CACHE REPLACEMENT FOR MULTI-THREADED APPLICATIONS USING CONTEXT BASED DATA PATTERN EXPLOITATION TECHNIQUE

Muthukumar S[1] and Dr P K Jawahar[2]

[1]Department of Computer Science and Engineering, Sri Venkateswara College of Engineering, Sriperumbudur, Chennai, Tamil Nadu, India.
[2]Department of Electronics and Communication Engineering, BS Abdur Rahman University, Vandalur, Chennai, Tamil Nadu, India.

Email: [1]muthukumar@svce.ac.in, [2]jawahar@bsauniv.ac.in

ABSTRACT

The impact of various cache replacement policies act as the main deciding factor of system performance and efficiency in Chip Multi-Core Processors (CMP). Many existing cache replacement polices such as the Least Recently Used (LRU), Most Recently Used (MRU), Not Recently Used (NRU) etc. have proved to work well in the shared L2 cache for most of the data set patterns generated by current applications. But when it comes to parallel multi-threaded applications which generate differing patterns of workload at different intervals, the above specified schemes might prove sub-optimal as they generally do not abide by the spatial and temporal locality theories. This paper proposes a novel cache replacement policy that is targeted towards such applications. Context Based Data Pattern Exploitation Technique (CB-DPET) assigns a counter for every block of the L2 cache. It then closely monitors the data access patterns of various threads and modifies the counter values appropriately to maximize the overall hit percentage. Experimental results obtained by using the PARSEC benchmarks have shown an average improvement of 8% to 9% in overall hits at L2 cache level when compared to the conventional LRU algorithm.

Keywords: ***Cache, Hits, L2, Replacement, Shared, Thread.***

1.0 INTRODUCTION

The advent of multi-core computers consisting of two or more processors working in tandem on a single integrated circuit has produced a phenomenal breakthrough in the performance over the recent years. They provide room for faster execution of applications by exploiting the available parallelism (mainly Instruction Level Parallelism), which implies the ability to work on multiple problems simultaneously. Memory management forms an integral part in determining the performance of CMPs. Cache memory is widely used in CMPs to enhance the data retrieval time. Generally every core in a CMP environment has a private L1 cache and all the available cores share a relatively larger L2 cache (which in most cases can also be referred to as the shared Last Level Cache). The L1 cache can be further divided into 'Instruction Cache' and 'Data Cache'. Let us assume that the mapping policy used is 'set-associative' mapping [1], where the cache is divided into a specified number of sets with each set having a specified number of blocks. The number of blocks in each set indicates the associativity of the cache. The size of the L1 cache has to be kept small owing to chip constraints. Since it is directly integrated into the chip that carries the processor, its data retrieval time is expedited.

Shared data present in L2 cache will be accessed by multiple cores, so it is essential that judicious replacement strategies need to be applied here, in order to improve the performance. At present, many applications make use of the LRU cache replacement algorithm [1], which discards the 'oldest possible' cache line available in the cache (i.e.) the cache line which has not been accessed over a period of time. For this purpose, the 'age' of every cache line needs to be updated after every access. This is done with the help of either a counter or a queue data structure. In the queue based approach, data that is referenced by the processor is moved to the top of the queue, pushing all the elements below by one step. At any point of time, to accommodate an incoming block, LRU deletes the block which is found at the bottom of the queue (least recently used data). MRU also works in a similar manner, with a slight difference that the victim is chosen from the opposite end of the data structure, i.e. the most recently referred data is

evicted to accommodate an incoming block. The above-stated algorithms work well when the 'locality of reference' is high, but for applications whose data sets do not exhibit any degree of regularity in their access pattern, these algorithms may result in sub-optimal performance. An example of such a pattern can be seen in Fig.1. From here on we refer the data items by enclosing them within curly braces.

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

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

As we can see from Fig.1, the starting pattern {1,5,7} seems to recur after a long burst of random data. If the LRU scheme is applied for the above set of data in a cache that can hold (say) 5 blocks at a time, eventually the recurring burst {1,5,7} will get replaced by newer data and finally will end up in 2 misses towards the end (where it again needs data elements {1,5,7}).

CB-DPET tries to maximize the hit-rate in such scenarios by closely watching and adapting itself to the pattern that occurs in the sequence. This is achieved by associating a 3-bit counter (which we call the PET counter from here onwards) with every cache block in the cache set. We define 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. The concept is then extended to avoid the thrashing condition and also to suit multi-threaded applications. The upper and lower bounds for the PET counter are chosen 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.

The rest of this paper is organized as follows:
Section 2 deals with related work, section 3 expounds the basic concept behind CB-DPET and its thread-aware policy, section 4 provides the details regarding the experimental set up, section 5 analyses the obtained results, and finally section 6 summarizes the paper.

2.0 RELATED WORK

The replacement policy forms the heart of any memory system. Considering the on-chip cache memory of a processor, choosing a perfect replacement strategy is extremely crucial in determining the performance of the system. The more adaptive a replacement strategy is to the incoming workload, the less is the overhead involved in transferring data to and from the cache. John Odule and Ademola Osingua have come up with a dynamically self-adjusting cache replacement algorithm [2] that makes page replacement decisions based on changing access patterns by adapting to the spatial and temporal features of the workload. Though it dynamically adapts to the workload needs, it does not detect changes or make decisions at a finer level (i.e.) at block level, i.e. which can be crucial when the working set changes quite frequently.

Many such algorithms which work better than LRU on specific workloads have evolved in recent times. One such algorithm is the Dueling Clock (DC) [3] algorithm, designed to be applied on the on-chip L2 cache, but it does not consider applications that involve multiple threads executing in parallel. Deviating from the traditional working environment, we look into the cache replacement strategies used in networking and mobile environments to gain a wide spread understanding. Considering the networking scenario, irregular flooding of data is a major problem. A new algorithm called Adaptive Least Frequently Evicted (ALFE) [4] analyses the data flow in a networking environment and makes appropriate replacement decisions at run time. Another similar type of workload is the online-video streaming application. Here the cache size will be relatively larger as it has to cache video objects. This gives rise to the proxy caching scheme which is very similar to the proxy server concept used to reduce the data transfer time. The Dual Cache Replacement Policy (DCRP) [5] works on those proxy caches and deals with the techniques that work best with such workloads.

In the case of Mobile Database Systems (MDS), the location and direction of movement of mobile clients need to be taken into account while making replacement decisions at cache level. Hariram Chavan et al. have come up with the Markov Graph Cache Replacement Policy (MGCRP) [6] to be applied across MDS. This models the future location and movement of the client by constructing a Markov Graph and makes appropriate replacement decisions in the cache. Improvements have been recorded against replacement algorithms like LRU, Furthest Away Replacement (FAR), etc.

In a multi-processor environment, efficient cache partitioning can help in improving the performance and throughput of the system. Adaptive Bloom Filter Cache Partitioning (ABFCP) [7] involves dynamic cache partitioning at periodic intervals using Bloom Filters and counters to adapt to the needs of concurrent applications. But since every processor core is allotted an array of Bloom Filters, the hardware complexity can increase as the number of cores becomes higher. Pseudo LRU algorithm (similar to LRU but with lesser hardware complexity) have been proven to work well with partitioned caches [8]. Studies have also been conducted in enhancing the performance of Non-Uniform Cache Architecture (NUCA). A dynamic cache management scheme [9] aimed at the Last-Level Caches (LLC) in NUCA claims to have improved the system performance. Although many schemes [4,9,17,27] work well by adapting to the dynamic data requirements of the applications, they do not attach importance to data that is shared between the threads of the parallel workloads. Hits on such data items not only improve the performance of those workloads, but also reduce the overhead involved in transferring data to and from the cache frequently.

Data prefetching can be very useful in any cache architecture since it helps in reducing the cache latency. The Prediction-Aware Confidence-based Pseudo LRU (PAC-PLRU) [10] algorithm efficiently makes use of pre-fetched information and also saves this information from unnecessarily being evicted from the cache. There can also be scenarios where the overhead can be reduced by promoting the data fetched from the main memory directly to the processor, bypassing the cache. Hongliang Gao and Chris Wilkerson's Dual Segmented LRU algorithm [11] has explored the positive aspects obtained from cache bypassing in detail. Though cache bypassing can help in reducing the transfer overhead, it may not work well with the 'locality principles', i.e. bypassing a data item that is likely to be referenced in the near future might not be a good idea. Different types of memory architectures have come up in recent times. Augmented cache architecture is one that comprises two parts- a larger direct-mapped cache and a smaller, fully-associative cache which are accessed in parallel. A replacement strategy similar to LRU known as the LRU-Like algorithm [12] was proposed to be applied across augmented cache architectures.

Performance of the shared LLC in multi-core architecture plays a crucial role in deciding the overall performance of the system, since all the cores will access the data present here. Versatile researches [13,14,15,16,17,18,19] have been conducted in improving the shared LLC performance. Phase Change Memory (PCM) with a LLC forms an effective low-power consuming alternative to the traditional DRAMs in existence. Writeback-aware Cache Partitioning (WCP) [13] efficiently partitions the LLC in PCM among multiple applications and Write Queue Balancing (WQB) replacement policy [13] helps in improving the throughput of PCM. However, one of the main drawbacks of PCM is that the writes are much slower compared to DRAM.

Cache Hierarchy Aware Replacement (CHAR) [14] makes replacement decisions in LLC based on the activities that occur in the inner levels of the cache. It studies the data pattern of hits and misses in the higher levels of cache and makes appropriate decisions in LLC. But propagating such information frequently across various levels of cache may slow down the system over a period of time. Cache space is another area of concern. Instead of utilizing the entire cache space at any point of time, Liqiang He et al. have proposed an Adaptive Subset Based Replacement Policy (ASRP) [15] where the cache space is divided into multiple subsets and at any point of time only one subset is active. Whenever a replacement needs to be made, the victim is selected only from the active set. However, there is a possibility of under-utilization of the available cache space in this technique. For any replacement algorithm to be successful, it is always essential to make good use of the available cache space. This can be achieved by eradicating the dead cache lines, i.e. the lines which will never be referenced by the processor in the future. Lot of researches have been carried out in this area [20,21]. Counter based cache replacement and cache bypassing algorithm [20] tries to locate the dead cache lines well in advance and choose them as replacement victims, thus preventing such lines from polluting the cache. But the problem which can arise here is that for applications with an unpredictable data access pattern, the process of dead line prediction becomes intricate and in the long run the predictor might lose its accuracy. Another approach [22] works on eliminating the LLC polluters dynamically using a special buffer that is controlled at the Operating System level.

The effect of the miss penalty can have a serious impact on the system efficiency. The Locality Aware Cost Sensitive (LACS) replacement technique [23], proposed by Rami Sheikh and Mazen Kharbutli, works on reducing the miss penalty in caches. It calculates the number of instructions issued during a miss for every cache block and when a victim needs to be selected for replacement, it selects the block which issues the minimum number of miss instructions. While it helps in reducing the miss penalty, it does not attach importance to the data access pattern of the application under execution. Another replacement policy which is called the LR+5LF [24] policy combines the concepts of LRU and LFU to find a replacement candidate. Again hardware complexity is an issue here.

Replacement algorithms that work well with L1 cache may not suit the L2 cache. Thus choosing an efficient pair of replacement algorithm for L1 and L2 becomes important for maximizing the hit rate [25]. Studies have indicated that execution history like the initial hardware state of the cache can also impact the sensitivity of replacement algorithms [26]. One predominant problem with LRU is that it may result in thrashing (a condition where most of the time is spent in replacement and there will not be any hits at all) when it is applied for memory intensive applications. Moinuddin K Qureshi et al. have tweaked the insertion policy of LRU to reduce the cache miss for memory intensive workloads. They call it the LRU Insertion Policy (LIP) [27] which has outperformed LRU with respect to thrashing.
Concurrent multi-threaded applications have become prevalent in today's CMPs. The Thread Aware Dynamic Insertion Policy (TA DIP) [28] takes the memory requirements of individual applications into consideration and makes replacement decisions. The Adaptive Timekeeping Replacement (ATR) [16] policy is designed to be applied primarily at LLC. It assigns 'lifetime' to every cache line. Unlike other replacement strategies like LRU, it takes the process priority levels and application memory access patterns into consideration and adjusts the cache line lifetime accordingly to improve performance. However, the condition which can lead to thrashing has not been considered in either method [16,28].

Thus our work in this paper goes as follows:
We have come up with a replacement algorithm for the LLC that is targeted mainly towards multi-threaded applications in a CMP environment. It deals at a finer block level and is thrash-proof to a large extent. It monitors the cache keenly and records the access patterns in the form of a counter. Data that is shared across multiple threads is given more importance than private data and is kept in the cache for a long time to reduce the ensuing transfer overhead and improve the hit rate.

3.0 CONTEXT BASED DATA PATTERN EXPLOITATION TECHNIQUE

Schemes like LRU can result in sub-optimal performance in cases where the data set follows a random pattern. To get a more detailed understanding, we need to look into the working of LRU. LRU maintains a queue data structure and keeps pushing the accessed blocks to the top of the queue while the unreferenced blocks remain at the bottom. When a replacement needs to be made, the block which is at the bottom of the queue is chosen as the victim. All the elements are pushed one position down and the new incoming data is placed at the top. But we consider multi-threaded applications which generate differing patterns of workloads at different intervals. So effectively the number of hits will be reduced and most of the time will be spent in adding new data and evicting the older ones. Thus the resulting number of misses will be predominantly higher. Let us consider the NRU (Not Recently Used) algorithm. It has a single bit counter associated with every cache line. This bit is set either a '0' or a '1'. If a hit occurs, the bit for the corresponding block is set to '0'. While replacing, the left-most block which has its bit set to '1' is chosen as the victim for replacement. Even in this method, as we will see in the forthcoming sections, the number of misses can be high.

3.1 Performance of LRU

To get a precise understanding of why LRU and NRU might not work well in certain cases, let us consider the data pattern shown in Fig.1. For illustrative purposes let us assume that there are 5 blocks in the cache set under consideration. Fig.2 shows the working of the LRU replacement policy over the given data set. The workload sequence (divided into bursts) is specified at the bottom of Fig.2. 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. So 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.

N	N	N	N	6	1	0	8	7
N	N	N	N	9	2	1	0	5
N	7	1	5	5	5	2	4	1
N	5	7	1	1	6	5	1	8
N	1	5	7	7	9	6	2	0
	1 5 7 7 m m m h	1 h	5 h	9 6 m m	5 2 1 h m h	0 m	4 0 8 m h m	1 5 7 h m m

Fig.2 Behaviour of LRU over the random data pattern shown in Fig.1

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 2 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.2. The overall number of misses encountered here amounts to eleven.

3.2 Performance of NRU

The working of NRU on the given data set is shown in Fig.3. As discussed earlier, 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 value of all the blocks is set to ‘1’ and a search is performed again. In Fig.3, counter values which get affected in every burst are underlined for easy understanding. As we can see 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 2 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 thing 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 (which gets pushed down) repeats itself after some point of time, it might have been tagged 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.

N_1	1_0	1_0	1_0	1_0	2_0	2_0	2_0	5_0
N_1	5_0	5_0	5_0	5_0	1_0	1_0	1_0	7_0
N_1	7_0	7_0	7_0	7_0	7_1	0_0	0_0	0_1
N_1	N_1	N_1	N_1	9_0	9_1	9_1	4_0	4_1
N_1	N_1	N_1	N_1	6_0	6_1	6_1	8_0	8_1
	1 5 7 7 m m m h	1 h	5 h	9 6 m m	5 2 1 h m m	0 m	4 0 8 m h m	1 5 7 h m m

Fig.3 Behaviour of NRU over the random data pattern shown in Fig.1

3.3 DPET

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 [14,15,25,28,29]. This paper hence proposes a scheme called CB-DPET which tries to address the shortcomings arising from the previously discussed methods and improve the hit rate. This section and the following section elaborate on the basic concepts that make up CB-DPET- namely DPET and DPET-V. We associate a counter called the PET counter 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 implies that we need only 3 bits to implement it at the hardware level, thereby not increasing the complexity. The minimum value that the counter can hold is '0'. We can see that the maximum upper bound value that the counter can hold is actually '7' (2^3-1) but we have chosen the value '4'. 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'.

A block with a lower PET counter value can be regarded as 'younger' when compared to a block having a higher PET value. 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 we find the oldest cache block. This block is then chosen for replacement. When a data hit occurs, that block is promoted to the 'youngest' level irrespective of which level it was in previously. By this way, we give more time for data items which repeat themselves in a workload to stay in the cache. The algorithm which implements this task, from here on will be referred as the 'Age Manipulation Algorithm' (AMA).

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, as with our scheme, the oldest block needs to be replaced. Since contention arises in our 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. So it goes for 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.

G. 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.

To understand more clearly what DPET does, consider the following input data set shown in Fig.1. For clarity it has been repeated here.

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

Fig.4 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 particular point. The PET counter value associated with every block is specified, 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. For simplicity, let us consider a cache having 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 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.

N_4	N_4	N_4	N_4	$6_{\underline{3}}$	$6_{\underline{4}}$	$0_{\underline{3}}$	$0_{\underline{1}}$	0_1
N_4	N_4	N_4	N_4	$9_{\underline{3}}$	$2_{\underline{3}}$	2_3	$8_{\underline{3}}$	8_3
N_4	$7_{\underline{0}}$	7_0	7_0	5_0	$5_{\underline{1}}$	5_1	$5_{\underline{3}}$	$5_{\underline{0}}$
N_4	$5_{\underline{3}}$	5_3	$5_{\underline{0}}$	1_0	$1_{\underline{0}}$	1_0	$1_{\underline{2}}$	$1_{\underline{0}}$
N_4	$1_{\underline{3}}$	$1_{\underline{0}}$	1_0	7_0	$7_{\underline{1}}$	7_1	$7_{\underline{3}}$	$7_{\underline{0}}$
	1 5 7 7 m m m h	1 h	5 h	9 6 m m	5 2 1 h m h	0 m	4 0 8 m h m	1 5 7 h h h

Fig.4 Behaviour of DPET over the random data pattern shown in Fig.1

Data item {0} arrives next, resulting in a miss. A similar process takes place and {6} is replaced by {0}. {4,0,8} forms the next input data burst. Data item {4} results in a miss and AMA follows the above mentioned steps to find the replacement candidate as {2}. Data item {0} results in a hit. {8} arrives thereafter and replaces {4}. Finally the pattern {1,5,7} occurs and all the three results in hits. It can be observed that DPET has recorded three misses fewer than NRU and two misses fewer than LRU.

3.4 DPET-V : A Thrash-Proof Version

We propose one more variant of DPET (which from here on will be referred to as DPET-V). Some times when the subsequent access time of all the data items in the cache is much longer than the size of the cache, 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’. So very rarely, AMA decrements the PET counter value by ‘2’ (instead of ‘1’) after an insertion is made. 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. The data pattern shown in Fig.5 can force DPET into thrashing mode. This pattern occurs in three bursts. {8,9,2,6,5,0} forms burst 1, {11,12,1,3,4,15} forms burst 2 and the remaining elements come under burst 3. For easy illustration purposes, the size of the cache is taken as 6 blocks.

. . . 8 9 2 6 5 0 11 12 1 3 4 15 2 9 5 8 6 . . .

Fig.5 Data sequence which will result in thrashing for DPET

It can be seen that if DPET is applied over the working set shown in Fig.5 there will not be any hits at all (Fig.6(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. So as with DPET, all counter values are incremented to ‘4’ and all the old blocks are replaced with the new ones. Now when the third burst arrives, again all the old blocks are replaced with new ones. If this pattern continues then we can see that there will not be any hits and the entire time will be spent only in replacement. Fig.6(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. So 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.

A few test sets 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 monitored. 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 on will be referred to as Decision Making register or DM register) is maintained to keep track of the amount of misses encountered in the test sets for both the techniques. Initially the value of this register is set to zero. 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 or just CB-DPET.

0_3	15_3	15_4
5_3	4_3	6_3
6_3	3_3	8_3
2_3	1_3	5_3
9_3	12_3	9_3
8_3	11_3	2_3

DPET
(i)

0_3	3_4	8_4
5_2	5_4	5_1
6_3	1_4	9_4
2_2	2_4	2_1
9_3	15_3	15_4
8_3	4_3	6_3

DPET - V
(ii)

Fig.6 (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

3.5 Thread Based Decision Making

The concept of CB-DPET can be extended to suit multi-threaded applications with a slight modification. For every thread we allocate a DM register and few test sets. 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. It is to be noted that the sets running DPET (or DPET-V) need not have to 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 we choose the policy as DPET.

3.6 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, we determine whether each block has shared or private data (using the id of the thread that accesses it) and decide on the counter values appropriately. For this purpose, there is a 'blockstatus' register associated with every cache block. Initially it is set to a value of '-1'. 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 (in our case we set it to 100). So if the register holds the value of 100, the corresponding block is regarded as shared. Otherwise it is treated as private.

Table 1: CPU architecture configuration parameters

Parameter	Value
Number of Cores	2
Instruction Set Architecture (ISA)	ALPHA
Clock Frequency	2Ghz
Supported Cache Levels	L1 and L2

3.6.1 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.

Table 2: Cache architecture configuration parameters

Parameter	L1 Cache		L2 Cache
Total Cache Size	i-cache	32 kB	2MB
	d-cache	64 kB	
Associativity	2-way		8-way
Miss Information/Status Handling Registers (MSHR)	10		20
Cache Block Size	64B		64B
Latency	1ns		10ns
Replacement Algorithm	LRU		CB-DPET

4.0 SIMULATION METHODOLOGY

For simulation purpose we use Gem5 [29] - a modular discrete event driven computer system open source simulator platform. It can simulate a complete system with devices and an operating system, in full system mode (FS mode), or user space only programs where system services are provided directly by the simulator in syscall emulation mode (SE mode). We use the FS mode.
Tables 1 and 2 contain the CPU and cache architecture parameters used for simulation.

4.1 Benchmark Used

The Princeton Application Repository for Shared-Memory Computers (PARSEC) [30,31] is a benchmark suite that comprises numerous large scale commercial multi-threaded workloads for CMP. For our simulation purposes, we have chosen seven workloads from PARSEC. Every workload is divided into three phases: an initial sequential phase, a parallel phase (which is further split into two phases), and a final sequential phase. We focus on the performance obtained at the parallel phase which is also known as the Region of Interest (ROI). It is to be noted that all the statistics collected to measure the cache performance correspond to the ROI of the workloads. The basic characteristics of the chosen benchmarks are shown in Table 3.

Table 3: Summary of the key characteristics of the PARSEC benchmarks

Program	Application Domain	Working Set
Blackscholes	Financial Analysis	Small
Canneal	Computer Vision	Medium
Dedup	Enterprise Storage	Unbounded
Ferret	Similarity Search	Unbounded
Fluidanimate	Animation	Large
Swaptions	Financial Analysis	Medium
Vips	Media Processing	Medium

5.0 RESULTS AND ANALYSIS

Fig.7 shows the overall number of hits obtained for each workload with respect to CB-DPET and LRU at L2 cache. The *x*-axis represents individual benchmarks and *y*-axis denotes the overall number of hits obtained in the ROI. These numbers have been scaled down by appropriate factors to keep them around the same range. It can be seen from the figure that CB-DPET has resulted in improvement in the number of hits compared to LRU for majority of the workloads. Increase in the number of hits recorded, varies from a minimum of 5.5% to a maximum of 12.3%.

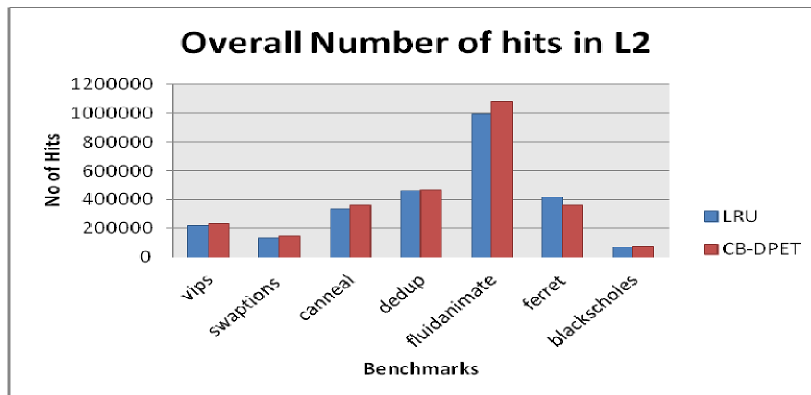

Fig.7 Overall number of hits recorded in L2 cache

Fig.8 shows the hit rate obtained for individual benchmarks. Hit rate can be calculated by the following formula.

Hit Rate = (Overall number of hits in ROI) / (Overall number of accesses in ROI)

Six of the seven benchmarks have reported marginal improvements in the hit rate. Fluidanimate shows the maximum increase whereas dedup had shown minimal improvement.

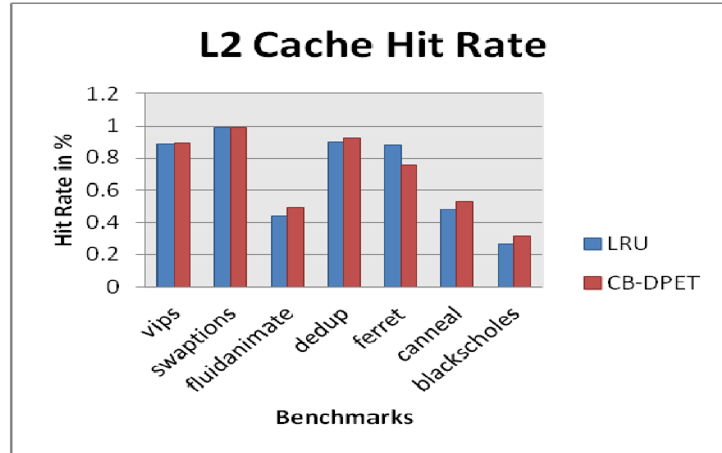

Fig.8 L2 cache hit rate

Fig.9 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). It is to be noted here that 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 captured in the graph demonstrated in Fig.9. CB-DPET has shown an increase in hits in most of the phases.

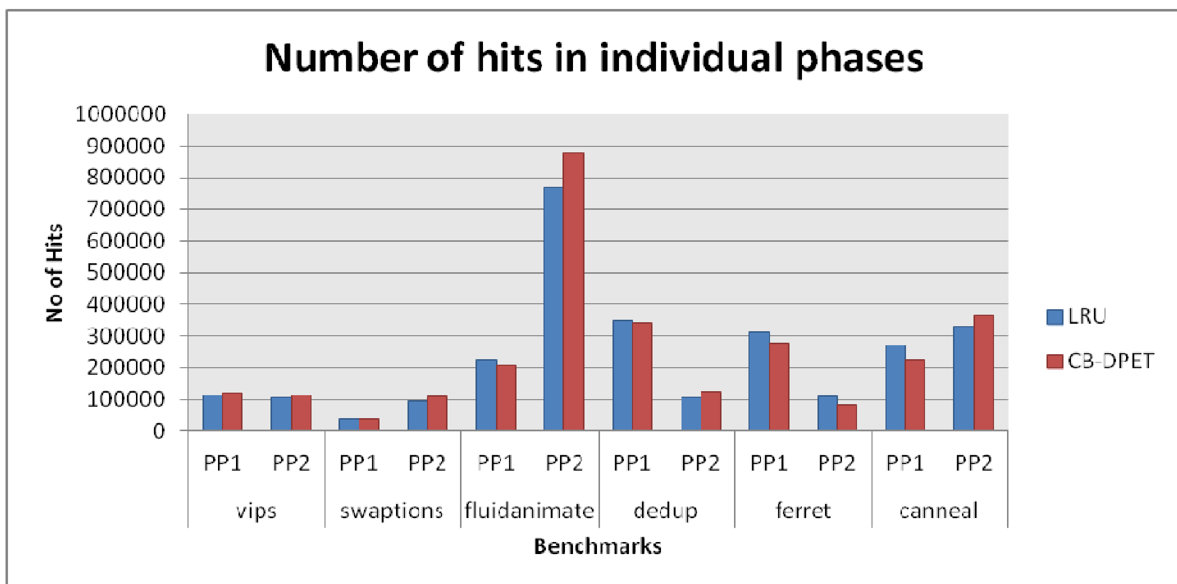

Fig.9 Hits recorded in individual phases of ROI

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 our case) following a miss in that level of memory. Miss latency at L2 cache for individual cores is shown in Fig.10. It is usually measured in cycles. To present it in a more understandable format, it has been converted to seconds using the following formula:

Miss latency (in seconds) = (Miss latency in cycles) / (CPU Clock Frequency)

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.

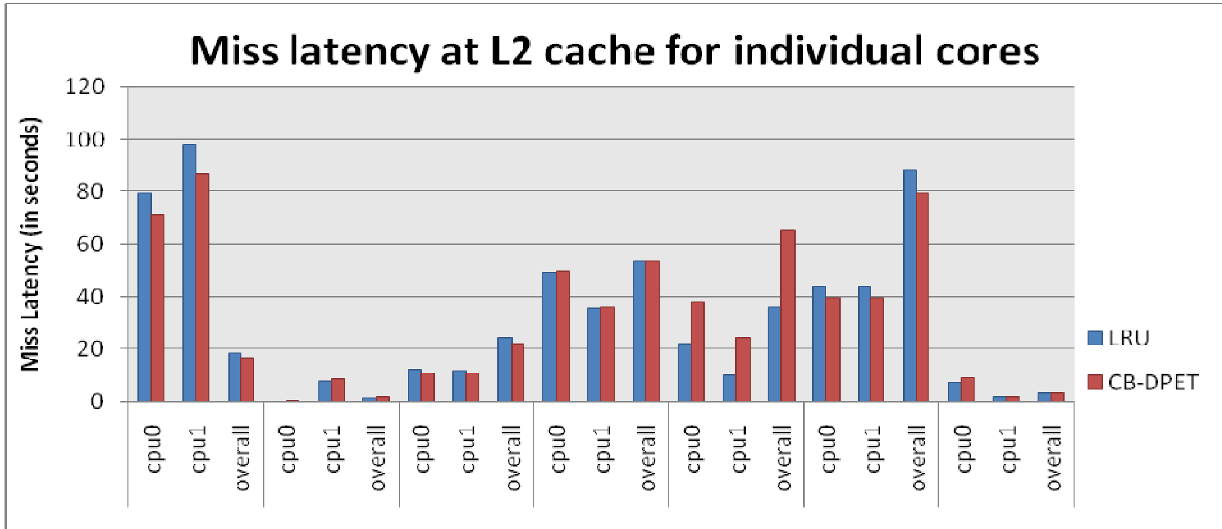

Fig.10 Miss latency at L2 cache for individual cores

Overall miss latency at L2 is also shown in Fig.10. Note that this is with respect to the ROI of the benchmarks. Considerable reduction in latency is observed across the majority of the benchmarks. Fig.11 shows the overall number of misses recorded by each benchmark. A marginal decrease in the number of misses is observed across the majority of the benchmarks compared to LRU.

$$\text{Hit Percentage} = ((\text{Hits in CB-DPET}) - (\text{Hits in LRU})) / (\text{Hits in CB-DPET}) \times 100$$

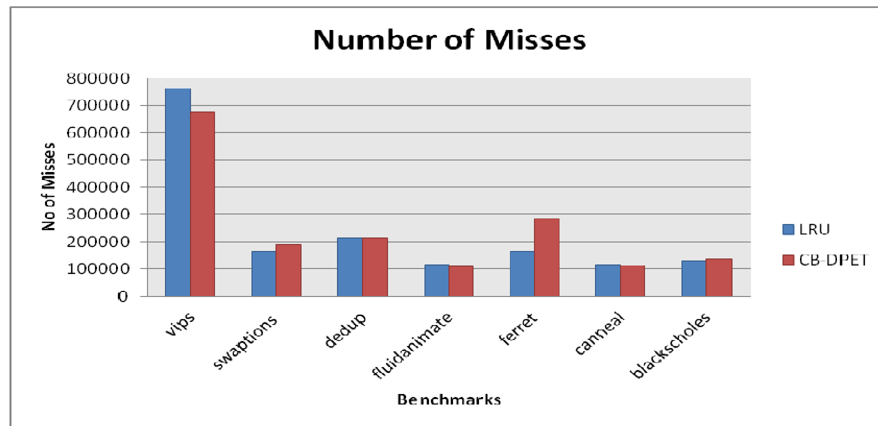

Fig.11 Overall number of misses at L2 cache

Fig.12 shows the percentage increase in hits obtained from CB-DPET compared to LRU. From the figure it can be seen that there is a marginal increase in the hit percentage for all the benchmarks except ferret. Approximately 9% of improvement over LRU is obtained in 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.

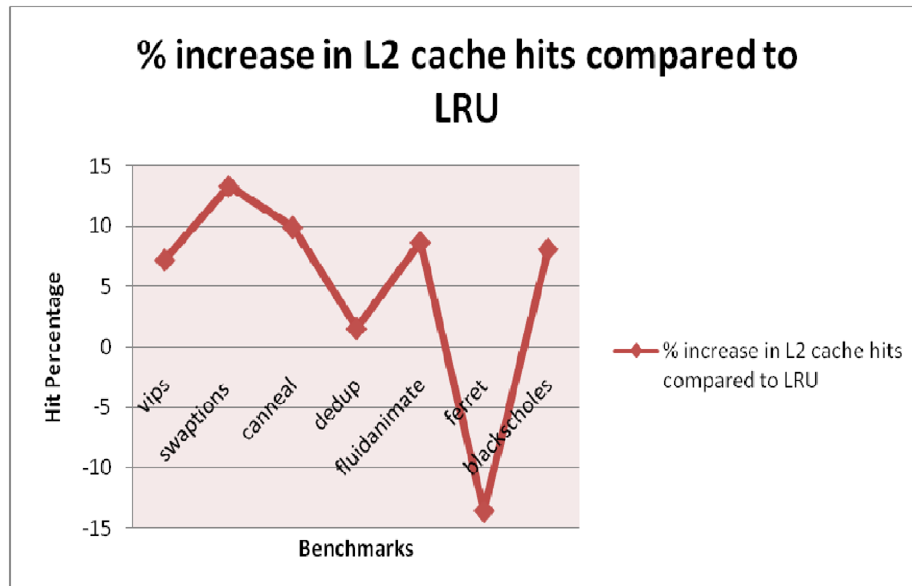

Fig.12 Percentage increase in L2 cache hits compared to LRU

6.0 CONCLUSION

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 like LRU, NRU, etc. may not be effective in many instances as they do not adapt dynamically to the data requirements. Numerous researches [13,14,15,16,17] are being conducted to find novel ways for improving the efficiency of the replacement algorithms applied at LLC. In this work, the following are our conclusions:

- We propose a novel replacement algorithm called CB-DPET which takes a more systematic and access-history based decision compared to LRU. It is further split into DPET and DPET-V. DPET associates a counter with every cache block to keep track of its recent access information. Based on this counter's values, our algorithm makes replacement, insertion and promotion decisions.

- DPET-V is a thrash proof version of DPET. Here occasionally we allow the data items to stay for a longer period in the cache by adjusting the counter value appropriately.

- In multi-threaded applications, we allocate a DM register and a specific number of 'test sets' for every thread. Based on tracking the misses obtained by applying both these techniques in their allocated test sets using the DM register, these threads go on to determine which method to apply for the remaining cache sets.

- These algorithms are made thread-aware by using the context id (or 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, we tag it 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 a marginal improvement in the overall hits (up to 9%) when compared to LRU at the L2 Cache level. Finally, these improved schemes can be applied to other fields such as artificial intelligence to improve the efficiency of algorithms as previously proposed [32, 33, 34].

REFERENCES

[1] John L. Henessey, David A. Patterson, "*Computer Architecture: A Quantitative Approach*", Fourth Edition, Elsevier Publications, 2006.

[2] Tola John Odule, Idowun Ademola Osinuga, "Dynamically Self-Adjusting Cache Replacement Algorithm", *International Journal of Future Generation Communication and Networking*, Vol. 6, No. 1, Feb. 2013.

[3] Andhi Janapsatya, Aleksandar Ignjatovic, Jorgen Peddersen and Sri Parameswaran, "Dueling CLOCK : Adaptive Cache Replacement Policy Based on the CLOCK Algorithm" in *Design in Automation & Test in Europe Conference & Exhibition (DATE)*, March 2010, p. 920-925.

[4] Tian Pan, Xiaoyu Guo, Chenhui Zhang, Wei Meng, Bin Liu, "ALFE: A Replacement Policy to Cache Elephant Flows in the Presence of Mice Flooding" in *IEEE International Conference on Communications (ICC)*, June 2012, p.2961-2965.

[5] Ponnusamy S P, Karthikeyan E, "Cache Optimization on Hot-Point Proxy (HPProxy) Using Dual Cache Replacement Policy" in *International Conference on Communications and Signal Processing (ICCSP)*, April 2012, p.108-113.

[6] Hariram Chavan, Suneeta Sane, H. B. Kekre, "A Markov-Graph Replacement Policy for Mobile Environment" in *International Conference on Communication, Information & Computing Technology*, Oct. 2012, p.1-6.

[7] Konstantinos Nikas, Matthew Horsnell, Jim Garside, "An Adaptive Bloom Filter Cache Partitioning Scheme for Multi-Core Architectures" in *IEEE International Conference on Embedded Computer Systems Architectures Modeling and Simulation*, 2008, p.25-32.

[8] Kedzierski K, Moreto M, Cazorla F.J, Valero M,"Adapting Cache Partitioning Algorithms to Pseudo-LRU Replacement Policies" in *International Symposium on Parallel & Distributed Processing (IPDPS),* April 2010, p.1-12.

[9] Fazal Hameed, Bauer L, Henkel J, "Dynamic Cache Management in Multi-Core Architectures Through Runtime Adaptation" in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, March 2012, p.485-490.

[10] Ke Zhang1, Zhensong Wang, Yong Chen3, Huaiyu Zhu4, Xian-He Sun, "PAC-PLRU: A Cache Replacement Policy to Salvage Discarded Predictions from Hardware Prefetchers", *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing,* May 2011, p.265-274.

[11] Hongliang Gao, Chris Wilkerson, "A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing", *JWAC – 1st JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship*, June 2010.

[12] Dongxing Bao, Xioming Li, "A Cache Scheme Based on LRU-Like Algorithm" in *International Conference on Information and Automation (ICIA),* June 2010, p.2055-2060.

[13] Miao Zhou, Yu Du, Bruce Chilers, Rami Melham, Daniel Mosse, "Writeback-Aware Partitioning and Replacement for Last-Level Caches in Phase Change Main Memory Systems", *ACM Transactions on Architecture and Code Optimization*, Vol. 8, No. 4, Article 53, Jan. 2012.

[14] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Srinivas Subramoney, Joseph Nuzman, "Introducing Hierarchy-Awareness in Replacement and Bypass Algorithms for Last-Level Caches", *ACM Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, p.293-304.

[15] Liqiang He, Yan Sun, Chaozhong Zhang, "Adaptive Subset Based Replacement Policy for High Performance Caching" in *1st JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship*, 2010.

[16] Carole-Jean Wu, Margaret Martonosi, "Adaptive Timekeeping Replacement: Fine-Grained Capacity Management for Shared CMP Caches", *ACM Transactions on Architecture and Code Optimization*, Vol. 8, No. 1, Article 3, April 2011.

[17] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely Jr., Joel Emer, "CRUISE: Cache Replacement and Utility-Aware Scheduling", *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vol. 40, Issue. 1, March 2012, p.249-260.

[18] Shekhar Srikantaiah, Mahmut Kandemir, Mary Jane Irwin, "Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors", *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vol. 36, Issue. 1, March 2008, p.135-144.

[19] R Manikantan, Kaushik Rajan, Govindarajan R, "NUcache: An Efficient Multicore Cache Organization Based on Next-Use Distance" in *IEEE 17th International Conference on High Performance Computer Architecture (HPCA)*, Feb. 2011, p.243-253.

[20] Mazen Kharbutli, Yan Solihin, "Counter Based Cache Replacement and Bypassing Algorithms", *IEEE Transactions on Computers*, Vol. 57, Issue. 4, April 2008, p.433-447.

[21] Haiming Liu, Michael Ferdman, Jaehyuk Huh, Doug Burger, "Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency", *41st Annual IEEE/ACM International Symposium on Microarchitecture*, Vol. 1, Issue. 12, 2008, p.222-233.

[22] Livio Soares, David Tam, Michael Stumm, "Reducing the Harmful Effects of Last-Level Cache Polluters with an OS-level, Software-Only Pollute Buffer", *41st Annual IEEE/ACM International Symposium on Microarchitecture*, 2008, p.258-269.

[23] Rami Sheikh, Mazen Kharbutli, "Improving Cache Performance by Combining Cost-Sensitivity and Locality Principles in Cache Replacement Algorithms" *in IEEE International Conference on Computer Design (ICCD)*, 2010, p. 76-83.

[24] Adwan AbdelFattah, Aiman Abu Samra, "Least Recently Plus Five Least Frequently Replacement Policy (LR+5LF)", *International Arab Journal of Information Technology*, Vol. 9, No. 1, Jan. 2012, p.16-21.

[25] Richa Gupta, Sanjiv Tokekar, "Proficient Pair of Replacement Algorithms on L1 and L2 Cache for Merge Sort", *Journal of Computing*, Vol. 2, Issue. 3, March 2010.

[26] Jan Reineke, Daniel Grund, "Sensitivity of Cache Replacement Policies", *ACM Transactions on Embedded Computer Systems*, Vol. 9, No. 4, Article 39, March 2010.

[27] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., Joel Emer, "Adaptive Insertion Policies for High Performance Caching", *ACM SIGARCH Computer Architecture News*, Vol. 35, Issue. 2, June 2007, p.381-391.

[28] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Joel Emer, "Adaptive Insertion Policies for Managing Shared Caches", *ACM Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, p.208-219.

[29] N. Binkert et al. "The gem5 simulator", *SIGARCH Computer. Architecture New*, Vol. 39, Issue. 2, May 2011, p.1-7.

[30] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, Kai Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications", *Princeton University Technical Report,* TR-811-08, Jan. 2008.

[31] M. Gebhart et al., "Running PARSEC 2.1 on M5", University of Texas at Austin, *Department of Computer Science*, Technical Report #TR-09-32, Oct. 2009.

[32] Raj, R.G. and Abdul-Kareem, S. (2009), Information Dissemination And Storage For Tele-Text Based Conversational Systems' Learning. Malaysian Journal of Computer Science, Vol. 22(2): Dec 2009. pp 138-159.

[33] Raj, R.G. and Abdul-Kareem, S. (2011). A Pattern Based Approach for The Derivation Of Base Forms Of Verbs From Participles And Tenses For Flexible NLP. Malaysian Journal of Computer Science, Vol. 24(2): Jun 2011. pp 63-72.

[34] Raj, R.G. and Balakrishnan, V. (2011). A Model For Determining The Degree Of Contradictions In Information. Malaysian Journal of Computer Science, Vol. 24(3): September 2011. pp 160-167

BIOGRAPHY

Muthukumar S

Muthukumar S received the BE degree in Electronics and Communication Engineering and the ME degree in Applied Electronics from Bharathiar University, Coimbatore in 1989 and 1992, respectively. He is currently pursuing the Ph.D. degree from BSA University, Chennai. His current research interests include Multi-Core Processor Architectures, High Performance Computing and Embedded Systems. He is working as Associate Professor in the Department of Computer Science and Engineering at Sri Venkateshwara College of Engineering, Chennai, India.

P K Jawahar

Dr P K Jawahar received the BE degree in Electronics and Communication Engineering from Bharathiar University, Coimbatore in 1989 and MTech degree in electronics and communication engineering from Pondicherry University 1998. He received the Ph.D. degree in Information and Communication Engineering from Anna University, Chennai in 2010. His current research interests include Computer Architecture, VLSI, Embedded Systems and Computer Networks. He is working as Professor in the Department of Electronics and Communication Engineering at BSA University, Chennai, India.