Understanding the LIRS algorithm

Raghunath Rajachandrasekar
The Ohio State University
rajachan@cse.ohio-state.edu
Big Data Analytics and Management in Distributed Systems, Winter 2012

Abstract. The LIRS algorithm proposed by Jiang and Zhang(1) aims at overcoming the limitations of the LRU algorithm which is solely based on the assumption that recently used data will be reused. This study tries to offer a deeper understanding of the LIRS algorithm and its relationship to the LRU mechanism. This paper also discusses the merits and demerits of a N-stack LIRS algorithm, and provides a solution to identify the perfect LIR set.

1 Introduction

The LRU algorithm(2) offers a simple solution to cache block replacement problem with the fundamental assumption that a data block that was recently accessed will be reused in the near future. However, this assumption does not prove beneficial in case of workloads that exhibit weak-locality access patterns. The following reference patterns are some instances of such weak-locality data accesses:

(a) Cyclic accesses to a data-set that is marginally larger than the cache size

(b) Arbitrary bursts of accesses to an infrequently accessed data-set that pollutes the cache by replacing the more frequently used entries.

(c) Accesses to blocks with varying access frequencies

However, the LRU algorithm has been widely adopted in caching systems owing to the simplicity it offers with implementation. The history information required by the LRU mechanism to make a decision with block replacement is very limited, which allows for efficient resource usage. This has found applications in virtual memory management, file buffer caches, database buffer management systems, etc. The drawback of LRU have been addressed by several researchers and replacement algorithms have been proposed based on user-level hints, tracing and richer history information and regularity detection. But these techniques impose high overheads on the performance and voids the simplicity of the LRU scheme. The LIRS(3) algorithm tries to find a balance between the amount of history information that needs to be maintained and the performance of the caching subsystem.

2 Summary of LIRS Algorithm

The LIRS algorithm revolves around the history information of data accesses in the form of two metrics - the Inter-Reference Recency (a.k.a IRR) and the Recency. The IRR of a data block refers to the number of other distinct blocks accessed between the last two consecutive accesses of the data block in question, while recency refers to the number of other distinct blocks accessed between the last reference to the current time. In essence, the LIRS replacement algorithm tries to maintain data blocks with a low IRR in the cache as much as possible, and evicts those with a high IRR when a replacement in required.This is achieved by means of two LRU-stack like data-structures that are used to maintain recency and IRR history information. However, unlike the LRU stack which stores only those data blocks that are residing in the cache, the LIRS stack stores any and all LIRS (Low IRR Set) and HIRS (High IRR Set) data blocks whose recency is lesser than the maximum recency of the LIR blocks. With such a framework, LIRS has three design goals - a) effective utilization of multiple sources of access history information; b) dynamic and responsive selection of blocks based on its possibility of being accessed in near future; and c) minimizing implementation overheads.

3 Finding the Perfect LIRS

In order to get a deeper understanding of the IRR trends in data blocks, it is desirable to see a better organization of the LIR set. In this context, this study provides a enhancement to the 2-stack LIRS algorithm, to get a perfect LIR set. This set contains a list of data blocks ordered by their IRR values.

For this purpose, maintain a third stack (P) in addition to the 2-stacks which are part of the LIRS data-structure is maintained. This stack P holds LIR data blocks that were residing in the cache at least once in their lifetime of the workload execution. Data blocks that were in the cache with HIR status, but evicted before its status could be elevated to LIR do not have counterparts in stack P. The stack size is variable and it increases as the number of unique data accesses increases. No data block is ever evicted from this auxiliary stack. Only one of two operations are possible on stack P - inserting a new data block and moving an existing data block to the top. The behavior of this new 3-stack structure in different scenarios is discussed below:

- **Referencing a non-resident HIR block**
  When the block being referenced was not resident in the cache, it is a miss and the data has to be brought in from main memory to the top of stack Q. The element at the bottom of the stack Q is evicted. If this block was originally present in stack S, its status is changed to LIR and is also pushed to the top of stack P that will eventually hold all the LIR blocks ranked by their IRR values at a given point

in time. The stack-pruning is also triggered after moving the block to the bottom of stack S. However, if the block was not originally present in stack S before the reference, the status is unchanged and it is not pushed to the stack P.

- **Referencing a cache-resident HIR block**
  In this case, the HIR block is moved to the top of stack S. If the data block was originally in stack S as well, then its status is changed to LIR and its counterpart in P is moved to the top of the stack. The LIR at the bottom of stack S is moved to Q and a stack-pruning on S is triggered. However, if the data block was not originally present in S, then it is neither updated in stack P, nor is its status changed to be an LIR block.

- **Referencing an LIR block**
  On accessing a data-block which has already been classified by the 2-stack approach to be of LIR type, it is not only moved to the top of the LIRS stack S, but its counter part is also moved to the top of stack P. If the block in question was at the bottom of the stack S to begin with, the stack-pruning operation is triggered after pushing the block to stack P.

Differences with LIRS 2-stack algorithm

A major difference between the LIRS 2-stack algorithm and the proposed 3-stack Perfect LIRS algorithm is the space requirement. The LIRS algorithm uses a fixed size HIR stack and a variable size LIR stack to maintain all the required history information. However, with the additional requirement of a set of blocks ordered by IRR distance, the space usage has also increased. The Perfect LIRS stack P grows in size with increasing number of misses. As indicated in the discussion above, the stack P gets a new element whenever a cache miss occurs, bringing along with it a data copy from main memory to the cache. This method was proposed to maintain the characteristic and behavior of the existing 2-stack approach.'

If that is not a constraint, then the Perfect LIRS set can be obtained with a much lower space requirement by extending the LIR stack S to on-load the functionality of the stack P by letting it grow indefinitely without ever having to delete an LIR data block that got a berth in the stack. The behavior with different data access patterns would still remain as discussed above, with minor modifications. Extending the LIR stack to maintain the ranking based on IRR would also reduce the operation latency introduced by the 3-stack approach.

4 Relationship with LRU Algorithm

The LRU and LIRS algorithms have a strong relationship and a common core goal. However, the fundamental assumptions made by each of them are significantly different. The LRU algorithm is based on the assumption that a data block that has not been accessed for the longest time would wait for the longest time to be accessed again. This associates future behavior of a block with its own previous reference. On the contrary, the LIRS algorithm defines a metric, the Inter-Reference Recency (IRR), which refers to the number of other distinct blocks accessed between two consecutive references to a block, and assumes that if the IRR of a block is large then its next IRR is also expected to be large. The combination of IRR and recency information overcomes the drawbacks of LRU algorithm, which had a poor performance in case of weak-locality access patterns as discussed in Section 1.

The data-structures used by the two algorithms is another aspect to look at. The LRU stack is of size L, which mirrors the size of the physical cache. This also means that the LRU stack only maintains the cache-resident data blocks in its stack. This structure gives the LRU system a constant time and space complexity. Although it captures the locality of reference, it does not exploit frequency effectively. A block that gains berth in the LRU stack will occupy space in it till it moves from the top of the stack to the bottom, without being referenced, and eventually getting evicted. The data-structure used by the LIRS system is a derivative of the LRU stack, which aims at maintaining two key pieces of data access history - the recency and the IRR. This is done by giving a LIR (Low IRR) or HIR (High IRR) status to data blocks, and managing them in two stacks. The first LIR stack tracks blocks with their recency less than the maximum recency of the LIR blocks. Its size is variable and it holds both cache-resident and non-resident blocks. The cache-resident HIR blocks are also tracked in separate fixed-size stack. When the need for a block eviction arises, the data block in the bottom of this HIR stack is evicted from the physical cache. However, unlike the LRU system, it is still kept in the LIR stack with its status changed to a non-resident block, if it was originally present in it. The bottom block in the LIR stack is always a LIR block, thanks to the stack-pruning operation that gets triggered in various scenarios to remove HIR blocks at the bottom.

5 Designing an N-Stack LIRS algorithm

The core component of the LIRS cache replacement algorithm is its 2-stack data structure. This structure gives the flexibility of maintaining key pieces of access history such as recency and IRR distance in a dynamic manner. Such an organization of data was inspired by the LRU's replacement behavior which assumed that temporal locality holds for block references. This 2-stack data structure can further be extended to an N-stack framework, with N being the total number of stacks that are used to maintain access

history information by appropriately placing data blocks in stacks. One approach to design this N-stack algorithm would be to have a data cascade. Data cascade here refers to the movement of data from one stack to the other in a cascaded fashion. This section discusses a design based on this model.

There are N-stacks that manages data blocks required by the caching subsystem. All stacks are of a fixed size L_{size}. The physical cache has the capacity to hold `N * L`$_{size}$ blocks of data. The stack size is not variable like the LIR stack, but is static like the HIR stack. Although, like LIR, these stacks can hold both cache-resident data and cache non-resident data. The last stack is the HIRS stack Q that holds the resident HIR blocks, meaning the cache can hold L_{size} HIR blocks. Stacks 1 through N-1, namely S_1 through S_{N-1}, hold the blocks which are given the LIR status. The LIRS principle based on which the threshold for changing block status from LIR to HIR, or vice-versa, is maintained as-is. The behavior of the algorithm in difference scenarios is as discussed below:

- **Referencing a non-resident HIR block**
  On referencing a data block that does not reside in cache, it is a miss. This data will be fetched from main memory and pushed to the top of the stack Q, the space for which will be made available after evicting the bottom element from Q. If the data block was also present in any of the stacks S_1 through S_{N-1}, its status is changed to LIR.

- **Referencing a cache-resident HIR block**
  Accessing a cache-resident HIR block will move that element from the stack in which it is present to the top of stack S_1. The other elements in the first stack are shifted down one position to make space for this update. The last element in S_1 is moved to the top of S_2. Such an update goes on from S_1 to S_X in a cascaded fashion, where S_X is the stack in which the data block in question was originally placed. This block is also removed from stack Q. The element in the bottom of S_{N-1} is moved to the top of stack Q.

- **Referencing an LIR block**
  This access, being an hit in the cache, just moves the data block in question to the top of stack S_1. A cascaded update operation takes place, as explained in the previous scenario.

Merits and Demerits of N-Stack LIRS

The major advantage of having such a N-Stack structure is the ability to provide quality of service to accesses. The overheads involved in accessing a LIR data block with IRR

1 will be NIL, while accessing a LIR block with IRR 5 will involve 4 cascading steps. The penalty curve increases gradually, and it takes a longer time for a LIR block to be completely kicked out of the S stack series. This longer retention period decreases the chances of a cache miss. This algorithm will also perform well in case of weak-locality access patterns. There will be a brief period of bubbling where the frequently accessed blocks get cascaded down to secondary stacks, but unlike LRU where it just drains out of the stack, the element is again moved up the stack hierarchy, thereby regaining a berth in the cache.

However, this algorithm is highly inefficient in average and worst-case scenarios. It is just a representative algorithm proposed to understand the behavior of an N-Stack approach. The amount of book-keeping involved will incur heavy performance penalties by increasing the access latency.

6 Conclusions

LIRS cache replacement algorithm is a very versatile one, and this study aims to get a deeper understanding into its behavior and merits. As part of this work, an algorithm to identify the perfect LIR set was proposed and its benefits compared to the 2-stack LIRS algorithm was discussed. Also, a design for an N-Stack LIRS structure was proposed and its behavior in different data access scenarios was discussed. Additionally, this work also outlines the relationship between the LIRS cache replacement and the LRU cache replacement mechanisms. Potential future work could be to implement a cache simulator for the proposed designs and evaluate their benefits using actual workloads on a real system.

References

[1] S. Jiang and X. Zhang, "LIRS: an Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance," in *SIGMETRICS*, 2002.

[2] "LRU Cache Replacement Algorithm," http://en.wikipedia.org/wiki/Cache-algorithms.

[3] S. Jiang and X. Zhang, "Making LRU friendly to weak locality workloads: a novel replacement algorithm to improve buffer cache performance," in *IEEE Transactions on Computers*, 2005.