

Implementing a Better Cache Replacement Algorithm in Apache Derby

Progress Report

Gokul Soundararajan

August 19, 2006

Abstract

The desire to have large amounts of memory with small access latency is desired by all. Caches provide this effect in a manageable fashion. In databases, the buffer pool acts as a cache to reduce the latency to access disk. However, the benefit of caching depends on the effectiveness of the replacement algorithm. In this report, I summarize my work over the past few months on Derby on different cache replacement algorithms and my two implementations. Through simple experiments, I show that both implementations provide better hit rates when compared to the existing Clock algorithm.

1 Introduction

There is speed to size tradeoff in memory hierarchies. This extends from computer architecture which employs the cache hierarchy of the the fast L1 cache to progressively slower L2, L3, and main memory. This structure is mirrored in software stacks as well. In databases, the time to access disk is often the limiting the limiting factor in database's throughput. To mitigate the long latencies, the buffer pool acts as a cache in memory that helps in reducing access time. To obtain high performance, databases cache items from disk in memory to reduce access time. The problem stated is that the existing replacement algorithm performs poorly so we need to research new algorithms for cache replacement and implement it in Apache Derby.

In this report, I document my research in various cache replacement algorithms and present a progress report evaluating the benefits of different designs of Derby's `CacheManager`. The rest of the report is structured as follows. Section 2 presents a background on the benefits of caching and performance issues of different algorithms. Section 3 presents the two designs I explored in my work. In Section 4, I present experimental results of simulation and a small benchmark evaluating the benefits of the new design. Finally, Section 5 presents conclusions and presents future work.

2 Background

In this section, I provide some background on the operation and benefits of using caches. In addition, I highlight the well known cache replacement algorithms.

2.1 Caches

As programmers, we would like to have access to vast amounts of memory as fast as possible. However, the cost of providing this would be astronomical. *Caching* is a design technique that provide the benefits of vast amounts of memory in a more economical manner. Caches exploit the property of *locality* which states that applications tend to re-use data which they have recently used. This is seen in two forms: *temporal* locality and *spatial* locality. Temporal locality states that data used now will be reused in the immediate future. Spatial locality states that if a piece of data is accessed now then, its neighbors will be accessed in the near future. Since I am focusing on cache replacement algorithms, we see how the property of temporal locality is exploited to give good cache performance.

2.2 Cache Replacement Algorithms

There have been several cache replacement algorithms proposed [1, 2, 3, 4, 5, 6]. Of these, I will describe a few which I implement in Derby. First, I will describe the LRU algorithm and the Clock algorithm. These two well known algorithms are easy to understand and provide a good background to understand the more advanced replacement algorithms. However, both these algorithms suffer if the access pattern does not exhibit temporal locality. One example is a *scan*. A memory scan is a sequence of one-time item access. Intuitively, items accessed during a scan should not be cached since they will not be accessed again. The next of replacement algorithms like the 2Q, ClockPro, and Approximate ClockPro filter accesses with temporal locality from scans and thus are *scan-resistant*.

2.2.1 Least Recently Used

As described earlier, caches exploit an application's *temporal* locality. One algorithm that uses this concept is the Least Recently Used (LRU) algorithm. The basic premise of this algorithm is that items that have not been accessed in a long period of time are not likely to be accessed soon. Therefore, these items are candidates for eviction. Conceptually, the cache is organized as a *stack* with the top of the stack containing the *most* recently accessed item and the bottom of the stack the *least* recently accessed item. On a cache access, if the item is found in the cache (a *hit*) then the item is moved from its current location to the top of the stack. If the item does not exist in the cache (a *miss*), the least recently used item is evicted from the cache to make room. Then the accessed item is fetched from disk and placed at the top of the stack.

2.2.2 Second Chance (*Clock*)

The main drawback of the LRU stack implementation is the overhead of shifting items to the top of the stack on each cache access. To avoid this, the second chance algorithm was proposed. It provides a good approximation to the LRU algorithm. The second chance algorithm is also referred to as the *clock* algorithm.

In this algorithm, the cache items are placed in a circular list. In addition, with each cache item, a reference bit is kept. On a cache access, there are two possibilities: (1) either the item exists in the cache (a *hit*) or (2) the item is not in the cache (a *miss*). On a cache hit, the reference bit is set to 1. On a cache miss, the *clock hand* is rotated to find an item to be evicted. If the *clock hand* sees an item with reference bit set to 1 then the reference bit is set to 0 and the *clock hand* proceeds to the next item. If the *clock hand* notices an item with a reference bit set to 0 then, this item is evicted. The accessed item is brought in from disk and placed in this space in the circular list.

2.2.3 2Q

As explained before, both LRU and Clock algorithms are not *scan-resistant*. In [4], the authors present the 2Q algorithm which separates cache items into *hot* and *cold* items. This algorithm uses 2 FIFO queues ($A1_{in}$ and $A1_{out}$) and 1 LRU stack (Am). The $A1_{out}$ queue holds the identifiers of recently evicted items. In addition, it uses 2 parameters K_{in} and K_{out} to size the $A1_{in}$ and $A1_{out}$ queues respectively.

The algorithm works as follows. On a page access, if the item is found in the $A1_{in}$ then, no action is taken. If the item is found in the Am stack, it is promoted to the top of the stack. On the other hand, if the item is not found in the cache, then it is brought in from disk. If the item's identifier is found the $A1_{out}$ queue then, the item is placed in Am . Otherwise, it is placed in the $A1_{in}$ queue. To make room for the new entry, if the size of $A1_{in}$, $|A1_{in}|$ is greater than K_{in} , then the oldest item from $A1_{in}$ is evicted and its identifier is placed in the $A1_{out}$ queue. Otherwise, the oldest entry from Am is evicted. If $|A1_{out}| > K_{out}$, then the oldest identifier is removed from $A1_{out}$.

2.2.4 ClockPro

The ClockPro cache replacement algorithm described in [2] is based on the authors previous work on the LIRS scheme [3]. ClockPro shifts away from the previous approaches by using *reuse distance* rather than *recency*. Recency is defined as the number of items accessed after the last reference to that item. Following the stack model described in LRU, recency is the distance away from the top of the stack. In

contrast, the reuse distance is the number of distinct cache accesses between the last reference to this item and its current reference.

Using reuse distance, ClockPro maintains three clocks within a single clock-like data structure: (1) the hot clock, (2) the cold clock, and (3) the test clock. On a cache access, the item is placed as cold page. It stays in the clock until it runs out of its test period. The test period is defined as the largest recency of the hot pages. If the item is accessed during its test period, then the item is promoted to a hot page. Similarly, the hot page with largest recency turns into a cold page when a cold page is promoted. An item in its test period can be evicted from the cache but its identifier is maintained until the item runs out of its test period.

2.2.5 Approximate ClockPro

Based on my study of the existing cache replacement algorithms, I decided to design an approximation to ClockPro by combining the framework of 2Q with the efficient implementation of Clock. In its core, the algorithm follows 2Q. I maintain 3 clock queues: (1) hot clock (C_{hot}), (2) cold clock (C_{cold}), and (3) test clock (C_{test}). The test clock contains only cache identifiers.

The algorithm works as follows. First, I build this algorithm using several clocks so oldest/newest entries are determined by the clock algorithm. The sz refers to the total cache size out of which K_{in} are cold items. The size of C_{test} is limited to K_{out} . On a page access, if the item is found in the C_{hot} or C_{cold} , then the reference bit is set (`setUsed(true)`). If the item is not found, then there are two cases: (1) either the item's identifier is found in the test clock (C_{test}), or (2) the item's identifier is not found in C_{test} . If the identifier is found, then the new item is added to the hot clock (C_{hot}). When added to C_{hot} , if the hot clock is full, C_{hot} ($|C_{hot}| > (sz - K_{in})$), the oldest item from C_{hot} is demoted into the cold clock (C_{cold}). Similarly, if the cold clock is full, the oldest item is pushed in the test clock. The oldest item from C_{test} (the identifier) is thrown out. In the second case, the new item is put into the cold clock C_{cold} . Again, if $|C_{cold}| > K_{in}$ then the oldest entry it pushed into C_{test} . Similarly, if $|C_{test}| > K_{out}$ then, the oldest identifier from C_{test} is thrown out. The biggest benefit of this algorithm is that many operations are lightweight since they operate in a clock-like fashion.

3 Implementation

I tried two different ways of implementing a better cache replacement algorithm in Derby. `Clock` is the current cache replacement policy implemented in Derby. First, I implemented the ClockPro algorithm directly. Second, I implemented a `PluggableCache` which follows `Clock` very closely except the decision to evict an item is given by a `ReplacementPolicy`. I will highlight the overall structure of both approaches and the benefits and drawbacks of each.

3.1 ClockProCache Implementation

3.1.1 Overall Structure

The `ClockProCache` implementation strictly follows the ClockPro algorithm as explained in [2]. In this implementation, I implemented my own design but followed some of the principles used in `Clock`. Specifically, I noticed that using a `ArrayList` provides better performance than `LinkedList`. In addition, I followed the general structure from `Clock`.

3.1.2 Benefits & Drawbacks

The main drawback of my `ClockProCache` is the poor performance some due to the algorithm's limitations and some due to my inexperience with Derby.

The main cause of problem is *synchronization*. Since the replacement decision is made by coordinating the clock hands (the hot hand, the cold hand, and the test hand), all misses are serialized. This is in contrast to `Clock`'s implementation of `rotateClock()` where there exists an optimization to look at only 20% of the cache. In addition, `Clock` has an additional property that multiple threads may move the clock hands concurrently. For example, `Thread-1` may move the clock hand by 2 and `Thread-2` may be move the clock hand by 5 spaces then `Thread-1` may move the clock hand 2 spots. Another execution sequence may be that `Thread-1` moves the clock hand by 4 spots and then `Thread-2` moves

the clock hand by 5 spots. In both execution sequences, the final position of the clock hand is the same. ClockPro does not provide this property. Therefore, access to move the clock hands need to be coordinated using synchronization. There also exists a temporary problem of a *race condition* which I have not fully understood. This problem occurs very rarely so it makes it harder to debug. For example, I ran 100 regression tests and this problem happened twice.

In conclusion, several optimization passes should help in improving performance of ClockProCache.

3.2 PluggableCache Implementation

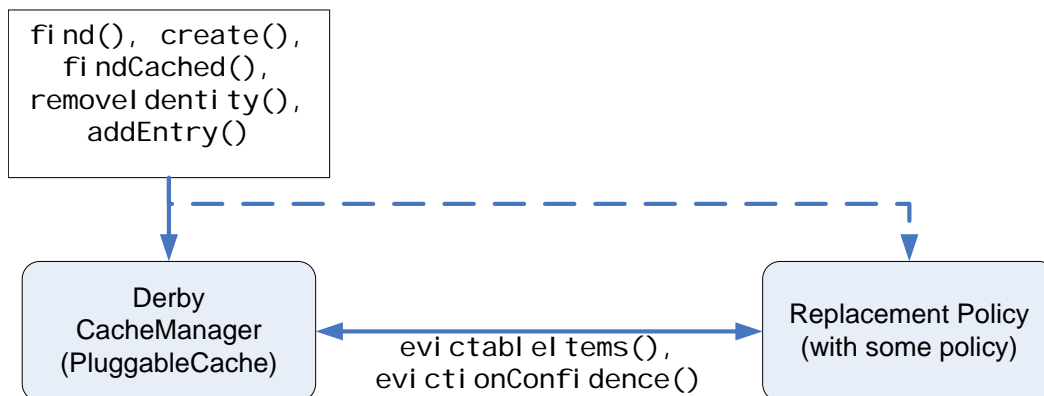


Figure 1: PluggableCache

3.2.1 Overall Structure

The PluggableCache implementation separates the cache implementation from the replacement policy. The basic idea is to maintain a *mirrored* cache in which only the cache item metadata (the identifiers) are kept. To maintain consistency, all cache access actions are duplicated. The mirrored cache updates its data structures based on a replacement policy which is *pluggable*. Basically, on every cache access from `find()`, `remove()`, `create()`, or `findCached()` I update the mirrored cache. If some item has to be removed from the mirrored cache, then the mirrored cache evicts it. Then, when the CacheManager runs `rotateClock()`, it checks whether an item passes the implementation conditions (like `!isKept()`) then calls the mirrored cache to see what to do. If the item does not exist in the mirrored cache (which means that it was evicted a while ago), then the mirrored cache says to evict the item. otherwise, no. If the engine says to evict, then the CacheManager will evict it. Otherwise, it skips the item and continues forward.

3.2.2 Benefits & Drawbacks

Having a mirrored cache is very useful in testing different replacement policies very quickly. Usually, every policy can be written in ~200-300 lines of code. In addition, a customized replacement policy can be used.

The only drawback is the overhead of updating the mirrored cache on every cache access. I did a very simple benchmarking test where I ran 100K cache accesses (by adding to the regression test `T_CacheService`) and compared the original Clock with my mirrored cache implementation running the `ClockPolicy`. In this case, there was a 10% slowdown. However, for Clock like algorithms, this overhead can be reduced by re-using the metadata from the original Clock rather than duplicating actions. By doing this, the overhead can be brought down to 1%. However, the regression test is just passing `T_Key`'s, so this test is only measuring the overhead. But, I should note that this part of the code uses only a small fraction of time used for a database operation.

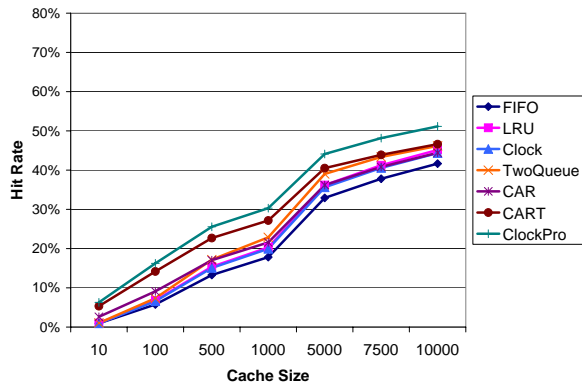


Figure 2: Simulation: Hit Rate

4 Experimental Results

In this section, I present results of simulation where I compare various cache replacement algorithms. From these results, I decided to implement ClockPro as the new replacement algorithm in Derby. Then, I present the results of running a microbenchmark using the regression test harness.

4.1 Test Configuration

The experiments are run on a dual processor Intel Pentium III running at 866 Mhz and 640 MB of RAM and a 40 GB hard drive. The operating system is Redhat Fedora Core 5 and I use Java JDK 1.4.2.11 to compile Derby. The Derby code is the latest trunk code.

4.2 Simulation

To evaluate the best algorithm to implement, I tried several well known cache replacement algorithms in a simulation. In this simulation, I tried the following algorithms: (1) First-In-First-Out (FIFO), (2) Least Recently Used (LRU), (3) Clock, (4) 2Q, (5) CAR, (6) CART, and (7) ClockPro. The data consisted of 100K items and the I ran 100K accesses in a 0.8 Zipf pattern. Figure 2 shows the results of the simulation.

4.3 Microbenchmark

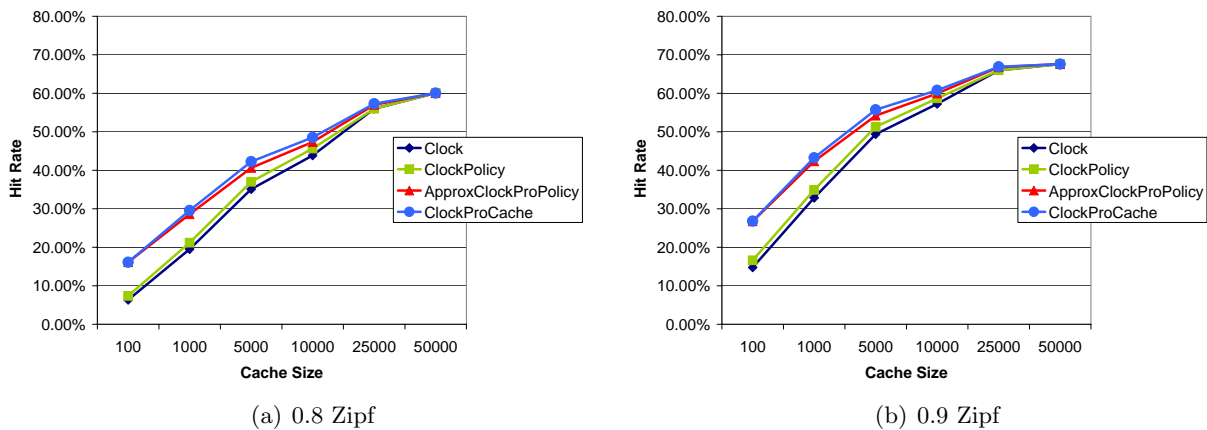


Figure 3: Microbenchmark: Hit Rates

In this experiment, I placed my own testing code with the cache service regression test `T_CacheService` unit test. All threads read from a trace file which contains identifiers in a Zipf access pattern. I ran the test with several configurations: (1) `Clock`, (2) `PluggableCache` with `ClockPolicy`, (3) `PluggableCache` with `ApproxClockProPolicy`, and (4) `ClockProCache`.

As Figure 3 shows, for both 0.8 and 0.9 Zipf access pattern, my `ApproxClockProPolicy` does comparable to `ClockProCache`. Both `ApproxClockPro` and `ClockPro` provide good benefits over the existing `Clock` algorithm. Specifically, for both the 0.8 and 0.9 Zipf patterns with a cache that can hold 1% of the entire data, the `ClockPro` and `ApproxClockPro` provide an almost 10% hit rate improvement. In addition, these two algorithms perform better until the cache is 25% of the data size. At this point, there is no difference in performance between `Clock` and `ClockPro/ApproxClockPro`.

5 Conclusions and Future Work

In this project, I aimed to implement a better cache replacement algorithm in Apache Derby. To achieve this goal, I implemented two designs. First, I implemented `ClockProCache` which implements the `ClockPro` algorithm. Second, I implemented the `PluggableCache` which supports many replacement policies. In addition, I implemented an approximation to `ClockPro` which is more lightweight and thus should have lower overhead. From simple experiments, I show that both `ClockPro` and `ApproxClockPro` achieve higher hit rates than the original `Clock` for different cache sizes and different access patterns.

There is still lots of work to do. First, I would like to optimize the code to integrate it better and lowering the overhead. Second, there are still other options on how to manage the `PluggableCache` which are good to explore. Finally, I would like to run industry standard benchmarks to measure the benefit of having a better cache replacement algorithm.

6 Acknowledgments

I would like to acknowledge the help of my mentor, Oystein Grovlen, Knut Anders Halten, Kristian Waagan, Mike Matrigali, and Andrew McIntyre and the rest of the Derby community for helping me setup my environment, understand Derby, and run experiments.

References

- [1] F. J. Corbato. A paging experiment with the multics system. *MIT Press*, pages 217–228, 1969.
- [2] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *the Annual Usenix Technical Conference (USENIX'05)*, Anaheim, CA, April 2005.
- [3] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS*, pages 31–42. ACM, 2002.
- [4] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB*, pages 439–450. Morgan Kaufmann, 1994.
- [5] Nimrod Megiddo and Dharmendra S. Modha. Outperforming lru with an adaptive replacement cache algorithm. *IEEE Computer*, 37(4):58–65, 2004.
- [6] Elizabeth J. O’Neil, Patrick E. O’Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In Peter Buneman and Sushil Jajodia, editors, *SIGMOD Conference*, pages 297–306. ACM Press, 1993.