Adapting ACME to the Database Caching Environment

A thesis presented in partial
fulfilment of the requirements for the degree of

Master of Science

in

Information Systems

at Massey University, Palmerston North, New Zealand.

Faizal Riaz-ud-Din

2003

Abstract

The field of database cache replacement has seen a great many replacement policies presented in the past few years. As the challenge to find the optimal replacement policy continues, new methods and techniques of determining cache victims have been proposed, with some methods having a greater effect on results than others. Adaptive algorithms attempt to adapt to changing patterns of data access by combining the benefits of other existing algorithms. Such adaptive algorithms have recently been proposed in the web-caching environment. However, there is a lack of such research in the area of database caching. This thesis investigates an attempt to adapt a recently proposed adaptive caching algorithm in the area of web-caching, known as Adaptive Caching with Multiple Experts (ACME), to the database environment. Recently proposed replacement policies are integrated into ACME's existing policy pool, in an attempt to gauge its ability and robustness to readily incorporate new algorithms. 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 point in time in its request stream. Although execution time increases by integrating more policies into ACME, the overall time saved increases by avoiding disk reads due to higher hit rates and fewer misses on the cache.

Acknowledgements

In the Name of Allah, The Most Beneficent, The Most Merciful

Praise be to Allah, The One and Only True God, who has blessed me with the opportunity to finally achieve my long-standing ambition to undertake my Masters degree.

I am immensely grateful to Markus Kirchberg, for agreeing to supervise me, and for his comments and invaluable criticism during the course of this dissertation. I have greatly appreciated his on-going commitment, dedication and patience with helping me to complete this project. This thesis would not have been possible without his contribution.

I am also grateful to Ismail Ari for providing the source code and traces for ACME, and for providing helpful comments. I would like to acknowledge Xiaodong Zhang, Song Jiang, Sam H. Noh, and Heung Seok Jeon for providing me with the database traces.

I would also like to acknowledge the financial support given to me by the Information Science Research Centre at Massey University.

I would like to thank my parents for their encouragement and support, and for giving me the opportunity to study through to university.

Last, but by no means least, I would like thank my dear wife Leah, for providing me with endless hours of encouragement and support. Thank you for all those hours you spent patiently sitting by me, with motivating me, with your helpful criticism, and with the final documentation. This thesis would not have been what it is without your input.

Table of Contents

List of Figures and Tables

Figures

Tables

1. Introduction

1.1 Cache replacement in databases

One of the biggest factors in a database's performance lies in its ability to effectively and efficiently cache frequently requested data. Databases store all of their data on secondary storage devices, such as hard disks, RAID arrays, and so on. The monetary cost of acquiring these secondary storage devices is decreasing, in stark comparison to the ever-increasing amounts of data that can be stored on them. The speed of accessing data from (and writing to) these devices is still not high enough to enable data to be read and written by database transactions whilst maintaining a reasonable performance threshold. The speed of accessing data from secondary storage devices is still far inferior to the speed of accessing data from the computer's main memory. A typical example of the time taken for accessing a location on disk is about 10 milliseconds, whereas accessing a location in main memory takes typically less than 60 nanoseconds. This means that main memory access is approximately 170,000 times faster than disk accesses [15].

Consequently, to improve performance, databases employ a buffer manager. The main task of a buffer manager is to retrieve the data from secondary storage into main memory, and the data in the main memory is then accessed by the transactions that request information from the database. Accessing data in this manner serves two purposes; firstly it ensures that subsequent 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 that results 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. The main memory cache is also known as the buffer pool.

The buffer manager is responsible for retrieving data from the secondary storage in the form of pages from the main memory based on demand from other database modules. It co-ordinates the writing of data pages back to the secondary storage.

The collection of pages stored by a buffer is collectively known as the buffer pool, whilst the term frames refers to the memory 'placeholders' that main memory pages in the buffer. Specifically, a single frame holds a single page, together with other organisational data relating to the page that it holds. For example, a frame may hold a page with a flag or a bit that indicates whether the page has been modified or not.

When other database modules request data from the buffer manager, the page is retrieved from secondary storage into the buffer pool unless it was already in the buffer pool. The buffer manager controls the release of the page back to secondary storage including whether or not a modified page should be written back to secondary storage. Other modules that use the data from the buffer pool have the responsibility of relaying information back to the buffer manager in respect to whether or not a page is still being used and if the data in the page has been updated.

When a page is requested from the buffer pool it is said to be 'fixed' (or 'pinned'), which means that it is not eligible for replacement. The page is said to be 'in use' when a module is currently using that page and this guarantees that the page will stay in the buffer. Once the page is no longer required by the calling module the page is said to become 'unfixed' (or 'unpinned'), which means that it is now eligible for replacement. [8]

The buffer manager performs the following tasks when a page has been requested by another module (this does not necessarily indicate the typical sequence of operations within the buffer manager):

1. Search for the corresponding page in the buffer and return the address of the page to the calling module if it was found.
2. Find a free frame if the page was not found in the buffer pool.
3. If a free frame was not found, determine the replacement victim according to the rules defined by the buffer replacement policy.
4. Write the victimised page back to secondary storage if it had been updated.
5. Read the requested block from secondary storage into the empty frame in the buffer pool.
6. Return the address of that frame to the caller [8].

The main memory cache (or buffer pool) area can be seen as a collection of pages in the case of database caches. The blocks that make up the segments in secondary storage are internally mapped to one page in the buffer pool and this mapping information is stored in the operating system or in the database-specific disk manager, as the case may be. There may be a number of blocks that may be mapped to a single page in memory.

The challenge in main memory caching is to determine which page of data to replace (the 'victim') out of the buffer pool once the buffer pool is full, in order to make space for any further data pages to be read in from secondary storage. The worst (existing) page to replace from main memory to make place for newly retrieved data from secondary storage would be the page that is requested by the database in the next request for a page. This is because in the next request, that data page would not be found in the buffer pool (since it was written back to secondary storage in the previous request), and it would need to be re-requested from secondary storage. The worst-case scenario in this situation would result in a disk-read with every request, which would mean that having the buffer pool results in no advantage at all, that is, every request requires a data page to be read in from secondary storage.

The best (memory-resident) page to replace from main memory to make place for newly retrieved data from secondary storage would be the page whose next request is the furthest possible request (out of the next request for all the other pages currently in memory) from the current request. This would ensure that the need to perform a read from secondary storage would be avoided for the longest possible time. The best-case scenario in this situation would result in the most appropriate times for disk reads, and would mean the best possible use of the buffer pool. However, the problem with determining which page to replace in this way would mean that future requests must be determined before they occur, which is impossible.

The challenge is to find an algorithm or heuristic that would result in the closest possible performance to that of the theoretical OPT algorithm, described in the paragraph above. An algorithm or heuristic that performs the task of determining which page should be replaced from memory in the event that a page is not found in

memory is known as a buffer replacement policy. It is also referred to as a cache replacement policy, or replacement algorithm.

A hit occurs when a page that is requested by a process or transaction is found in the buffer pool, and a miss occurs when a page that is requested by a process or transaction is not found in memory. In the case of a miss, a disk-read is performed to replace an existing page in memory with the newly read page from disk [8].

The ideal situation would be to have secondary storage devices that would have i/o speeds at least as fast as main memory. However, because of the fact that there is a latency issue involved with reading from secondary storage, and main memory is far more costly than secondary storage, the case for finding an optimal practical replacement policy still exists.

1.2 Benefits and drawbacks of cache replacement policies

A considerable amount of research has already gone into finding the optimal buffer replacement policy. From the outset, a number of policies have been presented. 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) amongst others.

The task was then not only to find a buffer replacement policy that would increase the hit count (the number of hits in the buffer pool), but also a policy that was also efficient in terms of its usage of processing power. LRU was found to be one of the best policies in terms of both easy implementation and a good number of hits in the buffer pool, and is used in a variety of modern day databases, such as Informix and Oracle 7. Microsoft SQL Server uses CLOCK replacement (which is similar to LRU but implemented in a different manner), and DB2 V6 supports FIFO. [15]

However, the drawback of using LRU is its susceptibility to the problem of sequential flooding. This occurs when a consecutive number of blocks are read from disk sequentially (where the number of blocks exceeds the size of the buffer pool), resulting in poor performance of the LRU algorithm. This has led to more research

into policies that would perform at least the same as LRU in respect to the cache hit rate (which is the percentage of hits encountered on a cache as a percentage of the number of requests), whilst at the same time being as easy to implement and avoiding the pitfalls of LRU.

More recently, policies have been introduced such as LRFU (Least Recently / Frequently Used), which attempts to combine the benefits of LRU and LFU [13]. O'Neil *et al* have presented LRU-K [14], which is another policy that attempts to improve on LRU, and has succeeded in achieving a higher hit-rate than LRU, however implementation is more difficult and execution time is higher. Even more recently, policies such as LIRS (Low Inter-reference Recency Set) [11], 2Q [12], and W^2R (Weighing Room / Waiting Room) [10] have been presented. Preliminary results from the experiments undertaken with theses policies suggest a marked improvement over their predecessors, although implementation is not as straightforward as LRU.

The trend has also started to shift towards an attempt to combine the benefits of two or more policies into one policy that can hopefully put together the benefits of all, whilst at the same time avoiding their pitfalls [5].

1.3 Cache replacement in operating systems and web servers

Databases are not the only systems that make use of caches. Other systems like operating systems and web servers also use caching to keep the most-used objects in memory, or another storage device, to reduce data retrieval delays, and to improve performance [16].

Operating systems use main memory to cache blocks from disk to improve performance, and are very similar to database caches except in the following ways:

1. Operating systems cache files (which can exist only on single disks), whereas a database buffer pool may need to cache data that may be stored across disks.

2. A database buffer pool needs to be able to pin a page in the buffer pool, force a page to disk , and order writes (which is important for implementing concurrency control and recovery)
3. A database buffer pool needs to adjust (tune) the replacement policy, and pre-fetch pages based on access patterns in typical database operations.

A web cache is typically implemented on a proxy server that is responsible for acting as a local cache of web objects (such as html files, pictures, and other media files) for internally networked computers. The caching in this case does not occur in memory, rather the objects are cached on disk on the proxy server, and the replacement policy acts on the objects cached on the disk. This enables frequently accessed objects on the Internet to be cached on the proxy server, and can save bandwidth and download time when other computers on the network access the same resources from the Internet.

Whilst the same method of caching these objects may be used in terms of buffer replacement policies, operating systems and web servers need to be aware of other factors that may affect their choice in keeping or ejecting blocks or objects respectively, from their cache spaces. Web servers, for example, also need to consider the distance of objects on the Internet from the local server, and the sizes of objects. These factors may not necessarily be applicable to a locally-stored database buffer pool. The differences between caches on web-servers and database buffers are discussed further in Section 3.

1.4 Motivation

This thesis attempts to trial and evaluate a recently presented cache replacement algorithm that uses machine learning to dynamically select the cache victim from a number of policies, or experts, during the cache replacement process. This algorithm, known as ACME, or Adaptive Caching with Multiple Experts, is adapted from its original web-caching environment to the database environment, in an attempt to explore its performance within the database environment and its adaptability to database-specific policies [2]. The rigidity and robustness of ACME is also tested by its ability to be integrated with more complex policies in its policy pool. These more

complex policies do not simply base their caches on a simple priority queue, but on a number of stacks and queues that make up the whole buffer pool, and use pre-defined heuristics to control the caching and uncaching processes.

Specifically, the objectives of this dissertation are to document the following:

1. *Survey of published database replacement policies*

   The exploration and identification of database specific replacement policies that have been presented in the field of database buffer management over the last few years, and selecting relevant ones for inclusion in the database-adapted ACME's policy pool. This will also enable the newly added policies to be classified according to the criteria that each policy uses to determine pages to be replaced from the buffer pool. The benefits and drawbacks of the policies will be examined at the same time.

2. *Investigation of ACME*

   a. Adaptation to the database environment

   This involves studying the ACME design and examining its potential to be used within the framework of database buffer management. This allows an analysis and description of the ACME architecture with a discussion on how it could be adapted to database buffer management as well as identifying the difference between database caches, web caches, and operating system caches.

   b. Design of a corresponding architecture

   This stage involves the design of an architecture based on ACME that will be proposed as a possible extension to be applied to the database buffer manager, and as a basis for the implementation within the database buffer management framework.

3. *Implementation*

   This involves the implementation of the database-adapted ACME architecture, with respect to the changes required to enable the new policies to be added to the policy pool, as well as any other changes required to adapt ACME to the database environment. This would also require implementing the policies in such a way that would enable them to be seamlessly integrated within the framework of the ACME implementation.

4. *Evaluation of the database-adapted ACME*

   The database-adapted ACME will be driven using live traces obtained from commercial databases. These traces will be used to evaluate the database-adapted ACME's performance with respect to hit rate, cache size and time. Synthetic traces manufactured with the express purpose of exposing well-known weaknesses in basic replacement policies will be used to observe the effects on the policies within the policy pool, as well as ACME's Real Cache. Both the live and synthetic traces will be driven using different combinations of policies in the policy pool to observe the behaviour of the Real Cache, and the individual policies as well.

1.5 Presentation of the remainder of this dissertation

The remainder of this dissertation is presented as follows: Section 2 presents related work, particularly other adaptive caching algorithms, and an overview of ACME. Section 3 describes the changes made to ACME to enable it to be adapted to a database environment. Section 4 relates the issues concerned with implementing ACME in the database environment. Section 5 discusses and analyses the results obtained from running the database-adapted ACME over live and synthetic traces. Section 6 suggests future work required following on from this research. Finally, Section 7 concludes and summarises the work presented in this dissertation.

2. Related Work

This section details related work in regards to adaptive algorithms and specifically Adaptive Caching with Multiple Experts (ACME).

2.1 Adaptive algorithms

Adaptive algorithms attempt to change their behaviour based on different patterns of access encountered within a request stream. In most cases, two or more static algorithms would be combined, to enable the resulting adaptive algorithm to use the appropriate traits of each of the static algorithms at different stages depending on the pattern of access that is encountered on the request stream.

Hybrid Adaptive Caching (HAC), presented by Castro *et al*, is an adaptive caching algorithm that combines the benefits of both page and object caching [5]. It attempts to combine the benefits of both methods of caching whilst at the same time 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. In this way it is able to adapt to the different access patterns in the request stream. This algorithm was presented within the environment of distributed objects [5].

Adaptive Caching with Multiple Experts (ACME) was presented recently by Ari *et al* [2]. This algorithm was presented in the web-caching environment, and is described in the following section. ACME is used as the basis of this research as it readily allows the incorporation of other existing policies within its architecture, and this provides the flexibility required to adapt it to the database environment [2].

2.2 ACME: Adaptive Caching with Multiple Experts

2.2.1 Description of ACME

ACME is an adaptive cache replacement policy that has been presented recently for caching web objects on web servers. It makes use of a policy pool that consists of a

number of different static replacement policies that represent experts. Each of these experts has their own virtual buffer space (known as the virtual cache) in which they cache incoming requests independently of the other experts in the policy pool. This means that at any particular point in time, each virtual cache may hold different objects depending on how the associated policy has been caching the incoming data request stream. Figure 1 below provides an architectural overview of the ACME design presented as an illustration [2].

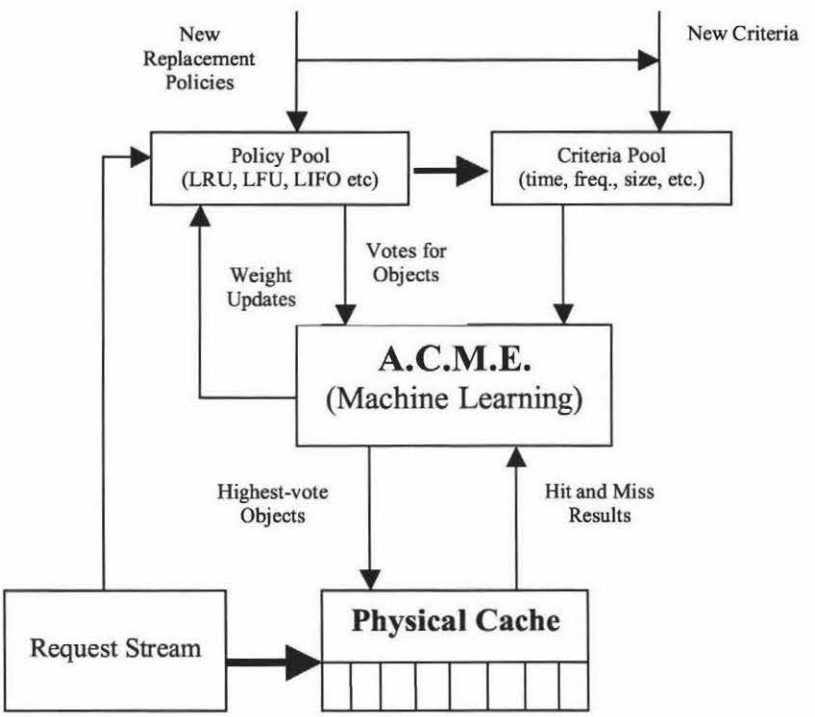


Figure 1 - Design of Adaptive Caching with Multiple Experts (ACME)

Each expert also has a physical cache, which contains references to the objects that are held in the Real Cache. The physical caches of all of the experts contain references to the same objects, but are ordered according to their corresponding algorithms [1].

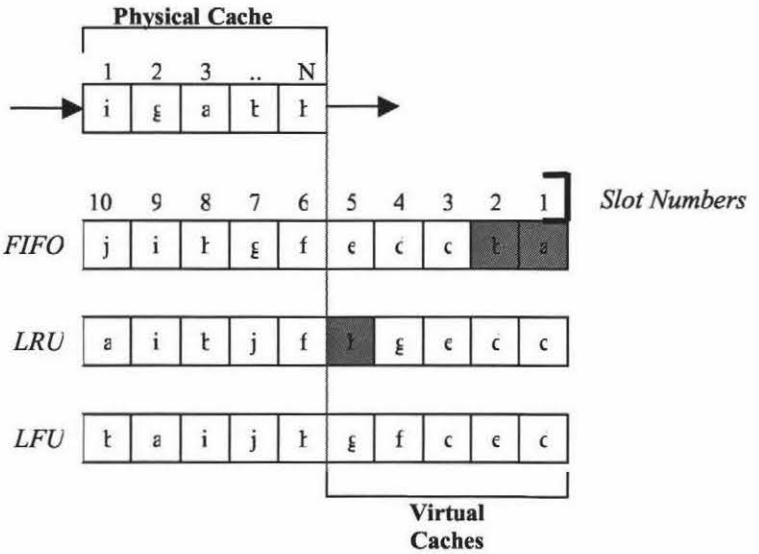
The Real Cache holds the actual data objects that are cached in and out by a particular policy in the policy pool. When the Real Cache is full, and an object needs to be replaced to make way for the requested object (to be retrieved from the data source), each of the policies in the policy pool is tested to see whether or not they already have the requested object in their respective virtual caches. If so, the

corresponding policy is rewarded with a vote, and if not, then the corresponding policy is penalised with a negative vote [1].

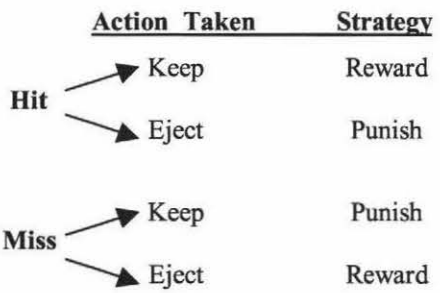
Once an object has been uncached from Real Cache, each of the policies in the policy pool updates its physical cache by releasing the corresponding reference to the Real Cache from its physical cache [1].

A machine-learning algorithm is used to allocate the weight of each policy, along a continuum from 0 to 1, based on the policy's history of votes. In other words, each policy has a weight between 0 and 1 that is continuously updated, and the weights of all the policies add up to 1. Once the weights have been updated, a policy is randomly chosen to select the replacement victim from the Real Cache. The probability of a policy being chosen at any time is directly affected by its weight, so if a policy has enjoyed a lot of success in ensuring that it caches the objects that are requested by the incoming requests, it will have a higher weight, and therefore a greater chance of being selected at random [1].

Once a replacement victim has been chosen by the selected policy, that victim is paged out of the Real Cache, and the requested page is paged in. Figure 2 below illustrates the adaptive caching features of ACME [2].



(a) *Multiple Cache "Experts"*



(b) *Evaluation Paths*

Figure 2 - Adaptive caching

2.2.2 Benefits of ACME

Ability to register new policies/criteria

ACME has been designed with an architecture that allows the addition of new policies into the policy pool, without the need to make major modifications to the rest of the existing structure.

As such, it is possible to add policies that use different types of criteria to determine the cache victims, without explicitly having to include those criteria in all the other existing policies. In this way it is possible to benefit from using a number of different

types of policies that use different criteria within the buffer replacement process. This is especially encouraging, as it would mean that the benefits of using all the criteria may be gained without having to mix them all up into one single policy, which may not be possible or practical.

The downside of this, however, is that as more policies are added to the pool, the need for additional virtual caches means that there is less main memory for storing the actual data objects, and more CPU cycles are required to process the caching and uncaching functions that are provided by the policies in the policy pool.

Isolation of experts (policies) and criteria

Replacement policies attempt to achieve the best possible performance by using the available information from past requests to keep the most appropriate pages in memory. Different policies determine the best pages to keep by using different criteria. However, due to the fact that some policies are better than other policies with regard to certain patterns of request streams, the effectiveness of one particular policy through a variety of patterns of requests may not be consistent.

ACME attempts to use the best possible policy (or expert) for the current pattern of access by adapting the Real Cache to the current best policy. This is done by the implicit isolation of the criteria from the mechanism that is responsible for determining the caching or uncaching of pages from the buffer. This ensures that the caching or uncaching of pages is not determined by a single set of parameters [1].

Machine-learning capabilities

ACME uses machine-learning capabilities to determine which expert to use for caching or uncaching at any point in time. The machine-learning algorithm within ACME 'learns' to use the current best policy over time, based on the past successes and failures of the individual experts in the policy pool [1].

3. Adapting ACME to the Database Environment

ACME was originally designed for use within a web-cache. As such, it includes certain features that are not necessarily required in a database caching environment. With regard to the target database cache simulation in mind, ACME needed to be slightly modified.

3.1 Defining the intended database environment

The intended database environment for which ACME is adapted to may be defined by a number of constraints. These constraints may not necessarily apply to a real-world database environment. However, for the purposes of experimenting with the concept of a new caching mechanism such as ACME, it is sufficient. The constraints are defined below:

1. *Uniform page-sizes in the cache* – The pages in the database cache have the same size.
2. *Data centrally stored* – The data in the database is stored on a single server or workstation, or it may be distributed across a number of servers or workstations. However, the buffer manager acts only on the physical machine on which it operates, and is not aware of the distributive nature of the database, if the database is distributed. As such, buffer replacement works within the context of a single machine.
3. *Single cache* – There is a single-level cache rather than multiple caches.
4. *Data is read-only (in this context)* – The operations on the database are read-only operations, so writing is not enabled. As such, the implementation of the database-adapted ACME design does not include the ability to take into account the latency associated with writing dirty pages back to disk.

3.2 Allowing for same sized pages

The database-adapted ACME is tested using buffers that consist of pages that have the same size. The typical size of a page in database buffers is 16 kilobytes, and this is the size of the pages that make up the target database buffer pool.

Taking the above considerations into account, the most obvious modification that was made to the original web-environment ACME architecture was removing the need to check the sizes of an object to be cached, against the size of the available cache space. In a web-caching buffer, objects with different sizes are cached and uncached, and it is necessary to continuously update the available cache size after each caching and uncaching process. However, with a database cache, the objects are represented by pages in the buffer that are all the same size, and therefore it is sufficient to keep track of the number of pages available in the buffer. This allows for caching and uncaching without the need to ensure that there is enough space in the cache by uncaching a number of objects before caching the requested object. This makes it easier to predict the number of I/O operations on the buffer pool.

3.3 Removal of web-specific policies

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 a factor. New policies were added to the buffer pool, with an emphasis on using policies that are applicable to the implemented cache simulation.

3.4 Additional demands of database specific policies

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 the database-adapted ACME 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 require careful considerations to be made before integrating into the ACME buffer pool.

Additionally, the release mechanisms would now need to be defined for each policy that uses more than a single queue or stack to manipulate the buffer pool. The papers from which the multi-buffer policies were sourced only described the caching and uncaching operations to be taken for their corresponding policies, and the release mechanisms on arbitrary pages need to be designed based on the heuristical analysis of the intrinsic behaviour of these policies. The policies are described in the next section (Section 4), along with the way in which the release mechanism for each policy has been defined.

In summary, the most obvious differences between a web-caching environment (for which ACME was originally developed), and the database environment (as defined in the previous sub-section) are:

- a web-cache caches objects of differing sizes, whereas a database cache caches objects of the same size, that is, pages in memory. These pages map to blocks on disk, and the mapping between memory pages and disk blocks is stored internally.
- a web-cache algorithm may take into account parameters such as the distance of the object from the cache (in the case of a network) and the time it would take to retrieve that object, whereas a database cache does not need to take into account such parameters. The reason for this is that the database caches only what is on the secondary storage on the machine on which it is running, rather than retrieving it externally.
- The new database-specific policies may use multiple queues or stacks to manipulate the buffer pool, and this means that release mechanisms would need to be defined for these new policies.

4. An Implementation of the Database-Adapted ACME Algorithm

This section discusses the issues related to implementing the additional database policies within the framework of the original ACME architecture, after having applied 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, this does not necessarily mean that the replacement victim is 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 an page from a priority queue. However, in the implementation of the database-adapted ACME, 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 could be invoked independent of the caching and uncaching mechanisms.

4.2 Choice of policies

The original ACME implementation employed the use of twelve different policies (or experts) within the policy pool. The policies used were Random, FIFO, LIFO, LRU, LFU (Least Frequently Used), LFUDA (Least Frequently Used with Dynamic

Aging), MFU (Most Frequently Used), MRU, GD (Greedy Dual) [18], GDS (Greedy Dual Size) [3], GDSF (GDS with Frequency) [6], and SIZE. These policies were shown to base their replacement victims on one or more of the following criteria: time, frequency, size and retrieval cost.

Since the size and retrieval cost (in terms of distance and time) criteria are redundant in database environments (as discussed previously in Section 3), the policies that employ the use of these criteria have been removed from the policy pool in the database-adapted version of ACME. As such, the GD, GDS, GDSF, and SIZE policies are not used in the database-adapted version of ACME.

4.3 Review of policies

A brief description of the original ACME policies that have been re-used, as well as the new policies that have been added to the policy pool of the database-adapted version of ACME are presented in this sub-section, along with the strategy used to implement the release mechanism for each of these policies.

4.3.1 Existing policies

The following eight policies in this sub-section are those that were part of the original ACME design and were subsequently selected for use in the database-adapted version of ACME. These policies are implemented as single priority queues, and as such, their release mechanisms simply evict the victim from their individual priority queues.

4.3.1.1 Random

The Random policy selects a page to replace from the existing pages in the buffer pool randomly (as the name suggests). As such, no criteria are used to select a victim.

4.3.1.2 FIFO

The FIFO (First-In-First-Out) policy replaces the earliest cached page in the buffer pool. FIFO's weakness lies in not being able to handle scan sequences where the number of unique pages being accessed in the scan are greater that the size of the buffer pool. In this case, the FIFO policy would need to read the pages from disk for each request.

4.3.1.3 LIFO

The LIFO (Last-In-First-Out) policy replaces the latest cached page from the buffer pool. As with FIFO, LIFO is susceptible to greater-than-buffer-size scans as well.

4.3.1.4 LRU

The LRU (Least Recently Used) policy replaces the page that has been referenced least recently. The difference between FIFO and this policy is that FIFO replaces the page that has been *cached* (that is, brought into the buffer pool) the earliest, whereas LRU replaces the page that has been *referenced* (that is, requested by an operation whether the page was already in the buffer pool or not) the earliest. LRU is also susceptible to greater-than-buffer-size scans (sequential scanning) as with FIFO. Additionally, LRU's weakness is also emphasized in the sequential flooding scenario, where a number of infrequently used pages are requested in an iterative fashion, causing LRU to evict commonly used pages, which results in a degradation of performance.

4.3.1.5 LFU

The LFU (Least Frequently Used) policy replaces the page that has been referenced least frequently. Each of the pages has a reference count attached to it which is updated every time the corresponding page is referenced. The page with the lowest reference count is selected as the victim. This policy is susceptible to traces that exhibit patterns where there is a high frequency of pages referenced at the start of a request stream, causing some pages to gain a high reference count. These pages are

then not used again, except rarely in later parts of the stream, yet they remain resident in the buffer pool for a long time due to their high frequency of references at the start of the stream. This may be typical in database applications that refer to certain data that is used for initialisation at the start of the application, and is then not used again, or used in exceptional circumstances. This policy, like FIFO, is also susceptible to greater-than-buffer-size scans.

4.3.1.6 LFUDA

The LFUDA (Least Frequently Used with Dynamic Aging) policy uses a function based on the frequency and age of the page, to work out the priority of each page in the buffer pool. The page with the lowest priority is selected as the replacement victim.

4.3.1.7 MFU

The MFU (Most Frequently Used) policy removes the page that has been accessed most frequently from the buffer pool. This policy is also susceptible to greater-than-buffer-size scans.

4.3.1.8 MRU

The MRU (Most Recently Used) policy removes the page that has been accessed most recently from the buffer pool. MRU, like some of the policies already mentioned, is also susceptible to greater-than-buffer-size scans as well.

4.3.2 New policies

The following six policies in this sub-section are those that have been added as part of the database-adapted ACME implementation.

4.3.2.1 LIRS

Overview

The LIRS (Low Inter-reference Recency Set) [11] policy employs the use of two primary buffers to control the caching of pages. The IRR (Inter Reference Recency) of a page refers to the number of other pages accessed between two consecutive references to the page. The page with the highest IRR is selected for replacement [11].

The LIR (Low Inter-reference Recency) buffer is the buffer that contains all pages that have an IRR of less than a specified threshold. The HIR (High Inter-reference Recency) buffer is the buffer that contains all pages that have IRR of more than a specified threshold. *Note*: The implementation dynamically adjusts the threshold to be the IRR of the LIR page with the highest IRR value [11].

The HIR buffer is normally a very small portion of the overall buffer pool (around 1%). A page's LIR or HIR status can be switched at any time as the recency of that page changes over successive requests [11].

A Stack S is used to control the HIR / LIR status of pages. The Stack S contains references to all the LIR pages, and references to non-resident HIR pages. A List Q is used to maintain a list of all resident HIR pages [11].

The benefits of the LIRS algorithm include eliminating the need for tuning or adapting sensitive parameters, effectively utilising multiple sources of access information, and addressing LRU limits with weak locality workloads without relying on the explicit regularity detections, thus increasing performance. Other benefits have also been mentioned [11].

Release Mechanism

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, in the case of releasing the page arbitrarily some other factors need to be taken into account.

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.

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 pushing it onto the top of the LIR page set (the implementation employs an 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 reason for designing the release mechanism in this manner is that 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).

4.3.2.2 LRFU

Overview

The LRFU policy associates a value with each page in the buffer pool [13]. This value is called the CRF (Combined Recency and Frequency) value. The CRF indicates the possibility that a page will be referenced in the near future. A function *f(x)* is used to calculate the CRF value for each page, where x is the time span from the page's reference in the past to the current time. Each reference to a page in the past contributes to this value [13].

The function *f(x)* associates a weight with each page based on the recency and frequency of the page's reference history. It allocates a greater weight to more recent references, so the page that is replaced is the one that has the lowest weight [13].

A major benefit of LRFU is that it exploits the positive features of both LRU and LFU and is consequently superior to both in terms of achieving higher hit rates than both. For small cache sizes, recency-based policies (such as LRU) perform better, whereas for larger cache sizes, frequency-based policies (such as LFU) perform better. LRFU attempts to incorporate both frequency and recency to gain the benefits of both methods [13].

Release Mechanism

The release mechanism in this case simply removes the page from the priority queue.

4.3.2.3 LRU-K

Overview

The LRU-K algorithm [14] looks at the times of the last k accesses to popularly accessed data pages, and uses this history to determine which page to replace. LRU-K avoids the correlated reference problem by defining a time-out period between which page access times should be considered for capture in the page access history. It avoids the problem of "forgetting" about recently accessed pages that have been

paged out by defining a Retained Information Period, after which the history for the page is dropped [14].

The benefits of LRU-K include that it is self-tuning, does not rely on external hints about workload characteristics, it is fairly simple, and incurs little bookkeeping overhead. It also adapts in real-time to changing patterns of access, and is better than LRU at discriminating between frequently referenced and infrequently referenced pages [14].

However, some weaknesses have also been identified. Lee *et al* [13] have noted that LRU-K ignores the recency of the k-1 references and considers only the distance of the kth reference, thus violating the rule of thumb that the more recent behaviour predicts the future better. Whilst it can quickly remove unpopular pages from the buffer pool when K is small, LRU-K is not very adaptive to changing workloads when K is large [13]. Jiang and Zhang [11] have noted that LRU-K can greatly increase complexity and / or cannot consistently provide performance improvement. It lacks an effective mechanism to identify the most deserved portion matching the currently available cache size. As a result of this, it either significantly increases the cost and / or introduces workload sensitive parameters [11].

The implementation of LRU-K used in the database-adapted ACME uses the LRU-2 version, which has been shown to be more efficient than the LRU-3 or greater values of k [14].

Release Mechanism

The release mechanism in this case simply removes the page from the priority queue.

4.3.2.4 SFIFO

Overview

SFIFO replacement [17] partitions the buffer into two areas: a primary buffer and a secondary buffer. The size of the secondary buffer is determined by a parameter, p, which indicates the amount of total buffer space that it uses [17].

Pages are placed into the buffer starting with the primary buffer. The primary buffer is managed as a FIFO queue. When the primary buffer is full, space needs to be created for the newly requested page in the primary buffer. The last page is removed from the bottom of the primary buffer and placed at the top of the secondary buffer. The secondary buffer is managed as an LRU queue. When the secondary buffer is full, the Least Recently Used page is removed from the secondary buffer. If a page is re-referenced while it is in the secondary buffer, it is removed from the secondary buffer and placed on top of the primary buffer. If p is 0 (that is, there is no secondary buffer), the algorithm degenerates to pure FIFO; if p is 100 (that is, there is no primary buffer), the algorithm becomes the same as LRU [17]. The p value used in the implementation was 30%.

Release Mechanism

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 design of the release mechanism 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.

4.3.2.5 2Q

Overview

The 2Q algorithm divides the main buffer area into two main areas, known as the Am buffer where the frequently referenced pages are kept, and the A1 buffer, where the most recent first accesses are kept. The Am buffer is managed as an LRU queue.

The A1 buffer is further divided into 2 areas, known as A1in and A1out. The purpose of the A1in buffer is to hold the most recently accessed pages that are not already in the main Am buffer. The A1in buffer is managed as a FIFO queue, and any pages that are referenced while they are in this buffer are treated as correlated references (that is, references that are somehow related to each other, and occur due to this

relationship, and not because of independent reasons). As such, no action is taken when a page is re-referenced during its residency in the A1in buffer [13].

The purpose of the A1out buffer is to hold the references to those pages that have been paged out from the A1in buffer. As such, the entries in the A1out buffer do not contain the pages themselves, but only references to those pages. If a page is requested during its residency in the A1out buffer, it is regarded as a valid and non-correlated reference, and is therefore promoted to the Am buffer. It is thus regarded as a frequently referenced page [13].

The benefits of 2Q include the following: cold pages (that is, unpopular pages) are quickly removed from buffer pool [13], it has a lower time complexity than LRU-K [13], it performs strongly when the buffer pool size is small [13], there is a constant time overhead, it performs as well as LRU-2, and there is a 5-10% improvement in hit rate over LRU for a wide variety of applications and buffer sizes.

Jiang and Zhang [11] have highlighted that the drawbacks of 2Q include the fact that it uses pre-defined parameters (Kin and Kout), which require careful tuning and are sensitive to the types of workloads.

Release Mechanism

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 reason for designing the release mechanism in this manner is that the sizes of each of the A1in and A1out buffers are checked when uncaching occurs, thereby enabling the arbitrary release of an page from either the Am or A1in buffer.

4.3.2.6 W^2R

Overview

The W^2R (Weighing Room / Waiting Room) algorithm [10] divides the buffer into two rooms - the Weighing Room and the Waiting Room. The Weighing Room is the main buffer area where pages are prioritised, or attain weights, to indicate which page should be replaced. The weight associated with each page depends on the policy that is used in the Weighing Room. In other words, any particular policy may be used to define the weights of pages in the Weighing Room.

The Waiting Room is used as an area where pre-fetched pages may be stored. A page is pre-fetched whenever it is found in either of the caches, or if a page needs to be replaced. The Waiting Room is managed as a FIFO queue. The pages from the Waiting Room are only promoted to the Weighing Room if they are referenced while they are in the Waiting Room.

The authors state that this policy has an improved hit rate of 14 - 23% over 2Q, and it alleviates the problem of replacing a hot page with a pre-fetched page that may be cold. The best hit rate occurs when the Waiting Room size is small [10].

Release Mechanism

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, then 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.

4.4 Classification of Policies

Table 1 below summarises the classification of the replacement policies used in the policy pool of the database-adapted ACME implementation, by the criteria that they use to determine replacement victims.

Criteria	Algorithm
-	RANDOM, FIFO, LIFO
Time	MRU, LRU, LFUDA
Frequency	MFU, LFU, LFUDA, LRFU, SFIFO, 2Q
Recency	LRU-K, LIRS, 2Q, LRFU, SFIFO, W^2R

Table 1 - Classification of policies used in database-adapted ACME

5. Experimental Results

This section provides details on the traces and experimental methodologies used to test the implementation of the database-adapted ACME design with the newly added policies, as well as an analysis and discussion of the experimental results.

5.1 Traces

The database-adapted ACME 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.

It was not possible to use the trace that was used to test the original web-cache ACME, as the trace was sourced from a proxy cache. Instead, two traces were used to simulate request streams. These two traces are the DB2 and OLTP (On-Line Transaction Processing) traces used in the papers by Jeon and Noh [10], Johnson and Sasha [12] and by O'Neil *et al* [14]. 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 a total of 914,145 requests to 186,880 distinct pages.

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

The synthetic traces are described below:

Synthetic Trace A - This trace initially includes some records from the DB2 trace, and then has a sequence of requests following a sequential flooding pattern to induce the weakness of LRU. This was to illustrate the adaptive behaviour of ACME when

encountering the positions along the trace when the LRU policy was at its worst, and when it was at its best. LRU was chosen as it is the most common replacement policy in practice, being used in Sybase ASE, IBM Informix and Oracle 7.

Synthetic Trace B – This trace included requests to unique pages throughout the stream to force each policy in the policy to perform uncaching, and therefore recording losses against their weights. This would cause all policies to have the same weights throughout the request stream, and no one policy would be favoured. This was necessary when testing the additional time taken for ACME to execute upon adding each policy to the policy pool.

Synthetic Trace C – This trace initially includes repeated requests for the same pages and then has a sequence of requests following a normal pattern of access. This was to induce the weakness of LFU when dealing with skewed high frequency requests. This would illustrate the behaviour of ACME when encountering the positions along the trace when the LFU policy was at its worst, and when it was at its best.

Synthetic Trace D – This trace initially includes some records from the DB2 trace, and then has a sequence of requests following a sequential scanning pattern to induce the weakness of FIFO. This aimed to illustrate the adaptive behaviour of ACME when encountering the positions along the trace when the FIFO policy was at its worst, and when it was at its best.

Synthetic Trace E – This trace is a combination of Synthetic Traces C and A above, in that order.

5.2 Experimental methodology

This sub-section provides details on the different sets of experiments that were used to test and evaluate the database-adapted ACME. Unless otherwise stated, all of the experiments listed below that tested the hit rates in relation to the request stream were based on a buffer pool size of 1000 pages.

5.2.1 Combined effect of all policies on the Real Cache

The first part of the experiment involved running the database-adapted ACME implementation with all of the fourteen policies enabled across the two live traces. This produced log files which recorded cache hit rates for different cache sizes as well as hit rates along different parts of the request stream. This was to determine the combined effect of all the policies on the Real Cache hit rates, and to show the hit rates achieved by the virtual caches of the individual policies themselves.

5.2.2 Real Cache adaptation to the current best policy

The second part of the experiment involved running the database-adapted ACME implementation with the best policy and an average policy to determine the extent to which the Real Cache adapted to the best policy in terms of its hit rate. Both live traces were used to drive this experiment.

5.2.3 Effect of the weaker policies on the Real Cache

The next experiment involved running the database-adapted ACME simulator with the policies that had the best hit rates across the two live traces. This was to determine whether or not using only the best policies would result in the same hit rate for the Real Cache as it would with all the policies enabled. This would show the effect that the other poorer performing policies have on the Real Cache hit rates.

5.2.4 The adaptive nature of database-adapted ACME

The next experiment involved running the database-adapted ACME simulator with Synthetic Trace A to determine the behaviour of ACME when one of the policies switched from one performance extreme to the other and how the presence of another policy in the policy pool should have a stabilising effect during the performance degradation of the first policy. In this case the policies used in the policy pool were LRU and 2Q.

5.2.5 Investigation of susceptibilities to well-known weaknesses

In line with the original objectives of this research, replacement policies were run using synthetic traces which were designed to expose the commonly known weaknesses of particular policies and the effects of this were observed. This would bring out the susceptibility of the other policies in the policy pool, and the effect would be observed on the Real Cache. Synthetic Traces A, C, and D were used to drive this experiment.

5.2.6 Average time to select a victim

The database-adapted ACME was tested with all of its policies in the policy pool using buffer pool sizes of 100, 200, 300, 500, 800, 1000, 2000 and 3000. This test was run using both live traces, but only for the first 50,000 requests of each trace. The objective of this test was to determine the effect of the cache size on the average time taken for each policy to select a victim when a miss was encountered in the Real Cache.

5.2.7 Average time loss for each additional policy

It is well known that the different replacement policies used in the database-adapted ACME vary in performance with regard to their time complexity. It is possible that the addition of certain policies to the policy pool would increase the overall time complexity of the database-adapted ACME, and thus may detract from the benefits of using such policies that may have a beneficial effect in terms of hit rate. Synthetic Trace B was used to drive this experiment.

5.2.8 Switching of current best policies

Using the OLTP trace the database-adapted ACME was run using only LRU-2 and LRU in the policy pool. 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.

5.2.9 The effect of disk reads on the total execution time

Synthetic Trace E was used to gauge 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 better or worse performance.

5.3 Analysis and discussion

This sub-section provides an analysis and discussion of the experiments that were described in the previous sub-section.

5.3.1 Relative performance of policies

For the details of the methodology used for this experiment, please refer to Sub-Section 5.2.1.

Figures 3 and 4 below show the hit rates for all of the policies in the database-adapted ACME's policy pool, as well as the hit rates for the Real Cache for the DB2 trace and the OLTP traces respectively. These are based on a buffer pool size of 1000 pages.

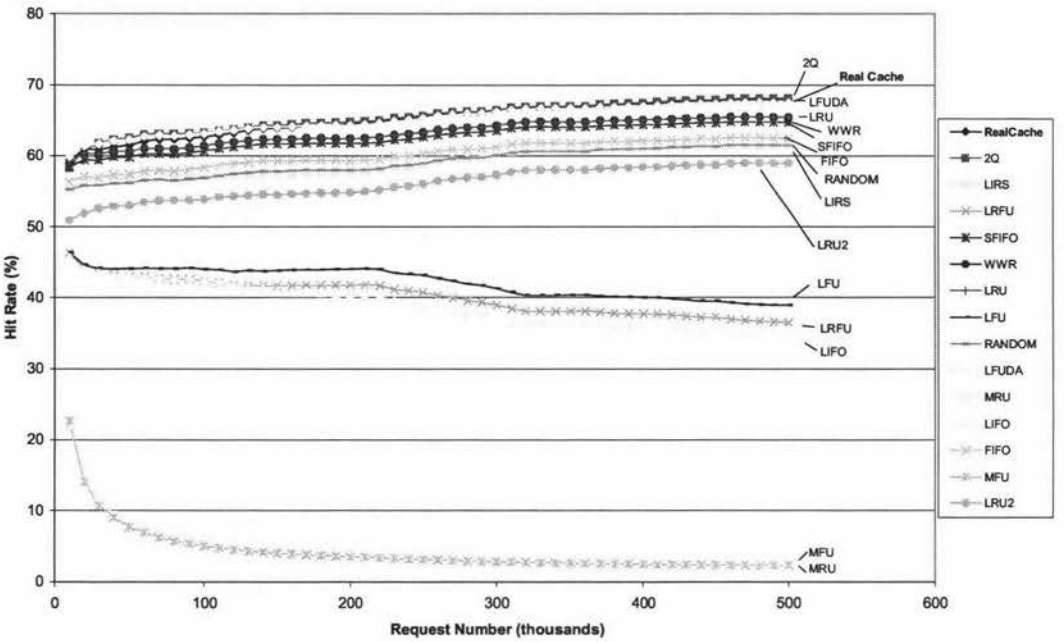


Figure 3 - Requests vs Hit rates (all policies), DB2 Trace

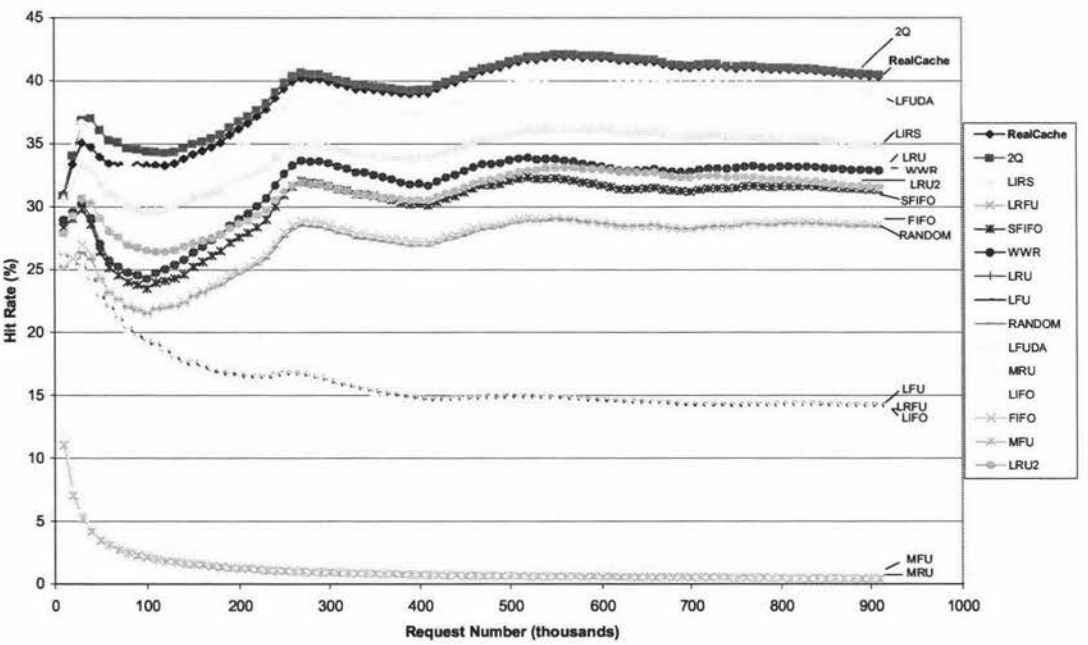


Figure 4 - Requests vs Hit rates (all policies), OLTP Trace

The first set of results (Figures 3 and 4 above) clearly indicates that there are three distinct groups of policies that are poor, average, and good in their relative performances, respectively. This is independent of which of the two live traces is used. Those policies that performed poorly were MRU and MFU, whilst those that performed well were 2Q and LFUDA, in particular.

5.3.2 The performance of the Real Cache

For the details of the methodology used for this experiment, please refer to Sub-Section 5.2.2.

The purpose of the ACME algorithm is to adapt to the best expert in the policy pool at any one point in time for a given request stream. This is reflected in the results of running both the DB2 and OLTP traces on the simulation.

Figures 3 and 4 above both clearly show the hit rates on the Real Cache following an almost identical trend to the 2Q policy, which performed the best of all the policies. This illustrates the ability of the database-adapted ACME algorithm to mimic the best policy. However, these figures do not clearly indicate whether it is the group of policies with the best hit rates towards the top of the graph that are contributing towards the high hit rate of the Real Cache, or if it is indeed a result of the Real Cache adapting to only the single best policy.

In order to test this adaptive behaviour by determining whether or not the Real Cache would stay closer to the best policy over a trace or another poorer performing policy, only 2Q and FIFO were used in the policy pool and the effect on the Real Cache was monitored. The outcome of this test is reflected in Figures 5 and 6 below for the DB2 and OLTP traces respectively.

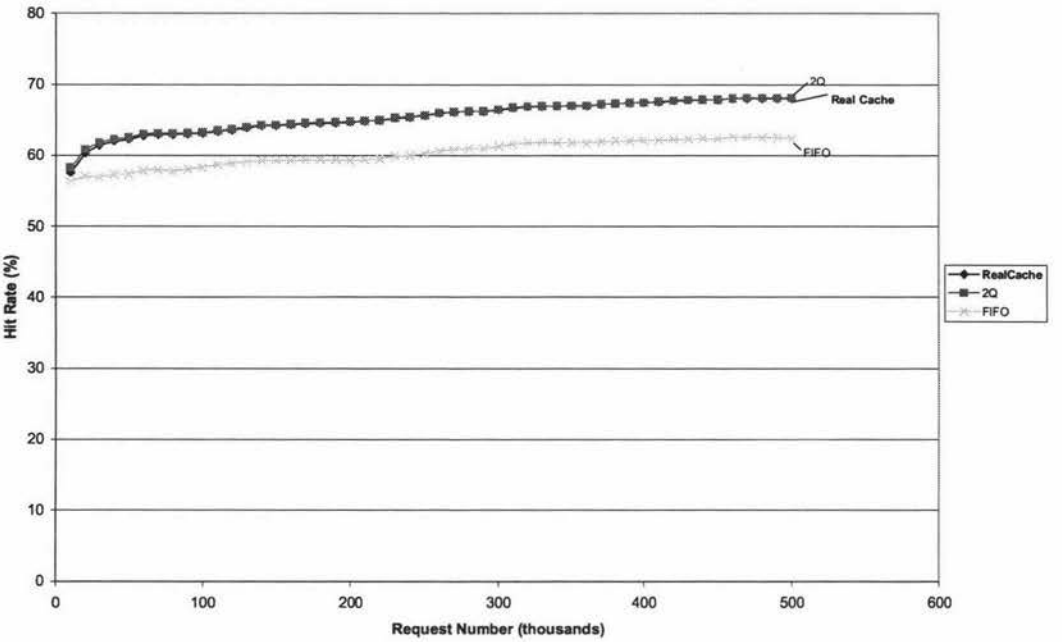


Figure 5 - Requests vs Hit rates (2Q and FIFO only), DB2 Trace

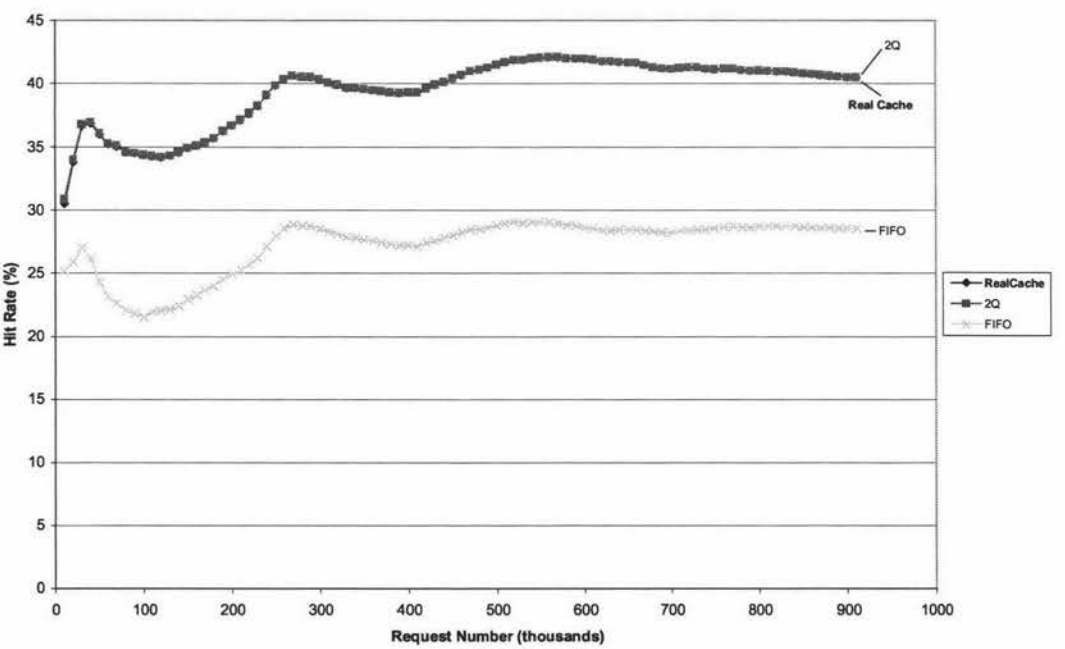


Figure 6 - Requests vs Hit rates (2Q and FIFO only), OLTP Trace

The two figures above show that even when only two policies are used in the policy pool, the Real Cache will try to stay as close as possible to the best performing expert, in terms of its cache hit rates.

The two experiments discussed above indicate that for the traces used, the Real Cache is using the 2Q policy to do the majority of its caching, thus resulting in hit rates similar to 2Q's hit rates. This would bring to mind the question as to whether there is any benefit in having other policies in the policy pool. In other words, do all the policies in the policy pool contribute towards the Real Cache hit rate in a synergistic manner?

This question is answered upon examining Figures 7 and 8 below, which are the results from the experiment outlined in Sub-Section 5.2.3. These two figures compare the Real Cache hit rates when all the policies are used in the policy pool, to when only the two worst performing policies and the two best performing policies are used in the policy pool.

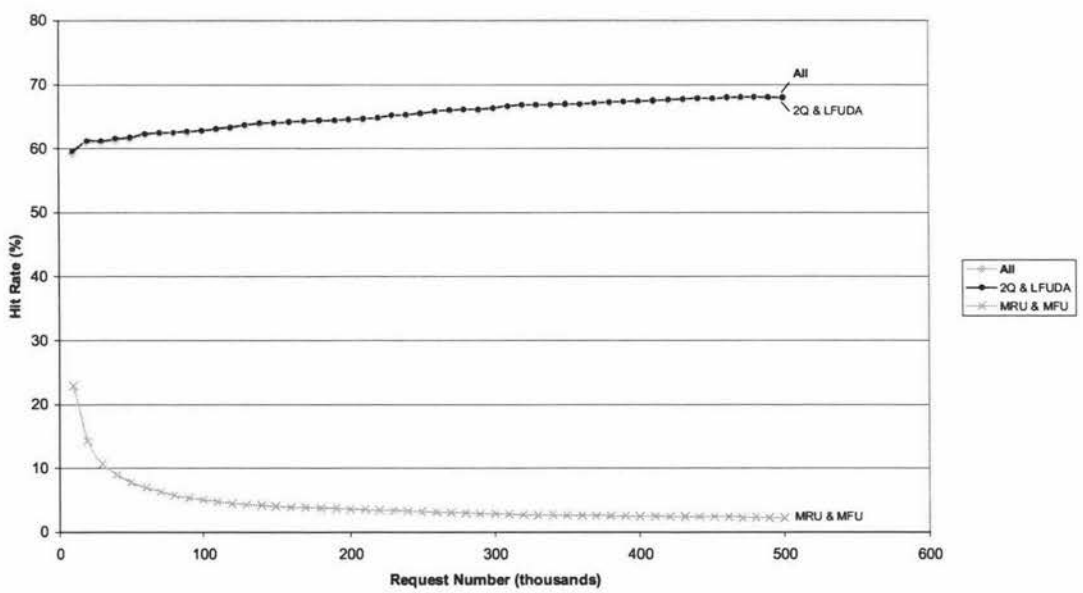


Figure 7 - Real Caches for different policy combinations, DB2 Trace

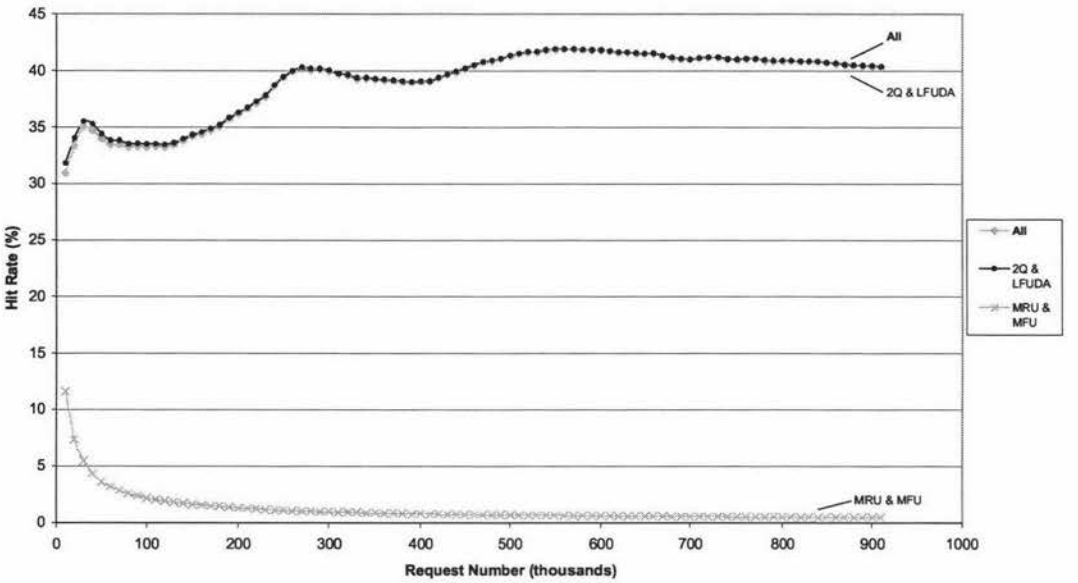


Figure 8 - Real Caches for different policy combinations, OLTP Trace

Figures 7 and 8 above clearly illustrate that there is no significant difference between the two scenarios that depict the Real Cache of all policies, and the Real Cache of the best two policies. This indicates that having a large number of policies in the policy pool would neither help nor hinder the Real Cache hit rate, in the case where the other policies do not become the current best policies at any point in time during the caching process.

Figures 7 and 8 above also show the Real Cache when only the worst-performing policies (MRU and MFU) are used in the policy pool. It has already been suggested above that the Real Cache can only perform as well as the current best performing policy, which may not necessarily have high hit rates, as in this case.

5.3.3 The adaptive behaviour of ACME

The above results may also bring to mind the following question: why not just use 2Q and forget about ACME? The results presented so far do indeed seem to suggest that 2Q performs best by itself.

Whilst 2Q has shown the best performance for the live requests shown here, there may be other request patterns for which it may not do as well. To illustrate this fact, the methodology outlined in Sub-Section 5.2.4 was used. Figure 9 below shows the results of running the database-adapted ACME on Synthetic Trace A, with only LRU and 2Q in the policy pool.

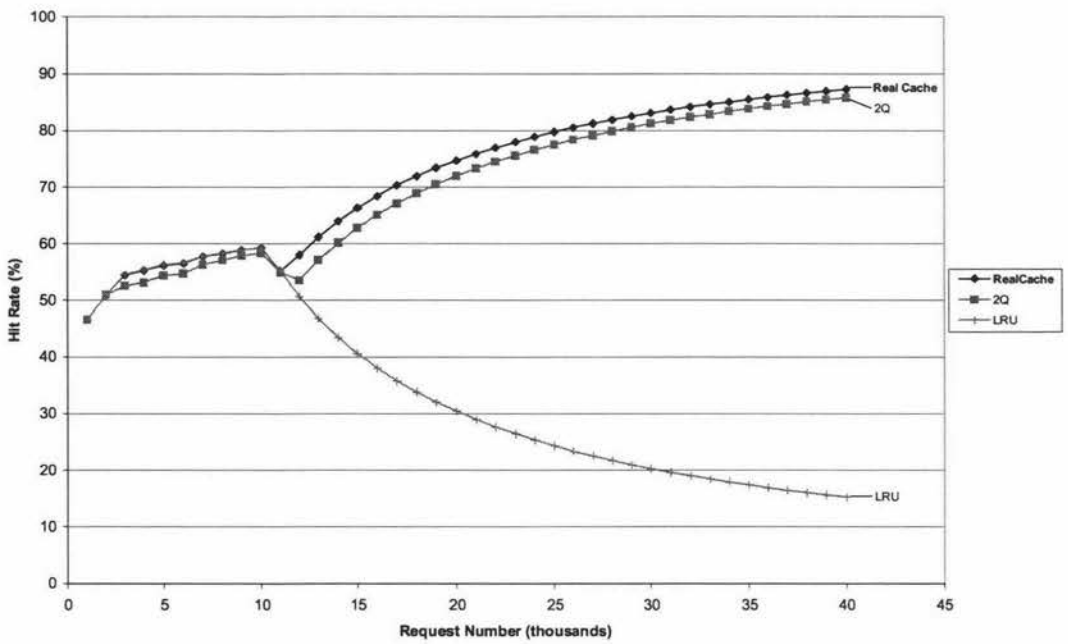


Figure 9 - Request vs Hit rate (2Q and LRU only), Synthetic Trace A

As is shown in Figure 9 above, LRU starts off showing a higher hit rate than 2Q for the start of the trace. 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 current best policy which is now 2Q. This illustrates the adaptive nature of ACME.

This points to the fact that having other policies would ideally result in having a 'safer' policy pool, in that, if at any one point in time, 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.

Figure 10 below further highlights the adaptive behaviour of ACME which shows the hit rates when only LRU and LRU-2 are used in the policy pool over the OLTP trace. The methodology for this experiment was outlined in Sub-Section 5.2.8.

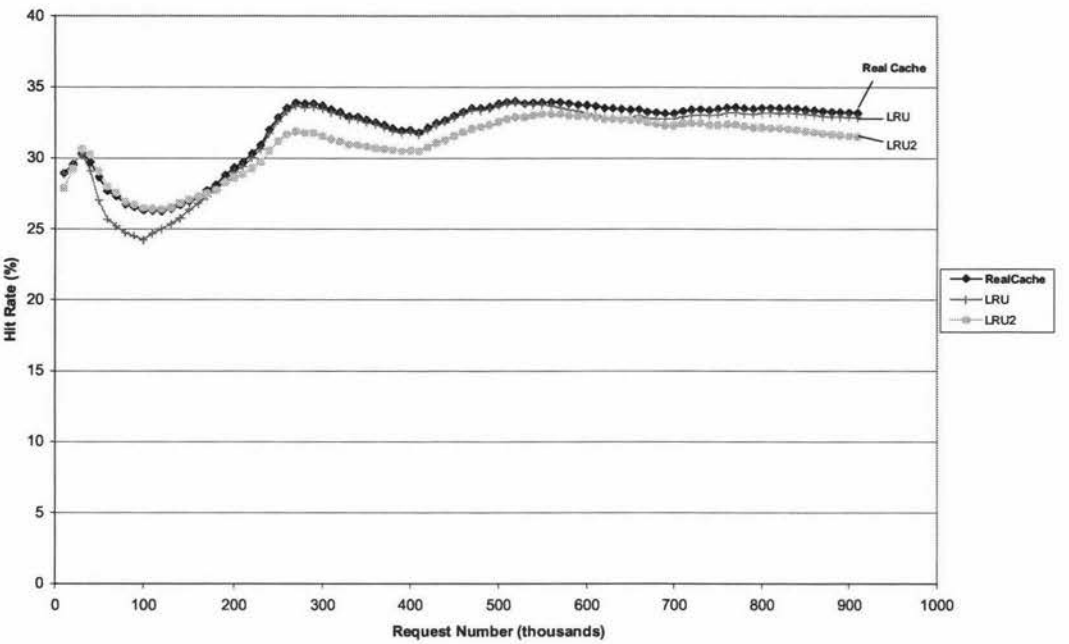


Figure 10 - Request vs Hit rate (LRU-2 and LRU only), OLTP Trace

Once again, it can be seen that the Real Cache adapts to the current best policy when the LRU-2 hit rate crosses over LRU and becomes the current best policy.

5.3.4 The effect of having different cache sizes

The live traces were used to drive the database-adapted ACME implementation with a number of different cache sizes to determine the effect of using different cache sizes on the ACME Real Cache. The results are shown in Figures 11 and 12 below which show the hit rate relative to the cache size for the DB2 and OLTP traces respectively.

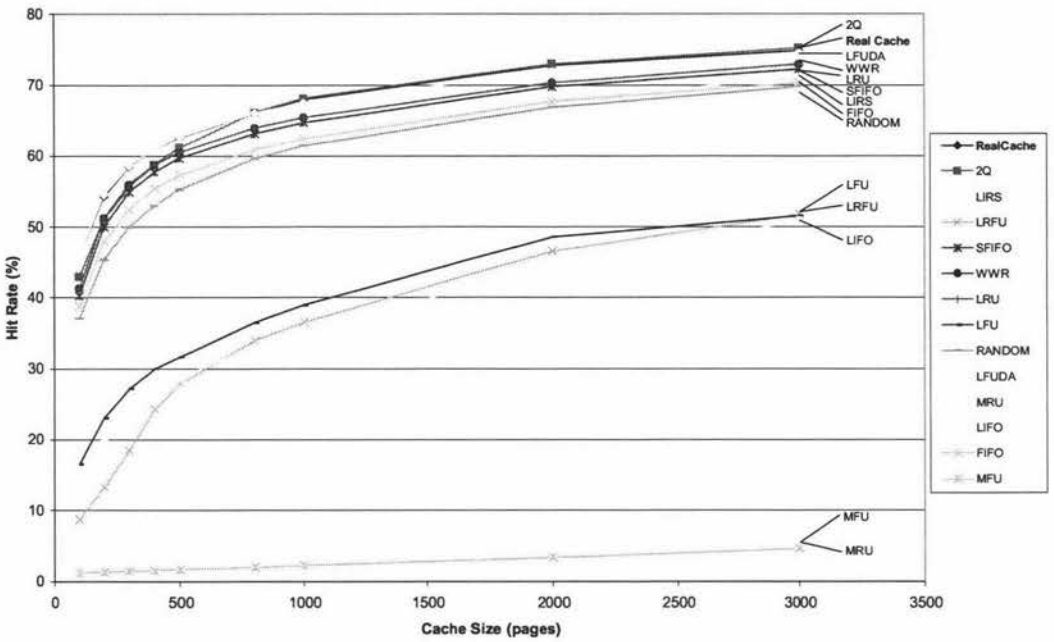


Figure 11 - Cache Size vs Hit rate (all policies), DB2 Trace

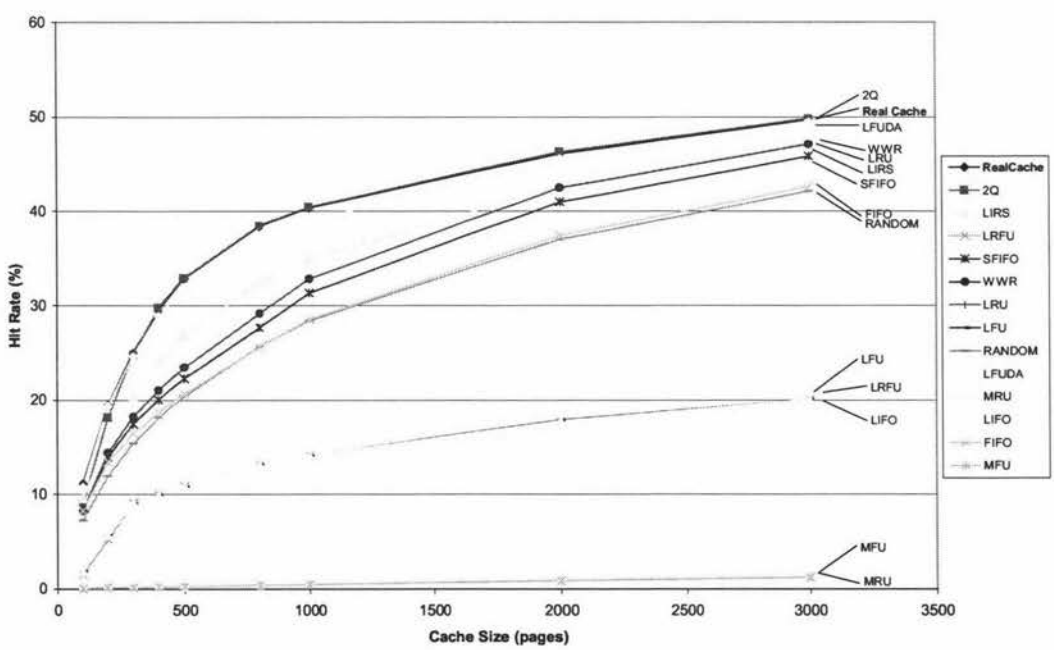


Figure 12 - Cache Size vs Hit rate (all policies), OLTP Trace

The effect of cache size on the Real Cache hit rate is summarised for each of the two live traces in Table 2 below.

Cache Size	Real Cache Hit Rates (%)			
	DB2		*OLTP*	
	Value	% Increase	Value	% Increase
100	46.1	--	11.2	--
200	54.3	18.0	19.4	74.0
300	58.6	7.8	25.0	28.7
400	61.0	4.2	29.5	18.0
500	62.7	2.7	32.9	11.5
800	66.1	5.5	38.4	16.8
1000	68.0	2.9	40.3	5.1
2000	72.7	7.0	46.1	14.3
3000	74.9	3.0	49.7	7.7

Table 2 - Real Cache hit rates for different cache sizes

The data displayed in Table 2 above is also shown diagrammatically in Figure 13 below.

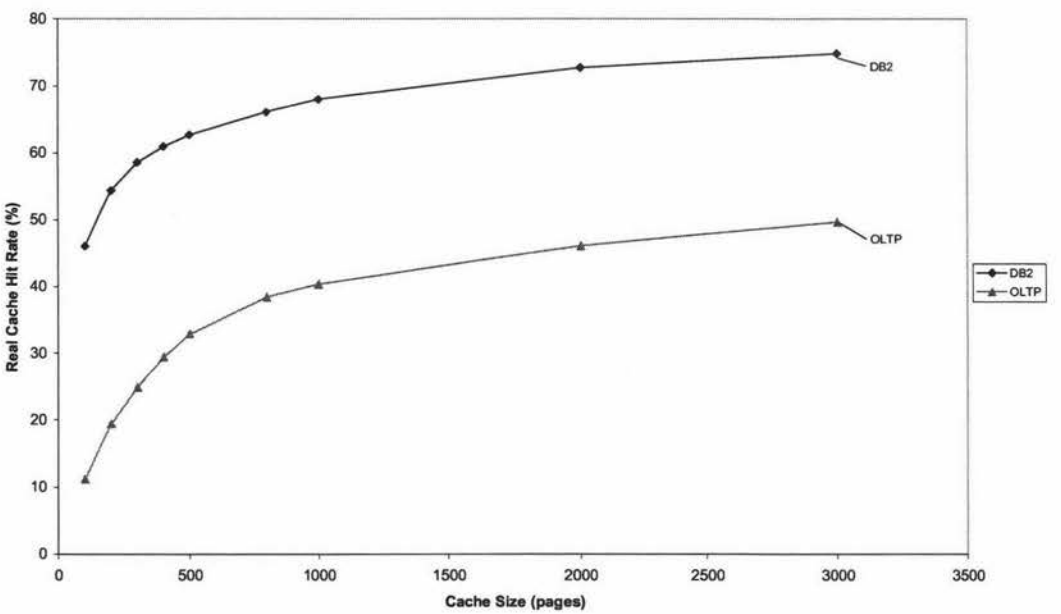


Figure 13 - The effect of cache size on the Real Cache hit rate

Figures 11, 12 and 13 cumulatively show that the Real Cache hit rate and the individual policies' hit rates increase as the cache size is increased. In the case of the Real Cache, this is due to the fact that the Real Cache follows the hit rate of the best policy in the policy pool, even when the cache size is changed. However, increasing the cache size also increases the time that is required to process the requests, more so

due to the fact that the time is proportionally related to the number of policies that are used in the policy pool (as each of the virtual caches perform their own caching as well).

It can also be seen in Figures 11 and 12 above and from the data in Table 2 above that whilst the percentage increase in the hit rate between cache sizes is significant when the cache size is in the hundreds, once the cache size reaches the thousands the hit rate begins to stabilise and level off, as shown by the relatively smaller percentage increases in hit rate. This suggests that increasing the cache size further will not result in significantly better hit rates.

If there is no significant increase in hit rate with an increase in cache size then it is possible that there is an optimal cache size, since it is desirable to get the maximum number of hits whilst spending as little time as possible searching for victims should a miss occur.

The time required by ACME to process requests is directly attributable to the time it takes for an arbitrary policy in the policy pool to select a page as the victim. As the cache size is increased, the time it takes to search for a particular page in the buffer pool increases as there are more pages to search through. This, in turn, results in more time being spent going through the request stream. In order to examine any benefits gained from increasing the cache size after there appears to be little increase in the hit rate, it is necessary to determine the time it takes to select a victim upon subsequent increases in cache size.

Table 3 below shows the increase in the time it takes for ACME to select a victim to be replaced from the Real Cache in the event of a miss, for increasing cache sizes. For the details of the methodology used for this experiment, please refer to Sub-Section 5.2.6.

Cache Size	Average Time to Select a Victim (*microseconds*)			
	DB2		*OLTP*	
	Absolute Value	Incremental Increase	Absolute Value	Incremental Increase
100	95.7	95.7	105.1	105.1
200	253.9	158.2	289.4	184.3
300	398.2	144.3	340.5	51
400	506.2	108	177.9	-162.5
500	652.7	146.5	218.6	40.7
800	1061.8	409	919.7	701.1
1000	1724.9	663.2	1743.7	824
2000	3246.7	1521.8	4243.3	2499.6
2500	18556.3	15309.6	3792.0	-451.3
3000	21070.1	2513.8	5668.9	1876.9
3500	16740.3	-4329.8	9195.4	3526.5

Table 3 - Average time to select a victim for different cache sizes

The data displayed in Table 3 above is also shown diagrammatically in Figure 14 below.

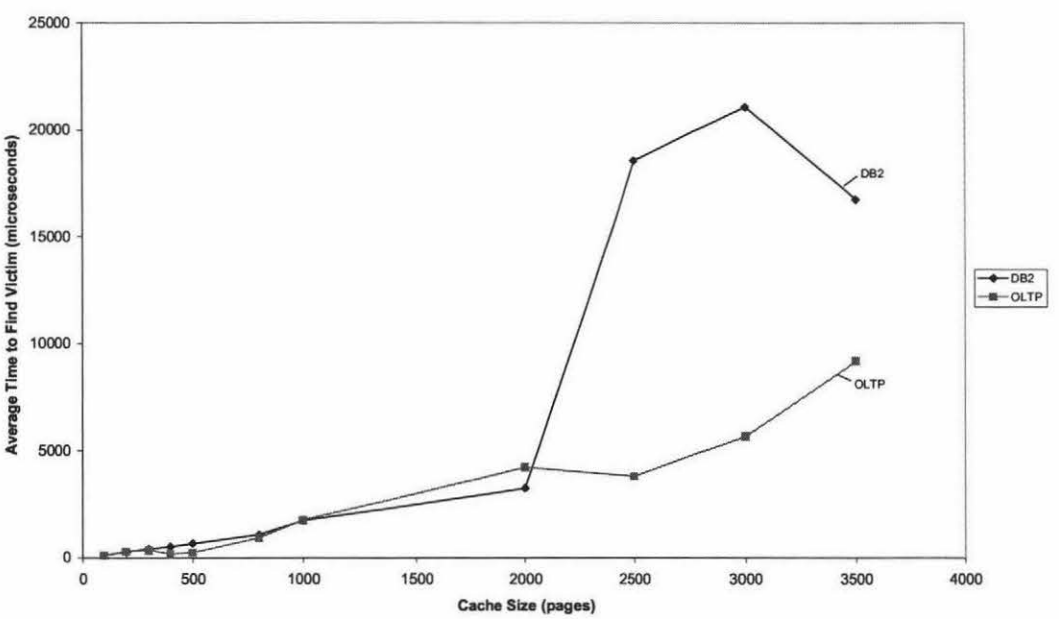


Figure 14 - The effect of cache size on the time taken to select a victim

Figure 14 above shows the average time to select a victim with each of the live traces for different cache sizes. The average time was calculated by measuring the time for:

1) ACME to randomly select a policy based on its weight; and

2) using the selected policy to uncache a page from the Real Cache based on that policy's uncaching criteria; and
3) the rest of the policies in the policy pool to release the selected victim from their physical caches.

These activities equate to the uncaching of an arbitrary policy, and thus it can be said to be the uncaching process of ACME itself.

As can be observed by viewing Table 3 and Figure 14 above, for the DB2 trace, the increase in cache size results in a steady increase in the victim selection time until a cache size of 2000 pages. After this, increasing the cache size to 2500 pages, results in a dramatic increase in the victim selection time. The victim selection time continues to rise until a cache size of 3000 pages after which it starts to fall again. This may be because the victim selection time is dependant on the pattern of access that is being exhibited in the request stream, and with the types of policies that are being employed in the policy pool.

By considering both the victim selection time and the hit rates for cache sizes shown in Figures 13 and 14 above respectively, it appears that a cache size of 2000 pages would offer the most benefit for a request stream similar to that of the DB2 Trace. This is because any increase in hit rate of cache sizes above 2000 pages is minimal, whilst the cost of selecting a victim starts to rise rapidly.

In the case of the OLTP trace, the general trend is the same as for the DB2 trace, however there are a couple of points to note. The first point is that the victim selection time decreases when the cache size is increased from 300 to 400. The increase in cache size from 400 to 500 also has a lower absolute time than for a cache size of 200. This could possibly be a result of the type of access patterns exhibited by the OLTP request stream in particular. As has been noted elsewhere [14], the accesses inherent in the OLTP trace exhibit an extremely high access skew for the most commonly referenced pages. Specifically, it was noted that 3% of the pages accessed in the trace were accessed 40% of the time. Thus, once the cache size is large enough to accommodate those pages which are referenced most often, there is little benefit in increasing the cache size.

The second point to note with the OLTP results is that they do not show the same dramatic increase in victim selection time when the cache size is increased from 2000 to 3000 pages, as for DB2. Once again, this may be attributable to the difference in patterns of access between the two traces. It is also apparent that the victim selection time for the OLTP trace starts to rise more steeply than before after a cache size of 3000 pages. Taking these considerations into account for the OLTP trace, it appears that the best cache size is around 2500 pages, where any greater increase in cache size does not achieve much more in terms of hit rates, but the time to select a victim is still low.

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 15 below illustrates this point by showing the additional time each policy adds to the total time it takes for the database-adapted ACME to process a request stream of 50,000 requests as described in the methodology outlined in Sub-Section 5.2.7.

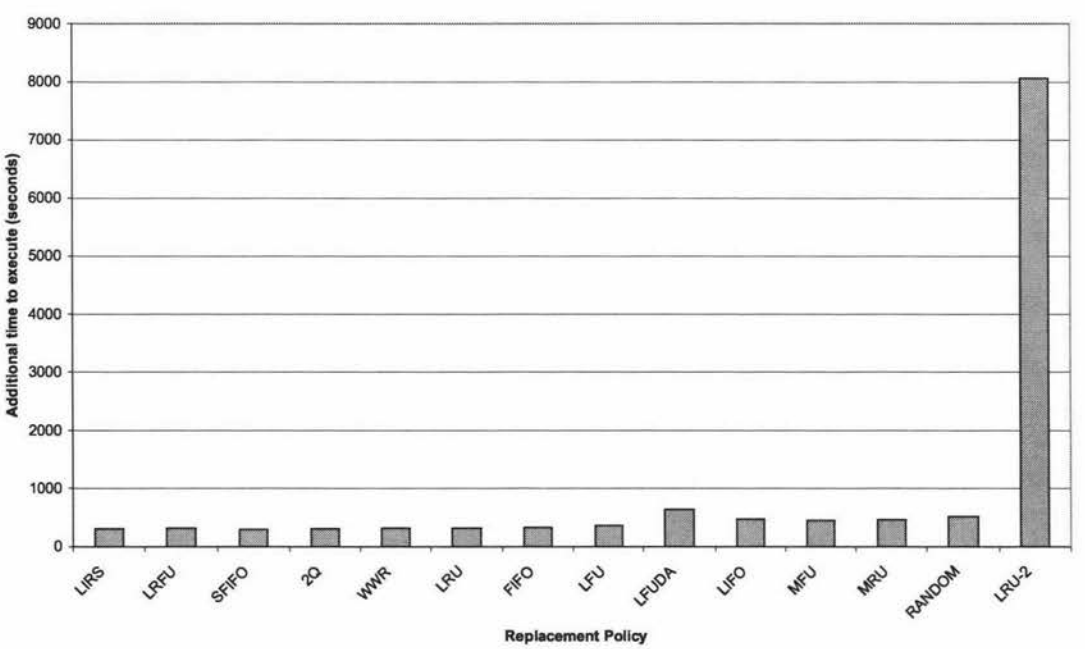


Figure 15 - The increase in time by adding each policy to the policy pool

As can be seen from Figure 15 above, all policies (with the exception of LRU-2) increase the total time for ACME to process the request stream in similar magnitude (around 300 to 500 seconds). Of this subset, LFUDA increased the time the most - by around 600 seconds (10 minutes). However, to put this in perspective, it can be seen that LRU-2 increased the total time by over 8000 seconds (more than 2 hours), which is 13 times slower than LFUDA. This certainly brings into question the practicalities of using LRU-2, especially given that its hit rate was more than ten percent less than LFUDA's hit rate for both live traces (see Figures 3 and 4). Thus LRU-2's performance in this respect does not warrant its inclusion in the policy pool. Indeed without the presence of LRU-2 the database-adapted ACME is not significantly slowed down by any one of the other policies in the policy pool.

2Q, which has been shown previously in this section to perform relatively on par with LFUDA using the live traces, only adds half as much time (approximately 300 seconds) as LFUDA does to the overall running time and therefore would arguably make it better overall when using request streams similar to those of the DB2 and OLTP traces.

5.3.6 Machine learning takes time to take effect

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. This is a function of the machine-learning algorithm, which 'learns' as the individual policies have their relative weights adjusted depending on their hit and miss history. 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.

This phenomenon is observable by viewing Figure 16 below, which depicts the hit rate of the Real Cache when only 2Q and LFUDA are used in the policy pool.

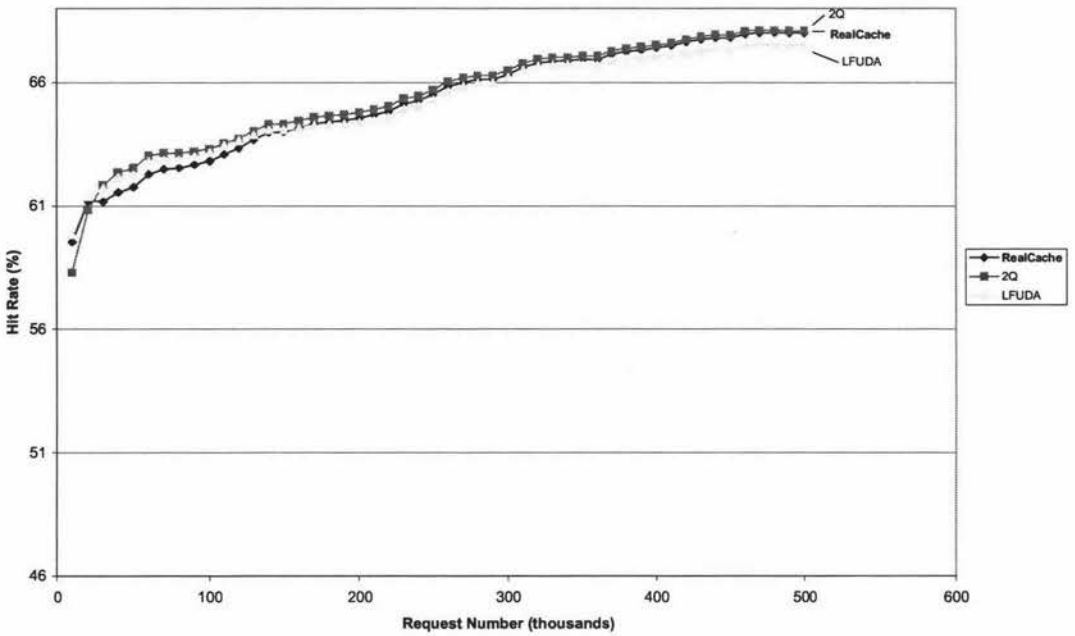


Figure 16 - The gradual adaptation of the Real Cache to the best policy, DB2 trace

Figure 16 above shows the Real Cache initially adapting to the current best policy, LFUDA until the 30,000 request mark, when the 2Q policy obtains the same hit rate as LFUDA. At this point, however, the Real Cache is below the hit rates of both policies. This is a result of the random selection of either policy in the policy cache, and reflects the fact that the machine learning algorithm is in the process of changing the weights of both policies to favour the new current best policy. However, the Real Cache takes quite a while to fully adapt to 2Q. This is because the difference between the hit rates of both 2Q and LFUDA is very small (less than 0.5%), and as such, 2Q's allocated weight is not significantly high enough relative to LFUDA's allocated weight to have an immediate effect on the Real Cache. This means both policies have almost equal weighting at the start, and only through time, does 2Q's slight advantage begin to accumulate and start to have a more obvious effect on the Real Cache.

This indicates that having two or more policies that are very close to each other in respect to their hit rates, will cause the machine learning algorithm to take a considerable amount of time to adapt to the current best policy. Compare this to Figures 5 and 6 (in Sub-Section 5.3.2 above) where the difference in the hit rates between the policies running in the policy pool was greater, and therefore the

machine learning algorithm was able to discriminate between the policies and adjust the weights allocated to each much faster, which resulted in the Real Cache immediately adapting to the current best policy. This was also the case when the current best policy changed from LRU to 2Q in Figure 9 (Sub-Section 5.3.3 above).

5.3.7 Effect on all policies by introducing well-known susceptibilities

Figure 17 below shows the effect of sequential flooding on the policies in the policy pool. Specifically, Synthetic Trace A was used to highlight the sequential flooding patterns to which LRU is susceptible.

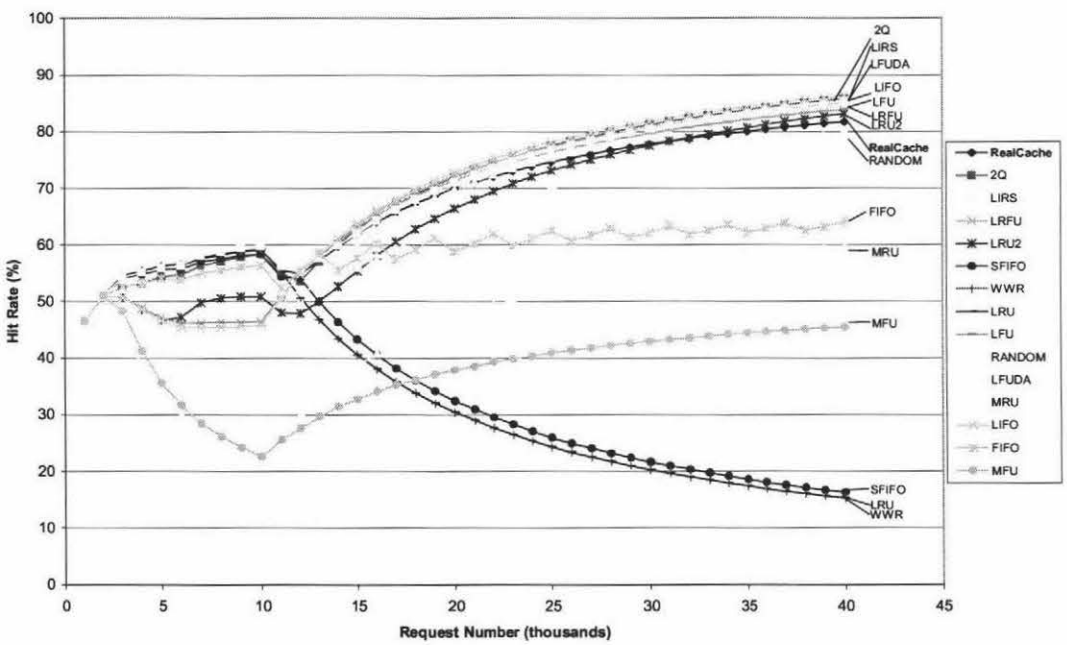


Figure 17 - The effect of sequential flooding on the policy pool

As Figure 17 above shows, the hit rates for all of the policies dramatically change at 10,000 requests. This is because the sequential flooding patterns are introduced into the request stream at this point. As expected, the hit rate for LRU, and W^2R (which is based on LRU in this implementation) significantly degrades. 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, and this is because most of these policies have been designed specifically to avoid this susceptibility, in addition to having other benefits. The erratic behaviour of the FIFO policy is due to the fact that the partial DB2 Trace that is included as 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. This means that the contents of the buffer pool at the point of encountering the sequential flooding includes some of those pages to be referenced. So when these pages are referenced during the sequential flooding part of the trace, the hit rates of the FIFO trace increases temporarily, and when other pages are not found the hit rates decrease again.

Figure 18 below shows the effect of sequential scans on the policies in the policy pool. Specifically, Synthetic Trace D used here was designed to highlight the sequential scanning patterns to which FIFO is susceptible.

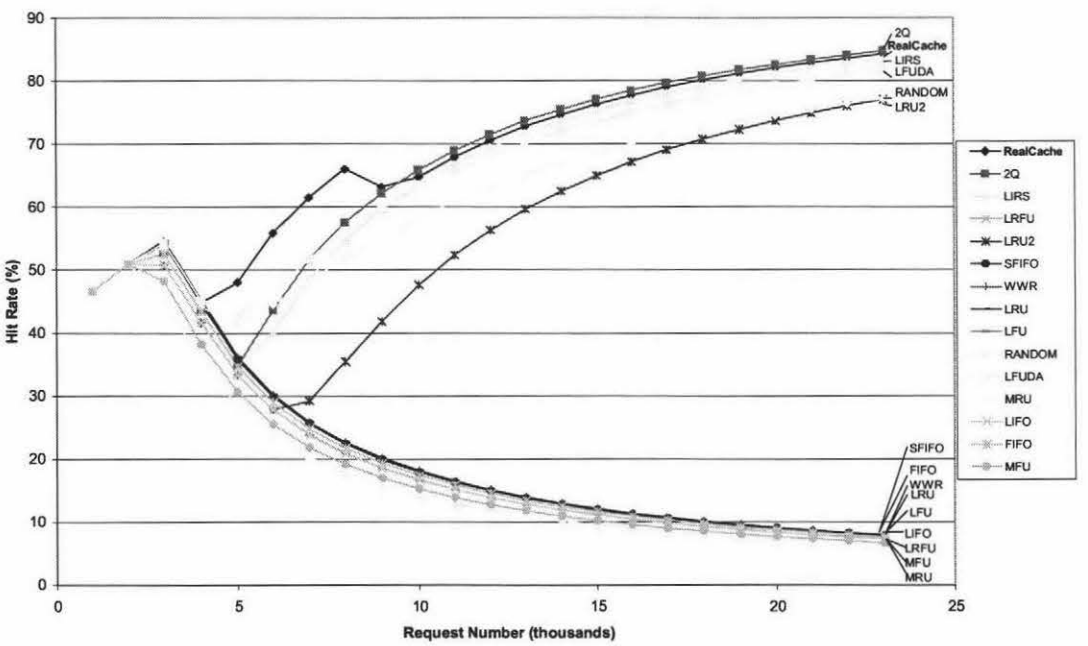


Figure 18 - The effect of sequential scans on the policy pool

The sequential scanning pattern has been introduced into this trace after about 3,000 requests, and this can be observed from Figure 18 above. In contrast to the sequential flooding effect, where some policies showed immediate degradation, whilst others

showed immediate improvement, the sequential scanning effect causes an immediate degradation in hit rates for all of the policies.

The Real Cache, and the LFUDA and Random policies are the first to recover from the degradation, followed by 2Q and LIRS. LRU-2 is the last policy to recover, whilst the rest of the policies' hit rates continue to degrade, without recovering.

It can also be observed that between 4,000 requests and 8,000 requests, the Real Cache is performing much better than all of the other policies, and this could be due to the fact that when the LFUDA and Random policies start to recover from the degradation, their weights are immediately adjusted as they are now starting to achieve more hits on the buffer pool, whilst the other policies are experiencing increasing misses. This sudden shift in trend and the effect of the rapidly increasing 2Q hit rate means that the Real Cache is temporarily achieving better results than all the other policies, until it finally adapts to 2Q.

Figure 19 below shows the effect of skewed high reference frequencies on the policies in the policy pool. Specifically, Synthetic Trace C used here was designed to highlight the frequency patterns to which LFU is susceptible.

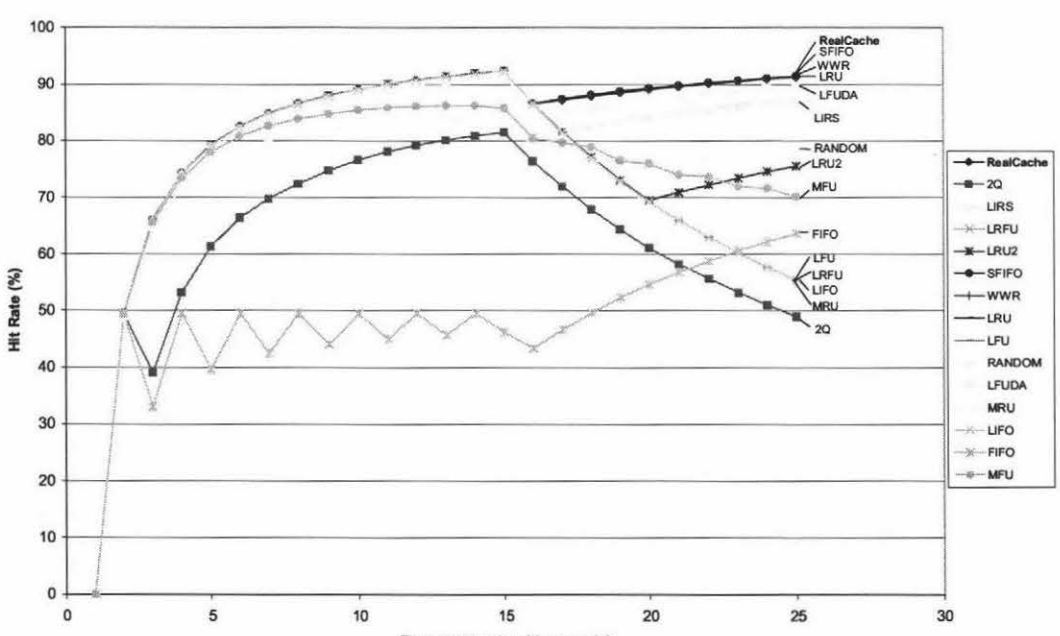


Figure 19 - The effect of skewed high reference frequencies on the policy pool

In this test, the skewed high reference frequency patterns were introduced at 15,000 requests, which is when all of the policies start to show an immediate degradation of their respective hit rates. The Real Cache, SFIFO, W^2R, and LRU are the first to recover, followed by LFUDA, LIRS, and Random. LRU-2 is the last to recover. The rest of the policies continue to degrade.

Interestingly, the 2Q policy, which has been the best performing policy with all of the tests so far, has not performed well at all for this particular trace. In fact, it belongs to the group of those policies that continued to decline when it encountered the skewed high reference frequency patterns. It is in fact, the worst performing policy.

This observation leads to the supposition that 2Q is also susceptible to skewed high frequency patterns in request streams. This supports the argument that was stated in Sub-Section 5.3.3, and is repeated here for the purposes of clarity: having other policies would ideally result in having a 'safer' policy pool, in that, if at any one point in time, 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, at least not to the same extent.

Based on the results so far, it would appear that a good combination of policies for the policy pool would include 2Q and LRU in particular. This is 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. Additionally, neither has a significant overhead in terms of execution time, when compared to other policies such as LFUDA.

The question of which policies to use, and how many to use, is one that would be 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.

5.3.8 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 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 discussed previously, 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.

In order to gauge the effect on the total time required for processing requests, including the time for disk reads, Synthetic Trace E was created by combining the skewed high reference request trace and the sequential flooding trace from the tests in the previous sub-sections. By combining these two traces, it was intended to induce the best and worst behaviour of the SFIFO and 2Q algorithms at opposite ends of the spectrum. The methodology for this experiment has been described in Sub-Section 5.2.9. The results are presented in Table 4 below.

Policies in Policy Pool	Time to execute without disk reads (seconds)	Number of misses	Time to execute including disk reads (seconds)
2Q	12	28197	**293.97**
SFIFO	8	58196	**589.96**
2Q & SFIFO	18	4852	**66.52**

Table 4 - Execution times with and without disk reads for 2Q and SFIFO, Synthetic Trace E

The above table shows the policies that were used in the policy pool, along with the time taken to execute the trace, the number of misses on the Real Cache, and the time to execute including disk reads. The column to note is the last one, which shows clearly 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. There is an overall gain of 227.45 seconds over 96,000 requests.

The above experiment has shown that when ACME encounters a request stream where the current best policy changes, the overall execution time (taking into account the disk reads avoided by maintaining a higher hit rate) decreases substantially. So what of the situation, as has been observed with the live traces previously, where the current best policy tends to be the same policy over a long period of time, or even over the entire request stream? In this case, what would the eventual loss in performance be by using ACME, 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 5 below.

Policies in Policy Pool	Time to execute without disk reads (seconds)	Number of misses	Time to execute including disk reads (seconds)
2Q	15	367020	**382.02**
SFIFO	4	392900	**396.9**
2Q & SFIFO	34	371170	**405.17**

Table 5 - Execution times with and without disk reads for 2Q and SFIFO, DB2 Trace

The results show that running 2Q by itself results in the quickest execution time overall. The slowest execution time is with running SFIFO and 2Q together, with an execution time of 405.17 seconds. The loss in time is 23.15 seconds over 100,000 requests, compared to the 227.45 seconds gained in the previous experiment.

There are a few points to consider with these results. The first point is that the loss in time is only a small fraction of the time that is potentially gained by using ACME in the case of encountering a request pattern to which 2Q is susceptible, as with the results shown in Table 4 above. The second point to note is that, if in fact, 2Q continues to be the current best policy, the Real Cache's hit rate will move closer and closer to 2Q's hit rate (as was demonstrated in Sub-Sections 5.3.3 and 5.3.6). This

would mean that 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 the hypothesis that the net gain by introducing both policies would indeed result in better overall performance. This is especially true in cases where the request stream exhibits differing patterns of access with time.

5.3.9 Flaw in the original ACME implementation

The original ACME implementation [1] included a logic flaw that needed to be corrected. This flaw was present in the part of the implementation that assigned votes to the experts in the policy pool, after it had queried each one to determine which of them had cached the requested page in its virtual cache. The flag requiring an expert to uncache a page to make space for the requested page was not being reset.

This resulted in the expert's vote not being correctly updated to reflect a miss in the virtual cache, and the uncaching of a victim and the caching of the requested page was not being performed. This series of events would only occur if one or more policies in the policy pool had a miss after an earlier policy had registered a hit, and had subsequently switched on the flag, which then (incorrectly) stayed switched on for the rest of the policies.

This would mean the weights of some policies would not be correct, and the machine learning algorithm would have allocated incorrect updates to the weights, resulting in a skewed result. In the case with the database-adapted ACME results, the hit rates for the LIRS algorithm seemed far better than the rest of the experts in the policy pool. Figure 20 below shows the performance of the 2Q, LIRS, and Real Cache hit rates when run on the DB2 trace using the flawed implementation. Figure 21 below shows the performance of the 2Q, LIRS, and Real Cache hit rates when run on the DB2 trace using the corrected implementation.

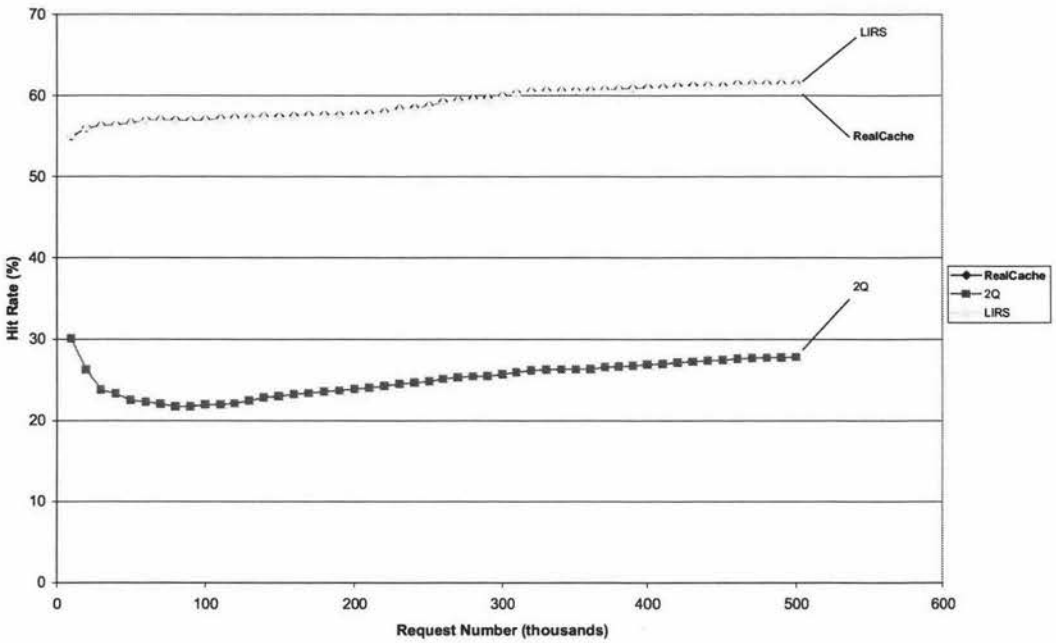


Figure 20 - Flawed results showing Request vs Hit rate (2Q and LIRS only), DB2 Trace

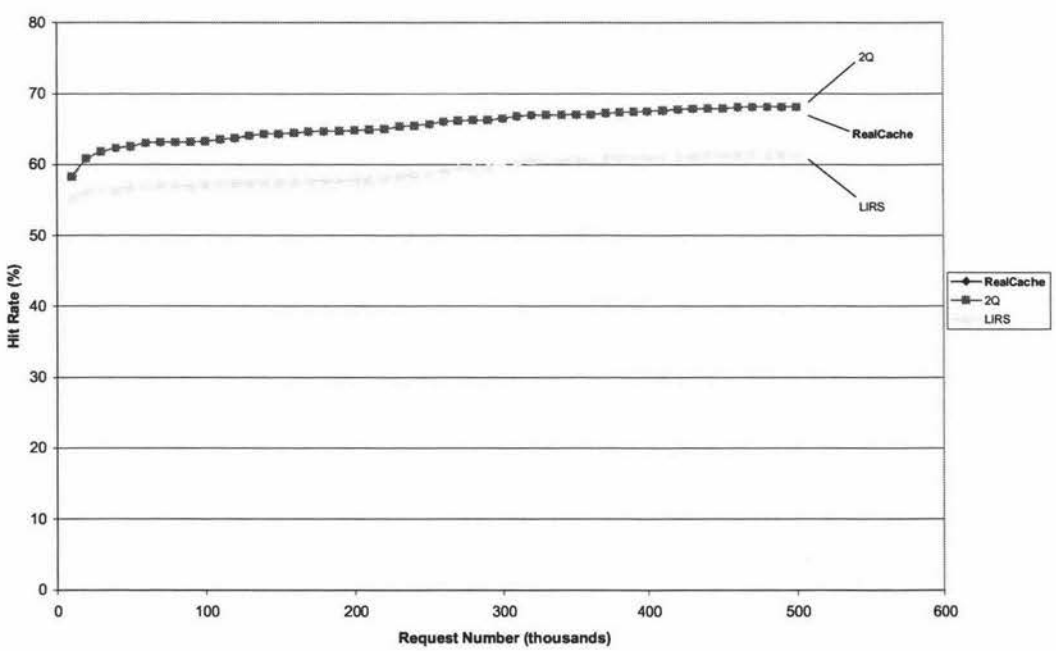


Figure 21 - Corrected results showing Requests vs Hit rate (2Q and LIRS only), DB2 Trace

As is apparent from the two figures above, the 2Q hit rates have been adversely affected by the flawed implementation, and 2Q does not achieve its potential hit rate. This results in the Real Cache being only as good as the LIRS hit rate, which is less than what it could possibly achieve had the implementation not been flawed.

As is shown in Figure 20 above, once the flaw had been corrected, the 2Q algorithm achieved its potential, and this meant that the Real Cache achieved higher hit rates as well.

5.3.10 Outcomes of research

When all the above results are taken into consideration several conclusions may be made. Firstly, it has been shown that ACME can indeed be successfully implemented in a database caching environment and that the addition of the 2Q policy in particular has increased the hit rate of the Real Cache.

Secondly, it has been clearly shown that the Real Cache hit rate closely resembles the hit rate of the current best policy, and if the current best policy changes then the Real Cache adapts to the new current best policy and changes its hit rate. This demonstrates the adaptive nature of ACME and that this trait is not lost when the algorithm is adapted to the database environment.

Thirdly, it has been shown that whilst for some traces it may appear that there is no apparent advantage to running ACME with a number of policies, due to the fact that one or two policies perform much better than others, there are times when those better performing policies become susceptible to certain patterns of requests. This shows that with carefully chosen policies in the policy pool, it is possible to maintain a higher overall hit rate for the buffer pool than would be possible with a single static policy for differing patterns of access. Furthermore, in this situation it means that there are less misses on the buffer pool, implying less overhead incurred in terms of reading from disk.

Finally, it has also been shown that more research into the effects of the optimal buffer pool size, the optimal number of policies used in the policy pool, and which policies should be used, would be beneficial in determining the best case scenario for implementing an efficient ACME based buffer manager.

6. Future Prospects

In order to further examine the integration of ACME into a real database buffer, other factors would also need to be examined.

The first factor to examine would be the memory requirements for the virtual caches used by the policies in the ACME's policy pool. Although the virtual caches store only references to the pages on disk, it may still be possible that by adding a few policies to the policy pool, the total buffer space available would decrease by so much as to have a significant impact on the hit rate performance. However, there could possibly be a trade-off between having a certain number of virtual caches using up vital buffering space and getting an overall improved hit rate performance.

The implementation so far has not considered the latency involved in writing pages from the buffer pool back to secondary storage. Writing to secondary storage usually takes longer than reading from secondary storage, and this would have an impact on the performance of ACME. However, it should be kept in mind that if a higher hit rate on the buffer pool means less I/O, writing to secondary storage would only occur when necessary. However, this also needs further investigation.

The implementation has also been restricted to working with the "normal" database environment, and does not take into account the requirements for priority-based databases, real-time databases, or databases that allow the user processes to submit working-set information. The types of replacement policies that apply to these types of databases use additional criteria or working environments such as working sets [4], priority hints [9], and Real-Time Active Database Systems (RTADB) [7]. The buffer managers used in these database environments may also benefit through using ACME.

Another direction that could be taken by further research would be to implement threads and parallel processing with the use of clusters to perform ACME based caching across a distributed database. This would mean that each policy in the policy pool could be installed on individual nodes of the cluster, performing their virtual

caching on their respective nodes in parallel, and propagate their results to the nodes that control the real cache.

7. Conclusions

Finding a cache replacement algorithm that performs as closely as possible to the optimal replacement scheme is a challenge that is frequently being pursued. Adaptive algorithms have recently been introduced in order to meet the challenge by combining the benefits of existing algorithms. The work described in this dissertation has looked at the way in which the recently proposed adaptive algorithm, known as Adaptive Caching with Multiple Experts (ACME), can be successfully adapted to 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. This work has also shown the inclusion of other policies besides the current best policy in ACME's policy pool neither negatively nor positively affects the hit rate of the Real Cache. Most significantly, it has also been shown that whilst adding more policies to the policy pool increases the execution time, the overall time to process requests can be dramatically improved. This is due to the fact that less overall misses on the Real Cache means lower latency delays from disk reads. This is particularly true for access patterns that expose weaknesses in some static policies.

8. References

[1] I. Ari, "Storage Embedded Networks (SEN) and Adaptive Caching using Multiple Experts (ACME), Ph.D. Proposal , May 2002.

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

[3] P. Cao and S. Irani, "GreedyDual-Size: A Cost-Aware WWW Proxy Caching Algorithm", *Proceedings of the Second Web Caching Workshop*, Colorado, 1997.

[4] R. W. Carr and J. L. Hennessy, "WSCLOCK - A Simple and Effective Algorithm for Virtual Memory Management", *Proceedings of the 8th SOSP Conference, Operating Systems Review*, Vol. 15, No. 5, December 1981, p. 87.

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

[6] L. Cherdasova, "Improving WWW Proxies Performance with Greedy Dual Size Frequency Caching Policy", *Technical Report, HP Laboratories*, November 1998.

[7] A. Datta, S. Mukherjee and I.R. Viguier, "Buffer Management in Real-Time Active Database Systems", *The Journal of Systems and Software*, Vol. 42, No. 3, 1998, pp. 227-246.

[8] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management", *ACM Transactions on Database Systems*, Vol. 9, No. 4, 1984, pp. 560 – 595.

[9] R. Jauhari, M. Carey, and M. Livny, "Priority-Hints: An Algorithm for Priority-Based Buffer Management", *Proceedings of the 16th Int'l VLDB Conference*, Brisbane, Australia, August 1990.

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

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

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

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

[14] E. J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering", *Proceedings of ACM MOD International Conference on Management of Data*, 1993, pp. 297 – 306.

[15] R. Ramakrishnan and J. Gehrke, "Database Management Systems", McGraw Hill, 2000.

[16] G. M. Sacco and M. Schkolnick, "Buffer Management in Relational Database Systems", *ACM Transactions on Database Systems*, Vol. 11, 1986, pp. 473 – 498.

[17] R. Turner and H. Levy, "Segmented FIFO Page Replacement", *Proceedings of ACM SIGMETRICS '81*, 1981, pp. 48 – 51.

[18] N. Young, "The k-server Dual and Loose Competitiveness for Paging", *Algorithmica*, Vol. 11, No.6, June, 1994, pp. 525-41. Rewritten version of "On-line Caching as Cache Size Varies", *The 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, 1991, pp. 241-250.