ARC will increase the size of T_2 and decrease the size of T_1 . In this way, ARC can automatically tune the size of T_1 and T_2 according to the characteristics of the workload. Self-adaptive policies can adapt to a wide range of workloads without the interference of human beings, but such adaptation is still limited to the policy. In many cases, the self-adaptive policies cannot behave as well as the optimal policy.

2.3 Inter-policy Adaptation

ACME (Adaptive Caching using Multiple Experts) [Ari et al. 2002] presents the idea of selecting the caching policies dynamically. ACME simulates several policies and assigns a weight to each policy. Each policy votes on the blocks it wants to keep, assigning higher values to blocks that it believes are more worth keeping. The block with the lowest weighted vote total is selected as the victim. ACME uses a machine learning method to adjust the weights of these policies. The advantage of ACME is that it can combine several policies to construct a new one. But for every missed access, the expert system has to give the victim block. The whole process introduces a large latency and overhead. Furthermore, ACME has to maintain several policies simultaneously, which costs much memory.

[Salmon et al. 2003] introduces a policy decision-making method for disk systems and suggests that it can also be used for cache systems. But it lacks implementation and verification for cache systems. Compared with disk systems, cache systems have a higher limitation of latency and efficiency. Like ACME, it is difficult for this method to satisfy these requirements.

2.4 Comparison with Our Work

Instead of introducing a new policy, SOPA makes better use of existing ones. Compared with intra-policy adaptation, inter-policy adaptation can adapt to a wider range of workloads, but it usually has a larger overhead. SOPA selects the optimal policy among the existing ones, so it is a kind of inter-policy adaptation. Different from ACME, SOPA selects the optimal policy, but not the optimal victim block, so the overhead is reduced. In most time, the overhead of SOPA is close to a normal caching policy.

3. THE ARCHITECTURE OF SOPA

The architecture of SOPA is depicted in Figure 1. SOPA consists of three components: Caching Policy Modules (CP-Mods), Trace Collector, and Trace Analyzer. Every CP-

Mod encapsulates the functions of a caching policy and exports these functions to other modules, which are called caching policy users (CP-Users), through a fixed interface. By policy reconstruction, the CP-User can switch between any two CP-Mods dynamically. The cache system itself is a CP-User and its caching policy is determined by the currently-used CP-Mod, which is called an “active CP-Mod”. The cache system can select a candidate CP-Mod and switch the active CP-Mod. It can also deploy a new policy by adding a new CP-Mod. The Trace Collector records I/O traces when policy decision-making is needed. It first stores the I/O traces in a memory trace pool, and when the pool is full, it moves them to a file on the disk and empties the memory pool. The Trace Analyzer is also a CP-User. It selects the optimal policy for the cache system when enough trace is collected. The Trace Analyzer replays the access trace collected by the Trace Collector with all the candidate policies one by one. Then it selects the optimal one and informs the cache system of the result.

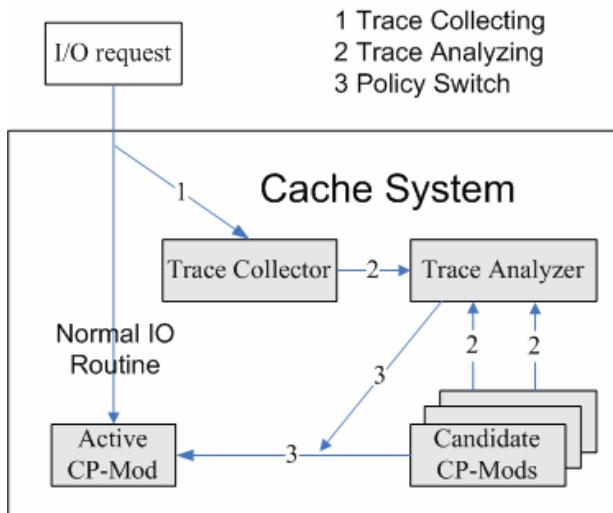


Fig.1. Architecture of SOPA. CP-Mod stands for caching policy module. 1. The Trace Collector collects I/O traces. 2. The Trace Analyzer decides the optimal policy. 3. The cache system performs a policy switch

This architecture has several advantages. First, the cache system can perform policy switching online, thus it can adapt to the varying workloads. Second, new policies can be deployed dynamically, thus the newest research achievements can be utilized. Third, the overhead is low. When no policy decision is performed, SOPA introduces only a little CPU overhead and memory cost.

3.1 Modularization of Caching Policies

The key idea of the modularization of caching policies is as follows: the CP-Mod implements a group of interface functions, and registers these functions to the CP-User;

then the CP-User calls the functions of the CP-Mod through this interface. The modularization of caching policies separates the CP-User from the implementation details of the CP-Mod, thus supporting online policy switching.

After studying the existing caching policies, we designed the interface and metadata organization for the modularization of caching policies (In this paper, metadata refers to the data used to maintain the cache system and the caching policy). The design complies with three rationales:

1. Efficiency: SOPA should not add much overhead to the execution of the caching policies.
2. Function separation: the CP-Mod should only deal with the logic of the caching policy, without considering the other functions of the CP-User. This allows the policy designers to focus on their policies and simplifies their work.
3. Metadata separation: the CP-User should not maintain the policy-specific metadata, which should be managed by the CP-Mod. This rationale enables the deployment of new CP-Mods and reduces the memory costs of SOPA.

The design of the interface and metadata organization in SOPA complies with these three rationales. However, there are conflicts among these rationales. For example, rationale 3 will complicate the process of metadata lookup, which will impair efficiency. SOPA provides a balance among these rationales.

3.1.1 Interface and Metadata Organization. To design the interface and the metadata organization, we have studied the design and implementation of various policies, including LRU, LFU, LRU-k, 2Q, FBR, LRFU, LIRS, MQ, ARC, CAR, DULO and SANBoost. The design is depicted in Figure 2. The interface is composed of three groups of functions, and the CP-Mod registers its implementation of this interface to the CP-User. The metadata is divided into two parts: universal metadata and policy metadata. The universal metadata is allocated and maintained by the CP-User and has a pointer to the policy metadata, which is managed by the CP-Mod. The CP-User does not understand or maintain the policy metadata, which satisfies the rationale of metadata separation.

The interface of the CP-Mod is composed of three groups of functions: 1) Initialization and release functions: including the *initialize* function to allocate and initialize the policy metadata and the *release* function to release the policy metadata. 2) Cache access processing functions: including the *blk_hit* function to process a cache block hit event, the *blk_new* function to process a cache block insertion event and the *blk_find_victim* function to process the cache block replacement event. The cache system will call these functions when the corresponding cache event occurs. 3) Cache access pre-

processing functions : including the *pre_blk_access*, *pre_blk_hit*, *pre_blk_new*, and *pre_blk_find_victim* functions to pre-process the corresponding cache events.

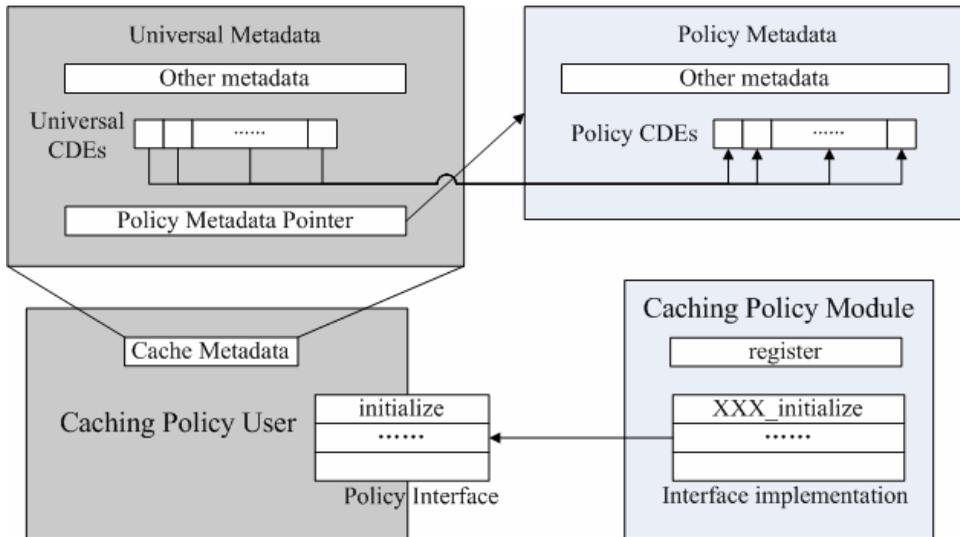


Fig.2. Design of the interface and the metadata organization. The cache directory entry (CDE) carries the information describing a cache block

Generally speaking, on-demand caching policies (every missed block is migrated into the cache) only require the implementation of the first two groups. The other caching policies such as SANBoost or prefetch policy may require the implementation of all the three groups.

The design of the interface has several advantages: 1) SOPA can make use of existing caching policies easily. They can be brought into SOPA with a small adjustment. 2) The CP-Mod does not need to deal with the details of the cache system, which satisfies the rationale of function separation.

The metadata organization is required to satisfy the rationale of metadata separation and it should also guarantee the efficiency of policy execution. SOPA takes both into account. A normal cache implementation uses a specific data structure to describe the information of a cache block, and it is called the Cache Directory Entry (CDE) structure [Robinson and Devarakonda 1990] in this paper. Some other data structures are used to look up the CDEs or to implement the caching policy, such as hashtable [Robinson and Devarakonda 1990], LRU list [O'Neal et al. 1993; Ari et al. 2002; Johnson and Shasha 1994; Zhou and Philbin 2001; Megiddo and Modha 2003], history records [Zhou and Philbin 2001; Megiddo and Modha 2003], etc. As shown in Figure 2, SOPA divides CDEs into Universal CDEs and Policy CDEs, and also divides the other metadata into two parts. SOPA recombines them into universal metadata and policy metadata. The

former is maintained by the CP-User, and the latter is allocated and managed by the CP-Mod. The CP-User does not understand or depend on the organization of the policy metadata, which satisfies the rationale of metadata separation. To improve the efficiency of metadata lookup, SOPA adds a pointer in each Universal CDE, which points to its corresponding Policy CDE. Additionally, SOPA keeps the most frequently and widely used policy metadata in the Universal CDE to improve the accessing efficiency. SOPA provides a balance between the rationale of efficiency and the rationale of metadata separation.

3.1.2 Switching the Caching Policy. The basic process of policy switching is as follows: the CP-User *releases* the current caching policy, then registers and *initializes* the new caching policy. The main problem is that after a period of time of running, some blocks have been migrated into the cache, but after the policy switching, the new policy metadata is blank. A simple solution is to empty the cache, but this has several drawbacks. First, all the dirty blocks have to be written to the disks, which brings a large latency. Second, the cache has to be re-warmed, which reduces the hit ratio and brings an extra cost of block re-migration.

SOPA reconstructs the policy metadata according to the existing cache blocks. It does not require any disk accesses or cache block movements. Furthermore, it can provide warm-up for the new caching policy and avoids the costs of block re-migration. The difficulty is that as a result of the rationale of metadata separation, the CP-User does not understand the organization of the policy metadata, and thus cannot build up the policy metadata directly. To solve this problem, the CP-User keeps an LRU History List (LHL) for the existing cache blocks: when a new block is migrated into the cache or when a block is hit, this block is moved to the tail of the LHL (just the adjustment of a pointer, requiring no data movement); when a block replacement occurs, the victim block is removed from the LHL, and the new block is moved to the tail of the LHL. In this way, the CP-User keeps an access history for the existing cache blocks in an LRU order. When a policy switching is performed, the CP-User calls the *blk_new* function of the new CP-Mod for each LHL record in order. In this way, the CP-Mod can reconstruct its own policy metadata indirectly through the *blk_new* interface function. The policy reconstruction does not require the CP-User to understand the policy metadata and can be used between any two CP-Mods. The cost of this method is that an LRU list entry has to be added to each Universal CDE to maintain the LHL.

3.2 Policy Decision-making

Policy decision-making is used to select the optimal policy. The process consists of three steps: first, the Trace Collector records the I/O traces. Second, when enough traces are collected, the Trace Analyzer replays these I/O traces with all the candidate caching policies and selects the optimal CP-Mod. Finally, a policy switch is performed if the Trace Analyzer finds that the Active CP-Mod is not the optimal one. The I/O traces are deleted after each decision process.

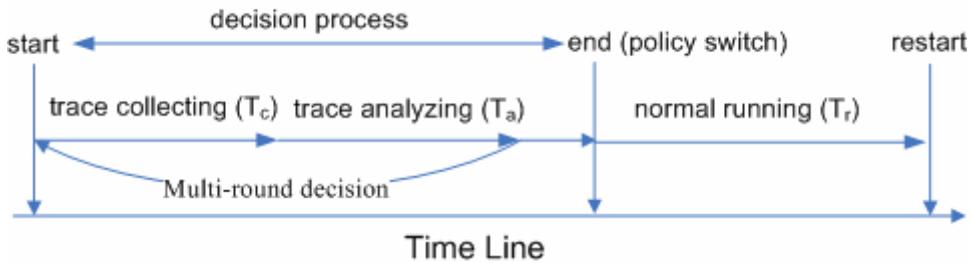


Fig.3 The decision process

Figure 3 depicts the decision process. T_c is the time for trace collecting, T_a is the time for trace analyzing and T_r is the time for normal running without additional operations. In most cases, $T_a \ll T_c \ll T_r$, so this overhead will not have a heavy impact on the cache system. Typically, replaying 1M accesses for one policy takes only tens of seconds on a modern CPU.

To make a correct and quick decision, SOPA introduces multi-round decision, which is discussed in Section 3.2.1. When the hit ratio of the cache system drops greatly, which means that the workload has changed greatly and the last decision result may not be useful any more, a new decision process will start.

3.2.1 Multi-Round Decision. One major problem of the decision process is the value of T_c . If T_c is too small, the traces collected may not have enough information about the workload and the decision process will not achieve the correct result. If T_c is too long, the trace collecting process will introduce more overhead. To solve this problem, SOPA introduces multi-round decision to both ensure the correctness of the decision and limit T_c : when a fixed number of traces are collected, the Trace Analyzer replays these traces and makes a decision, and then the Trace Collector continues to collect traces for the Trace Analyzer for another round of decision. The decision process does not stop until two consecutive rounds of decision reach the same result. Empirically, we found that in

each round, collecting 1M block requests is enough, and normally two to four rounds of decision can reach the final decision.

3.2.2 Decision criterion. The Trace Analyzer requires a criterion to determine which policy is optimal. The hit ratio is the most widely used criterion (in LRU-k, 2Q, FBR, LRFU, LIRS, MQ, ARC, CAR, etc) and some others are also used, such as the hit density (in SANBoost). In SOPA, the total cost of different kinds of operations is used as the criterion to evaluate all the candidate caching policies:

$$C_{all} = C_{hit} \times N_{hit} + C_{migration} \times N_{migration} + C_{direct} \times N_{direct} \quad (1)$$

C is the average cost of an operation, and N is the number of the operation executed. The Trace Analyzer selects the policy with the minimum C_{all} .

For on-demand caches, since every missed block is migrated into the cache, N_{direct} is zero, and

$$\begin{aligned} C_{all} &= C_{hit} \times N_{hit} + C_{migration} \times (N_{all} - N_{hit}) \\ &= N_{all} \times [C_{hit} \times h + C_{migration} \times (1 - h)] \quad h = N_{hit} / N_{all} \end{aligned} \quad (2)$$

Since $C_{hit} \ll C_{migration}$ and N_{all} is a fixed number, the larger the h (hit ratio) is, the smaller the C_{all} is. So for on-demand caches, C_{all} is equivalent to the hit ratio. For caching policies not only based on hit ratio, such as SANBoost, N_{direct} is not zero, and C_{all} can evaluate these policies more precisely. C_{all} is compatible with the most widely used hit ratio criterion. It can also describe various cache environments and policies. So SOPA can be applied to a wide range of cache systems.

The values of C depend on the cache system. In the current version of SOPA, they are set manually. In [Chen et al. 2005], the latencies of different storage systems are measured and they can be used as the values of C. Some tools such as lmbench [Mcvoy and Staelin 1996] can measure the bandwidth and latency of the disks and memories, which can also help the setting of C. Improving the C_{all} expression (for example to distinguish read/write or sequential/random operations) and automatically collecting the required parameters are our future works.

3.2.3 Dropping less frequently used policies. The number of candidate policies will increase with the emergence of new caching policies, and this will add work to the trace analyzing process, so less frequently used policies should be dropped. In SOPA, the

usage ratio $\frac{\text{time as Active CP-Mod}}{\text{elapsed time since deployed}}$ is used to evaluate each policy. When the number of candidate policies goes beyond a threshold, SOPA drops the policy with the lowest usage ratio.

4. SIMULATION EVALUATION

We implemented a simulator for the evaluation. We simulated four kinds of representative workloads and observed whether SOPA could correctly select the optimal policy.

4.1 Traces and Caching Policies

The traces we used include two SPC-1 Financial traces (F1, F2), three SPC-1 Websearch traces (W1-W3) [SPC-1 traces], one Cello99 trace (C1) [Ruemmler and Wilkes 1993; HP traces], and one TPC-C trace (T1) [TPC-C trace].

SPC-1 Financial traces contain two I/O traces (F1, F2) from OLTP applications running at two large financial institutions. F1 contains 5.3 million references and accesses 17.2GB data in 12.1 hours. F2 contains 3.7 million references and accesses 8.4GB data in 11.4 hours.

SPC-1 Websearch traces contain three I/O traces (W1-W3) from a popular search engine. W1 contains 1.1 million references and accesses 15.2GB data in 0.88 hours. W2 contains 3 million references and accesses 43.0GB data in 2.5 hours. W3 contains 4.3 million references and accesses 62.6GB data in 83.0 hours (99.98% of them are in the first 6.3 hours).

The Cello99 trace is a low-level disk I/O trace collected on an HP UNIX server with 2GB memory [Chen et al. 2003]. We use one week trace (C1) between 12/01/1999 and 12/07/1999. C1 contains 43 million references and accesses 350.8GB data in a week.

TPC-C contains a disk trace (T1) of the TPC-C database benchmark with 20 warehouses. The client ran 20 iterations for Postgres 7.1.2. It was collected with DTB v1.1. T1 contains 10 million references and accesses 438.4GB data in 14.5 hours.

LRU, 2Q, MQ, ARC and LIRS were used in our evaluation, since they are all hit ratio based general-purpose policies and require no additional information from the cache system. LRU-k and LRFU were dropped because of their large overhead [Megiddo and Modha 2003]. Nine caching policies were selected as candidates, including the same policy with different parameters (cache contains c blocks): LRU, ARC, MQ (with 8 queues and $4c$ history records), $2Q_1$ (K_{in} is $0.2c$, K_{out} is $0.5c$), $2Q_2$ (K_{in} is $0.3c$, K_{out} is $0.5c$), $LIRS_1$ (L_{hir} is $0.01c$), $LIRS_2$ (L_{hir} is $0.1c$), $LIRS_3$ (L_{hir} is $0.2c$) and $LIRS_4$ (L_{hir} is

0.3c). These policies were all implemented according to their articles and modified to suit SOPA. The parameters of these policies were all suggested or used by the authors in these articles. In the evaluation, the initial policy of SOPA was ARC.

These evaluations ran on a machine with four Xeon 700MHz CPUs and block size is 4KB in all tests. Virtual time, which is the number of blocks requested, was used in this section to depict the variation of hit ratio in the process.

4.2 The Process of policy switching

First, we observed the fluctuation of the hit ratio in the process of policy switching. LRU policy was first used for trace W3, and it was switched to the new policy ARC at virtual time 5 ($\times 10^6$). The fluctuation of the hit ratio is shown in Figure 4.

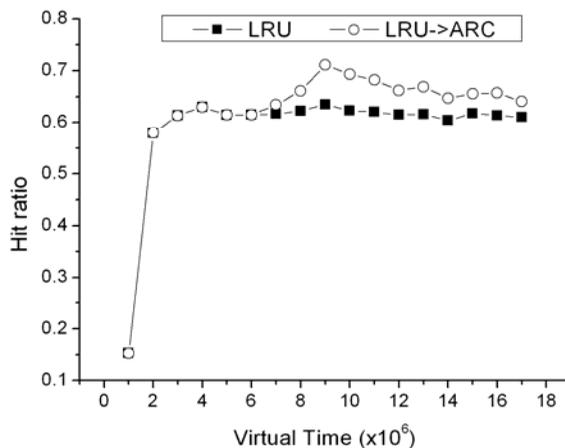


Fig.4. Fluctuation of the hit ratio on W3, switching from LRU to ARC at time 5. Cache size was 4G.

Figure 4 depicts the fluctuation of hit ratio with time. After the policy switching was performed at time 5, the hit ratio was retained between times 5 and 6. From time 7, the hit ratio of the new policy overcame that of the original LRU policy. This reveals that the old policy can be switched to a new one smoothly, without a performance vibration.

4.3 Simulation results

In this section, we simulated all the traces in the on-demand cache environment. The results of SOPA and other policies are compared in Table I. Since for on-demand caches, C_{all} is equivalent to the hit ratio, hit ratios are used in this table. The highest hit ratio achieved by a single policy was underlined.

Trace	LRU	ARC	MQ	2Q		LIRS				SOPA	
				0.2	0.3	0.01	0.1	0.2	0.3		Choice
SPC-1 Financial											
F1,4M	66.4	66.4	<u>67.3</u>	65	66.1	57.6	65.6	66.2	66.4	67.2	MQ
F1,8M	68.3	68.3	<u>68.9</u>	68.5	<u>69.1</u>	62.4	68	68.2	68.2	68.7	2Q(0.3)
F1,16M	70.1	70.2	<u>70.3</u>	71	<u>71.4</u>	67.1	70	70.1	70.1	71	2Q(0.3)
F1,32M	71.8	72	<u>72</u>	72.7	<u>72.9</u>	70.6	71.9	71.9	71.9	72.7	2Q(0.3)
F1,64M	73.2	73.3	<u>73.2</u>	73.8	<u>74</u>	72.6	73.2	73.3	73.3	73.3	LIRS(0.3)
F2,4M	53.5	54.2	<u>56.5</u>	56.1	56.1	49.6	53.4	54	54.1	56.1	MQ
F2,8M	61.9	62.3	<u>64.3</u>	<u>64.6</u>	64.5	59.1	61.9	62.1	62.2	64.1	2Q(0.3)
F2,16M	71.2	70.8	<u>72.3</u>	<u>73</u>	<u>73.2</u>	68.5	70.3	70.7	70.9	72.7	2Q(0.3)
F2,32M	78.2	77.7	<u>78.6</u>	<u>80.1</u>	<u>80.1</u>	76	77.5	78.1	78.3	79.5	2Q(0.2)
F2,64M	83.2	83.1	<u>83.3</u>	<u>84.6</u>	84.5	81.5	83.3	83.5	83.5	84.2	2Q(0.2)
SPC-1 Websearch											
W1,1G	0.3	0.7	5.4	1.3	1.3	<u>13.2</u>	12.2	11.1	10	10.5	LIRS(0.01)
W1,2G	11.2	20.6	17.9	19.9	18.7	<u>30.8</u>	28.8	26.5	24.1	27.5	LIRS(0.01)
W1,4G	48.9	50.7	49.9	44.2	44.2	<u>53.5</u>	51.7	50.6	50.1	51.6	LIRS(0.01)
W2,1G	0.4	2	12.2	4.2	4.2	<u>14.8</u>	13.7	12.5	11.1	13.7	LIRS(0.01)
W2,2G	12.5	30.2	27.3	29	25	<u>35.1</u>	32.9	30.5	27.7	33.8	LIRS(0.01)
W2,4G	57.5	61.6	61.2	57.9	56.9	<u>65.4</u>	62.9	61.4	60.9	64.6	LIRS(0.01)
W3,1G	0.4	2.9	<u>15.7</u>	5.8	5.8	15.2	14	12.7	11.2	14.5	LIRS(0.01)
W3,2G	11.9	31.5	31.5	30.8	26.2	<u>36.2</u>	33.9	31.3	28.3	35.4	LIRS(0.01)
W3,4G	58.7	62.8	64.6	60.3	58.8	<u>67.5</u>	65	63.3	62.7	67	LIRS(0.01)
Cello99											
C1,4M	34.8	36.5	36.9	<u>38.3</u>	38.2	25.5	36.3	36.3	36.2	38.2	2Q(0.2,0.3)
C1,8M	37.3	38.7	38.7	<u>40.2</u>	<u>40.2</u>	37.4	38.5	38.6	38.5	40.1	2Q(0.2,0.3)
C1,16M	40.4	41.3	41.3	<u>42.7</u>	<u>42.7</u>	40.4	40.9	40.9	41	42.5	2Q(0.3), LIRS(0.1), ARC
C1,32M	44.4	44.9	44.8	45.4	<u>45.6</u>	42.9	43.4	43.6	43.8	45.4	2Q(0.3), LIRS(0.2), ARC
C1,64M	48.8	49	49	49.4	<u>49.8</u>	45.9	46.6	47.1	47.4	49.2	ARC, 2Q(0.3)
C1,128M	53	53	53.1	53.7	<u>54.1</u>	49.2	50.4	51.1	51.7	53.3	MQ, 2Q(0.2)
C1,256M	56.2	56.2	56.2	57.2	<u>57.6</u>	52.4	54.2	55.3	55.7	56.2	MQ, LIRS(0.3)
C1,512M	59	59.2	59	60.4	<u>60.7</u>	55.9	58.3	59	59.2	59.1	LRU, LIRS(0.2)
C1,1G	61.6	61.8	61.6	63.5	<u>63.7</u>	59.3	61.8	62.1	62.2	61.7	LRU, ARC
TPC-C											
T1,128M	5.2	12.5	5.8	7.4	7.4	<u>19.6</u>	18.3	16.8	15.4	19.5	ARC, LIRS(0.01)
T1,256M	5.4	19.5	19.6	5.4	5.4	<u>34.3</u>	31.7	28.8	25.8	33.9	ARC, LIRS(0.01)
T1,512M	9.3	13.4	35.6	43.6	31.5	<u>63.6</u>	58.4	52.8	47.4	63.3	LIRS(0.01)
T1,1G	94	93.9	91.7	91.8	<u>96</u>	<u>96</u>	95.1	94.2	93.4	94.2	LIRS(0.2)
T1,2G	<u>99.6</u>	LRU									

Table 1 Hit ratio of different policies on different traces. Underlined numbers designate the highest hit ratio achieved by a single policy.

In the listed 19 cases of SPC-1 Financial and SPC-1 Websearch, where only one decision was needed, SOPA selected the optimal policy in 16 cases and selected the second optimal policy in 2 cases. In the Cello99 and TPC-C cases, where two or more decisions were needed, the hit ratios of SOPA were also close to the optimal ones.

This test shows that SOPA could correctly select the policy among the candidate ones. Conversely, no single policy could behave satisfactorily on all the traces. In our experiments, LIRS performs best on Websearch and TPC-C traces, 2Q performs best on Cello and Financial traces, but they cannot perform equally well on all the traces and parameter tuning is also required.

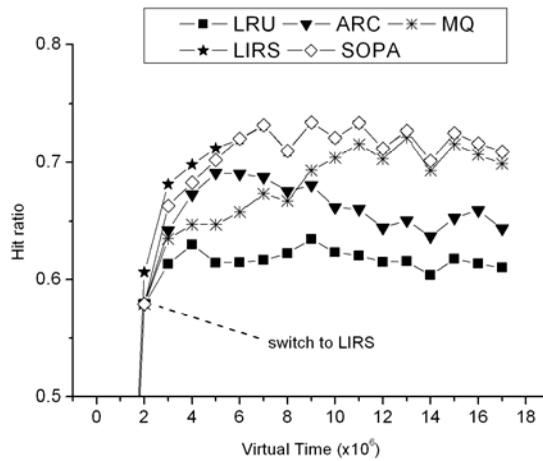


Fig.5 The variation of hit ratio with time on (W3, 4G)

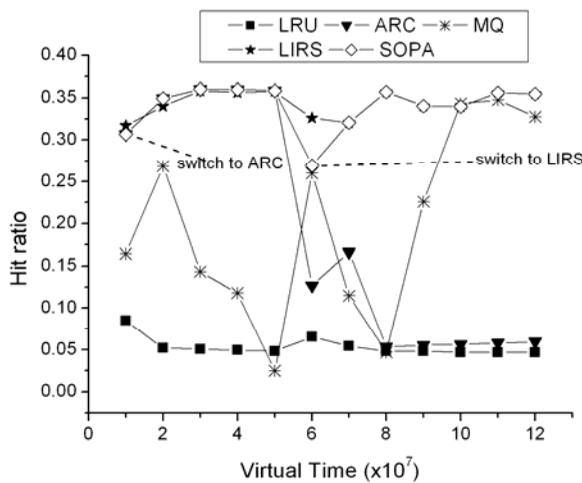


Fig.6 The variation of hit ratio with time on (T1, 256M)

Figure 5 takes a detailed view of one trace W3. After the decision at time 2, SOPA selected LIRS and performed the policy switch. Then its hit ratio went beyond those of LRU, ARC and MQ.

Figure 6 depicts the variation of hit ratio on T1. T1 is a trace with varying characteristic that more decisions were needed. SOPA selected ARC in the first decision, but after time 5, the hit ratio dropped and SOPA started a new decision. This time, SOPA selected LIRS and went beyond other policies.

5 REAL SYSTEM EVALUATION

We implemented a write through cache system for an iSCSI target [unh project]. SOPA was applied to this system and compared with two adaptive policies ARC and MQ. The target machine was equipped with Xeon 2.8GHz CPU, 4GB memory and 6 Maxtor MaxLine Plus II 250GB SATA disks. The initiator machine was equipped with Xeon 2.4GHz CPU and 1GB memory. The target machines installed Linux 2.4.31 and the initiator machine installed Windows 2003 and Microsoft iSCSI Initiator. Two machines were connected through Gigabyte Ethernet. The cache block size was 4KB in all experiments. SPC-1 Websearch and TPC-C traces were used in this evaluation. SPC-1 Financial and Cello99 traces were not used since they require too many disks (more than 20). Traces were played on the initiator machine through direct I/Os.

5.1 Results on Websearch Workload

In this experiment, trace W3 was played on the initiator machine. The cache size was set to 1G, 2G and 3G. Since the total length of W3 is 63 hours and 99.98% of the requests are in the first 6.3 hours, the last 0.02% requests are dropped to save time. The average response time were shown in Table II.

	LRU	ARC	MQ	SOPA
W3, 1G	9.5	9.4	8.5	8.3
W3, 2G	8.3	6.9	6.6	6.6
W3, 3G	6.1	5.4	5.4	5.0

Table II Average response time (ms) of different policies on W3.

The evaluation results showed that on W3, SOPA could reduce the average response time by up to 20.3% compared with LRU, up to 11.8% compared with ARC, and up to 6.7% compared with MQ.

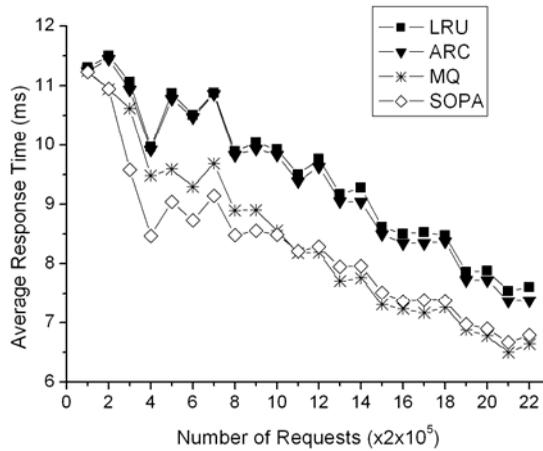
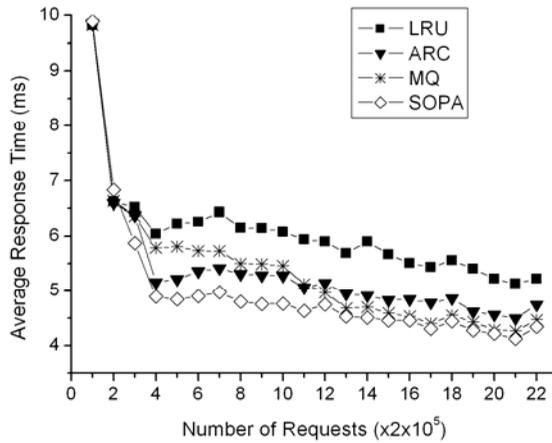


Fig.7 The variation of average response time on (W3, 3G) and (W3, 1G). Each point is the average response time of 200,000 requests. For example, point 2 is the average response time of the 400,001st to the 600,000th requests.

Figure 7 depicted the variation of average response time in the whole process. In each figure, SOPA made the decision at the first and second point. It can be observed that at the first two points, the average response time of different policies were almost the same, since warm-up process had not finished. SOPA could successfully reduce the average response time after selecting an appropriate policy.

5.2 TPC-C Workload

In this experiment, trace T1 was played on the initiator machine. The cache size was set to 256M and 512M. The average response time were shown in Table III.

	LRU	ARC	MQ	SOPA
T1, 256M	6.5	6.3	6.5	6.1
T1, 512M	6.4	6.3	6.2	5.5

Table III Average response time (ms) of different policies on T1.

The evaluation results showed that on T1, SOPA could reduce the average response time by up to 13.6% compared with LRU, up to 11.9% compared with ARC, and up to 10.5% compared with MQ.

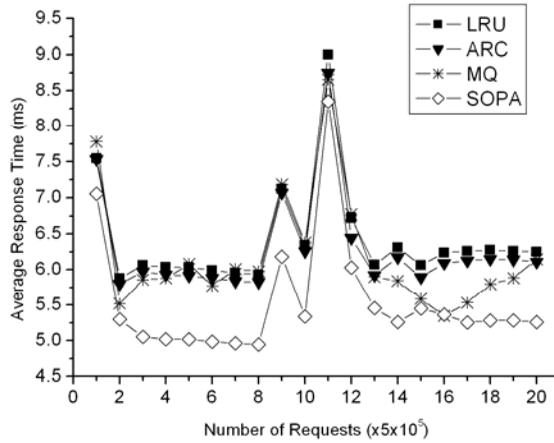


Fig.8 The variation of average response time on (T1, 512M). Each point is the average response time of 500,000 requests.

Figure 8 depicted the variation of average response time on T1. Similarly, after the policy decision at the first point, SOPA could successfully reduce the average response time.

In these two evaluations, LIRS were selected as the optimal policy. In fact, LIRS “will work well for stable workloads drawn according to the IRM, but not for those LRU-friendly workloads drawn according to the SDD” [Megiddo and Modha 2003], and our simulation also showed that LIRS performed well on Websearch and TPC-C traces, but not on Financial and Cello99 traces. Thus, it was not a policy that can be adaptive to many workloads. But SOPA could overcome this drawback, since SOPA only selected LIRS when it was “good” in the decision process.

These evaluations showed that by automatically choosing and switching among multiple policies, SOPA could adapt to different workloads and achieve lower average response time.

5.3 Overhead of Trace Collecting and Analyzing

In this section, we presented the overhead of the trace collecting and analyzing process.

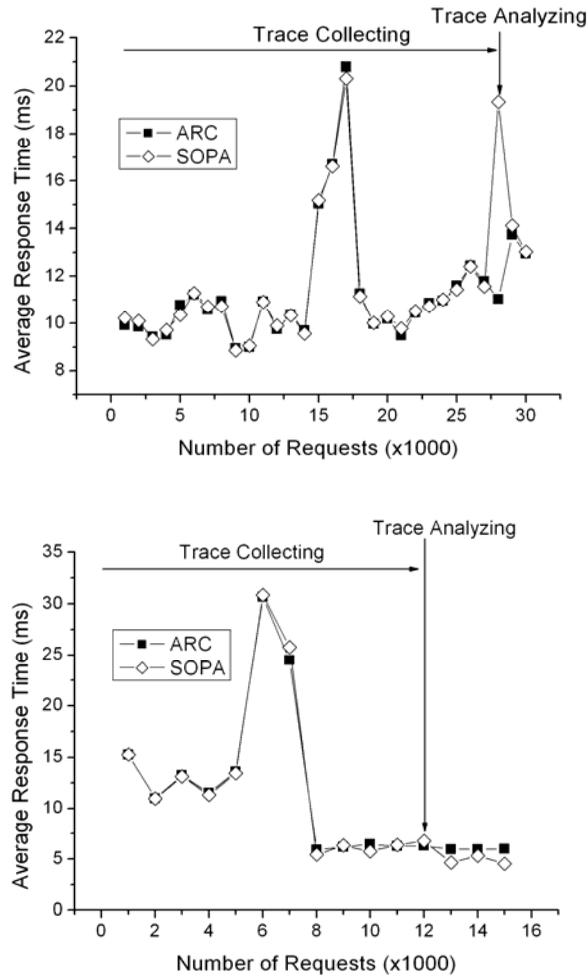


Fig.9 The variation of average response time on (W3, 1G) and (T1, 256M). Each point is the average response time of 1000 requests.

Figure 9 depicted the overhead of the first round of the decision process on W3 and T1, where in Figure 7 and 8, this process was compressed into several points. Since the initial policy of SOPA was ARC, so the average response time of SOPA and ARC at the same period of time were compared. In Figure 9, it can be observed that the overhead of trace collecting was almost negligible. The trace analyzing was a CPU-intensive work. On T1, where the workload was light at the time of trace analyzing, the overhead of trace analyzing was also negligible. On W3, where the workload was heavier, this process introduced a large average response time at point 28. However, the time for trace

analyzing was very short, thus even a large average response time at a single point had little influence on the overall performance. Compared with the benefit gained by the decision-making, this overhead was worthwhile.

6. CONCLUSIONS AND FUTURE WORKS

This paper presents SOPA to select the optimal policy adaptively for a cache system. On the one hand, it encapsulates the functions of a caching policy into a module, and by policy reconstruction, it allows dynamic switching between any two policies and the online deployment of new caching policies. On the other hand, it can select the optimal caching policy on a varying workload. SOPA also reduces the overhead of the policy decision-making by asynchronous decision.

The simulation evaluation showed that no single caching policy could perform well under all of the different workloads. By choosing and switching among multiple policies, SOPA could achieve hit ratios close to the optimal policies. The real system evaluation on an IPSAN system showed that SOPA can reduce the average response time by up to 20.3% compared with LRU and up to 11.9% compared with ARC.

One of our future works is to build a decision-making system based on experience. The Trace Analyzer stores the past decision results as experience and when a new workload comes, it can make a decision based on its experience. This can shorten the decision-making time needed and reduce the overhead of the trace collecting and analyzing.

ACKNOWLEDGEMENT

We are grateful to Ken Bates and Bruce McNutt for providing the SPC-1 traces, to Hewlett-Packard Company (HP) for providing the Cello99 traces, to Brigham Young University for providing the TPC-C trace, and to Bing Yu for providing the earlier prototype.

References:

- ARI, I., AMER, A., GRAMARCY, R., MILLER, E., BRANDT, S., AND LONG, D. 2002. ACME: Adaptive caching using multiple experts. In Proceedings of the Workshop on Distributed Data and Structures, Carleton Scientific, 2002.
- ARI, I., GOTTWALS, M., AND HENZE, D. 2004. SANBoost: Automated SAN-Level Caching in Storage Area Networks, In Proceedings of the International Conference on Autonomic Computing (ICAC '04), May 2004.
- BANSAL, S. AND MODHA, D. S. 2004. CAR: Clock with adaptive replacement. In Proceedings of the USENIX File and Storage Technologies Conference (FAST '04), 2004, 142–163.
- CAO, P., AND IRANI, S. 1997. Cost-aware WWW proxy caching algorithms. in Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, 1997.
- CHEN, Z., ZHOU, Y., AND LI, K. 2003. Eviction-based Cache Placement for Storage Caches. In Proceedings of the USENIX Annual Technical Conference, General Track 2003, Jun. 2003, 269-281.

- CHEN, Z., ZHANG, Y., ZHOU, Y., SCOTT, H., AND SCHIEFER, B. 2005. Empirical. Evaluation of Multi-Level Buffer Cache Collaboration for Storage. System. In Proceedings of the 2005 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS. '05), 2005.
- CHOU, H.T., AND DEWITT, D.J. 1985. An Evaluation of Buffer Management Strategies for Relational Database Systems. In Proceedings of the VLDB Conference, 1985.
- GILL, B.S., AND MODHA, D.S. 2005. WOW: Wise Ordering for Writes - Combining Spatial and Temporal Locality in Non-Volatile Caches. In Proceedings of the USENIX Conference on File and Storage Technologies (FAST '05), San Francisco, CA, December 2005.
- HP TRACES. http://tesla.hpl.hp.com/public_software/
- JIANG, S., AND ZHANG, X. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, June 2002, 31–42.
- JIANG, S., DING, X., CHEN, F., TAN, E., AND ZHANG, X. 2005. DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Localities. In Proceedings of the USENIX Conference on File and Storage Technologies (FAST '05), San Francisco, CA, December 2005.
- JOHNSON, T., AND SHASHA, D. 1994. 2Q: A low overhead high performance buffer management replacement algorithm.” In Proceedings of the VLDB Conference. 1994. 297–306.
- LEE, D., CHOI, J., KIM, J.H., NOH, S.H., MIM, S. L., CHO, Y., AND KIM, C. S. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. IEEE Transactions on Computers, vol. 50, no. 12, pp. 1352–1360.
- LI, X., ABOULNAGA, A., SACHEDINA, A., SALEM, K., AND GAO, S.B 2005. Second-tier cache management using write hints. In Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05), 115-128, December 2005.
- MCVOY, L., AND STAELIN, C. 1996. Imbench: Portable tools for performance analysis. In Proceedings of the 1996 USENIX Technical Conference. San Diego, CA, January 1996, 279-295.
- MEGIDDO, N., AND MODHA, D.S. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In Proceedings of the USENIX File and Storage Technologies Conference (FAST '03), March 2003, 115-130.
- MENON, J. 1994. Performance of RAID5 disk arrays with read and write caching. Distributed and Parallel Databases, vol. 2, no. 3, pp. 261–293.
- MENON, J. AND HARTUNG, M. 1988. The IBM 3990 disk cache. In Proceedings of the IEEE Computer Society International COMPCON Conference, 1988.
- NELSON, M. N., WELCH, B. B., AND OUSTERHOUT, J. K. 1998. Caching in the Sprite network file system. ACM Transactions on Computer Systems, vol. 6, no. 1, pp. 134–154.
- O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. 1993. The LRU-K page replacement algorithm for database disk buffering. In Proceedings of International Conference on Management of Data, Washington, DC, USA, 1993, 297~306
- ROBINSON, J.T., AND DEVARAKONDA, M. V. 1990. Data cache management using frequency-based replacement. In Proceedings of SIGMETRIC on Measuring and Modeling of Computer Systems, Boulder, Colorado, USA, 1990, 134~142
- RUEMMLER, C., AND WILKES, J. 1993. A trace-driven analysis of disk working set sizes. Technical Report HPL{OSR}{93}{23, Hewlett-Packard Laboratories, Palo Alto, CA, USA, April 5, 1993.
- SALMON, B., THERESKA, E., SOULES, C.A.N, AND GANGER, G.R. 2003. A two-tiered software architecture for automated tuning of disk layouts. In Workshop on Algorithms and Architectures for Self-Managing Systems. San Diego, CA, June 11, 2003.
- SPC-1 TRACES. <http://traces.cs.umass.edu/index.php/Storage/Storage>
- TPC-C TRACE.
<http://tds.cs.byu.edu/tds/tracelist.jsp?searchby=attribute&type=Disk+I%2FO&length=All&platform=All&cache=All&pageNum=0&searchAction=Go&x=52&y=19>
- UNH PROJECT. <http://unh-iscsi.sourceforge.net/>
- YADGAR, G, AND FACTOR, M. 2007. Karma: Know-it-All Replacement for a Multilevel cAche. in In Proceedings of the USENIX File and Storage Technologies Conference (FAST '07), 2007
- ZHOU, Y., AND PHILBIN, J.F. 2001. The multi-queue replacement algorithm for second level buffer caches. In Proceedings of the 2001. USENIX Annual Technical Conference, Boston, MA, June 2001, 91–104.