4. LOGICAL CACHE PARTITIONING TECHNIQUE

It is imperative for any level of cache memory in a multi-core architecture to have a well defined, dynamic replacement algorithm in place to ensure consistent superlative performance [92,93,94,95]. At L2 cache level, the most prevalently used LRU replacement policy does not acquaint itself dynamically to the changes in the workload. As a result, it can lead to sub-optimal performance for certain applications whose workloads exhibit frequently fluctuating patterns. Hence many works have strived to improve performance at the shared L2 cache level [91,98,99].

To overcome the limitation of this conventional LRU approach, This chapter proposes a novel counter-based replacement technique which logically partitions the cache elements into four zones based on their ‘likeliness’ to be referenced by the processor in the near future. Experimental results obtained by using the PARSEC benchmarks have shown almost 9% improvement in the overall number of hits and 3% improvement in the average cache occupancy percentage when compared to LRU algorithm.

4.1. Introduction

Cache hits play a vital role in boosting the system performance. When there are more hits, the number of transactions with the next levels of memories is reduced thereby expediting the overall data access time. Many replacement techniques use the cache miss metric to determine the replacement victim but in this chapter, choosing the victim is based on the number of hits received by the data items in the cache. Based on the hits received, the cache blocks are logically partitioned into separate zones. These virtual zones are searched in a pre-determined order to select the replacement candidate.

Contribution of this chapter includes,

- A novel counter based replacement technique that logically partitions the cache into four different zones namely – Most Likely to be Referenced (MLR),
Likely to be Referenced (LR), Less Likely to be Referenced (LLR), Never Likely to be Referenced (NLR) in the decreasing order of their likeliness factor.

- Replacement, insertion and promotion of data elements within these zones in such a manner that the overall hit rate is maximized.
- Association of a 3-bit counter with every cache line to categorize the elements into different zones. On a cache hit, the corresponding element is promoted from one zone to another zone.
- Replacement candidates selection from the zones in the ascending order of their ‘likeliness factor’ i.e. the first search space for the victim would be the never likely to be referenced zone, followed by the subsequent zones till the most likely to be referenced zone is reached.
- Periodic zone demotion of elements also occurs to make sure that stale data does not pollute the cache.

4.2. Logical Cache Partitioning Method

LCP uses a 3-bit counter to dynamically shuffle the cache elements and logically partition them into four different zones based on their likeliness to be referenced by the processor in the near future. This counter will be referred to as LCP (Logical Cache Partitioning) counter in subsequent sections. The lower and upper bounds for the counter are set to ‘0’ and ‘7’ respectively. The prediction about the likeliness of reference of the data items is made with the help of the hits encountered in the cache. As the number of hits for a particular element increases, it is moved up the zone list till it reaches the MLR region.

Only if it stays unreferenced for quite an amount of time, it is evicted from the cache. As their names imply, the zones are arranged in the decreasing order of their likeliness factor. Table 4.1 shows the counter values and their corresponding zones. Any replacement policy consists of three phases- Replacement, Insertion and Promotion and so does this method. Each of which is explained in the subsequent sub-sections.
Table 4.1 LCP zone categorization

<table>
<thead>
<tr>
<th>Counter Value Range</th>
<th>Zone</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-7</td>
<td>MLR</td>
</tr>
<tr>
<td>3-5</td>
<td>LR</td>
</tr>
<tr>
<td>1-2</td>
<td>LLR</td>
</tr>
<tr>
<td>0</td>
<td>NLR</td>
</tr>
</tbody>
</table>

### 4.3. Replacement Policy

When a miss is encountered in the cache, the data item is fetched from the secondary memory and brought into the cache thereby replacing one of the elements which was already present in the cache. This element is often referred to as the victim. The process of selecting a victim needs to be efficient in order to improve the hit rate. In this method, the victim is selected from the zones in the increasing order of their likeliness factor. Any cache line which comes under the NLR zone is considered first for replacement. If no such line is found search is performed again to check if any element falls in the LLR region and so on till MLR is reached. If two or more lines possess the same counter value that is considered for replacement during that iteration, the line that is encountered first is chosen as the victim.

Once the replacement is made, LCP counters of all the other data elements are decremented by ‘1’. This is done to carry out gradual zone demotion as discussed earlier to flush out unused data items from the cache.

### 4.4. Insertion Policy

This phase is encountered as soon as the replacement candidate is found. The new incoming data item is inserted into the corresponding cache set and its LCP counter value is set to ‘2’ (LLR zone).
4.5. Promotion Policy

A hit on any data item in the cache calls for the promotion phase. The LCP value associated with the cache line is incremented to the final value of its immediate upper zone. For example, if it was earlier in the LLR zone (LCP value ‘1’ or ‘2’), its LCP value is set to ‘5’ i.e. the element now falls within the LR zone. By transferring elements within the zone according to the workload pattern, LCP has proven to be more dynamic than LRU.

Figure 4.1(i) LCP flow diagram
4.6. Boundary Condition

It is essential that the LCP counter value does not overshoot its specified range. Thus whenever it is modified, a boundary condition check is carried out to ensure that the value does not go below ‘0’ or beyond ‘7’. Fig. 4.1(i), 4.2(ii) shows the working of LCP.

---

**Fig 4.1 (ii) Victim Selection flow for LCP**

1. **Find replacement victim in the cache**
   - LCP algorithm starts searching for a replacement candidate ‘c’ in the cache from zone ‘X’ where ‘X’ denotes NLR initially.
2. **Search for first block ‘c’ which has the LCP counter value that falls within the range of zone ‘X’**
   - If ‘c’ is found:
     - Choose ‘c’ as replacement victim
     - Insert new data item here and set LCP counter value to ‘2’ (LLR zone)
   - No:
     - Increment ‘X’ to point to the next higher zone
3. **If ‘c’ is found**
   - Yes:
     - Choose ‘c’ as replacement victim
     - Insert new data item here and set LCP counter value to ‘2’ (LLR zone)
Algorithm 2: Implementation for Cache Miss, Hit and Insert Operations

Init:
Initialize all blocks with LCP counter value ‘0’.

Input:
A cache set instance

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

Miss:
/** Invoke FindVictim method. Set is passed as parameter **/

FindVictim (set):
  \( i = 0 \)
  removing_index = -1
  while \( i < 8 \) do
    /** Scanning the set **/
    for \( j = 0 \) to associativity do
      /** Search from the NLR zone. LCP counter = 0 **/
      if set.blks[j].lcp == i then
        /** Block Found. Exit Loop **/
        removing_index = i;
        break;
      if removing_index ! = -1 then
        break;
    i = i + 1;
  for \( i = 0 \) to associativity do
    /** Gradual Zone Demotion **/
    if set.blks[j].lcp>0 then
      set.blks[j].lcp = set.blks[j].lcp – 1;
  return removing_index

Hit:
/** Boundary Condition Check **/
if set.blk.lcp ! = 7 then
/** NLR Zone. Promote to LLR **/
if set.blk.lcp == 0 then
    set.blk.lcp = 2;
/** LLR Zone. Promote to LR **/
else if set.blk.lcp == 1 or set.blk.lcp == 2 then
    set.blk.lcp = 5;
/** LR Zone. Promote to MLR **/
else
    set.blk.lcp = 7

Insert:
/** LR Zone **/
set.blk.lcp = 2;

Correctness of Algorithm

Invariant: At any iteration i, the LCP counter value of set.blk[i] holds a value in the range [0,7].

Initialization:
When i = 0, set.blk[i].lcp will be ‘0’ initially. After being selected as victim and after the new item has been inserted, set.blk[i].lcp will be set to ‘2’. Hence invariant holds good at initialization.

Maintenance:
At i = n, set.blk[i].lcp will contain
    either zero (Cache block yet to be occupied for the first time) or
    set.blk[i].lcp will have ‘2’ (If it has been newly inserted) or
    set.blk[i].lcp will have ‘1’ or ‘3’ or ‘4’ or ‘6’ (As a part of gradual zone demotion)
    set.blk[i].lcp will hold ‘5’ or ‘7’ (On a cache hit from previous zone).
“if blk.lcp ! = 7” condition ensures that the lcp counter value does not overshoot its range. **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.**

<table>
<thead>
<tr>
<th>LRU</th>
<th>LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Fig 4.2 Working of LRU and LCP for the data set
Termination:

Increment of the loop control variable ‘i’ ensures that the loop terminates within finite number of runs which is ‘8’ in this case. The condition "if set.blks[j].lcp>0" ensures that the lcp counter value of the block does not go below ‘0’. The invariant can also be found to hold good for all the cache blocks present when the loop terminates.

4.7. Comparison with LRU

The following data set is considered.

... 2 9 1 7 6 1 7 5 9 1 0 7 5 4 8 3 6 1 7 ...

It is observed that data items 1 and 7 occur frequently. Fig. 4.2 shows the working of LRU and LCP on this data pattern. Assume that the cache can hold 4 blocks at a time. Incoming data items at that point of time are shown in the leftmost column.

In the right most column, ‘m’ indicates a miss and ‘h’ indicates hit. Initially the cache contains invalid blocks. Counter values associated with all the blocks are set to -1. After applying both the techniques, LCP has resulted in 3 hits more than LRU. Frequently occurring data items 1 and 7 have resulted in hits towards the end unlike LRU. This is primarily because LRU follows the same approach irrespective of the workload pattern and tags 1 and 7 as ‘least recently used’.

4.8. Results

Full system simulation using Gem5 simulator with PARSEC benchmarks (discussed in Chapter 6) as input sets shows improvement in many cache metrics. To measure the performance of LCP, metrics like hit percentage increase, number of replacements made at L2, average cache occupancy and miss rate are used. Results are shown for two, four and eight core systems.
4.8.1. LCP Performance

Subsequently, the term LCP will be used to represent this method in the graphs. Percentage increase in the overall number of hits at L2 is shown in Fig. 4.3.

Fig 4.3 Percentage increase in overall number of hits at L2 cache

Fig 4.4 Difference in number of replacements made at L2 cache

Fig. 4.4 shows the total number of replacements made at L2. Fig. 4.5 shows the
average L2 cache occupancy percentage for all the workloads. Fig. 4.6 shows the miss rate recorded by the individual cores and also the overall miss rate at L2.

![Average cache occupancy (in percentage)](image)

**Fig 4.5 Average cache occupancy percentage**

Miss rate is defined as the ratio of number of misses to the total number of accesses. As it is observed from Fig. 4.6 there is reduction in the core-wise miss rate and overall miss rate across majority of the benchmarks when compared to LRU.

Apart from two workloads, all the others have shown significant improvement in hits. Percentage increase in hits varies from a minimum of 0.1% (*dedup*) to maximum of up to 11% (*ferret*) as shown in Fig. 4.3.

Number of replacements reflects the efficiency of any replacement algorithm. A maximum of almost 20% reduction in the number of replacements is observed across the given workloads compared to LRU from Fig. 4.4. Cache occupancy refers to the amount of cache that is being effectively utilized to improve the performance for any workload. Fig. 4.5 indicates cache utilization is higher for majority of the benchmarks when LCP is applied compared to LRU. *Vips* utilizes the cache space to the maximum possible extent.
This chapter discusses about a dynamic and a structured replacement technique that can be adopted across the LLC. Key points pertaining to this replacement policy are as follows:

- Elements of the cache are logically partitioned into four zones based on their likeliness to be referenced by the processor with the help of a 3-bit LCP counter associated with every cache line. The minimum value that the counter can hold is ‘0’ and the maximum value is ‘7’.

- Replacement candidates are chosen from the zones in the increasing order of their likeliness factor starting from the NLR zone. Initially all the counter values are set to ‘-1’. Conceptually all the blocks contain invalid data.

- For every hit, the corresponding element is moved up by one zone by adjusting the LCP counter value. If it has reached the top most zone (MLR) the counter value is incremented by one and so on.
value is left untouched.

- For every miss the counter value is decremented by ‘1’ to prevent stale data items from polluting the cache as the algorithm executes. When a new data arrives, its LCP value is set to ‘2’. Boundary condition check needs to be applied whenever the LCP counter value is modified to make sure that it does not overshoot its designated range.

Experimental results obtained by applying the method on PARSEC benchmarks have shown a maximum improvement of 9% in the overall number of hits and 3% improvement in the average cache occupancy percentage when compared to the conventional LRU approach.