6.894 Final Project
Periodic Reference Buffer Management (PRBM)

Eric Perkins, Rusty Lee & Eddie Byon

December 7, 2000

Massachusetts Institute of Technology
Department of Electrical Engineering & Computer Science
6.894 Distributed Operating Systems Laboratory

Abstract

Cache performance is determined by the ability of cache management software to predict when blocks will be re-referenced, so as to maximize cache utilization, and minimize disk accesses. Commonly this is achieved with a least recently used (LRU) eviction policy. While attractive in its simplicity, LRU fails take advantage of patterns in block accesses such as periodic referencing. The Unified Buffer Management Scheme (UBM)[1] improves performance by classify block access patterns, and managing them in a dynamically partitioned cache according to targeted schemes. The resulting performance improvement comes at the expense of a complex policy involving marginal-gain dynamic partitioning of the cache. We propose a new method which manages all block references as quasi-periodic, and ignores correlations in sequential referencing. Our scheme does not impose the overhead of dynamic partitioning or multiple cache management policies, but preserves much of the predictive capabilities of UBM. Using a trace-driven simulator, on a variety of trace inputs, we show that the performance of the two schemes is comparable.

Contents

1 Introduction

A common goal among developers today is to efficiently manage the buffer cache. Fetching data from disk proves a 1000 times more time consuming then fetching data from the RAM buffer. Thus proper buffer management proves to be essential to improve throughput and response times of any system. Block replacement schemes improve performance when we place limits on the buffer cache size, and thus most work is concentrated on creating new schemes based upon different methods. For example, schemes can be based on frequency and/or recency factors, user-level hints, or utilizing regularities such as sequential and looping references. UBM, an example of the latter, improves performance over the LRU scheme while remaining simple to implement. In this paper, we propose a new scheme which caches blocks according to the average Inter-Reference Gap (IRG). This scheme, which proves to be simpler to deploy then the UBM while retaining the same ratio hits found in the UBM through trace-driven simulations.

The goal of this paper is to simplify the UBM scheme such that our scheme performs as well as, if not better than the UBM scheme. We will describe our approach and test our results on similar traces ran under the UBM scheme. If successful, our hit ratio percentage should be at least equal if not greater than the UBM scheme with similar cache sizes. Equal performance is tolerated due to the fact that our scheme is much simpler than that of the UBM scheme.

The remainder of the paper is organized as follows. The next section describes the approach we took in our scheme while Section 4 analyzes our system and determines if we fulfill our goals stated earlier. Section 5 describes other schemes being implemented, and how our scheme compares to them. Finally, we review our conclusions in Section 6.

2 PRBM

The scheme developed in this paper draws from the UBM scheme of[1] but simplifies the approach by considering each block independently; UBM invests a significant amount of resources and effort in dividing block-reference patterns between sequential, looping, and other. Once thus divided, the blocks are assigned to portions of cache reserved for their determined reference pattern. The result is that repetitive consecutive (looping) references are evicted from cache based on their reference period, and all non-consecutive references are evicted using LRU. Simple consecutive references are evicted immediately once the cache is sufficiently warm.

While this method does achieve significant performance improvements, it is not evident that much is gained by the identification of consecutive references; little use is made of the information that the blocks are always referenced consecutively. Furthermore, since only consecutive references are examined for reference period, UBM ignores the possibility of periodic reference patterns in non-consecutive accesses.

Noting that consecutively referenced blocks will individually exhibit similar reference periods similar to that of the entire sequence taken as a whole, we can effectively manage consecutive periodic references as individual periodic refereneces. By ignoring the differences between consecutive and non-consecutive block references, those two UBM cache partitions are collapsed. Futhermore, since the partitioning mechanism all but eliminates the simple sequential partition, we can eliminate cache partitioning altogether. The only requirement imposed is that the eviction policy of our single partition removes non-recurring references sufficiently quickly.

3 Implementation

The approach of PRBM is to track the accesses of the blocks individually. A running average of the IRG for each block is used to determine the block's access period, and this period is used to prioritize the block cache. The actual cache works like a priority queue, into which each referenced block is inserted, and from which the longest period block is evicted. A secondary ghost cache is required to maintain the relevant block metadata (period, last reference time, etc.) after a block has been evicted from cache. This ghost cache is managed according to an LRU policy.

In order to track the access period for the data blocks, our scheme associates a cache record with each block. This record contains all of the information needed to prioritize the blocks, and to determine a new period estimate for the blocks when they are accessed. The record consists of the tuple real,blockno,reftime,period; real is a boolean indicating whether the block is actually cached, or is being held in the ghost cache, blockno uniquely identifies each block, reftime gives the most recent access time, and period gives the current estimate of the access period.

In order to service page references, the records of all known blocks are indexed by a hashtable. The records of blocks which are currently cached are also indexed by a heap-structured priority queue, facilitating insertion and eviction. Each time a block is referenced, the hashtable is entered to find the cache record. If the record is found, the access period is updated, and the block is returned to the caller. If the block is not currently cached (i.e. it is in the ghost cache) it must first be retrieved from disk.

Because the cache queue is prioritized by the period estimate, it is possible to evict blocks immediately following a reference if the estimated period is sufficiently long with respect to the rest of the queue. For first-time references, this estimate can be little more than a guess, and if the initial estimate of the access period is too long, the block may be evicted before it has a chance to establish a shorter period. If the record is subsequently discarded, the next reference is treated as a first-time reference and the problem remains unresolved; the block is never cached, and the system takes numerous cache-misses, degrading performance. To solve this problem, we use a ghost cache.

The ghost cache indexes the records of evicted blocks, so that access statistics can be collected. The blocks themselves, however, are not cached, and must be re-fetched from disk if referenced. Since the ghost cache is intended to maintain the cache records of blocks for which the period may not be accurate, it is not appropriate to prioritize them according to period. What is in fact required is a sort of holding station in which to place records while the system has a chance to reevaluate its estimate of the access period. For this purpose an LRU scheme is ideal; records are evicted when they have not been referenced for a long time, validating their low estimated period. Furthermore, since the actual blocks are not cached, it is possible to have a fairly large ghost cache, ensuring the blocks ample time to be referenced again.

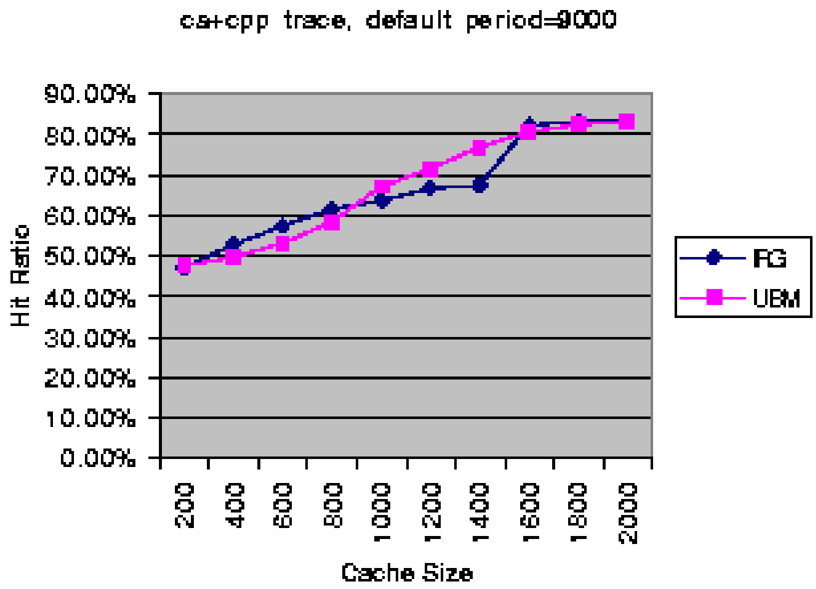
The ghost cache is used to solve the problem of short-period blocks with long estimated periods. The inverse problem can also occur if a block gets assigned a period which is very short, but is never referenced. Since the period is not updated until the next reference, such blocks could remain unevicted indefinitely, and potentially waste a significant amount of cache space. In order to limit this effect, the records are periodically reexamined to determine if the period is still valid, and the queue is reprioritized, ensuring that a record in the queue never misrepresents its block's access period for long.

4 Analysis

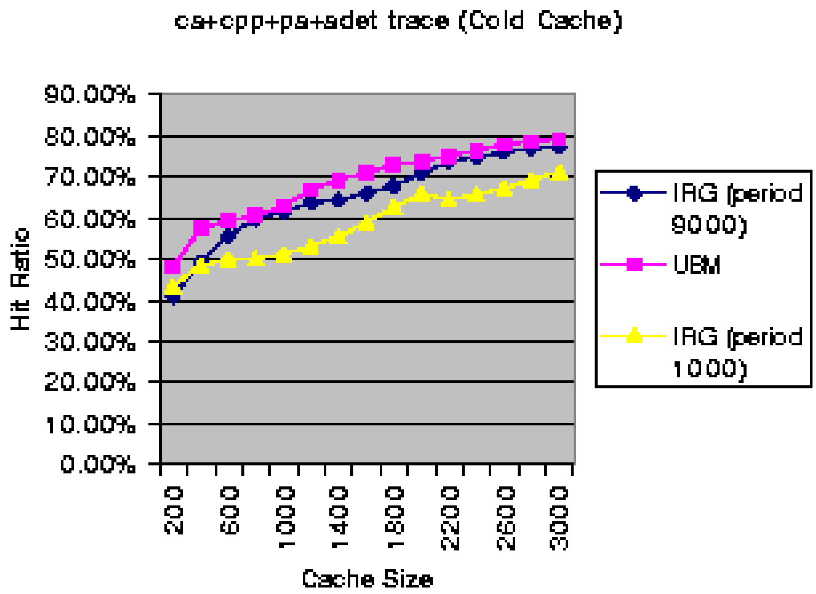
We modified a program developed to simulate the Unified Buffer Management scheme to evaluate the performance of our caching scheme. The new simulation results were compared with the results of the original UBM simulation. A variety of traces created for the UBM simulator were run through our modified simulator to compare the performance under different types of loads. These traces include a number of applications simultaneously executing under FreeBSD on an Intel Pentium PC. The memory access patterns of these applications are described below[1].

cpp, the GNU C compiler pre-processor, exhibits mostly sequential block references. cscope, the C source analysis tool, exhibits primarily looping references. glimpse, the text information retrieval utility, has a diverse combination of looping, sequential, and other block references. gnuplot, the GNU plot generator exhibits mostly looping block references. postgres, a relational database system from UC Berkeley, exhibits sequential, looping, and other block references. sdet, a benchmark that simulates a multiprogramming environment, exhibits sequential, looping, and other block references[1].

These traces were done using a ghost buffer size equal to the size of the cache. Our

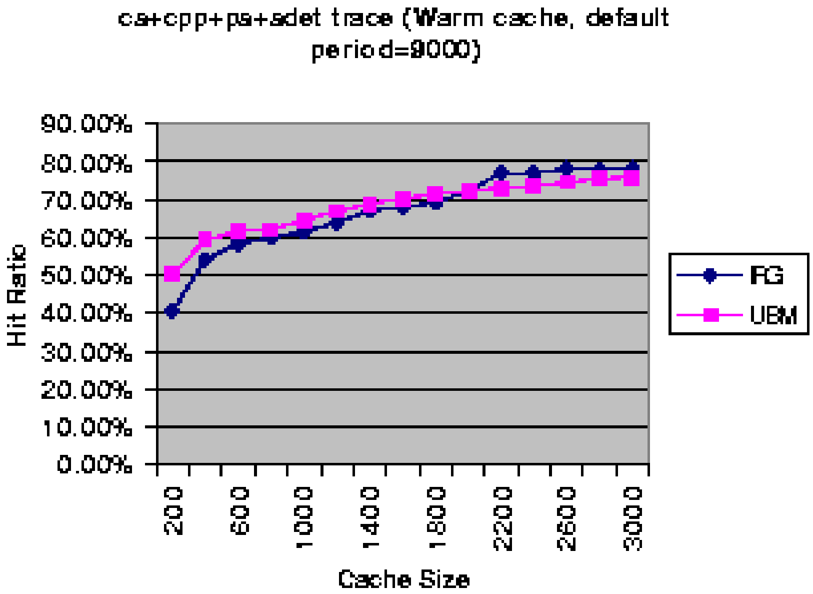


ca+cpp trace, default period=9000
Hit Ratio
90.00%
80.00%
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
200
400
600
800
1000
1200
1400
1600
1800
2000
Cache Size
IRG
UBM



ca+cpp+ps+sdet trace (Cold Cache)
Hit Ratio
90.00%
80.00%
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
200
600
1000
1400
1800
2200
2600
3000
Cache Size
IRG (period 9000)
UBM
IRG (period 1000)

replacement scheme performs almost as well or better than the UBM scheme, given identical cache sizes and ghost buffer lengths. It outperforms SEQ, 2Q, EELRU, and LRU, except for very small cache sizes[1]. The greater complexity of the UBM scheme only yields a very slight hit-ratio enhancement. Increasing the default period to grow closer to the average period between references of a block improves the hit ratio.



With a warm cache, the hit-ratio of our scheme exceeds that of UBM for large caches, and is very close for smaller caches. Our scheme seems to deal slightly better with warm caches than UBM does, and much better than SEQ, 2Q, EELRU, and LRU for large caches[1].

Since our scheme is so much simpler than the UBM scheme, it performs much better in terms of run-time speed. Increasing the size of the ghost buffer to hold more blocks than the size of the cache results in only a small improvement in the hit ratio, even for small caches.

5 Related Work

This section surveys two previous developed schemes and explains their advantages and disadvantages.

O'Neil et al. proposes an LRU-K replacement policy, which keeps track of the last K references and uses this knowledge when making a decision to replace a block.[2] The advantages lie in its simplicity and low overhead, but this scheme ignores sequential and looping references[1] and ignores the recency of the K-1 references[2].

The 2Q scheme improves upon the LRU scheme by quickly removing cold pages from the buffer. As described by Johnson and Shasha[3], the LRU scheme removes cold pages from the buffer to make space for a faulted page. If this faulted page is cold, LRU displaces a warm page in favor of the cold one. 2Q on the other hand only allows hot pages into the buffer. 2Q inserts the most recent first accesses into the A1in queue up to a certain threshold, thereafter flushing the older first accesses into A1out. If any page is re-referenced and resides in the A1in queue, it is considered hot and moved to an Am queue, which is managed as an LRU queue. The main drawback to this scheme however lies in tuning the sizes of A1in and A1out[1]. Johnson and Shasha explain that given a buffer size B, divide this value into A1in, A1out, and Am[3], without giving an exact tuning solution.

6 Conclusions

Besides 2Q and LRU-K, other schemes exist, such as EELRU, LRFU, LFU, and SEQ which try to attack the problem of buffer management from different perspectives. LRFU inherits the benefits of the LRU and Lease Frequently Used (LFU) policies while the SEQ scheme detects page faults and applies a Most Recently Used (MRU) scheme to those pages. All schemes prove to be efficient when compared against the LRU scheme, yet results from the UBM paper illustrate UBM's improvements over them. Thus, by proving that our scheme performs better than the UBM scheme, we have shown that our scheme is perhaps a better alternative to all schemes.

References

[1] Y. Cho, J. Choi, C.S. Kim, J. Kim, J.M. Kim, S.L. Min, S.H. Noh, "A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References,"

[2] E.J. O'Neil, P. E. Oneil, and G. Weikum, "The LRU-K Page Replacement Algorithm For Database Disk Buffering," in *Proceedings of the 1993 ACM SIGMOD Conference*, pp. 297-306, 1993.

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