5. SHARING AND HIT BASED PRIORITIZING TECHNIQUE

The efficiency of cache memory can be enhanced by applying different replacement algorithms. An optimal replacement algorithm must exhibit good performance with minimal overhead. Traditional replacement methods that are currently being deployed across multi-core architecture platforms, try to classify elements purely based on the number of hits they receive during their stay in the cache. In multi-threaded applications data can be shared by multiple threads (which might run on the same core or across different cores). Those data items need to be given more priority when compared to private data because miss on those data items may stall the functioning of multiple threads resulting in performance bottleneck. Since the traditional replacement approaches do not possess this additional capability, they might lead to inadequate performance for most of the current multi-threaded applications.

To address this limitation, this chapter proposes a Sharing and Hit based Prioritizing (SHP) replacement technique that takes the sharing status of the data elements into consideration while making replacement decisions. Evaluation results obtained using multi-threaded workloads derived from the PARSEC benchmark suite shows an average improvement of 9% in the overall hit rate when compared to LRU algorithm.

5.1. Introduction

In multi-threaded applications, when threads run across different cores, the activity in L2 cache tend to shoot up as multiple threads try to store and retrieve shared data. At this point there are many challenges that need to be addressed. Cache coherence has to be maintained, the replacement decisions made need to be judicious and the available cache space must be utilized efficiently. When there is a miss on a data item which is shared by multiple threads, the execution of many threads gets stalled. This is because when the first thread, which had encountered a miss, is attempting to fetch the data from the next level of memory, any subsequent thread which tries to access the same data will result in miss.
Hence it becomes imperative to handle shared data with more care compared to other data items. Traditional replacement algorithms like LRU, MRU etc do not possess this capability. They classify elements based on when they will be required by the processor but do not check for the status of the cache block, i.e. whether it is shared or private at any point of time. Also when there are more than thousands of threads running in parallel, it may not be very useful in tagging the block simply as ‘shared’ or ‘private’. In those cases, additional information about their shared status can prove handy. This chapter provides an overview on a novel counter based cache replacement technique that associates a ‘sharing degree’ with every cache block. This degree specifies a set of ranges which includes – not shared (or private), lightly shared, heavily shared and very heavily shared. This information combined along with the number of hits received by the element during its stay in the cache is used to produce a priority for that element based on which judicious replacement decisions are made.

Contributions of this chapter includes,

- A novel counter based replacement algorithm that makes replacement decisions based on the sharing nature of the cache blocks
- Association of a ‘sharing degree’ with every cache line, which indicates the extent to which the element is shared depending on the number of threads that try to access that element.
- Four degrees of sharing namely – not shared (or private), lightly shared, heavily shared and very heavily shared.
- A replacement priority for every cache line which is arrived upon by combining the sharing degree along with the number of hits received by the element based on which the replacement decisions are made.

5.2. Sharing and Hit Based Prioritizing Method

Similar to the other two replacement algorithms, every cache block is associated with a counter. This 2-bit counter is called as the Sharing Degree Counter or SD counter. Table 5.1 shows all the four possible values that the counter can hold
and their individual descriptions. Depending on the number of threads that access a cache block, it is classified into any one of the sharing categories as shown in the table. To collect the sharing status of the cache blocks, it is essential to have an efficient data structure in place to track the number of threads that accesses the data item. For this purpose there is a filter which is referred to as the Thread Tracker filter (TT filter). It is a flexible dynamic software based array that gets created and is associated with every cache block during run time.

When a thread tries to access a data item, a search is conducted in the TT filter to check if the thread id is already present in it. If not, then the id is stored in the corresponding cache block’s TT filter. Size of the filter expands as and when a thread id gets added to it. Once a cache block is about to be evicted, the memory allocated for the associated TT filter is freed. At any point of time, with the help of thread ids that are found in the TT filter, the sharing degree counter is populated for every cache block.

### 5.3. Replacement Priority

The sharing status of the blocks alone cannot be used to make replacement decisions. Number of hits garnered by the element is also an important factor to consider when performing a replacement. Hence a priority for every individual cache block has been arrived at, that not only gives more weightage to sharing nature but also attaches importance to the number of cache hits garnered by the element. Priority computation equation is as follows

\[
\text{Priority} = (\text{Sharing Degree} + 1) \times \frac{\text{Hits Count}}{2} \quad \ldots \ldots (4)
\]

Sharing degree indicates the sharing degree counter value for the particular cache block at that particular point of time. A software based hit counter is associated with every cache block that provides an estimate of the hits received by the element during its stay in the cache. This counter is refreshed every time a
new element enters the block. The upper bound of the counter is fixed to a specific value. If that value is reached, the counter is considered saturated and any hits received beyond that will not be taken into account. Experimental results have shown that in a unified cache memory, the average number of instructions is 50% and the remaining 50% forms the data. For calculation of priority, the analysis is mainly upon the data part as it is accessed much more frequently compared to instructions. Also due to locality of reference, several threads can access the data several times. To highlight the former part, the hit count value is divided by 2 and to highlight the latter part, hit count value is scaled by a factor of the sharing degree counter value. Whenever any one of these counters change, the priority of the element needs to be recomputed.

Cache replacement policy consists of deletion, insertion and promotion operations. Each phase of the algorithm is explained in detail in the subsequent sections. The mapping method used is set associative mapping.

5.4. Replacement Policy

When the cache becomes full, replacement has to be made to pave way for new incoming data items. SHP makes replacement decisions based on the computed priority values. Elements are evicted in the increasing order of their priority. The element with the least priority in the list is chosen as the victim. If more than one element has the same least priority, then the one which is encountered first while scanning the cache is taken as the victim as a tie-breaking mechanism. It is also essential to ensure that stale data do not pollute the cache for longer periods of time.

From this aspect the hit counter can be regarded as an ageing counter. Every time a victim is found, the hit counter of all the other elements in the cache is decremented and their corresponding priorities are re-computed. If any element remains unreferenced for a long period of time, its priority will gradually decrease and the element will eventually be flushed out of the cache.
Algorithm 3: Implementation for Cache Miss, Hit and Insert Operations

Input:
A cache set instance

Output:
removing_index /* Block id of the victim */

Miss:
/** Invoke FindVictim method. Set id is passed as parameter **/
FindVictim(set)

FindVictim(set):

min = ∞
for i = 0 to associativity do
    /** Choose the minimum priority element **/
    if set.blk[i].priority < min then
        min = set.blk[i].priority;
        removing_index = i;
    return removing_index;

Hit:
/** Scan the TT Filter **/
for i = 0 to length(TT_Filter) do
    if TT_Fitler[i] == accessing_thread_id then
        thread_found = 1;
/** If thread id not found in TT Filter, add the thread id to it **/
if thread_found != 1 then
    TT_Filter[i] = accessing_thread_id;
set blk.sharing_degree counter value[0 to 3] according to TT_Filter length
priority = (set.blk.sharing_degree +1) * (set.blk.hit_counter)/2;
**Insert:**

```c
/** Private block **/
set.blk.sharing_degree = 0;
refresh blk.hit_counter;
```

**Correctness of Algorithm**

**Invariant:** At any iteration \( i \), the sharing degree counter value of set.blk[\( i \)] holds a value in the range \([0,3]\).

**Initialization:**

When \( i = 0 \), set.blk[\( i \)].sharing_degree will be set to ‘0’ initially. Hence invariant holds good at initialization.

**Maintenance:**

At \( i = n \), set.blk[\( i \)].sharing_degree will contain

either zero (Private) or

set.blk[\( i \)].sharing_degree will have ‘1’ (Lightly shared) or

set.blk[\( i \)].sharing_degree will have ‘2’ (Heavily Shared) or

set.blk[\( i \)].sharing_degree will hold ‘3’ (Very Heavily Shared).

Hence invariant holds good for \( i = n \).

Same case when \( i = n+1 \) as the individual iterations are mutually exclusive. Thus invariant holds good for \( i = n + 1 \).

**Termination:**

The priority computation loop is limited by the associativity of the cache which will always be finite. TT filter iteration is limited by the size of the TT filter which will be equal to the total number of threads executing. It will also be finite. The invariant can also be found to hold good for all the cache blocks present when the loops terminate.
Processor looks for a particular block ‘c’ in the cache

SHP algorithm starts search for block ‘c’ in the cache

If ‘c’ is found

Increment Hit Counter

HIT

MISS

Processor brings the data item from next level of memory

Find replacement victim in the cache

Update TT Filter if needed and sharing degree counter

Re-compute priority

Finish

Fig 5.1 (i) SHP flow diagram
**Find replacement victim in the cache**

SHP algorithm starts searching for a replacement candidate ‘c’ in the cache

**Search for first block ‘c’ which has the minimum priority value**

**Choose ‘c’ as replacement victim**

**Insert new data item here and set Sharing degree counter value to ‘0’ (private) and re-compute priority**

Fig 5.1 (ii) Victim Selection flow for SHP

---

**Table 5.1 Sharing degree values and their descriptions**

<table>
<thead>
<tr>
<th>Number of Accessing Threads</th>
<th>Sharing Degree Counter Value</th>
<th>Nature of Sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Private/Not Shared</td>
</tr>
<tr>
<td>2-3</td>
<td>1</td>
<td>Lightly Shared</td>
</tr>
<tr>
<td>4-7</td>
<td>2</td>
<td>Heavily Shared</td>
</tr>
<tr>
<td>8-10</td>
<td>3</td>
<td>Very Heavily Shared</td>
</tr>
</tbody>
</table>
5.5. Insertion Policy

After evicting the victim, the new data item needs to be inserted into the cache. Sharing degree counter is set to ‘0’ to indicate that the incoming block is currently not shared by any other threads. The corresponding hit counter is refreshed.

5.6. Promotion Policy

When a cache hit happens, the hit counter of the corresponding block is incremented. Also the accessing thread id is compared against the ids which are already present in the TT filter and if it is not present there, it is added into the TT filter. Sharing degree counter is adjusted accordingly and the priority is recomputed. Fig. 5.1(i), 5.1(ii) shows the basic flow of SHP technique.

5.7. Results

To evaluate the performance of SHP, simulation has been carried out with GEM5 simulator using PARSEC workloads (discussed in Chapter 6) and the results are compared with LRU. Most of the applications are benefited through this replacement policy. The results are shown in this section.

5.8. SHP Performance

It is observed from the following figures, that the performance of the proposed replacement technique is better than LRU for multithreaded workloads. The overall number of hits obtained at LLC cache for SHP method compared to LRU is shown in the graph in Fig. 5.2. In Fig. 5.2, blackscholes has shown the maximum improvement. On an average, a 9% percent improvement in the overall number of hits is observed across the given benchmarks.
Fig. 5.3 shows the overall number of replacements made at L2 for SHP and LRU. The more the number of replacements made, the more will be the overhead involved. So it is always desirable to keep this parameter as low as possible. In this method, the average number of replacements made at L2 has decreased compared to LRU. Compared to other benchmarks, blackscholes has exhibited the least number of replacements (1.6% lesser than that of LRU) and swaptions has shown a 0.55% decrease when compared to LRU.

Fig. 5.4 shows the L2 miss rate for different workloads. Miss rate is computed from the overall number of misses and the overall number of accesses. To obtain a high performance, LLC must provide a lower miss rate. Majority of the benchmarks have shown improvement in this metric when compared to LRU with dedup showing superior performance.
Fig 5.3 Percentage decrease in number of replacements made at L2

Fig 5.4 Overall miss rate and core-wise miss rate at L2 cache
5.9. Summary

Shared data plays a critical role in determining the performance of cache memory systems, especially in a multi-threaded environment. Conventional LRU approach does not attach importance to such data and hence this chapter proposes a novel counter based prioritizing algorithm.

- Every cache block is associated with a 2-bit sharing degree counter which iterates from 0 to 3.
- A dynamic software based TT filter is associated with every block to keep track of the threads that are accessing that block.
- A hit counter is used to keep track of the hits received by the data item.
- Values of the sharing degree and the hit counters are used to compute a priority for each cache block.
- This priority is then used to make judicious replacement decisions.

Evaluation results have shown an average improvement of up to 9% in the overall number of hits when compared to the traditional LRU approach.