Adaptive Replacement Policy with Double Stacks for High Performance

Hongliang YANG

School of Informatics, Linyi University, Linyi, 276009, China

email: lytuyhl@163.com

Keywords: LRU; Access Mode; Adaptive Replacement; High Performance

Abstract. LRU policy showed the good performance and has been used widely in modern computer, however, the accesses mode of the application is not all preferences of LRU algorithm, there were some drawbacks. In the paper, by researching the existed policy about LRU, we propose an adaptive policy base on the LRU, in this policy, we set double LRU stacks to respectively save the recency and frequency block, and according their efficiency value to adjust dynamically their size. The experiments showed that this method could improve the hit ratio under the different circumstance.

Introduction

The cache replacement policies play an important role in cache management. They can improve system performance by reducing cache miss counts. Especially, the least recently used (LRU) policy shows good performance in recent decades and has been used as a de-facto standard [14]: 1) When the workloads is larger than the cache size, LRU algorithm will happens jitter meeting, the result is less efficient. 2) It is oblivious to recent history and lacking of other reuse information, LRU captures the recency features of a workload, it does not capture and exploit the frequency features of a workload, when the programs require memory strongly, LRU is not as good as LFU with low efficiency. 3) In some certain systems, the performance of processors will be affected by processors stall without dealing with well.

In view of the above points of the defects of traditional LRU algorithm, the recent solution mainly concentrated in this aspect: Improving LRU algorithm to solve the problem of a kind of traditional LRU solving badly, such as LRFU [1], the BIP [2], LIN [3].

In our research, we find that it is very easy to find simple policies to capture a single locality characteristic behavior, but it is hard to design a good replacement policy that behaves comparably to LRU in replacing blocks that were not recently access and eagerly replaces blocks that are accessed only once, even when the workload locality characteristics are well-known. The idea of adaptive replacement policy based on LRU is to adaptively switch between polices LRU and other police, so that it uses LRU whenever recent behavior indicates that LRU would outperform the other. In this paper, we proposed an adaptive policy with double stacks, we set double LRU stacks to respectively save the recency and frequency block, and according their efficiency value to adjust dynamically their size to choice the suited policy between the LRU and the LFU polices. By performing simulations with a large number of traces form previous studies, we showed that our adaptive policy performs was very well under the three kinds of access patterns compared with the LRU, LRFU and ARC of the typical policies, although the whole hit ratio is not high.

Related Work

There have been many novel caching algorithms, such as FBR, LRU-2, 2Q, LRFU, MQ, and LIRS, have been proposed, which have attempted to combine LFU and LRU in order to remov one or more disadvantages of LRU. However, all of the above cited policies, using LRU as the building block, for a detailed overview of these algorithms, it turns out, that each of these algorithms leaves something to be desired, and, hence, continue to suffer from some certain drawbacks of LRU. Finally, the idea is certainly not new idea, but the adaptivity of past schemes was limited to a very

specific policy. For example, EELRU is based on the principle of timescale relativity, which helps it detect and adapt to phase changes, only suiting for loop detection pattern, the SEQ replacement algorithm is one that bases its decisions on address information (detecting sequential address reference patterns). Consequently, it is lacking in generality，and so on. More specifically, SEQ and EELRU primarily are used managing virtual memory, while LRFU, and ARC are more targeted towards file system data caching, either at the OS or the disk controller level.

The ARC [10] is self-tuning, low overhead, scan-resistant, exploits both the recency and the frequency features of the workload in a self-tuning fashion, has low space and time complexity, and outperforms LRU across a wide range of workloads and cache sizes. It seemed to eliminate essentially all drawbacks of the above mentioned policies, but ARC adopted double stacks, the size of the two stacks was adjusted by using a parameter p, but there was not a good way to determine the size of p value.

LRFU [5] is the novel caching algorithms that have attempted to combine recency and frequency (CRF) with the intent of removing one or more disadvantages of LRU. The police always choose a block with minimum value of CRF to evict. The CRF value is calculated by weight function $C_{t_{present}}(b) = \sum_{i=1}^{k} F(x)$, $F(x) = (1/p)^{\lambda x}$, among them, $C_{t_{present}}(b)$ is the CRF value of b block in the current time, x is the time interval from the last access to the current, λ is the balance parameter of locality and frequency. For example, the accessed time of some block is at 2, 5 and the current time is at 7, then the value of the CRF of the block is F(7-2)+F(7-5). It is obvious that if the parameter is the greater, the policy is more inclined to focus on the access locality, if the λ value gets smaller, the policy is more inclined to focus on access frequency features. But the λ value is fixed, so the LRFU is not a self-tuning policy.

Principles of the Policy

In this section, we discuss the idea of adaptive replacement based on the LRU and the implementation.

As a general adaptive policy, the goal is to adaptively switch between policies A and B, so that it uses B whenever recent behavior indicates that B would outperform A. Here, we used for reference the idea of ARC policy and set double adaptive stacks LRS and HRS adopted LRU policy to manage and third stack HS, adopted FIFO policy to manage, was used for recording the history information. Among them, LRS was used for placing recency smaller blocks and HRS was used for placing recency bigger and higher access frequency blocks. When the workloads were frequency base access pattern, the HRS value was increased and the LRS value was decreased, on the contrary, the HRS value was decreased and LRS value was decreased. In order to determine which value was be increased or decreased, we introduced an efficient value to measure the two stacks. In a certain time period, the efficient value was expressed by the following: effi_value=numbers of hit block/ max numbers of stack. According to the effi_value, we can modify the value of the HRS and LRS.

We assumed that the size of cache was expressed by S_cache and the size of the HRS and LRS stacks were expressed respectively by size_HRS and size_LRS, and S_cache=S_HRS + S_LRS. The value P was modified by the following pseudo code:

```
Modify_value /* used for modifying the P*/
H_times=0; /* hit times in the HRS stack*/
L_times=0; /* hit times in the LRS stack*/
Request data
Case 1: (hit in the HRS) H_times ++
If (S_HRS+S_LRS==S_cache)
   {
move data to the MRU position of the LRS
}
 Case 2: hit in the LRS L_times ++
 If (S_HRS+S_LRS==S_cache)
```

```
   {
move data to the MRU position of the HRS
}
   Case 3: hit in HS
        If S_HRS+S_LRS==S_cache∩S_LRS>MAX_S-LRS
          {
              Delete LRU block in the LRS
           }
         Else
          {
                Delete LRU block in the HRS
           }
              Move data to MRU position of HRS;
         Case 4: hit in HRS∪LUS∪HS
                {
           If S_HRS+S_LRS<S_cache
              {
              move data to MRU position of HRS or LRS
         }
         Else
         {
              Delete LRU blocks in LRS or HRS
           Move data to MRU position of HRS
       }
 If request numbers > S_cache
    {
Adjust stack size
      }
   Adjust stack size
    {
Case 1: E_LRS>E_HRS; /* efficient of LRS and HRS stack */
            {
            S_LRS=Modify_value+S_LRS
            S_HRS=Modify_value-S_HRS
            }
Case 2: E_LRS<E_HRS;
           {
            S_LRS=Modify_value+S_LRS
            S_HRS=Modify_value-S_HRS
            }
        H_times=0
        L_times=0
        Request_time=0
    }
```

Experiment and Evaluation

In order to evaluate the adaptive replacement policy based on LRU, We use a trace-driven cache simulation compared with the present popular replacement policies. The experiment environment is the Memory Buddies Trace of Linux server [1]. Memory Buddies Trace is the memory access trace of the data virtualization center server combined the resources of the multiple devices and provided users with a unified logical interface, the users can access the different resources by the different

patterns [7]. In practical application, the different replacement algorithm has different performance in application environment, we choose three kinds of trace simulation, representing respectively three kinds of access pattern: linear access pattern, probability access pattern, locality access pattern.

In this experiment, the block size is 64B, and the size of the parameters S_HRS, S_LRS, MAX_S_HRS, MAX_S_LRS and Modify_value respectively was 75%, 25%, 99%, 99%, 1%. Although the whole hit ratio is not high, Fig1~Fig3 shows that the A-LRU policy can improve the hit ration in different circumstance compared with LRU, LRFU and ARC.

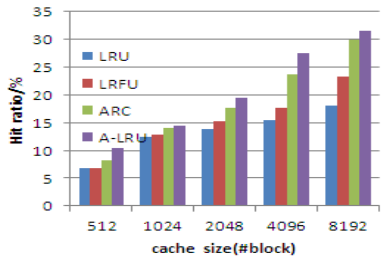

Fig.1. Compare hit rate in linear access pattern

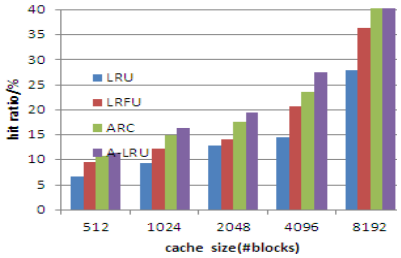

Fig.2. Compare hit rate in probability access pattern

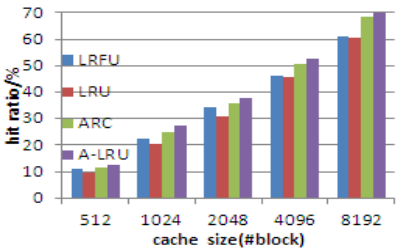

Fig.3. Compare hit rate in locality access pattern

Conclusion

While there has been much work on improving the performance of caches, our work makes several new contributions. In this paper, we introduce a scheme based on LRU with adaptivity which provides a higher hit ratio under the different circumstance. Thus, we hope that the new ideas would be presented in the design of future caching mechanisms..

Acknowledgement

In this paper, the research was sponsored by the Natural Science Foundation of Shandong Province (Project No. ZR2014FL012), Achievements Transformation Major Projects of Shandong Province (Project No. 2014ZZCX02702) .

References

[1] S. Jiang and X. Zhang, “LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance,” in Proc. ACM SIGMETRICS Conf., 2002.

[2] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson, “EELRU: Simple and efficient adaptive page replacement,” In ACM SIGMETRICS Conference on Measurement and Modeling of computer systems, pages 122-133, 1999.

[3] T. Johnson and D. Shasha, “2Q: A low overhead high performance buffer management replacement algorithm,” in Proc.VLDB Conf., pages 297–306, 1994.

[4] G. Glass and P. Cao, “Adaptive page replacement based on memory reference behavior,” In ACM SIGMETRICS Conference on Measurement and Modeling of computer systems, pages 115-126, 1997.

[5] Yannis Smaragdakis, “general adaptive replacement polices,” ISSM’04, Vancouver, British Columbia, Canada, October 24-25, 2004.

[6] E. Perelman, G. Hamerly, and B. Calder, “Picking Statistically Valid and Early Simulation Points,” In Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques, New Orleans,LA, USA, pages 244–255, September 2004.

[7] B. Fields, “Focusing Processor Policies via Critical-Path Prediction,” In Proceedings of the 28th International Symposium on Computer Architecture, pages 74–85, GÖteborg, Sweden, June 2001.

[8] A. Glew, “MLP Yes! ILP No! Memory Level Parallelism, or, Why I No Longer Worry About IPC,” In Proceedings of theASPLOS Wild and Crazy Ideas Session, San Jose, CA, USA,October 1997.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “MiBench: A Free, Commercially Representative Embedded Benchmark Suite,” In Proceedings of the 4th Workshop on Workload Characterization, Austin, TX, USA, pages 83–94, December 2001.

[10] N. Megiddo and D. S. Modha, “ARC: A self-tuning, low overhead replacement cache,” in Proc. 2nd USENIX Conference on File and Storage Technologies (FAST 03), San Franciso, CA, pages 115–130, 2003.

[10] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, “LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies,” IEEE Transactions on Computers, 50(12):1352-1361, Dec. 2001.

[11] Ruirui Guo, Jose G.Delgado-Frias and Stephan Wong, “cache replacement policies for IP address lookups,” Proceeding of the fifth IASTED International conference circuits, signals, and systems, July 2-4, 2007, Banff, Alberta, Canada.

[12] N. Megiddo and D. S. Modha, “ARC: A self-tuning, low overhead replacement cache,” In USENIX File and Storage Technologies (FAST), Mar. 2003.

[13] K. Albayraktaroglu, A. Jalell, X. Wu, M. Franklin, B. Jacob, C. W. Tseng, and D. Yeung, “BioBench: A Benchmark Suite of Bioinformatics Applications,” In Proceedings of the International Symposium on Performance Analysis of Systemsand Software, pages 2 – 9, Austin, TX, USA, March 2005.

[14] Liqing He, Yan Sun and Chaozhong Zhang, “adaptive subset based replacement policy for high performance caching,” JWAC 2010-1st JILP workshop on computer architecture competitions.

[15] J. T. Robinson and M. V. Devarakonda, “Data cache management using frequency-based replacement,” in Proc. ACM SIGMETRICS Conf., pages. 134–142, 1990

[16] E. J. O’Neil, P. E. O’Neil, and G. Weikum, “The LRU-K page replacement algorithm for database disk buffering,” in Proc. ACM SIGMOD Conf., pages. 297–306, 1993. [11] Chunmei Xu. Mechanical servo system based on fuzzy neural network for complex control [J]. Control Engineering .2010:17 (2):146-148.