Comparison between TreePLRU and ARC

Seyed Reza Sajjadinasab, Bin Xu, yilei feng

Electrical and Computer engineering Department

This paper presents a comparative study of two popular cache replacement algorithms, Adaptive Replacement Cache (ARC) and Tree Pseudo-Least Recently Used (TreePLRU). The aim of the study is to evaluate the performance of the two algorithms. The experiments were conducted using different benchmark applications. The results indicate TreePLRU performs better in terms of miss rate and CPI. Overall, the study provides insights into the strengths and weaknesses of the two algorithms and can help inform cache replacement decisions in real-world systems.

Introduction

Cache replacement algorithms(policies) is important in improving the performance of by reducing the number of cache misses. The effective processing of the contemporary big data applications depends on cache replacement policies. Any high speed computer system's cache memory performance has a significant impact on the system's performance. The critical blocks can be located closer to the core thanks to a smarter replacement policy. Many algorithms have been proposed over the years, each with its own strengths and weaknesses. In this paper, we compare two popular cache replacement algorithms, Adaptive Replacement Cache (ARC) and Tree Pseudo-Least Recently Used (TreePLRU), in terms of their miss ratio performance across different benchmark applications.

We conduct our experiments using five widely used benchmark applications: gcc, perlbench, deeps, xalancbmk and omnetpp. We evaluate the performance of the two algorithms on each application with different cache sizes for both the L1 and L2 caches. The aim of this study is to provide a comprehensive evaluation of the two algorithms under different cache sizes and workloads.

The first replacement policy we use is ARC is a cache replacement algorithm that dynamically adjusts the size of its replacement policies based on the changing workload of the system. And the second one is TreePLRU is a classic algorithm that uses a binary search tree to maintain the LRU order of the cache blocks.

We hypothesize that ARC may perform better in workloads with highly dynamic access patterns, while TreePLRU may perform better in workloads with more static access patterns. Additionally, we expect the cache size to have a significant impact on the performance of both algorithms.

The contributions of this paper are twofold. First, we provide a comparative analysis of two popular cache replacement algorithms, which can help system designers make informed decisions about which algorithm to use based on their specific requirements. Second, we provide a detailed evaluation of the performance of the algorithms across different benchmark applications with varying cache sizes, which can help identify the optimal cache configurations for specific workloads.

Replacement Policy Description

Tree Pseudo-Least Recently Used (TreePLRU) is a cache replacement algorithm that uses a binary search tree data structure to maintain the Least Recently Used (LRU) order of the cache blocks. In this algorithm, each cache block is associated with a node in the binary search tree. The nodes are arranged in the order of their access times, with the least recently used node at the root of the tree.

Whenever a cache miss occurs, the algorithm evicts the block associated with the root node of the tree, which is the least recently used block. The algorithm then reorders the nodes in the tree to reflect the new access times of the remaining blocks. If a block is accessed again after being evicted, the algorithm adds a new node to the tree to represent the block's new access time.TreePLRU is a static algorithm, which means that it does not adapt its replacement policy to changes in the workload. As a result, the algorithm may perform poorly in workloads with highly dynamic access patterns. However, it is known to perform well in workloads with more static access patterns, where the LRU order is more predictable.

On the other hand, Adaptive Replacement Cache (ARC) is a cache replacement algorithm that dynamically adjusts the size of its replacement policies based on the changing workload of the system. The algorithm maintains two lists of cache blocks, the LRU list and the MRU (Most Recently Used) list. The LRU list contains blocks that have not been accessed recently, while the MRU list contains blocks that have been accessed recently.

Whenever a cache miss occurs, ARC decides which list to evict from based on the current state of the cache. If the cache is full and the cache block that was missed is not in the MRU list, then ARC evicts a block from the LRU list. If the missed block is in the MRU list, ARC evicts a block from the MRU list instead. If the cache is not full, ARC adds the missed block to the MRU list. To dynamically adjust the size of its replacement policies, ARC uses two parameters, p and c. The p parameter determines the relative size of the MRU and LRU lists, while the c parameter determines how much of the cache space should be allocated to the MRU list. These parameters are updated periodically based on the hit rates of the MRU and LRU lists.

ARC is designed to adapt to changing workload patterns by allocating more space to the list with a higher hit rate. For example, if the workload becomes more dynamic and the MRU

list begins to perform better, ARC will allocate more space to the MRU list and decrease the size of the LRU list.

Implementation and Testing

Following the gem5 data structure, we implement touch, reset, getVictim and invalidate. Because we need to get the tag of the data, we override the reset and touch method including packetPtr for getting the address of replacement data.
In the implementation, We use 8-way associated sets with 64 default block size to calculate the size of t1, t2, b1 and b2. Using (countif) which is from the STL algorithm to find the size of these vectors.
In the reset function, we need to address the miss condition for the algorithm. The main logic of handling the data flow among t1, t2, b1 and b2 is implemented in it. At the end of the method, the subroutine replacement policy of the ARC algorithm is implemented and passes the victim value to the getVictim function.
Touch function is called every time when the cache hit occurs in the t1 and t2. When the data is hit in the t1 and t2, we implement logic for moving data to the MRU position in t2. Also the packetPtr and lastTouchTick need to be updated.

Evaluation Results

Tree	CPI	MKPI
L1 cache size fixed at 16 KB		
gcc-16-1	3.57	8.53
gcc-16-2	3.53	8.49
gcc-16-4	3.50	8.46
omn-16-1	3.99	15.11
omn-16-2	3.95	15.37
omn-16-4	3.91	15.69
perl-16-1	3.97	13.38
perl-16-2	3.97	13.20
perl-16-4	3.96	13.43
deep-16-1	4.24	12.74
deep-16-2	4.22	12.69
deep-16-4	4.18	12.69
xal-16-1	3.57	8.54
xal-16-2	3.55	8.56
xal-16-4	3.51	8.46

Fig. 1. TreePLRU L2 cache - 1 MB, 2MB and 4MB, L1 cache size fixed at 16 KB.

L1 cache size fixed at 32 KB		
gcc-32-4	3.38	4.69
omn-32-4	3.75	8.05
perl-32-4	3.96	8.5
deep-32-4	3.92	7.3
xal-32-4	3.38	4.7

Fig. 2. TreePLRU L2 cache 4MB, L1 cache size fixed at 32 KB.

L1 cache size fixed at 64 KB		
gcc-64-4	3.3	2.65
omn-64-4	3.72	6.5
perl-64-4	3.9	5.15
deep-64-4	3.83	4.3
xal-64-4	3.32	2.7

Fig. 3. TreePLRU L2 cache 4MB, L1 cache size fixed at 64 KB.

ARC	CPI	MKPI
L1 cache size fixed at 16 KB		
gcc-16-1	4.334	20.08
gcc-16-2	4.26	19.27
gcc-16-4	4.21	19.35
omn-16-1	4.34	20.067
omn-16-2	4.26	19.25
omn-16-4	4.197	19.44
perl-16-1	4.337	20.7
perl-16-2	4.25	19.26
perl-16-4	4.18	19.37
deep-16-1	4.197	19.34
deep-16-2	4.242	19.5
deep-16-4	4.197	19.3
xal-16-1	4.337	20
xal-16-2	4.26	19.3
xal-16-4	4.21	19

Fig. 4. ARC L2 cache 1MB, 2MB, 4MB, L1 cache size fixed at 16 KB.

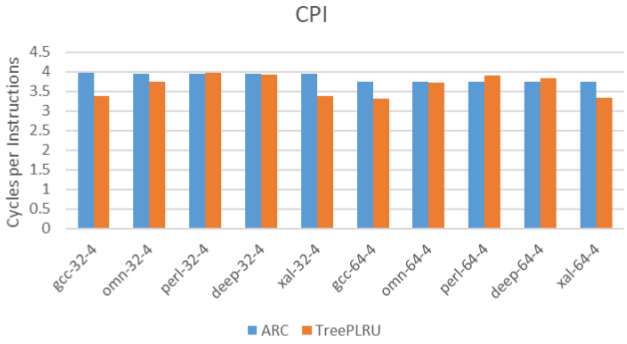

Fig. 7. compare ARC and TreePLUR in miss CPI.

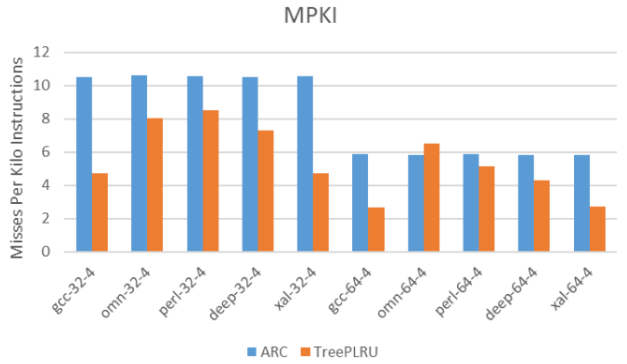

Fig. 8. compare ARC and TreePLUR in miss ratio.

L2 cache size fixed at 32KB		
gcc-32-4	3.97	10.48
omn-32-4	3.95	10.6
perl-32-4	3.95	10.58
deep-32-4	3.94	10.5
xal-32-4	3.95	10.56

Fig. 5. ARC L2 cache 4MB, L1 cache size fixed at 32 KB.

L1 cache size fixed at 64KB		
gcc-64-4	3.75	5.87
omn-64-4	3.75	5.8
perl-64-4	3.74	5.85
deep-64-4	3.74	5.83
xal-64-4	3.73	5.81

Fig. 6. ARC L2 cache 4MB, L1 cache size fixed at 32 KB.

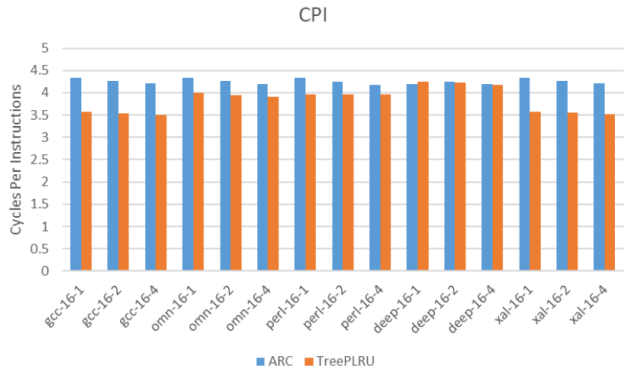

Fig. 9. compare ARC and TreePLUR in CPI.

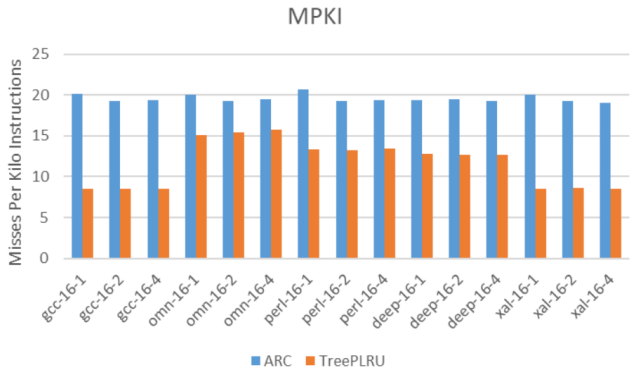

Fig. 10. compare ARC and TreePLUR in miss ratio.

The given data compares the performance of two different cache replacement policies, ARC and TreePLRU, for various benchmarks and cache configurations. The performance is measured in terms of CPI (cycles per instruction) and miss rate.

Based on the data, it can be observed that in most cases, TreePLRU has a lower miss rate and CPI than ARC. For example, when L1 cache size is fixed at 16KB and for benchmark 'gcc-16-1', TreePLRU has a miss rate of 8.53 and CPI of 3.57, while ARC has a miss rate of 20.08 and CPI of 4,334. Similarly, for most other benchmarks and cache configurations, TreePLRU consistently has a lower miss rate and CPI than ARC.

One reason why ARC performs better than TreePLRU could be attributed to the way TreePLRU algorithm works. TreePLRU is better, one possible reason for that could be that TreePLRU has a more deterministic behavior than ARC. This means that TreePLRU provides a guaranteed number of ways that are never evicted at any point in time. In contrast, ARC is more adaptive and dynamically adjusts the number of ways allocated to LRU and LFU lists based on the workload patterns. This dynamic behavior can lead to some level of unpredictability in the cache eviction behavior, which may result in higher miss rates or CPI in some cases.TreePLRU could be better in some cases is that it has a simpler structure than ARC. While ARC's dynamic behavior can make it effective for handling different workload patterns, it also requires more complex logic and additional hardware resources to implement. TreePLRU, on the other hand, can be implemented more efficiently and with less hardware overhead, which can be an advantage in some scenarios.

Conclusions

Overall, based on the given data, we can conclude that TreePLRU generally outperforms ARC in terms of both miss rate and CPI for most benchmarks and cache configurations.

Reference

ARC: A Self-Tuning, Low Overhead Replacement Cache, USENIX Conference on File and Storage Technologies (FAST 03), San Francisco, CA, pp. 115-130, March 31-April 2, 2003.