ACME-DB: AN ADAPTIVE CACHING MECHANISM USING MULTIPLE EXPERTS FOR DATABASE BUFFERS

Faizal Riaz-ud-Din, Markus Kirchberg
Information Science Research Centre, Department of Information Systems, Massey University, Private Bag 11-222, Palmerston North, New Zealand.

Keywords: Database caching, adaptive caching, buffer management

Abstract: An adaptive caching algorithm, known as Adaptive Caching with Multiple Experts (ACME), has recently been presented in the field of web-caching. We explore the migration of ACME to the database caching environment. By integrating recently proposed database replacement policies into ACME's existing policy pool, an attempt is made to gauge ACME's ability to utilise newer methods of database caching. The results suggest that ACME is indeed well-suited to the database environment and performs as well as the best currently caching policy within its policy pool at any particular moment in its request stream. Although execution time increases by integrating more policies into ACME, the overall processing time improves drastically with erratic patterns of access, when compared to static policies.

1 INTRODUCTION

One of the key factors affecting a database's performance is its ability to effectively and efficiently cache frequently requested data. Main memory access is approximately 170,000 times faster than disk accesses (Ramakrishnan and Gehrke, 2000). The challenge in main memory caching is to determine which page of data to replace (the 'victim') from the buffer pool once it is full, to make space for future data pages to be read in from secondary storage. The selection of victims is performed by a cache replacement algorithm, which chooses victims according to certain criteria based on past access patterns.

This paper attempts to trial and evaluate a recently presented adaptive cache replacement algorithm, known as ACME, or Adaptive Caching with Multiple Experts (Ari et al., 2002). ACME uses machine learning to dynamically select the cache victim from a number of policies, or experts, during the cache replacement process. Whilst ACME was originally presented for the web-caching environment, this paper describes the evaluation of the adaptation of ACME to the database environment. The rigidity and robustness of ACME is also tested by its ability to integrate more complex policies in its policy pool, including optimizations achieved in terms of overall processing time.

The remainder of this paper is presented as follows: Section 2 presents related work, including adaptive caching algorithms, with a focus on ACME. Sections 3 and 4 respectively, detail the way in which ACME has been adapted to the database environment and the issues concerned with implementation. Section 5 analyses the results obtained from driving the database-adapted ACME (referred to as ACME-DB in this paper) with live and synthetic traces. Finally, section 6 presents conclusions.

2 RELATED WORK

The main task of a buffer manager is to retrieve the data from secondary storage into main memory, thus allowing the data in the main memory to be accessed by the transactions that request information from the database. There are two purposes for accessing data in this manner: firstly it ensures that subsequent

Riaz-ud-Din F. and Kirchberg M. (2004).
ACME-DB: AN ADAPTIVE CACHING MECHANISM USING MULTIPLE EXPERTS FOR DATABASE BUFFERS.
In *Proceedings of the Sixth International Conference on Enterprise Information Systems*, pages 192-201
DOI: 10.5220/0002598101920201

accesses to the same data are much faster in future references (since they are now in the main memory and do not need to be accessed from secondary storage again), and secondly, it ensures that the data is presented to the database in a synchronous manner, resulting in data consistency. Any data in the main memory that has been changed by a transaction is written back to where it was retrieved from on the secondary storage device.

This process of using the main memory area as an efficient data delivery mechanism is known as caching, whilst the main memory cache is also known as the buffer pool. A cache replacement algorithm is often referred to as a buffer replacement policy, and its function is to select a victim from the buffer pool.

Ideally, secondary storage devices would have I/O speeds at least as fast as main memory. However, because there is a latency issue involved with reading from secondary storage (Sacco and Schkolnick, 1986), and main memory is far more costly than secondary storage, the need to find an optimal practical buffer replacement policy still exists.

A number of buffer replacement policies have been presented in the literature in the quest for that optimal replacement policy. The more well-known amongst them have been FIFO (First-In-First-Out), LIFO (Last-In-First-Out), MRU (Most Recently Used), LRU (Least Recently Used), and LFU (Least Frequently Used), as well as others (Effelsberg and Haerder, 1984).

More recently, LRFU (Least Recently / Frequently Used), has been presented, which is a policy that attempts to combine the benefits of LRU and LFU (Lee et al., 1999). Whilst LRU-K (O'Neil et al., 1993) has succeeded in achieving a higher hit-rate than LRU, implementation is more difficult and execution time is higher. Even more recently, policies such as LIRS (Low Inter-reference Recency Set) (Jiang and Zhang, 2002), 2Q (Johnson and Shasha, 1994), and W^2R (Weighing Room / Waiting Room) (Jeon and Noh, 1998) have been presented. Whilst these recent policies suggest a marked improvement over their predecessors, implementation is not as simple as for LRU.

Adaptive caching algorithms go one step further by using two or more policies, in the attempt to reap their benefits, whilst avoiding their pitfalls (Castro et al., 1997).

Hybrid Adaptive Caching (HAC) is an adaptive caching algorithm presented by Castro et al. (Castro et al., 1997) within the environment of distributed objects. It combines the benefits of both page and object caching, whilst at the same avoiding their respective disadvantages. It behaves like a page caching system when spatial locality is good, and when spatial locality degrades, it behaves like an object caching system, thereby adapting to the different access patterns in the request stream.

Adaptive Caching with Multiple Experts (ACME) is another adaptive caching algorithm presented recently by Ari et al. (Ari et al., 2002) within the context of the web-caching environment. ACME makes use of a policy pool that consists of a number of different static replacement policies ('experts'). Each of these experts has its own cache (known as a Virtual Cache). The request stream is processed by each of the experts and each performs its own caching independent of the others. When the actual cache that holds the data (the Real or Physical Cache) is full, and a page needs to be replaced, the experts are queried to see if they would have held the requested page in their virtual caches. If so, they are awarded a vote, otherwise they are penalised. The votes that each expert gains over time are used to determine its probability in choosing the next replacement victim.

ACME also possesses a machine-learning algorithm and it 'learns' to use the current best policy over time, based on the past successes and failures of the individual experts in the policy pool (Ari, 2002). The architecture of ACME is illustrated in Figure 1 below for clarity.

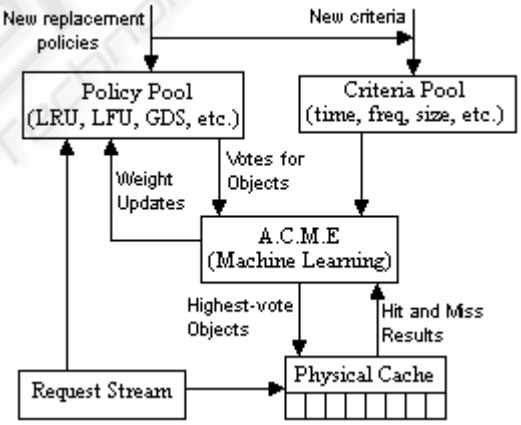

Figure 1: ACME architecture (Ari, 2002)

ACME is used as the basis of this research because it readily allows the incorporation of other policies within its architecture. Not only does this mean that ACME has the ability to use many different criteria when selecting a victim, but importantly, it provides the flexibility required to adapt it to the database environment (Ari et al., 2002). However, it should be noted that adding more policies increases the memory usage, and increases processing time. For detailed information on ACME, please refer to (Ari et al., 2002).

3 ADAPTING ACME FOR DATABASE CACHING

ACME was originally designed for use within a web-cache and as such, it includes certain features not necessarily required in a database-caching environment. With regard to the target database cache simulation in mind, ACME needed to be modified (Riaz-ud-Din, 2003).

3.1 Required modifications

The original implementation of ACME allows for objects of different sizes to be cached. This ability is removed with the assumption that the pages that exist in memory are of the same size, thereby removing the need to correlate the sizes of the pages and freeing buffer space. This makes it easier to predict the number of I/O operations on the buffer pool.

The other modification required was to remove those policies from the ACME buffer pool that used the sizes of the objects to determine the object's priority in the cache. Since the size of all objects is the same in the database environment, these policies are redundant in that one of the criteria (the object size) used to determine the object's priority is no longer an issue. New policies have been added to the buffer pool, with an emphasis on using policies applicable to the implemented cache simulation.

3.2 Additional demands

The additional demands of adapting ACME to the database environment include adding policies to the policy pool which are intrinsically dissimilar to the policies in the original ACME policy pool. The original ACME policies are based on single priority queues, and objects are assigned priorities based on pre-defined functions that use parameters such as time, size, and so on. However, the new policies that have been added to the policy pool for ACME-DB include policies that use more than one priority queue or divide the main buffer pool into a number of buffers. Using such policies requires more sophisticated buffering operations for caching and uncaching, and requires careful considerations to be made before integrating them into the ACME-DB buffer pool.

In addition, release mechanisms now need to be defined for each policy using more than a single queue or stack to manipulate the buffer pool. The literature that details the multi-buffer policies that have been added to the policy pool only describe the caching and uncaching operations of the respective policies. Consequently, the release mechanisms (described in Sub-Section 4.2) on arbitrary pages needed to be designed based on the heuristic analysis of the intrinsic behaviour of these policies.

4 IMPLEMENTATION

This section discusses issues related to implementing the additional database policies within the framework of the original ACME architecture, after applying the necessary adaptations as described in the previous section.

4.1 Release Mechanisms

In addition to being able to cache and uncache pages, each of the policies also needs to know how to release pages arbitrarily when required. The release functionality is invoked by each policy when one of the policies has selected a replacement victim to expel from the Real Cache. At this point the other policies would also need to expel the reference to the replacement victim from their physical caches as well. However, the replacement victim is not necessarily the same page that would be selected by each of the other policies' uncaching rules.

This process of releasing an arbitrary page from the physical cache of each policy requires a release mechanism that discards the selected page from cache. Discarding a page with the original ACME required removing a page from a priority queue. However, in the ACME-DB implementation, the release mechanisms are more complex since they affect more than one queue or stack for the new policies that have been added to the policy pool.

The release mechanisms have been determined based on the way in which the buffers are used by each of the policies concerned. In the same way that the caching and uncaching functionality needs to be separated from each other for the purposes of being able to integrate them into the overall ACME architecture, the release mechanism for each policy needs to be defined separately so that it can be invoked independent of caching and uncaching.

4.2 Choice of policies

Random, FIFO, LIFO, LRU, LFU, LFUDA, MFU, and MRU were the only policies that were re-used from the original ACME implementation, and their release mechanisms simply evict victim pages from a priority queue. Six new policies were also added to the policy pool. These new policies are aimed more

specifically to work in a database-specific environment, and have more complex behaviour that the other existing policies in the policy pool. The respective release mechanisms for each of the six policies that have been added as part of ACME-DB will now be described.

LIRS. This policy is described in (Jiang and Zhang, 2002). In the case of uncaching a page from the buffer, the page at the front of the List Q queue is ejected, creating space for a new page in the buffer pool. However, when releasing the page arbitrarily some other factors need to be taken into account.

(i) In the case that the page to be released exists in List Q, that is, the page is a resident HIR page, the page is released from List Q (wherever in the queue it may be). This case is identical to the case of uncaching, except that instead of replacing the page at the front of the queue, any page in the queue could be potentially replaced.

(ii) In the case that the page to be replaced exists in the LIR page set, that is, the page is an LIR page, the page is released from the LIR page set. This creates a space in the LIR page set, which needs to be filled in before normal caching / uncaching processes can proceed on the buffers. The space in the LIR page set is filled by ejecting the resident HIR page at the tail of List Q, and pushed onto the top of the LIR page set (the implementation employs a LIR page set to hold LIR pages). If this page was not in Stack S, it is pushed onto Stack S, and flagged as a resident LIR page.

The release mechanism is designed in this manner because it is presumed that the HIR page at the tail of List Q is the HIR page that had the least recency out of all the other resident HIR pages. In releasing one of the LIR pages from the LIR page set, it is considered that the HIR page in List Q with the least recency should be added to the LIR page set to fill the space left by the released page. Due to the fact that a resident HIR page is therefore turned into an LIR page, it needs to be added to Stack S, if it is not already there, and flagged as a resident LIR page. (Note: All pages are flagged as either resident or non-resident in the implementation).

LRFU. This policy is described in (Lee et al., 1999). The release mechanism in this case simply removes the page from the priority queue.

LRU-K. This policy is described in (O'Neil et al., 1993). The release mechanism in this case simply removes the page from the priority queue.

SFIFO. This policy is described in (Turner and Levy, 1981). The release mechanism in this policy checks the primary buffer for the page to be released, and releases it from there if found. If not, then the secondary buffer is checked for the page to be released, and is released from there. The release mechanism design in this policy reflects the fact that pages are not cached to the secondary buffer until the primary buffer is full, thereby enabling the arbitrary removal of pages from either buffer.

2Q. This policy is described in (Johnson and Shasha, 1994). The release mechanism in this policy checks to see whether the page to be released exists in the Am buffer, and releases it from there if found. If not, then the page to be released is searched for in the A1in buffer, and released from there if found. The release mechanism was designed in this manner so that the sizes of each of the A1in and A1out buffers are checked when uncaching occurs, thereby enabling the arbitrary release of a page from either the Am or A1in buffer.

W^2R. This policy is described in (Jeon and Noh, 1998). The release mechanism in this policy checks the Weighing Room buffer for the page to be released, and releases it from there if found. If not, the Waiting Room buffer is checked for the page to be released, and is released from there. The Weighing Room is implemented as a simple LRU queue, and the Waiting Room as a FIFO queue, enabling the simple arbitrary removal of a page from either queue, without needing any extra queue adjustments.

5 EVALUATION

This section describes the evaluation environment for which ACME-DB was implemented, the methodology used to design the experiments, the results of those experiments, and an analysis of the results.

5.1 Evaluation environment and traces

The ACME-DB simulation was implemented using C++, compiled and tested on Microsoft Windows XP, Linux RedHat 6.2 and Linux Debian. The execution time tests were performed on a Pentium IV 2GHz PC with 256 MB of RAM.

Two traces were used to simulate request streams. These traces are the DB2 and OLTP (On-Line Transaction Processing) traces used by Jeon and Noh (Jeon and Noh, 1998), Johnson and Sasha (Johnson and Shasha, 1994) and by O'Neil et al. (O'Neil et al., 1993). The DB2 trace was originally obtained by running a DB2 commercial application and contains 500,000 page requests to 75,514 distinct pages. The OLTP trace contains records of page requests to a CODASYL database for a

window of one hour. It contains 914,145 requests to 186,880 distinct pages.

Further to these two live traces, five synthetic traces were also created to simulate well known susceptibilities in replacement policies, and to simulate request streams that would not favour any particular policy in the policy pool.

5.2 Experimental methodology

This sub-section provides details on the experiments used to test and evaluate ACME-DB. Unless otherwise stated, all the experiments that tested hit rates in relation to the request stream were based on a cache size of 1000 pages. This cache size was chosen in an attempt to make comparisons with other experiments that used cache sizes of 1000 as well.

5.2.1 Combined effect of all policies

This experiment involved running ACME-DB across the two live traces with all fourteen policies enabled. This was to determine the combined effect of all the policies on the Real Cache hit rate, and to show the hit rates achieved by the virtual caches of the individual policies.

5.2.2 Real Cache adaptation to the current best policy

The second part of the experiment involved running ACME-DB with the best policy and an average-performing policy to determine the extent to which the Real Cache adapted to the best policy in terms of hit rate. The purpose of this test was to examine the switching of the current best policy and how this would affect the Real Cache in terms of hit rate.

5.2.3 The adaptive nature of ACME-DB

This experiment involved running ACME-DB with Synthetic Trace A to determine the behaviour of ACME when one of the policies switched from one performance extreme to the other and to illustrate how the presence of another policy in the policy pool has a stabilising effect during the performance degradation of the first policy.

5.2.4 Different cache sizes

ACME-DB was tested with all of its policies in the policy pool using cache sizes of between 100 and 10000 pages. The objective of this test was to determine the effect of the cache size on the cache hit rate.

5.2.5 Average time loss for each additional policy

It is well known that the different replacement policies used in ACME-DB vary in performance with regard to time complexity. It is possible that the addition of certain policies to the policy pool could increase the overall time complexity of ACME-DB, and thus detract from the benefits of using policies that perform well in terms of hit rate.

5.2.6 Investigation of susceptibilities to well-known weaknesses

Replacement policies were run using synthetic traces designed to expose the commonly known weaknesses of particular policies. This aimed to expose the susceptibility of other policies in the policy pool, and observe the effect on the Real Cache.

5.2.7 The effect of disk reads on the total execution time

This experiment gauged the effect of disk reads on total execution time, and whether the ability of the Real Cache to adapt to the current best policy would result in improved or poorer performance. The results of this test would provide the best value practically, since disk accesses are taken into account.

5.3 Simulation results

This sub-section presents the main results and corresponding analyses. For the purposes of this paper, in most cases, only the results for the DB2 trace have been shown. For more details, please refer to (Riaz-ud-Din, 2003).

5.3.1 Relative performance of policies.

Figure 2 below shows the hit rates with all policies in the policy pool using the methodology given in Sub-Section 5.2.1.

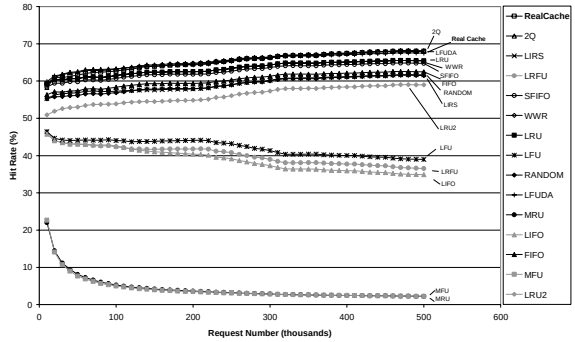

Figure 2: Requests vs. Hit rates

The graph above clearly defines three distinct groups of policies that are poor, average, and good in their relative performance. This reflects the relative strengths and weaknesses inherent in the heuristics that have been used to drive the victim selection processes of these policies. MRU and MFU performed poorly whilst 2Q and LFUDA performed particularly well.

5.3.2 The performance of the Real Cache

In order to test the desired adaptive behaviour (that is, the Real Cache always stays closer to the best policy), only 2Q and FIFO were used in the policy pool (see Sub-Section 5.2.2 for methodology).

Experiments in (Riaz-ud-Din, 2003) have shown that even when only two policies are in the policy pool, the Real Cache attempts to stay with the best performing (in terms of its cache hit rate) expert. Here the Real Cache is using the 2Q policy to do the majority of its caching, so its hit rates are similar to 2Q's hit rates. So if 2Q performs so well, is there any benefit in having other policies in the policy pool? Do the policies act synergistically to improve the Real Cache hit rate?

Results from tests performed in (Riaz-ud-Din, 2003) show no significant differences between the Real Cache hit rates where only 2Q and LFUDA are used, and where all the policies are used. Thus, having a large number of policies in the policy pool seems to neither help nor hinder the Real Cache hit rate. However, this would affect execution time and memory, which will be examined later in this paper.

Tests in (Riaz-ud-Din, 2003) also show that the Real Cache can only perform as well as the current best performing policy, which may not always have high hit rates. This shows the importance of finding the best mix of the minimum number of policies for inclusion in the policy pool.

5.3.3 The adaptive behaviour of ACME

Whilst 2Q has so far performed the best for the live requests, there may be other request patterns for which it may not do as well (refer figure 7). The methodology given in Sub-Section 5.2.3 is used to highlight this fact and Figure 3 below displays the results.

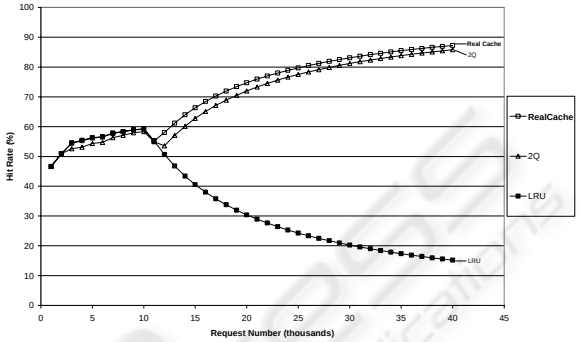

Figure 3: Request vs. Hit rate (2Q and LRU only)

It illustrates how LRU starts off with a higher hit rate than 2Q and at this point in time, the Real Cache adapts to LRU's caching behaviour. However, when the processing reaches the part of the trace that inhibits the performance of LRU, 2Q continues to climb the 'hit rate chart', while LRU's performance starts to degrade significantly. At this point, the Real Cache does not drop with LRU, but rather adapts to the new best policy (2Q) to clearly illustrate ACME's adaptive nature.

This is an important point because it indicates that having other policies should result in a 'safer' policy pool. Therefore, if at any particular moment, the performance of the current best policy degrades due to a known or un-known weakness, other policies in the policy pool would ensure that the Real Cache hit rate would not degrade as well, or at least not to the same extent.

In all of the relevant figures, it can be seen that the Real Cache hit rate starts with individual policy hit rates from one point and then starts to converge on the best policy as the number of requests increases. As more requests are made, the machine learning adjusts the weights until the best performing policies have a greater probability of being selected for performing caching and uncaching actions on the Real Cache, which then starts resembling the current best policy with respect to its hit rate.

5.3.4 The effect of having different cache sizes

Live traces were driven using different cache sizes to show the effect of cache size on the Real Cache with Figure 4 below showing hit rate versus the cache size for the DB2 trace.

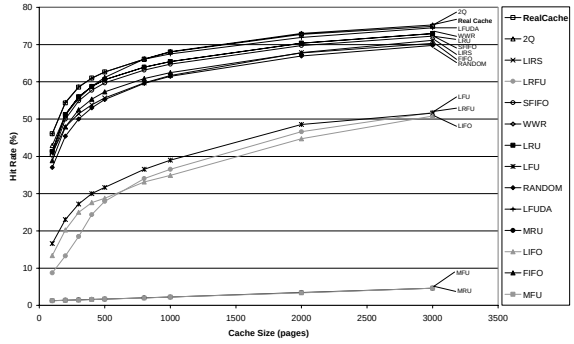

Figure 4: Cache Size vs. Hit rate, DB2 Trace

The effect of cache size on the Real Cache hit rate is summarised for each of the live traces in Table 1 below.

Table 1: Real Cache hit rates for different cache sizes

Cache Size	Real Cache Hit Rates (%)			
	DB2		*OLTP*	
	Value	% Increase	Value	% Increase
100	39.5	--	7.1	--
200	47.8	20.9	13.9	94.9
300	52.2	9.3	17.1	23.0
400	55.2	5.7	21.7	27.5
500	57.1	3.5	24.9	14.6
800	60.5	5.9	33.7	35.2
1,000	61.6	1.8	33.9	0.7
2,000	67.0	8.7	40.0	17.9
3,000	69.1	3.3	45.8	14.5
5,000	71.11	2.9	50.1	9.5
10,000	73.27	3.0	55.3	10.4

The choice of cache sizes shown above were determined by the cache sizes that have been used to test the new policies that have been added to the policy pool, in their respective papers. Doing so provides a basis for comparison with the other policies using similar cache sizes.

Figure 4 cumulatively shows that increasing the cache size also increases the individual policies' hit rates, and therefore the Real Cache hit rate (since the Real Cache follows the best policy at any time regardless of cache size). However, the major sacrifice for achieving better hit rates as a result of larger cache sizes is that the time required to process the requests also increases (since there are more pages to search through). This is largely due to the fact that processing time is positively correlated to the number of policies in the policy pool (as each of the virtual caches also perform their own caching).

5.3.5 Effect of replacement policies on time performance

Just as replacement policies vary in performance with regard to hit rates, so too do they vary with regard to their time complexities. Figure 5 below illustrates this point by showing the additional time each policy adds to the total time it takes for ACME-DB to process a request stream of 50,000 requests as described in the methodology outlined in Sub-Section 5.2.5.

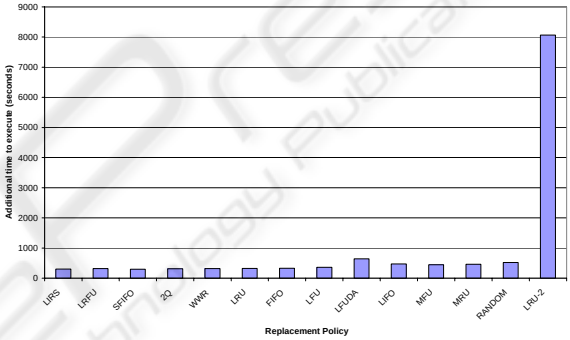

Figure 5: The increase in time by adding each policy

In the above figure, all policies (except LRU-2) increase the total time for ACME to process the request stream in similar magnitude (around 300 to 500 seconds) for 50,000 requests. Of this subset, LFUDA increased the time the most - by around 600 seconds (10 minutes). However, to put this in perspective, LRU-2 increased the total time by over 8000 seconds (more than 2 hours), which is 13 times slower than LFUDA. Thus, LRU-2's performance in this respect does not warrant its inclusion in the policy pool.

2Q, which has been shown to perform relatively on par with LFUDA using the live traces, only adds half as much time (around 300 seconds) as LFUDA does to the overall running time and therefore would arguably make it better for request streams similar to the DB2 or OLTP traces. Once again, this shows the importance of reducing the number of policies, and of finding a good mix of policies.

5.3.6 Effect on all policies by introducing well-known susceptibilities

Figure 6 below shows the effect of sequential flooding. Specifically, a synthetic trace was used to highlight the sequential flooding patterns to which LRU is susceptible.

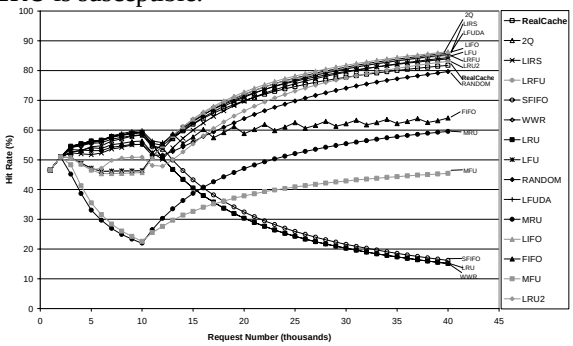

Figure 6: The effect of sequential flooding

As Figure 6 above shows, the hit rates for all of the policies dramatically change at 10,000 requests (this is where the sequential flooding patterns are introduced into the request stream). As expected, the hit rates for LRU, and W^2R (which is based on LRU in this implementation) significantly degrade. SFIFO also shows a similar weakness to this susceptibility, and this is because the primary and secondary buffers in SFIFO combine to act in a similar manner to LRU.

The remaining policies improve upon encountering the sequential flooding because most of them have been designed specifically to avoid this susceptibility. The erratic behaviour of the FIFO policy is due to the fact that the partial DB2 Trace that is included at the start of the synthetic trace used here has already referenced the same pages that are referenced in the sequential flooding part of the trace. Thus, the contents of the buffer pool upon encountering the sequential flooding includes some of those pages to be referenced, and when these pages are referenced during the sequential flooding part, the hit rates of the FIFO trace increases temporarily, and when other pages are not found the hit rates decrease again.

Introducing sequential scanning patterns in the request stream shows an immediate degradation of performance in terms of hits for all policies. Some policies recover, whilst others continue to degrade, but the Real Cache stays with the better policies. Please see (Riaz-ud-Din, 2003) for the details of this experiment.

Figure 7 below shows the effect of skewed high reference frequencies on the policies in the policy pool. A synthetic trace was designed to highlight the frequency patterns to which LFU is susceptible.

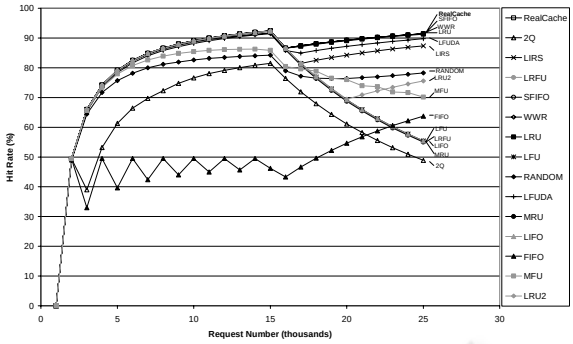

Figure 7: The effect of skewed high reference frequencies

The skewed high reference frequency patterns were introduced at 15,000 requests, as shown by the immediate degradation of hit rates for all policies. Interestingly, 2Q, which has performed the best with all of the tests so far, is the worst performing policy for this trace and along with other policies it continues to decline upon encountering the skewed high reference frequency patterns. This indicates that 2Q is susceptible to skewed high frequency patterns in request streams and once again confirms that having other policies results in a 'safer' policy pool than if just one policy, such as 2Q, was relied upon.

So far, it would seem that a good choice of policies for the policy pool would include 2Q and LRU in particular because 2Q performs very well with most of the request streams, and for those that it does not, LRU performs very well. Thus, they complement each other well and neither has a significant overhead in terms of execution time, in comparison to other policies such as LFUDA or LRU2.

The question of which policies and how many policies to use, is one that is well worth answering. However, this cannot be addressed by studying the effects of running through only two traces, but needs more in-depth examination across a wide range of live access patterns. It would depend highly on the databases expected access patterns.

5.3.7 The effect of misses on the total processing time

Until now the discussion of processing time with regard to the time taken to find a victim and the additional time added per policy has only dealt with the time taken for ACME-DB to do the processing required to perform the caching, uncaching, release and other activities within the scope of the implementation. The time taken to read from disk, which occurs in the event of a miss, has until now been ignored. However, the time that is taken to

read from disk is significant, and forms a major part of the latencies concerned with the overall database performance. As noted, the time needed to read from disk is some 170,000 times greater than reading from memory and each read from disk takes around 10 milliseconds.

To gauge the effect on the total time required for processing requests (including the time for disk reads), the methodology given in Sub-Section 5.2.7 was used. The trace used was intended to induce the best and worst behaviour of SFIFO and 2Q at opposite ends of the spectrum. The results are presented in Table 2 below.

Table 2: Execution times with and without disk reads

Policies in Policy Pool	Time to execute without disk reads (seconds)	Number of misses	Time to execute including disk reads (seconds)
2Q	12	28,197	**293.97**
SFIFO	8	58,196	**589.96**
2Q & SFIFO	18	4,852	**66.52**

The above table shows the time taken to execute the trace, the number of misses on the Real Cache, and the time to execute including disk reads, for the two policies. The column to note is right-most, which shows that the performance was significantly better by introducing two policies in the policy pool, each of which contributed to the hit rate when the other was susceptible to the pattern in the request stream. Overall a gain of 227.45 seconds (almost 4 minutes) over 96,000 requests is made.

The above experiment has shown that when ACME-DB encounters a request stream where the current best policy changes, the overall execution time (when accounting for the disk reads avoided by maintaining a higher hit rate) decreases substantially.

However, what if the current best policy is the same policy over a long period of time (or even the entire request stream), as with the live traces used here? In this case, what would the eventual loss in performance be by using ACME-DB, rather than the policy just by itself? In order to answer this all important question, the above experiment was run once again, but this time with the first 100,000 requests from the DB2 trace, which has shown to favour 2Q over all the other policies. The results are presented in Table 3 below.

Table 3: Execution times with and without disk reads

Policies in Policy Pool	Time to execute without disk reads (seconds)	Number of misses	Time to execute including disk reads (seconds)
2Q	15	367,020	**382.02**
SFIFO	4	392,900	**396.9**
2Q & SFIFO	34	371,170	**405.17**

Running 2Q by itself results in the fastest execution time overall, whilst the slowest is when running SFIFO and 2Q together. The loss in time by adding SFIFO is 23.15 seconds over 100,000 requests, compared to the 227.45 seconds gained in the previous experiment. This time loss is only a small fraction of the time that is potentially gained by using ACME-DB should a request pattern to which 2Q is susceptible be encountered. Furthermore, if 2Q continues to be the current best policy, the Real Cache's hit rate will continue to move closer to 2Q's hit rate. Consequently, the Real Cache's misses will be more or less the same as 2Q's misses, resulting in fewer disk reads, and ultimately faster execution times.

These experiments confirm that the net gain by introducing both policies would indeed result in better overall performance, especially where the request stream exhibits differing patterns of access with time.

6 CONCLUSIONS

The work described in this paper examines the implementation of a recently proposed adaptive algorithm, known as Adaptive Caching with Multiple Experts (ACME), within the database environment. The results indicate that ACME works well with single-sized page caches and with replacement policies that are readily applied to the database buffer pool. It has also been shown that ACME maintains its adaptive behaviour when caching database pages, and stays with the current best policy. Most significantly, it has also been shown that whilst adding more policies to the policy pool increases the execution time, the overall processing time is dramatically reduced due to a greater number of hits. The results are based on an implementation that is efficient, can be readily integrated into the real world environment, and should provide great incentive for further database research. The results of this work provide an excellent platform for further research in the field of database buffer replacement.

ACKNOWLEDGEMENTS

We would like to thank Ismail Ari for the original ACME source code and Xiaodong Zhang, Song Jiang, Sam H. Noh, and Heung Seok Jeon for the live traces.

REFERENCES

Ari, I. (2002). *Storage Embedded Networks (SEN) and Adaptive Caching using Multiple Experts (ACME)*, Ph.D. Proposal.

Ari, I., Amer, A., Miller, E., Brandt, S., and Long, D. (2002). Who is more adaptive? ACME: Adaptive Caching using Multiple Experts. *Workshop on Distributed Data and Structures (WDAS 2002)*.

Castro, M., Adya, A., Liskov, B., and Myers, A. C. (1997). HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, 102–115.

Effelsberg, W. and Haerder, T. (1984). Principles of Database Buffer Management. In *ACM Transactions on Database Systems*, 9 (4), 560 – 595.

Jeon, H. S. and Noh, S. H. (1998). A Database Disk Buffer Management Algorithm Based on Prefetching. In *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM '98)*, 167-174.

Jiang, S. and Zhang, X. (2002). LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 31-42.

Johnson, T. and Shasha, D. (1994). 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Databases*, 439 - 450.

Lee, D., Choi, J., Kim, J. H., Noh, S. H., Min, S. L., Cho, Y. and Kim, C. S. (1999). On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. In *Proceedings of ACM SIGMETRICS'99 International Conference on Measurement and Modeling of Computer Systems*, 134 - 143.

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 ACM MOD International Conference on Management of Data*, 297 – 306.

Ramakrishnan, R. and Gehrke, J. (2000). *Database Management Systems*, McGraw Hill, USA.

Riaz-ud-din, F. (2003). *Adapting ACME to the Database Caching Environment*. Masters Thesis, Massey University.

Sacco, G. M. and Schkolnick, M. (1986). Buffer Management in Relational Database Systems. In *ACM Transactions on Database Systems*, 11, 473 – 498.

Turner, R. and Levy, H. (1981). Segmented FIFO Page Replacement. In *Proceedings of ACM SIGMETRICS*, 48 – 51.