라스트 레벨 캐시 성능 향상을 위한 캐시 교체 기법 연구
(A New Cache Replacement Policy for Improving Last Level Cache Performance)

두콩 튜안† (Cong Thuan Do) 손동오† (Dong Oh Son) 김종면** (Jong Myon Kim) 김철홍*** (Cheol Hong Kim)

요약 캐시 교체 기법은 캐시 미스를 감소시키기 위해서 개발되었다. 마이크로프로세서와 주기억장치의 속도 차이를 해결하기 위해서는 캐시 교체 기법의 성능이 중요하다. 일반적인 캐시 교체 기법으로는 LRU 기법이 있으며 대부분의 마이크로프로세서에서 캐시 교체 기법으로 LRU 기법을 사용한다. 그러나, 최근의 연구에 따르면 LRU 기법과 최적 교체(OPT) 기법 간의 성능 차이는 매우 크다. LRU 기법의 성능은 많은 연구를 통해서 검증되었지만, 캐시 사상방식이 높아짐수록 LRU 기법과 OPT 기법의 성능 차이는 증가한다. 본 논문에서는 기존의 LRU 기법을 활용하여 캐시 성능을 향상시키는 캐시 교체 기법을 제안하였다. 제안된 캐시 교체 기법은 캐시 블록의 접근율에 따라 교체 대상을 선정하여 캐시 내의 블록을 교체시킨다. 제안된 캐시 교체 기법은 512KB L2 캐시에서 기존의 LRU 기법과 비교하여 평균 15%의 미스율 감소시켰고, 프로세서 성능은 4.7% 향상된 것을 알 수 있다.

키워드: 캐시 교체 기법, LRU, 라스트 레벨 캐시, 캐시 적중율

Abstract Cache replacement algorithms have been developed in order to reduce miss counts. In modern processors, the performance gap between the processor and main memory has been increasing, creating a more important role for cache replacement policies. The Least Recently Used (LRU) policy is one of the most common policies used in modern processors. However, recent research has shown that the performance gap between the LRU and the theoretical optimal replacement algorithm (OPT) is large. Although LRU replacement has been proven to be adequate over and over again, the OPT/LRU performance gap is continuously widening as the cache associativity becomes large. In this study, we observed that there is a potential chance to improve cache performance based on existing LRU mechanisms. We propose a method that enhances the performance of the LRU replacement algorithm based on the access proportion among the lines in a cache set during a period of two successive replacement actions that make the final replacement action. Our experimental results reveals that the proposed method reduced the average miss rate of the baseline 512KB L2 cache by...
Cache management is increasingly important in designing modern processors especially when the performance gap between processor and main memory is continuously widening. Hence, the design of an effective memory hierarchy becomes even more crucial in order to reduce the average memory access time seen by the processor. In fact, effective last level caches have been the center of much research work in recent years.

A current trend in designing of last level caches is increasing in cache capacity and cache associativity. With non-blocking loads and high miss latencies, last level cache lookup is not as time critical, therefore, enabling larger cache associativity. For instance, in case last level caches becomes 8-way or 16-way set associative, strategies that emphasize reuse of cache lines already present in the cache might be more important than those that target conflict misses. With highly-associative caches and more corresponding victim choices, it might be time to revisit the conventional wisdom that cache replacement algorithms are not much important of a performance factor.

One of the most popular replacement policies is Least Recently Used (LRU). This policy generally performs well and has remained as the de-facto standard for replacement policy over the last several decades. LRU replacement policy and its approximations are practical for reasonable set sizes. However, some research work [1] has shown that in the cache of larger associativity there is noticeable room for improvement between LRU replacement policy and optimality as in the off-line MIN [2] or the equivalent OPT algorithms [3]. Therefore, increasing the number of lines per set can raise the probability that the least recently used line is not the optimal victim. This suggests that alternative replacement may be able to improve performance significantly over the LRU replacement policy.

In this paper, we propose a method to improve performance of LRU replacement policy that increases the effectiveness of last level cache (second-level cache in this study). The basic idea is that we modify the original LRU replacement policy to help it select the victim more accurately. More specifically, the proposed policy is based on the proportion of accesses (access proportion) among cache lines in the corresponding set during the period between two consecutive LRU replacements to make the final decision. According to our evaluations, the proposed replacement policy reduces the number of cache misses on last level cache by 15% on average over the pure LRU replacement policy that translates into an average performance improvement by 4.71% on selected workloads from SPEC CPU2000.

The rest of this paper is organized as follows. In section 2 we will summarize the related work. Section 3 describes our proposed technique. Section 4 shows the experimental results and discussion. Finally, section 5 concludes this paper.

2. Related Work

Cache replacement policies have received much attention from both industry and academia. Many studies have been researched about replacement algorithms for decades. Belady’s MIN algorithm [2] is an off-line algorithm that gives the minimum number of page faults for a given program. Mattson introduced OPT algorithm [3] which, like LRU, has the “stack” property. Moreover, variations on LRU [4,5] have been extensively studied. Two types of dead-block predictors, sequence-based prediction [6,7] and time-based prediction, replace dead blocks as soon as possible. Cost-sensitive replacement algorithms have been proposed to reduce the cost of cache misses rather than miss rates [8,9]. Recently, it is a renewed interest in bypassing. Several cache bypassing schemes have been proposed and they can be categorized into static and dynamic approaches. Static approaches [10,11] rely on the profile-guided compiler to identify lines that should bypass the cache. Dynamic cache bypassing uses runtime behavior learning and prediction to identify bypassing opportunities [12,13].
3. Proposed Technique

In this section, we will briefly describe LRU replacement policy and then explain the implementation of cache line structure in last level cache as well as the algorithm.

3.1 LRU Replacement Policy

Under LRU replacement policy, the data in cache memory is sorted by the last used time. On a request to cache line \( x \), if there is a miss and the cache is full, the line at the least recently used position will be evicted. Theoretically, LRU replacement policy tries to accommodate temporal locality by keeping recently used lines away from replacement in the hope that, when they are used, they will still be in the cache. Thus, LRU replacement policy has the advantage of good performance for high-locality workloads. However, it may have a pathological behavior for memory intensive workloads which have a working set greater than the available cache size.

3.2 Cache Line Structure

Figure 1 shows the cache line structure used by the proposed replacement policy. In this work we modeled a processor architecture that includes two levels of cache hierarchy. The first level cache consists of two separate caches, instruction cache and data cache while the second level cache is unified cache. Therefore, the second level cache is the last level cache. In order to adapt the proposed policy, several fields were added to each cache line of the last level cache to keep the profile of each cache line during the period between two consecutive LRU replacement events. As depicted in the figure, first three fields are original fields of each cache line that help the processor to access the cache. The fourth field, \( H \) field, is used to keep the number of cache hits to the own line while the last field, \( A \) field, keeps the number of cache accesses to the corresponding set. In fact, these fields act like increment counters, this can reduce logic complexity.

In order to make a replacement decision on a L2 cache miss, the mechanism needs to track all the number of set accesses and the number of hits for each line. Therefore, the size of \( A \) and \( H \) directly affects the cache performance. It is clear that a particular size of \( A \) and \( H \) cannot satisfy all cases since it depends on characteristics of workloads. However, due to hardware overhead, the range of \( A \) and \( H \) should be large enough to satisfy in almost all cases. We find that 5 bits for each additional filed is appropriate for our workloads. This bit-width guarantees each counter can count up to 31. This bit-width is referenced from the experiments to obtain a tradeoff between hardware overhead and performance. The impact of average L2 cache miss rate on various size of \( A \) and \( H \) is shown in Figure 2. When the size of \( A \) and \( H \) is increased beyond the need, not only the performance is not improved remarkably, but it also wastes hardware areas. On the other hand, if the range of \( A \) and \( H \) is not large enough, the counter will be overflowed and thus, the replacement decision will not be accurate resulting in high miss rate. With 5 bits for each extra field, the storage overhead per cache line is \( 5 + 5 = 10 \) bits, a reasonable size in comparison to a typical size of a last level cache line of 64 or 128 bytes.

![Modified Cache Line Structure and Memory Architecture](image1)

![Average Cache Miss Rate with Various Sizes for A and H](image2)
3.3 Algorithm Details

The algorithm implementation is shown in Table 1. During the period of two consecutive LRU replacements, on an access to a line \( l \) in a set \( s \), the counter \( A \) of the line is incremented. If \( l \) is found in the cache, there is a hit and the line is sent to processor, then the counter \( H \) of line \( l \) is incremented. If, on the other hand, \( l \) is not available in any cache - a miss, a victim line from the corresponding set needs to be selected for replacement. In order to find the victim, all lines in the corresponding set are checked for expiration. A line \( b \) is considered safe if the ratio between \( b \)'s number of hits and \( b \)'s number of set accesses, \( b.H/b.A \), is greater than or equal to the threshold \( \theta \) (\( \theta = \text{set associativity}/A \)), otherwise, it is considered expired. After expired lines are identified, a line \( x \) that is at the least recently used position among expired lines in the LRU chain is chosen for replacement. If the group of expired lines contains the original least recently used line, the victim line will be the original least recently used line. After \( l \) is placed from the main memory to the cache, \( H \) and \( A \) counters of all cache lines in the corresponding set are reset to start a new process.

The basic idea of this algorithm is based on the fact that when the access proportion of a line in the set is distributed non-uniformly, it may become the LRU line despite of its high access frequency before and it can be accessed again. By doing that, the proposed replacement policy can help the original LRU replacement policy selects victim blocks more appropriately. Logically, the cache line has the lowest hit proportion should be the new victim in our algorithm, however, we wanted to utilize the advantage of LRU policy. Hence, the algorithm sets a threshold to receive a number of potential victims and based on the position of those potential victims in the LRU chain, the algorithm will decide the final victim. In this algorithm, the value of the threshold is chosen by an assumption that at first each cache line has an equal probability to be a potential victim. If its hit proportion is lower than the threshold, it is considered expired, otherwise, it is considered safe.

Figure 3 illustrates the working of algorithm by some examples. This example depicts a cache set in an 8-way set associate cache. The lines are sorted from the MRU (left) to the LRU (right). In this example, we assume that the number of hits during the period of two consecutive LRU replacements is 25. The access number 26 to the set is a miss on the line \( I \). In case the cache is full, the replacement policy needs to find a victim line. The threshold \( \theta \) is 0.32 (8/25 = 0.32). Based on the proposed algorithm, replacement policy selects victim blocks more appropriately.

Table 1 Algorithm Implementation

<table>
<thead>
<tr>
<th>During a period of two consecutive original LRU replacements: On an access to line ( l ) in set ( s ):</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Increment the number of accesses ( A ) to the set ( s ):</td>
</tr>
<tr>
<td>( s[i].A = s[i].A + 1 )</td>
</tr>
<tr>
<td>2. If there is a hit, increment ( H ) counter of ( l ):</td>
</tr>
<tr>
<td>( LH = LH + 1 )</td>
</tr>
<tr>
<td>3. If there is a miss, identify expired lines in the set ( s ):</td>
</tr>
<tr>
<td>a. Line ( b ) is safe if</td>
</tr>
<tr>
<td>( b.H/b.A \geq \text{threshold } \theta )</td>
</tr>
<tr>
<td>else ( b ) is expired</td>
</tr>
<tr>
<td>b. Find the victim ( x ):</td>
</tr>
<tr>
<td>if there is at least one expired line in the set ( s ):</td>
</tr>
<tr>
<td>( x = \text{the LRU among the expired lines} )</td>
</tr>
<tr>
<td>if the group of expired lines contains the original LRU:</td>
</tr>
<tr>
<td>( x = \text{the original LRU line} )</td>
</tr>
<tr>
<td>c. Place line ( l ) from the main memory into the cache</td>
</tr>
<tr>
<td>d. Reset information (( H, A ) counters) of all lines in the set ( s )</td>
</tr>
</tbody>
</table>

![Fig. 3 Example of Algorithm Implementation for an 8-way Set](image-url)
the lines which have $H/A$ value greater than or equal to 0.32 are safe lines, the other lines are in the group of expired lines. The replacement policy will select the least recently used line among expired lines as the victim line. In example $a$, the victim line is $H$, the proposed replacement policy have the same decision with the original LRU replacement policy. However, in example $b$, the proposed replacement policy chooses line $F$ as the victim line instead of choosing line $H$ according to the original LRU policy.

**Hardware Overhead and Complexity:** With extra 10 bits per cache line, assuming the size of last level cache is 512 KB, 64 B cache line; we can calculate the total hardware overhead that is: $(512\text{KB}/64\text{B}) \times 10$ bits $= 10$ KB, about 2% of last level cache size, it is a small addition area in comparison to typical sizes of last level cache ranging from a half to a few megabytes. Moreover, the proposal needs some 5-bit dividers for integer numbers and comparators to do logic computations in the algorithm. In the normal case, an integer division operation requires some CPU cycles in modern processors. Please note that the division operation in the algorithm occurs on a L2 cache miss only, it means that the mechanism has ample time to implement this operation while waiting the requested data due to high latency between L2 cache and main memory (hundreds of CPU cycles). Although our proposal may cause some logic complexity when updating counters and finding expired lines, using $A$ and $H$ as increment counters can reduce remarkably the complexity of the mechanism. About selecting the victim, in case a cache line is considered expired, the mechanism can use a bit to mark it. Furthermore, because the proposed policy utilized the existing mechanism of LRU policy, based on the position cache lines in the LRU chain, the mechanism can look for the victim quite simply.

### 4. Experimental Results

We extended SimpleScalar toolset v.3.0 [14], an event-driven simulator that provides detailed instruction-level and cycle-driven simulation, to evaluate our proposed replacement policy. Table 2 shows the parameters used for component of the simulated architecture. Our workloads are selected from the SPEC CPU2000 benchmarks [15] including both integer and floating-point benchmarks. These workloads are compiled for little-endian alpha binary using gcc compiler.

Figure 4 shows cache miss rates of various cache replacement policies. Our replacement policy was compared against FIFO, random (RND), LRU and OPT replacement policies. As shown in the figure, the OPT algorithm achieved the lowest miss rate as expected, however, this algorithm is not practical. The average cache miss rates under FIFO, RND and LRU replacement policies are 35.44%, 33.45% and 35.66%, respectively. We can see that there is a little difference in terms of miss rate on selected workloads of various typical replacement policies. Meanwhile, the proposed policy achieved the lowest miss rate on average in comparison to the three policies above, 30.29%, an improvement of about 15% over FIFO, RND and LRU replacement policies. If we look at closely, we can find that the proposed policy achieved considerable improvement on almost floating-point workloads such as *apsi*, *art* or *mgrid* (for instance, 42.16% in case of *apsi*). This reflects the proposed policy is friendlier with floating-point workloads than with integer workloads.

Figure 5 presents IPC of FIFO, RND and LRU replacement policies as well as the proposed replacement policy. Overall, the IPC of LRU replacement policy is better than FIFO and RND replacement policies although in some cases cache miss rates under FIFO replacement policy or RND replacement policy are lower than that under LRU replacement policy. As shown in Figure 5, the average performance of the proposed replacement policy is the best on selected applications from SPEC CPU2000. Figure 6
shows the percentage of performance improvement of different cache replacement policies normalized to LRU replacement policy. On average, the proposed replacement policy achieved a speedup of 4.71% over the original LRU replacement policy without slowing down any workload, demonstrating its robust performance. For example, mgrid shows the greatest performance gain, up to 15.4% speedup over the LRU replacement policy. In addition, the proposed replacement policy outperformed FIFO and Random replacement policies, providing an improvement of 6.03% over FIFO replacement policy and 7.67% over the RND replacement policy. From the experimental results, we found that the improvement in miss rate does not result in linear improvement in IPC. For example, mgrid did not achieve the largest miss rate improvement but it obtained the largest IPC improvement. Based on our analysis, there are many factors affecting on the relation between miss rate improvement and IPC improvement. For instance, recent research work [8] has proved that miss cost do not have the same cost. More specifically, this work proved that store and load instructions have different costs, this explains in part why some benchmarks achieved high improvement in terms of miss rate but not in terms of IPC.

5. Conclusions

In this study, we have presented a new cache replacement algorithm that based on the LRU replacement algorithm. In this approach, we modify the original LRU replacement algorithm to help it selects the victim block more accurately. The proposed cache replacement policy is based on access proportion among lines in the corresponding set during the period of two consecutive LRU replacements to make the final replacement decision.

Our proposed cache replacement policy performed almost well, reducing cache miss rate on selected workloads of SPEC CPU2000 by 15% on average that leads to an average performance improvement of 4.71% over the pure LRU replacement policy.

References


Do Cong Thuan

손 동 오
2010년 전남대학교 전자컴퓨터공학부 공학사. 2012년 전남대학교 전자컴퓨터공학부 석사. 2012년~현재 전남대학교 전자컴퓨터공학부 박사과정. 관심분야는 컴퓨터구조, 임베디드시스템

김 종 멜

김 철 홍

김철홍


Do Cong Thuan