3. CONTEXT-BASED DATA PATTERN EXPLOITATION TECHNIQUE

Data access patterns of workload vary across applications. A replacement algorithm can aid in improving the performance of the system only when it works well majority of the applications in existence. In short, if a replacement algorithm could make decisions based on the access history and the current access pattern of the workloads, the number of hits received can increase considerably [87,88,89].

The impact of various cache replacement policies act as the main deciding factor of system performance and efficiency in Chip Multi-Core Processors. Conventional replacement approaches work well for applications which have a well-defined and a predictable access pattern for their workloads. But when the workload of the applications is dynamic and constantly changes pattern, traditional replacement techniques may fail to produce improvement in the cache metrics. Hence this chapter discusses a novel cache replacement policy that is targeted towards applications with dynamically varying workloads. Context Based Data Pattern Exploitation Technique assigns a counter for every block of the L2 cache. It then closely observes the data access patterns of various threads and modifies the counter values appropriately to maximize the overall performance. Experimental results obtained by using the PARSEC benchmarks have shown an average improvement of 8% in overall hits at L2 cache level when compared to the conventional LRU algorithm.

3.1. Introduction

Memory management plays an essential role in determining the performance of CMPs. Cache memory is indispensible in CMPs to enhance the data retrieval time and to achieve high performance. The level 1 cache or the L1 cache needs to be small in size owing to the hardware limitations but since it is directly integrated into the chip that carries the processor, accessing it typically takes only a few processor cycles when compared to the accessing time of the next higher levels of memories. Accessing the L2 cache may not be as fast as accessing the L1 cache but since it is larger in size and is shared by all the available cores,
effectively managing it becomes more important. This chapter puts forth a
dynamic replacement approach called CB-DPET that is targeted mainly towards
the shared L2 cache.

3.1.1. Different Types of Application Workloads

The following are the types of workload patterns [90] exhibited by many
applications: Let ‘x’ and ‘y’ denote sequence of references where ‘xi’ denotes the
address of the cache block. ‘a’, ‘p’ and ‘n’ are integers where ‘a’ denotes the
number of times the sequence repeats itself.

Thrashing Workload:

[.. (x₁, x₂,..,xₙ)ᵃ ..] a>0, n>cache size

These workloads have a working set size that is greater than the size of the
available shared cache. They may not perform well under LRU replacement
policy. Sample data pattern for a thrashing workload for a cache which can hold
8 blocks at a time can be as follows
. . . 8, 9, 2, 6, 11, 20, 4, 5, 10, 16, 80 . . .

Here, cache_size = 8 blocks, x₁ = 8, x₁₁ = 80, n = 11 and a = any value > 0.

On first scan, elements ‘8’ till ‘5’ will be fetched and loaded into the cache. When
the next element ‘10’ arrives, the cache will be full and hence replacement has to
be made. Same is the case for remaining data items like ‘16’, ‘80’ and so on.

Regular Workload:

[... (x₁, x₂,..,xₙ₋₁, xₙ, xₙ₋₁, ... x₂, x₁)ᵃ...] n>0, a>0

Applications possessing this type of workload benefit hugely from the cache
usage. LRU policy suits them perfectly since the data elements follow stack based
pattern. Sample data pattern for regular workload for a cache which can hold 8 blocks at a time can be as follows

\[ \ldots 8, 9, 2, 6, 11, 20, 4, 5, 4, 20, 11 \ldots \]

Here, cache\_size = 8 blocks, \( x_1 = 8, x_7 = 4, x_8 = 5, n = 8 \) and \( a = \) any value > 0.

After the cache gets filled up with the eighth element ‘5’, the remaining elements ‘4’, ‘20’ and ‘11’ result in back to back cache hits.

**Random Workload:**

\[ \ldots (x_1, x_2, \ldots x_n) \ldots ] n \rightarrow \infty \]

Random workloads possess poor cache re-use. They generally tend to thrash under any replacement policy. Sample data pattern for a random workload for a cache which can hold 8 blocks at a time can be as follows

\[ \ldots 1, 9, 2, 8, 0, 12, 22, 99, 101, 50, 3 \ldots \]

**Recurring Thrashing Workload:**

\[ \ldots (x_1, x_2, \ldots x_n)^a \ldots (y_1, y_2, \ldots y_p) \ldots (x_2, x_1, \ldots x_n)^a \ldots ], a > 0, p > 0, n > \text{cache size} \]

Sample data pattern for a recurring thrashing workload for a cache which can hold 8 blocks at a time can be as follows

\[ \ldots 8, 9, 2, 6, 11, 20, 4, 5, 7, 17, 46, 10, 16, 80, 90, 99, 8, 9, 2 \ldots \]

Here, cache\_size = 8 blocks, \( x_1 = 8, x_{11} = 46, y_1 = 10, y_3 = 99, n = 11 \) and \( a = 1 \).

On first scan, elements ‘8’ till ‘5’ will be fetched and loaded into the cache. When the next element ‘7’ arrives, the cache will be full and hence replacement has to be made. Same is the case for remaining data items like ‘17’, ‘46’ and so on. By the time the initial pattern recurs towards the end, it would have been replaced resulting in cache misses towards the end.
Recurring Non-Thrashing Workload:

\[ [... (x_1, x_2, ..., x_n)^a \ldots (y_1, y_2, ..., y_p) \ldots (x_{2}, x_{1}, ..., x_{n})^a \ldots ] \ldots, a > 0, p > 0, n = cache size \]

It is similar to the thrashing workload but with a recurring pattern in between. Sample data pattern for a recurring non-thrashing workload for a cache which can hold 8 blocks at a time can be as follows

\[ \ldots 8, 9, 2, 6, 11, 20, 4, 5, 10, 16, 80, 8, 9, 2 \ldots \]

Here, cache_size = 8 blocks, \( x_1 = 8, x_8 = 5, y_1 = 10, y_3 = 80, n = 8 \) and \( a = 1 \).

On first scan, elements ‘8’ till ‘5’ will be fetched and loaded into the cache. When the next element ‘10’ arrives, the cache will be full and hence replacement has to be made. Same is the case for remaining data items like ‘16’ and ‘80’. Again the initial pattern recurs towards the end. If LRU is applied here, it will result in 3 cache misses towards the end as it would have replaced data items ‘8’, ‘9’ and ‘2’.

In the recurring non-thrashing pattern, if ‘n’ is large, then LRU can result in sub-optimal performance. Data items \( x_2 \) to \( x_n \) which reoccur after a short burst of random data \( (y_1 \) to \( y_p) \) may result in cache miss. The problem here is even if a data item receives many cache hits in the past, if it is not referenced by the processor for a brief period of time, LRU evicts the item from the cache. When the processor looks for that data item again it results in a miss. This stresses the need for a replacement technique that adapts itself to the changing workload and makes replacement decisions appropriately.

This chapter’s contribution includes,

- A novel counter based replacement algorithm called CB-DPET which tries to maximize the cache performance by closely monitoring and adapting itself to the pattern that occurs in the sequence.
- Association of a 3-bit counter (which will be referred to as the PET counter from here onwards) with every cache block in the cache set.
- A set of rules to adjust the counter value whenever an insertion, promotion or a replacement happens such that the overall hit percentage is maximized.
• Extension of the method to suit multi-threaded applications and also to avoid cache thrashing.
• PET counter boundary determination such that there will not be any stale data in the cache for longer periods of time, i.e. non-accessed data ‘mature’ gradually with every access and gets removed at some point of time to pave the way for new incoming data sets.

3.2. Basic Structure of CB-DPET:
This section and the following sub-sections elaborate on the basic concepts that make up CB-DPET- namely DPET and DPET-V. Mapping method is set-associative mapping [1]. A counter called the PET counter is associated with every block in the cache set. Conceptually this counter holds the ‘age’ of each and every cache block. The maximum value that the counter can hold is set to 4 which imply that only 3 bits are required to implement it at the hardware level, thereby not increasing the complexity. The minimum value that the counter can hold is ‘0’. The maximum upper bound value that the counter can hold is actually ‘7’ (2^3-1) but the value of ‘4’ has been chosen. This is because, any value that is chosen past 6 can result in stale data polluting the cache for longer periods of time. Similarly if a value below ‘3’ had been chosen, the performance will be more or less similar to that of NRU. Thus the upper bound was taken to be ‘4’. The algorithm which implements the task of assigning the counter values based on access pattern is referred as the ‘Age Manipulation Algorithm’ (AMA).

3.3. DPET
A block with a lower PET counter value can be regarded as ‘younger’ when compared to a block having a higher PET value. Initially the counter values of all the blocks are set to ‘4’. Conceptually every block is regarded as ‘oldest’. So in this sense, the block which is considered to be the ‘oldest’ (PET counter value 4) is chosen as a replacement candidate when the cache gets filled up. If the oldest block cannot be found, the counter value of every block is incremented continuously till the oldest cache block is found. This block is then chosen for
replacement. Once the new block has been inserted into this slot, AMA decrements the counter associated with it by ‘1’ (it is set to ‘3’). This is done because the block can no more be regarded as ‘oldest’, as it has been accessed recently. Also it cannot be given a very low value as the AMA is not sure about its future access pattern. Hence it is inserted with a value of ‘3’. When a data hit occurs, that block is promoted to the ‘youngest’ level irrespective of which level it was in previously. By this way, more time is given for data items which repeat themselves in a workload to stay in the cache.

To put this in more concrete terms,
A. DPET associates a counter, which iterates from ‘0’ to ‘4’, with every cache block.
B. Initially the counter values of all the blocks are set to ‘4’. Conceptually every block is regarded as ‘oldest’.
C. Now when a new data item comes in, the oldest block needs to be replaced. Since contention can arise in this case, AMA chooses the first ‘oldest’ block from the bottom of the cache as a tie-breaking mechanism.
D. Once the insertion is made, AMA decrements the counter associated with it by ‘1’ (it is set to ‘3’). This is done because the block can no more be regarded as ‘oldest’, as it has been accessed recently. Also it cannot be given a very low value as the AMA is not sure about its future access pattern. Hence it is inserted with the value of ‘3’.
E. In the case where all the blocks are filled up (without any hits in between), there will be a situation where all the blocks will have a PET counter value of ‘3’. Now when a new data item arrives, there is no block with a counter value of ‘4’. As a result, AMA increments all the counter values by ‘1’ and then performs the check again. If a replacement candidate is found this time, AMA executes steps C and D again. Otherwise it again increments the values. This is carried out recursively until a suitable replacement candidate is found.
F. Now consider the situation where a hit occurs for a particular data item in the cache. This might be the start of a specific data pattern. So AMA gives high
priority to this data item and restores its associated PET counter value to '0' terming it as the 'youngest' block.

On the other hand, if a miss is encountered, the situation becomes quite similar to what AMA dealt with in the initial steps, i.e. it scans all the blocks from the bottom to find the 'oldest' block. If found it proceeds to replace it. Otherwise the counter values are incremented and once again the same search is performed and this process goes on till a replacement candidate is found.

3.4. DPET-V : A Thrash-Proof Version

This method is a variant of DPET (In the following sections, it will be referred to as DPET-V). Some times for workloads with a working set greater than the available cache size, there is a possibility that there will not be any hits at all. The entire time will be spent only in replacement. This condition is often referred to as thrashing. The performance of cache can be improved if some fraction of randomly selected working set is retained in the cache for some more time which will result in increase in cache hits Hence very rarely, AMA decrements the PET counter value by '2' (instead of '1') after an insertion is made. So the new counter value post insertion will be '2' instead of '3'. Conceptually it allows more time for the data item to stay in the cache. This technique is referred to as DPET-V. The probability of inserting with '2' is chosen approximately as 5/100, i.e. out of 100 blocks, the PET counter value of (approximately) 5 blocks will be set to '2' while the others will be set to '3' during insertion.

\[\ldots 8 9 2 6 5 0 1 1 1 2 1 3 4 1 5 2 9 5 8 6 \ldots\]

Fig 3.1 Data sequence which will result in thrashing for DPET

Algorithm 1: Implementation for Cache Miss, Hit and Insert Operations
Init:
Initialize all blocks with PET counter value '4'.

Input:
A cache set instance

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

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

FindVictim(set):
/** Infinite Loop **/

while 1 do
  removing_index = -1;
  /**< Scanning from the bottom **/
  for i=associativity-1 to 0 do
    if set.blk[i].pet == 4 then
      /**< Block Found. Exit Loop **/
      removing_index = i;
      break;
    if removing_index == -1 then
      for i=associativity-1 to 0 do
        /**< Increment all blocks pet value by 1 and search again **/
        set.blks[i].pet = set.blks[i].pet + 1
      else
        break; /**< break from main while loop **/
  return removing_index;

Hit:
set.blk.pet = 0;
**Insert:**

\[ set.blk.pet = 3; \]

---

**Correctness of Algorithm**

**Invariant:**
At any iteration i, the PET counter value of set.blk[i] either holds a value of ‘4’ or a value in the range of ‘0’ to ‘3’.

**Initialization:**
When \( i = 0 \), set.blk[i].pet will be ‘4’ initially. After being selected as victim and after the new item has been inserted, set.blk[i].pet will be ‘3’. Hence **invariant holds good at initialization**.

**Maintenance:**
At \( i=n \), set.blk[i].pet will contain either zero (if a cache hit has happened) or set.blk[i].pet will have ‘3’ (if it has been newly inserted).

set.blk[i].pet will hold ‘1’ or ‘2’ or ‘4’ (if a replacement needs to be made but block with PET counter ‘4’ not yet found). **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 internal infinite loop will terminate when a replacement candidate is found. A replacement candidate will be found for sure in every run since the counter value of individual blocks is proven to be in the range ‘0’ to ‘4’. A candidate is found as and when a counter value reaches ‘4’. Since an increment by ‘1’ is happening for all the blocks until a value of ‘4’ is found, the boundary condition “if set.blk[i].pet == ‘4’” is bound to hit for sure, terminating the loop after a finite number of runs.

The **invariant can also be found to hold good for all the cache blocks present when the loop terminates.**
It is observed that if DPET is applied over the working set shown in Fig 3.1 there will not be any hits at all (Fig 3.2(i)). After the first burst \{8,9,2,6,5,0\}, the PET counter value of all blocks is set to ‘3’. When the next burst \{11,12,1,3,4,15\} arrives, none of them results in a hit. Now when the third burst arrives, again all the old blocks are replaced with new ones. Fig 3.2(ii) shows the action of DPET-V over the same data pattern. According to DPET-V, AMA has randomly selected two blocks (containing data items \{5\} and \{2\}) whose counter values have been set to ‘2’ instead of ‘3’ while inserting the data burst 1. Now when burst 2 arrives, those two blocks are left untouched as their PET counter values have not matured to ‘4’ yet. Finally when the third burst of data \{2,9,5,8,6\} arrives, it results in two hits more (for \{5\} and \{2\}) compared to DPET. Hence by giving more time for those two blocks to stay in cache, the hit rate has been improved. Fig 3.3(i), 3.3(ii), 3.3(iii) and 3.3(iv) shows the working of CB-DPET in the form of a flow diagram.

Fig 3.2 (i) Contents of the cache at the end of each burst for DPET

(ii) Contents of the cache at the end of each burst for DPET-V
Processor looks for a particular block ‘c’ in the cache

AMA starts search for block ‘c’ in the cache

If ‘c’ is found

- HIT: Set PET counter value of ‘c’ to 0
- MISS: Finish

Processor brings the data item from next level of memory

Find replacement victim in the cache

Insert new data item

Finish

Fig 3.3 (i) CB-DPET flow Diagram
AMA algorithm starts searching for a replacement candidate ‘c’ in the cache

Find replacement victim in the cache

Search for first block ‘c’ from cache bottom with PET counter value of ‘4’

If ‘c’ is found

Choose ‘c’ as replacement victim

Yes

No

Increment counter values of all blocks by 1

Fig 3.3 (ii) Victim Selection flow
Fig 3.3 (iii) Insert flow - DPET

Insert new data item - DPET

Insert incoming data item into the victim block

Set PET Counter value of the block to 3

95 out of every 100 blocks

5 out of every 100 blocks

Fig 3.3 (iv) Insert flow DPET-V

Insert new data item - DPET-V

Insert incoming data item into the victim block

Set PET Counter value of the block to 3

Set PET Counter value of the block to 2
3.5. Policy Selection Technique

A few test sets in the cache are allocated to choose between DPET and DPET-V. Among those test sets, DPET is applied over a few sets and DPET-V is applied over the remaining sets and they are closely observed. Based on the performance obtained here, either DPET or DPET-V is applied across the remaining sets in the cache (apart from the test sets). A separate global register (which from here onwards will be referred to as Decision Making register or DM register) is maintained to keep track of the number of misses encountered in the test sets for both the techniques. Initially the value of this register is set to zero.

Fig 3.4 Policy selection using DM register
For every miss encountered in a test set which has DPET running on it, the register value is incremented by one and for every miss which DPET-V produce in its test sets, AMA decrements the register value by one. Effectively, if at any
point of time the DM register value is positive, then it means that DPET has been issuing many misses. Similarly if the register value goes negative, it indicates that DPET-V has resulted in more misses. Based on this value, AMA makes the apt replacement decision at run time. This entire algorithm which alternates dynamically between DPET and DPET-V is termed Context Based DPET. Fig 3.4 depicts the policy selection process. Fig 3.5 shows the Victim selection process.

3.6. Thread Based Decision Making

The concept of CB-DPET is extended to suit multi-threaded applications with a slight modification. For every thread a DM register and few test sets are allocated. Before the arrival of the actual workload, in its allocated test sets each thread is assigned to run DPET for one half and DPET-V for the remaining half of the test sets. The sets running DPET (or DPET-V) need not be consecutive. These sets are selected in a random manner. They can occur intermittently i.e. a set assigned to run DPET can immediately follow a set that is assigned to run DPET-V or vice versa. On the whole, the total number of test sets allocated for each thread is equally split between DPET and DPET-V. Now when the actual workload arrives, the thread starts to access the cache sets. Two possibilities arise here.

Case 1:
The thread can access its own test set. Here, if there is data miss, it records the number of misses in its DM register appropriately depending on whether DPET or DPET-V was assigned to that set as discussed in the previous section.

Case 2:
The thread gets to access any of the other sets apart from its own test sets. Here, either DPET or DPET-V is applied depending on the thread’s DM register value. If this access turns out to be the very first cache access of that thread, then its DM register will have a value of ‘0’, in which case the policy is chosen as DPET.
3.7. **Block Status Based Decision Making**

When more than one thread tries to access a data item, it can be regarded as shared data. Compared to data that is accessed only by one thread (private data), shared data has to be placed in the cache for a longer duration. This is because multiple threads try to access those data at different points of time and if a miss is encountered, the resulting overhead would be relatively high. Taking this into consideration, it is possible to find out whether each block has shared or private data (using the id of the thread that accesses it) and decide on the counter values appropriately. To serve this purpose, there is a ‘blockstatus’ register associated with every cache block. When a thread accesses that block, its thread id is stored into the register. Now when another thread tries to access the same block, the value of the register is set to some random number that does not fall within the accessing threads’ id range. So if the register holds that random number, the corresponding block is regarded as shared. Otherwise it is treated as private.

3.8. **Modification in the Proposed Policies**

In case of DPET, when a block is found to be shared, insertion is made with a counter value of ‘2’ instead of ‘3’. Similarly in DPET-V, for shared data, insertion is always made with a PET counter value of ‘2’. If a hit is encountered in either of the methods, the PET counter value is set to ‘0’ (as discussed earlier) only if the block is shared. For private blocks, the counter value is set to ‘1’ when there is a hit.

\[\ldots\,1\,5\,7\,7\,1\,5\,9\,6\,5\,2\,1\,0\,4\,0\,8\,1\,5\,7\,\ldots\]

**Fig 3.6 Data sequence containing a recurring pattern {1,5,7}**

3.9. **Comparison between LRU, NRU and DPET**

3.9.1. **Performance of LRU**

The behaviour of LRU for the data sequence in Fig 3.6 is shown in Fig 3.7. It is assumed that there are five blocks in the cache set under consideration. Fig. 3.7
shows the working of the LRU replacement policy over the given data set. The workload sequence (divided into bursts) is specified in the Fig. 3.7. Below every data, ‘m’ is used to indicate a miss in the cache and ‘h’ is used to indicate a hit. Alphabet ‘N’ in the cache denotes Null or Invalid data. The cache set can be viewed as a queue with insertions taking place from the top of the queue and deletion taking place from the bottom. Additionally, if a hit occurs, that particular data is promoted to the top of the queue, pushing other elements down by one step. Hence at any point of time, the element that has been accessed recently is kept at the top and the ‘least recently used’ element is found at the bottom, which becomes the immediate candidate for replacement. In the above example, initially the stream \{1, 5, 7\} results in miss and the data is fetched from the main memory and placed in the cache.

![Fig 3.7 Behaviour of LRU for the data sequence](image)

Now data item \{7\} will hit and it is brought to the top of the queue (in this case it is already at the top). Data items \{1\} and \{5\} results in two more consecutive hits bringing \{5\} to the top. The ensuing elements \{9, 6\} results in misses and are brought into the cache one after another. Among the burst \{5, 2, 1\} the data item \{2\} alone misses. Then \{0\} too results in a miss and is brought from the main
memory. In the stream \{4,0,8\} the data item \{0\} hits. And finally the burst \{1, 5, 7\} re-occurs towards the end where \{5, 7\} will result in two misses as indicated by Fig. 3.8. The overall number of misses encountered here amounts to eleven.

3.9.2. Performance of NRU

The working of NRU on the given data set is shown in Fig. 3.8. NRU associates a single bit counter with every cache line which is initialized to ‘1’. When there is a hit, the value is changed to ‘0’. A ‘1’ indicates that the data block has remained unreferenced in the cache for a longer period of time compared to other blocks. Thus the block from the top which has a counter value of ‘1’ is selected as a victim for replacement. Scanning from the top helps in breaking the tie. If such a block is not found, the counter values of all the blocks are set to ‘1’ and a search is performed again. In Fig. 3.8, counter values which get affected in every burst are underlined for more clarity. As it is observed from the figure, NRU results in one miss more than the overall misses encountered in LRU. Towards the end, the burst \{5,7\} result in two misses. This is primarily because NRU tags the stream \{5,7\} as ‘not-recently used’ as it had not seen them for some time and thus chooses them as replacement victims. In the long run, as the data set size increases in a similar fashion, the number of misses will increase considerably.

One feature that is common between both algorithms is that they follow the same technique irrespective of the pattern that the data set follows. This might minimize the hit rate in some cases. In case of LRU, if an item is accessed by the processor, it is immediately taken to the top of the queue thereby pushing the remaining set of elements one step below, irrespective of whatever pattern the data set might follow thereafter. Thus even if any element in this group, that gets pushed down, repeats itself after some point of time, it might have been marked as ‘least recently used’ and evicted from the cache. NRU is also not powerful as it has only a single bit counter, which does not allocate enough time for data items which might be accessed in the near future.
3.9.3. Performance of DPET

Fig. 3.9 shows the contents of the cache at various points when DPET is applied, along with the burst of input data which is processed at that moment. The PET counter value associated with every block is shown, as a sub-script to the actual data item and the counter value of the block which is directly affected at that instant is underlined for more clarity. The input data set has the letters ‘m’ and ‘h’ under every data item to indicate a miss or a hit respectively. The behaviour is studied with a cache set consisting of five blocks. Initially AMA assigns a value of ‘4’ to PET counters associated with all the blocks. As seen from figure, the burst \{1,5,7,7\} arrives initially. Data items \{1,5,7\} are placed in the cache in a bottom up fashion and their PET values are decremented by ‘1’. Now when the data item \{7\} arrives again, it hits in the cache and thus the counter value associated with \{7\} is made ‘0’. Data items \{1,5\} arrive individually and result in two hits. The corresponding PET values are set to ‘0’ as indicated in the figure. The next burst contains elements \{9,6\}, both resulting in cache misses. They are fetched from the main memory and stored in the remaining two blocks of the cache set and
their counter values are decremented appropriately. The fifth data burst comprises the data items \{5,2,1\}. Among these, \{5\} is already present in the cache and hence its counter value is set to ‘0’. Data item \{2\} is not present in cache. So as with DPET, AMA searches for a replacement candidate bottom up, which has its counter value set to ‘4’. Since such a block could not be found, the PET values of all the blocks are incremented by ‘1’ and the search is performed again. Now the PET value of the block containing \{9\} would have matured to ‘4’ and it is replaced by \{2\} and its counter value is decremented to ‘3’. Data item \{1\} hits in the cache. The overall number of misses is reduced compared to LRU and NRU.

<table>
<thead>
<tr>
<th>N4</th>
<th>N4</th>
<th>N4</th>
<th>N4</th>
<th>6</th>
<th>6</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>N4</td>
<td>N4</td>
<td>N4</td>
<td>N4</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>N4</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>N4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>N4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Fig 3.9 Behaviour of DPET for the data sequence

3.10. Results

To evaluate the effectiveness of the replacement policy on real workloads, a performance analysis has been done with Gem5 simulator using PARSEC benchmark programs, discussed in Chapter 6, as input sets. The results are shown in this section.

3.10.1. CB-DPET Performance
The effectiveness of this method is visible in the following figures as it has surpassed the LRU. Fig. 3.10 shows the overall number of hits obtained for each workload with respect to CB-DPET and LRU at L2 cache. The overall number of hits obtained in the ROI is taken. These numbers have been scaled down by appropriate factors to keep them around the same range. It is observed from the figure that CB-DPET has resulted in improvement in the number of hits compared to LRU for majority of the workloads with *fluidanimate* showing a high performance with respect to baseline policy.

![Overall number of hits in L2](image)

**Fig 3.10 Overall number of hits recorded in L2 cache**

Fig. 3.11 shows the hit rate obtained for individual benchmarks. Hit rate is given by the following expression.

$$\text{Hit Rate} = \frac{\text{Overall number of hits in ROI}}{\text{Overall number of accesses in ROI}} \quad \ldots \ldots (1)$$

Six of the seven benchmarks have reported improvements in the hit rate. *Fluidanimate* shows maximum improvement of 8% whereas *swaptions* and *vips* have exhibited hit rates on par with LRU.
Fig. 3.12 shows the number of hits obtained in the individual parallel phases. As discussed earlier, the ROI can be further divided into two phases denoted by PP1 (Parallel Phase 1) and PP2 (Parallel Phase 2). These phases need not be equal in complexity, number of instructions, etc. There can be cases where PP1 might be more complex with a huge instruction set compared to PP2 or vice versa. The number of hits recorded in these phases is shown in the graph in Fig. 3.12. CB-DPET has shown an increase in hits in most of the phases of majority of the benchmarks.

![L2 cache hit rate](image)

**Fig 3.11 L2 cache hit rate**

Miss latency at a memory level refers to the time spent by the processor to fetch a data from the next level of memory (which is the primary memory in this case) following a miss in that level of memory. Miss latency at L2 cache for individual cores is shown in Fig. 3.13. It is usually measured in cycles. To present it in a more readable format, it has been converted to seconds using the following formula:

\[
\text{Miss latency (in secs)} = \frac{\text{Miss latency in cycles}}{\text{CPU Clock Frequency}} \quad \cdots \cdots (2)
\]
The impact of miss latency is an important factor to be taken into consideration. If the miss latency is greater, it means that the overhead involved in fetching the data will be high thereby resulting in performance bottleneck.

![Number of hits in individual phases](image1)

**Fig 3.12** Hits recorded in individual phases of ROI

![Miss latency at L2 cache for individual cores](image2)

**Fig 3.13** Miss latency at L2 cache for individual cores

Overall miss latency at L2 is also shown in Fig. 3.13. Note that this is with respect to the ROI of the benchmarks. Considerable reduction in latency is observed.
across the majority of the benchmarks. Vips has shown a maximum reduction in Overall miss latency. Except ferret and swaptions, all the other workloads have shown reduction in the overall miss latency.

\[
\text{Percentage increase in Hits} = \frac{\text{Hits in CBDPET} - \text{Hits in LRU}}{\text{Hits in LRU}} \times 100 \quad (3)
\]

Fig. 3.14 shows the percentage increase in hits obtained from CB-DPET compared to LRU. The figure shows that there is an increase in the hit percentage for all the benchmarks except ferret. Swaptions exhibiting the highest percentage increase of 13.5% whereas dedup has shown a 2% increase. An approximate 8% improvement over LRU is obtained as the average hit percentage when CB-DPET is applied at the L2 cache level. This is because CB-DPET adapts to the changing pattern of the incoming workload and retains the frequently used data in the cache, thus maximizing the number of hits. Fig. 3.15 shows the overall number of misses recorded by each benchmark. Decrease in the number of misses is observed across majority of the benchmarks compared to LRU.
3.11. Summary

When it comes to concurrent multi-threaded applications in a CMP environment, LLC plays a crucial role in the overall system efficiency. Conventional replacement strategies may not be effective in many instances as they do not adapt dynamically to the data requirements. Numerous researches [45,46,79,90] are being conducted to find novel ways for improving the efficiency of the replacement algorithms applied at LLC. Both LRU and NRU schemes tend to perform poorly on data sets possessing irregular patterns because they do not store any extra information regarding the recent access pattern of the data items. They make decisions purely based on the current access only [15,45,51,79]. This chapter hence proposes a scheme called CB-DPET which tries to address the shortcomings arising from the conventional replacement methods and improve the performance.

In this chapter, the following are the conclusions

- CB-DPET takes a more systematic and access-history based decision compared to LRU. A counter associated with every cache block tracks the
behavior of application access pattern and makes the replacement, insertion and promotion decisions.

- DPET-V is a thrash proof version of DPET. Here some randomly selected data items are allowed to stay in the cache for a longer time by adjusting the counter values appropriately.
- In multi-threaded applications, a DM register and some test sets are allocated for every thread. Based on the number of misses in the test sets, a best policy is chosen for the remaining sets.
- These algorithms are made thread-aware by using the thread id. Based on the thread id, it can identify which thread is trying to access the cache block, so when multiple threads try to access the same block, it is tagged as shared and allow that block to stay in the cache for more time compared to other blocks by appropriately adjusting their counter values.

Evaluation results on PARSEC benchmarks have shown that CB-DPET has reported an average improvement in the overall hits (up to 8%) when compared to LRU at the L2 cache level.