LETTER

Adaptive Insertion and Promotion Policies Based on Least Recently Used Replacement

Wenbing JIN[†,††a)], ***Nonmember***, **Xuanya LI**[†], ***Student Member***, **Yanyong YU**[††], ***and*** **Yongzhi WANG**[††], ***Nonmembers***

SUMMARY To improve Last-Level Cache (LLC) management, numerous approaches have been proposed requiring additional hardware budget and increased overhead. A number of these approaches even change the organization of the existing cache design. In this letter, we propose Adaptive Insertion and Promotion (AIP) policies based on Least Recently Used (LRU) replacement. AIP dynamically inserts a missed line in the middle of the cache list and promotes a reused line several steps left, realizing the combination of LRU and LFU policies deliberately under a single unified scheme. As a result, it benefits workloads with high locality as well as with many frequently reused lines. Most importantly, AIP requires no additional hardware other than a typical LRU list, thus it can be easily integrated into the existing hardware with minimal changes. Other issues around LLC such as scans, thrashing and dead lines are all explored in our study. Experimental results on the gem5 simulator with SPEC CUP2006 benchmarks indicate that AIP outperforms LRU replacement policy by an average of 5.8% on the misses per thousand instructions metric.

key words: *last-level cache, replacement, insertion, promotion, thrashing*

1. Introduction

Modern computers rely on efficient cache management to mitigate the wide speed gap between processor and memory. Traditional cache management focuses on cache replacement with the Least Recently Used (LRU) policy. The LRU policy works well for workloads with high locality but results in ineffective cache management for the Last-Level Cache (LLC), where temporal locality is partly filtered by preceding levels of the cache hierarchy [1]. Hence, the Least Frequently Used (LFU) policy was introduced to identify the presence of frequently reused lines in the cache [2], given that frequency is another feature of referenced lines in addition to recency. Although LFU improves cache performance for a number of workloads with many frequently reused lines, it fails for others where recency is the preferable choice for replacement. The two policies seem unrelated and independent; however, they trade off with each other for different access patterns.

Apart from recency and frequency, a number of issues persist around LLC management, such as scans, thrashing and dead lines. The last decade has seen the development of several novel cache management approaches that have attempted to combine the LRU and LFU policies with the intent of solving one or more problems in this field; however, many of these approaches incur huge storage overhead and require significant changes in the existing design, along with limited improvement on performance.

In this letter, we propose Adaptive Insertion and Promotion (AIP) policies based on LRU replacement. AIP provides dynamic insertion and changeable promotion, which combines the LRU and LFU policies together under a single unified scheme. As a result, AIP copes well with changed access patterns from recency to frequency, benefiting workloads with high locality as well as with many frequently reused lines. At the same time, scans, thrashing and dead lines are all effectively addressed under AIP. Moreover, AIP is simple in combination, flexible in adaptation and scalable in structure.

The remainder of this letter is organized as follows: In the next section, existing solutions are discussed. Section 3 proposes AIP policies in detail, followed by experiments in Sect. 4. The last section presents our conclusion.

2. Existing Solutions

The LRU policy depends on the recency of referenced lines to select the evictee for replacement. In contrast, the LFU policy selects the evictee according to the frequency of referenced lines. The Least Recently/Frequently Used (LRFU) policy [3] subsumes the LRU and LFU policies together to balance between the recency and frequency of the referenced lines as the basis for a replacement decision. With complicated algorithms and several parameters, which must be tuned on a per workload basis, the LRFU policy is too slow to be implemented in the current cache structure [4].

The Adaptive Replacement Cache (ARC) [4] maintains one list each for recency and frequency. The recency list contains pages that are used only once while in residence, whereas the frequency list contains pages that are used at least twice. ARC dynamically tunes the number of pages devoted to each list to detect and track temporal locality and frequency of referenced lines so as to keep those pages that have the greatest likelihood of being used in the near future in the cache. Scavenger policy [11] divides the total storage budget into two parts: a conventional cache and a novel victim file architecture. The latter employs additional components including a skewed bloom filter in conjunction with a pipelined priority heap to identify and retain

Manuscript received March 30, 2012.
Manuscript revised August 3, 2012.

[†]The authors are with the School of Computer Science, Beijing Institute of Technology, Beijing, China.
[††]The authors are with the North Automatic Control Technology Institute, Taiyuan, China.
a) E-mail: wenbbo@sina.com

DOI: 10.1587/transinf.E96.D.124

the blocks that most frequently missed in the conventional part of the cache in the recent past. More adaptive cache management approaches can be found in [6], [7]. Although they successfully combine LRU and LFU replacement policies together via additional data structures, they significantly increase hardware overhead and complexity. A recent study on Shepherd Cache [12] even changes the organization of the existing cache, leading to further verification and testing efforts for it.

Dynamic Insertion Policy (DIP) relies on the LRU policy for workloads with high locality and the LRU Insertion Policy (LIP) for other workloads with thrashing. The LIP component of DIP inserts lines directly into the LRU position to minimize the residency time of thrashing [5], resulting in a number of distantly reused lines in the large working set being partly preserved in the cache. Although DIP effectively protects LLC from scans and thrashing using auxiliary tag directories and a policy selector, it's unable to retain frequently reused lines in the cache.

Various other solutions for efficient cache management have been reported in [8]–[10], but they either require significant hardware or drastically change the organization of the existing cache design. A single cache management scheme, with minimal changes to the existing structure and capability to effectively address all of the problems in this field, is highly desirable.

3. AIP Policies

3.1 Insertion and Promotion Policies Based on LRU Replacement

AIP employs the same structure as that commonly used by the LRU replacement policy. Figure 1 illustrates a typical cache set with eight lines in a list, logically organized left-to-right from the Most Recently Used (MRU) position to the LRU position. The traditional LRU replacement policy inserts a missed line in the MRU position while the line in the LRU position is evicted, and a reused (hit) line is promoted to the MRU position. A number of studies have proposed different insertion and promotion policies, such as insertion in the LRU position [5] or the promotion of a single increment each time for the reused line in the list [13]. AIP differs from all the previous proposals in that it inserts missed lines in the middle of the list. The insertion position in AIP

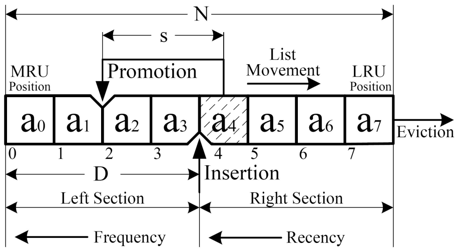

Fig. 1 Insertion and promotion based on the LRU replacement structure.

splits the list into two parts, namely, the left and right sections. All inserted Lines with high locality have the opportunity to be reused and promoted to the left section when they travel from the middle position to the LRU position, but lines with poor locality such as dead-on-arrival lines will smoothly pass by the right section and finally get out of the cache. In particular, AIP promotes reused lines with several positions left each time, rather than a single increment. With this interval promotion, frequently reused lines are promoted to the left section of the cache list where they have a relatively long lifetime in the cache. In contrast, the occasionally reused lines will stay in the right section of the cache list where they are pushed right by the inserted lines step by step and finally evicted. Thus, the traditional LRU and LFU policies are deliberately combined in AIP. To our knowledge, ours is the first study to investigate interval promotion in cache management. Earlier studies have proposed the promotion of a reused line either to the MRU position or to the left position beside the line. However, these proposals are of no benefit to the frequently reused lines because they are incapable of distinguishing different frequencies on the lines. Moreover, one-step promotion in the right section will soon be offset by the list movement to the right when a missed line arrives.

Several important parameters are depicted in Fig. 1. N denotes the number of ways in a cache set, D denotes the distance from the MRU position to the insertion position ($D \leq N$), and s denotes the promotion interval that is generally set to be $2, 4 \cdots n^2$. Thus, cache efficiency can be evaluated by a function as follows:

$$\lambda = f(D, s) \tag{1}$$

where λ stands for the cache efficiency, which can be derived from the hit ratio, that is, the division between the number of cache hits and the total number of cache accesses. Both values can be tracked as the program is running.

Parameter D indicates the insertion position. Scans and dead-on-arrival lines will be placed on the right section and eventually be evicted from the cache without polluting the left section, where valuable lines can be preserved. Thus, at least a fraction of a working set which is larger than the available cache size, can be retained in the left section to provide cache hits, resulting in a resistance to thrashing. At the same time, a number of high frequently reused lines can also be preserved in the left section for more contributions to cache hits.

Parameter s controls the promotion interval. The value of s should be at least two so that the list movement to the right will not offset the promotion to the left. In other words, a line has a theoretically better chance of staying in the cache while promoted to the left at least twice as many steps as being pushed to the right when another line is inserted before it. This promotion is, in effect, similar to LFU replacement, where frequently reused lines are highlighted by accumulated counts.

Parameter s is the most important parameter that distinguishes our proposal from that of others on combined LRU and LFU policies. With a reasonable value for parameter *s*, promotion can be performed several times until the line enters the MRU position. Thus, the frequently reused lines are highlighted by continuous movement to the left, but the infrequently reused lines are highlighted when a hit occurs and are soon decayed after the emergent hits. On the other hand, the limited advance to the left alleviates the effect of the dead lines in the left section, which often occurs seriously as stale lines under the LFU replacement policy because of its accumulated counts for frequently reused lines.

3.2 Adaptation of Insertion and Promotion Policies

The AIP mechanism contains two tunable variables, namely, *D* and *s*. *D* is relative to recency, and *s* is relative to frequency. *D* and *s* should be adaptively tuned individually in response to an evolving workload in terms of increased cache efficiency. A decreased *D* benefits lines with locality because more spaces are reserved for them in the right section of the cache list. In contrast, an increased *D* results in an expanded left section where more frequently reused lines can be preserved. In the presence of scans or thrashing, relatively fewer hits will occur in the right section compared with the left section. Hence, the adaptation rule on cache efficiency will tend to increase the value of *D* at the expense of the shorter right section, thus decreasing the residency time of scans and thrashing in the list. Meanwhile, lines that are more valuable are preserved in the left section, preventing the effects of scans and thrashing on the cache. Even though, *D* should be limited to a particular scope so that a new line with temporal locality can traverse all the way from the insertion position to the LRU position for several cycles, whereas any hit on this line will promote it back to the left. Without a proper limitation on the value of *D*, the new line will be evicted soon after its insertion, such as a scan, which damages the working set with temporal locality. On the other hand, when the workload exhibits high locality, *D* should be pushed back to the left while an increased cache efficiency is achieved. In the extreme scenario of *D* being set to zero, this scheme degenerates to the LRU replacement policy by inserting a missed line in the MRU position. If the value of *D* is seven, the scheme is effectively identical to the LIP. Theoretically, the preferable value of *D* is half of the set associativity; however, it should be adapted according to different access patterns on cache efficiency.

The variable *s* introduces frequency of reused lines into the scheme. An increased *s* results in increased highlights on the reused lines, thus more lines can be promoted to the left section of the cache list. In contrast, a decreased *s* prevents some frequently reused lines from entering into the left section of the cache list.

To simplify the adaptive process, the variable *s* should be initially fixed with a preferable value, so that variable *D* can be tuned according to the feedback value of the hit ratio (λ), which can be obtained by tracking the cache hits and accesses across pieces of workload. After the value of *D* is determined under a relatively stable λ, *s* is tuned in terms of the improved λ. The adaptation here is triggered automatically by a threshold value (ε) set for λ and stops when the value of λ falls into a small scope. Figure 2 depicts the pseudocode of the adaptive tuning algorithm.

```
Main Function: (Threshold: ε)
 1: Sample λ
 2: If λ < ε Then
 3:     Decrement D
 4:     Sample λ
 5:     If λcurr > λprev Then
 6:         If λcurr > ε Then
 7:             Exit
 8:         Else
 9:             Goto 3
10:         EndIf
11:     Else
12:         Increment D
13:         Sample λ
14:         If λcurr ⩾ λprev Then
15:             If λcurr > ε Then
16:                 Exit
17:             Else
18:                 Goto 12
19:             EndIf
20:         Else
21:             Call FuncTuneS
22:         EndIf
23:     EndIf
24: EndIf
```

```
Subroutine FuncTuneS:
 1: Increment s
 2: Sample λ
 3: If λcurr > λprev Then
 4:     If λcurr > ε Then
 5:         Exit
 6:     Else
 7:         Goto 1
 8:     EndIf
 9: Else
10:     Decrement s
11:     Sample λ
12:     If λcurr ⩾ λprev Then
13          If λcurr > ε Then
14:             Exit
15:         Else
16:             Goto 10
17:         EndIf
18:     Else
19:         Return
20:     EndIf
21: EndIf
```

Fig. 2 Pseudocode of the adaptive tuning algorithm.

4. Experiments and Analysis

4.1 Experiments

All simulations were run on the execution-driven gem5 simulator, which merges the best aspects of M5 and GEMS [14]. Gem5 provides cycle accurate simulation of a complete out-of-order processor with a hierarchical memory system. For the purposes of this study, only the LLC aspects of gem5 were modified to provide LLC management under AIP policies. The baseline LLC is a 2 MB eight-way set associative cache with the LRU replacement policy for all performance comparisons. All caches in the simulation used a 64 B line size. Table 1 shows the baseline configuration for the simulation system.

The SPEC CPU2006 benchmarks used in our study were compiled for the ALPHA ISA with -O2 optimizations. The reference input set was used for each benchmark. We ran the benchmark to completion with about one-fifth of the instructions forwarded for system warm-up.

Initially, *D* was set as 4, and s was set as 2. The threshold of parameter λ was 0.95 while the stable scope was 0.04. The value of λ in the experiment was calculated as follows:

Table 1 Baseline configuration for the simulation system.

Parameter	Value
Processor	Single-core; 8 wide decode/commit; 8 wide issue; out-of-order
Branch predictor	Tournament; BTB:4096; RAS:16; Choice:8192
L1 I-cache	64KB; Line size: 64B; two-way LRU repl.; 1 ns latency
L1 D-cache	64KB; Line size: 64B; two-way LRU repl.; 1 ns latency
L2 cache(LLC)	2MB; Line size: 64 B; eight-way LRU repl.; 10 ns latency
Memory	infinite size; 300 ns latency
Simulation mode	System-call Emulation

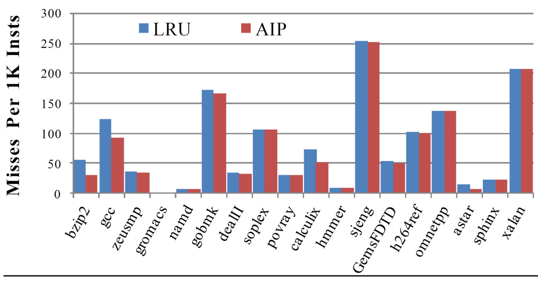

Fig. 3 Performance comparison of AIP and LRU.

$$\lambda = \frac{h}{n} \tag{2}$$

where h denotes the number of cache hits, and n denotes the total number of cache accesses. Both of the numbers were tracked in the adaptation period and sampled once with every 1 M instructions. While λ was less than 0.95, D was tuned first, and then s was determined after D.

Figure 3 details the performance comparison measured on the Misses Per 1000 Instructions (MPKI) metric of the baseline system with the LRU replacement policy, and the same system with the AIP policies. With fluctuations in D from 2 to 5 and in s from 2 to 4 for different benchmarks, the AIP system consistently outperformed the baseline system by an average of 5.8% on MPKI. AIP's strong performance results from its effective cache management, which can be viewed as a combination of the LRU, LFU and LIP policies.

4.2 Analysis

The set associativity affects the residency time of referenced lines. Sometimes a few lines are evicted before they are reused because the re-reference interval is greater than the available set associativity. While keeping the same cache size, we repeated the experiments with different configurations of set associativity from 16-way to 64-way and 128-way. The performance for benchmarks with frequently reused lines in distance interval was improved gradually. For brevity, these results were omitted because more experiments are necessary before we can confirm this judgment.

We explored the preferable value of D under the conditions wherein the best performance was achieved and found that it's about half of the set associativity, which differs from results reported in previous studies [1], [5]. In our view, a middle insertion in the list is a tradeoff between localities and thrashing. Meanwhile, this method is effective in preventing scans, as well as stale lines, by evicting them as soon as possible.

5. Conclusion

Based on LRU replacement, the AIP provides dynamic insertion and changeable promotion under a single unified scheme for workloads with temporal locality, as well as with frequently reused lines. AIP is effective in scan-resistance, thrash-resistance, and dead-line removal. Furthermore, it is simple, flexible and scalable because neither complicated computations for trade-off between LRU and LFU nor dynamic arbitration between sets under different policies to choose the best one are required. As a result, the integration of AIP into existing LRU approximations is easy, with minimal changes to LLC.

AIP can be extended to a shared LLC too. The major idea of AIP, such as middle insertion and interval promotion, can be used to protect a shared LLC from scans and thrashing. At the same time, valuable lines are retained in the left section of the cache list, contributing more hits too. Similar to PIPP [13], AIP partitions a shared LLC approximately among demanding applications targeting at maximum cache efficiency, as well as relative fairness. As a result, it makes full use of the cache space, overcoming the problem of degraded cache efficiency incurred by the strictly partitioning approaches. On the other hand, AIP should be improved because the original feedback mechanism cannot be effectively used in a sharing LLC. We are now exploring new ways for AIP to learn each application's characteristic in a multi-core system.

References

[1] A. Jaleel, K.B. Theobald, S.C. Steely Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," Proc. ISCA'10, pp.60–71, Saint-Malo, France, June 2010.
[2] J.T. Robinson and N.V. Devarakonda, "Data cache management using frequency-based replacement," Proc. ACM SIGMETRICS'90 Conf. Measurement and Modeling of Computer Systems, pp.134–142, New York, USA, May 1990.
[3] D. Lee, J. Choi, J. 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 Trans. Comput., vol.50, no.12, pp.1352–1360, Dec. 2001.
[4] N. Megiddo and D.S. Modha, "ARC: A self-tuning, low overhead replacement cache," Proc. FAST'03, pp.115–130, San Francisco, USA, March 2003.
[5] M.K. Qureshi, A. Jaleel, Y.N. Patt, S.C. Steely Jr., and J. Emer, "Adaptive insertion policies for high-performance caching," Proc. ISCA'34, pp.381–391, San Diego, USA, June 2007.
[6] R. Subramanian, Y. Smaragdakis, and G. Loh, "Adaptive caches: Effective shaping of cache behavior to workloads," Proc. MICRO-39, pp.385–396, Orlando, USA, July 2006.
[7] M.K. Qureshi, D.N. Lynch, O. Mutlu, and Y.N. Patt, "A case for MLP-aware cache replacement," Proc. ISCA'33, pp.167–178, Boston, USA, July 2006.
[8] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," IEEE Trans. Comput., vol.57, no.4, pp.433–447, April 2008.
[9] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," Proc. Micro-41, pp.222–233, Lake Como, Italy, Nov. 2008.
[10] M. Chaudhuri, "Pseudo-LIFO: The foundation of a new family of replacement policies for last-level caches," Proc. Micro-42, pp.401–412, New York, USA, Dec. 2009.
[11] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez, "Scavenger: a new last level cache architecture with global block priority," Proc. Micro-40, pp.421–432, Chicago, USA, Dec. 2007.
[12] K. Rajan and G. Ramaswamy, "Emulating optimal replacement with a shepherd cache," Proc. Micro-40, pp.445–454, Chicago, USA, Dec. 2007.
[13] Y. Xie and G.H. Loh, "PIPP: Promotion/insertion pseudo-

partitioning of multi-core shared caches," Proc. ISCA'36, pp.174–183, Austin, USA, June 2009

[14] N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, and D.A. Wood, "The gem5 simulator," ACM SIGARCH Computer Architecture News, vol.39, no.2, pp.1–7, May 2011.