A Computationally Efficient P-LRU based Optimal Cache Heap Object Replacement Policy

Burhan Ul Islam Khan
Department of ECE
Kulliyyah of Engineering
IIUM, Malaysia

Rashidah F. Olanrewaju
Department of ECE
Kulliyyah of Engineering
IIUM, Malaysia

Roohie Naaz Mir
Department of CSE
National Institute of Technology
Srinagar, Kashmir

Abdul Raouf Khan
Department of Computer Sciences
King Faisal University
Saudi Arabia

S. H. Yusoff
Department of ECE
Kulliyyah of Engineering
IIUM, Malaysia

***Abstract*—The recent advancement in the field of distributed computing depicts a need of developing highly associative and less expensive cache memories for the state-of-art processors i.e., Intel Core i6, i7, etc. Hence, various conventional studies introduced cache replacement policies which are one of the prominent key factors to determine the effectiveness of a cache memory. Most of the conventional cache replacement algorithms are found to be as not so efficient on memory management and complexity analysis. Therefore, a significant and thorough analysis is required to suggest a new optimal solution for optimizing the state-of-the-art cache replacement issues. The proposed study aims to conceptualize a theoretical model for optimal cache heap object replacement. The proposed model incorporates Tree based and MRU (Most Recently Used) pseudo-LRU (Least Recently Used) mechanism and configures it with JVM's garbage collector to replace the old referenced objects from the heap cache lines. The performance analysis of the proposed system illustrates that it outperforms the conventional state of art replacement policies with much lower cost and complexity. It also depicts that the percentage of hits on cache heap is relatively higher than the conventional technologies.**

***Keywords*—cache heap object replacement; garbage collectors; Java Virtual Machine; pseudo LRU**

I. Introduction

To bridge the performance gap between the main memory, cache, and the processor, the current research trends towards computer hardware engineering are more focused on designing efficient memory hierarchy to reduce the average memory access time required by the CPU. Numerous research works highlight that computer scientists performed an in-depth investigation on Level 2 (L2) caches for several reasons such as; firstly, processors can create a level of abstract to hide the Level 1 (L1) cache misses followed by the L2 cache hits [1]. The processor and the cache schedule exploit the Instruction Level Parallelism (ILP) to determine out of order execution phases and non-blocking phases of cache lines. Therefore, it is very difficult to hide the L2 cache miss penalty [2]. Secondly, it can also be seen that the optimization of L1 caches considering short hit times depicts more complex scenario during execution time as compared to the less critical L2 cache hits. The involvement of L2 caches allows efficient cache replacement optimizations on smarter replacement policies [3] [4]. All the conventional state-of-art cache replacement policies except the random policies can detect the cache memory line to be eliminated by looking into its past reference. In the case of Least Recently Used (LRU) policy implementation, a set of state transition signals (control status bits) is required to update the cache schedule about when each cache block is accessed [5]. Therefore, set-associativity in between cache and main memory increases the number of bits and it imposes cost and computational complexity. Possibly, the best way to reduce the complexity associated with LRU, the random policy has been chosen but only to an extent. Most of the researchers and computer designers opted for Pseudo LRU heuristic algorithm to minimize the hardware cost and enhance the performance of the system by approximating the LRU mechanism [6]. Though, most of the recent studies on cache replacement policies usually incorporate LRU techniques with limited associativity but few of them initiated the enhancement of LRU by improving replacement decisions [7][8][9].

There are very few state-of-the-art optimal cache replacement policies that are feasible as well as useful with Java's garbage collection mechanism. Moreover, a few of the policies such as OPT L2, FIFO L2, and random page replacement policies lead to an uncertain scenario where the tall cache miss rate could be higher. It has also been observed that most of the conventional studies are repetitive in nature with regard to consideration of fewer efficient performance parameters. Therefore, addressing the above-stated research issues, the proposed study aims to combine both tree based and MRU pseudo-LRU based cache heap replacement policies which are further integrated with the Java's garbage collection scenario to further improve the performance scenario on cache miss rate. The experimental outcomes obtained from the prototype simulation show that it gives more precise comparative analysis considering different cache models and it performs very less iteration during implementation process at a

This work was partially supported by Ministry of Higher Education Malaysia (Kementerian Pendidikan Tinggi) under Fundamental Research Grant Scheme (FRGS) number FRGS-15-254-0495 and Research Initiative Grant Scheme (RIGS) number RIGS15-150-0150.

faster rate and lesser space complexity from all the experimental aspects.

Based on the motivation stated above, this study aims to develop a cache heap object replacement policy which is based on Java's indirect garbage collection mechanism [10]. It also evaluates two different types of cache object replacement policies which are Tree based Cache Heap Object Replacement and MRU bits based Cache Heap Object Replacement mechanisms respectively. The experimental analysis performed considering a test bed highlights that our proposed model achieves optimal computational efficiency and very less L2 cache miss rates during the Java's object replacement and allocation process execution. It also shows the performance improvement for different cache configurations.

The rest of the manuscript is organized as follows: Section II discusses the literature survey followed by the theoretical analysis of conventional memory hierarchy discussed in Section III. Section IV discusses design methodology and the algorithm implementation of the proposed P-LRU based cache heap object replacement policy and the functionality of garbage collectors on cache replacement operation. Section V describes the experimental analysis followed by the conclusions in Section VI.

II. Literature Survey

This section highlights most of the significant studies carried out towards designing optimal cache replacement algorithms to reduce the operational cost during the instruction read, write and fetch operations from cache memories.

Authors in [11] developed an optimal scheme for cache replacement namely, Min-SAUD that determines the cache objects to be replaced by incorporating validation delay. Various factors affecting cache performance are taken into consideration such as update frequency, retrieval delay, cache validation cost, data size and access probability. It has been assumed that the cache has zero access latency because it can be easily neglected in comparison to the server access latency. This study is known to be the first of its kind to analyze the effect of factors like cache validation delay and access latency on cache performance; and thus functions as a fundamental guideline for designing cache management policies. In this paper, stretch has been employed as the main performance metric as it takes into account data service time which is therefore fair for different-sized items. Furthermore, the performance of this scheme has been thoroughly evaluated through successive simulation experiments for diverse system configurations. The results reveal that Min-SAUD performs much better than the two existing algorithms viz. LRU and SAIU. In terms of stretch, the proposed cache replacement policy yields much better access cost as compared to other replacement policies. However, an optimal solution can be obtained only when the data size is comparatively smaller than cache size. Authors have avoided updating access rate of all cached items during every replacement in order to lower the computational complexity. In future, this policy can be extended to cache admission for caching client data. Moreover, simulation results show that if improvements are made in estimation methods, performance of Min-SUAD can be enhanced further towards that of an ideal cache replacement policy.

In [12] authors have addressed the cache replacement problem for transcoding proxy caching. Generally, cache replacement algorithms replace cached objects with minimum profit for accommodating the incoming object to be cached. This algorithm considers the interrelationship among various versions of a single multimedia object and replaces any version as per the aggregate profit unlike the traditional algorithms that performed summation of the individual profits of the versions to be replaced. Moreover, cache consistency has also been considered which was not included in existing cache replacement schemes. A complexity analysis has been performed for demonstrating efficiency of this algorithm. Simulation of the proposed algorithm is performed by considering several metrics viz. request-response ratio (RRR), delay saving ratio (DSR), staleness ratio (SR) and object-hit ratio (OHR); and results reveal its superior performance when compared to conventional algorithms.

An adaptive cache replacement policy called CRFP i.e., Combined LRU and LFU Policy was put forward by authors in [13]. CRFP is found to respond to access pattern changes effectively and dynamically by switching among the cache replacement policies. This policy makes use of cache directory for learning access pattern at run-time. Also, a SWITCH value is maintained by cache manager for recording existing replace policy. The proposed cache replacement scheme is built on LRU stack which is used for maintenance of pages in the cache and LRU queue is used for maintaining the pages replaced recently. CRFP was implemented by authors in PostgreSQL which is an open-source database management system and its performance was compared with LRU, ARC, LFU and LRFU. It was found that CRFP performed better than the other algorithms in most of the situations thus making it suitable for several cache management systems.

Authors in [14] presented Locality-Aware Cost-Sensitive (LACS) cache replacement strategy that brings together cost sensitivity and locality principles. The cost of a cache block is estimated by LACS from number of instructions issued by processor during cache-miss on the block and then the blocks with poor locality and less cost are victimized for maximizing overall cache performance. The proposed cost estimation policy has been found to be effective in uniprocessor as well as multiprocessor architectures. LACS accelerates the 10 L2 cache performance controlled SPEC CPU2000 benchmarks by about 85 per cent and 15 per cent on an average without degrading any 20 SPEC CPU2000 benchmarks in its evaluation on uniprocessor architecture. On the other hand, it accelerates 6 SPEC CPU2000 pairs of benchmark by about 44 per cent and 11 per cent on an average during its evaluation on dual core multiprocessor architecture. Furthermore, this algorithm has proven to be effective over varied associativities and sizes of L2 cache. But there have been some problems in case of shared L2 caches such as cache partitioning issue. Although LACS brings down miss count in comparison to LRU, it is not clear if private or shared threshold values should be used.

In [15], Reuse and Memory Access cost-aware eviction Policy (ReMAP) has been proposed by the authors that, considers memory access cost, Post Eviction Reuse Distance (PERD) and recency for making eviction decisions. This policy shows superior performance since it takes into account the interaction of last-level-cache (LLC) with main memory for better cache management decision making. In this policy, a cost is assigned to every cache line that indicates the eviction cost for a specific cache line as opposed to retaining it in the cache, unlike assignment of a fixed counter value as seen in LRU. ReMAP has been evaluated through an open source simulator, namely gem5 and the full system evaluation showed about 13 per cent reduction in number of misses in SPEC2006 applications as compared to LRU and 6.5 per cent reduction on an average. However, DRRIP and MLP aware replacement schemes have shown only -0.7 and 5 per cent reduction in miss count respectively. Notably, the proposed scheme achieved an IPC performance gain of about 4.6 per cent as against 1.8 and 2.3 per cent in MLP-aware and DRRIP replacement schemes respectively.

The drawbacks of existing cache replacement algorithms motivated the authors in [16] to put forward a policy called Recency Frequency Replacement (RFR) that combines recency of a cache block with frequency i.e., a hybrid of LRU and LFU. Two weighing values are associated to every cache block that corresponds to LRU and LFU thereby maintaining a balance between them and then cumulative weight is also computed from the two values. This policy includes three important steps viz. weighing LRU/LFU, fusing LRU/LFU and predicting line to be evicted. The RFR scheme has been simulated by authors using the multi-core heterogenous user-customized simulator, CUBEMACH that captures and compares the cache dynamics of LRU, FIFO and LFU. The effectiveness of RFR has been analyzed using various benchmarks viz. GCC, equake, VPR and parser which revealed about 9 per cent better performance than LRU, FIFO and LFU with respect to miss ratio.

In [17] authors have proposed two algorithms, wildcard rules caching and Rule Cache Replacement (RCR) for solving the TCAM problem in software defined networking. In these algorithms, the accumulated contribution value of a rule-set is calculated instead of individual value. The wildcard rules caching policy caches the wildcard rules that are matched frequently without incurring additional cache cost and shows efficient TCAM space utilization in comparison to cover set caching. Further, the rule replacement cache policy outperforms LRU, Adaptive Replacement Cache (ARC) and random replacement (RR) algorithms in maintaining a high hit ratio as it considers traffic locality. Due to the inability to get real data-center traffic, ClassBench has been used for generating synthetic rule policy with diverse packet classification. In this simulation, the maximum capacities of TCAM have been set as 3K and 2K for wildcard rules caching and cache replacement algorithms respectively. It was observed that both the algorithms presented improved performance with an average ratio of 10 per cent for different ranges of traffic volume as against cover set caching policy.

In [18] a light-weight caching strategy called Optimized Cache Replacement algorithm in Information Centric Networks (OCRICN) has been designed in order to reduce redundancy and maximize the cache efficiency. This can be achieved by storing small-sized and high frequency chunks which are closer to end-user so that a router nearby may satisfy the request packet rather than the burden of traversing a lengthy path to the actual server. As a result, both bandwidth consumption and cache resource usage can be optimized with this algorithm that greatly enhances the system performance. When simulated, the proposed algorithm performs much better than the traditional schemes in terms of server bandwidth consumption and access latency which is depicted by 60 per cent improvement in hit ratio and 30 per cent reduction in server messages. The algorithm is believed to show further improvement in performance in case of a complex topology owing to effect of multiple caching metrics on hierarchical cache level.

Regional Popularity-Aware Cache replacement (RPAC) algorithm has been presented in [19] that prolongs the lifetime of Solid State Drive (SSD) cache by reducing number of erasure operations and cache replacements that are unnecessary. This algorithm records region (formed by consecutive disk blocks) popularity instead of block popularity to select the block to be replaced. Thus, sequential I/O blocks are grouped in SSD leveraging the disk-access spatial locality. RPAC has been evaluated in real system by several workloads. In the simulation with CacheSIM, two types of I/O traces from real systems viz. Mail and Webvm are employed. On analyzing the simulation results, it is revealed that RPAC has better applicability for small caches. Further, it is seen that more memory and time are consumed in block level popularity statistics versus region level popularity statistics. For validation of the algorithm, the authors have implemented the same in Facebook's flashcache and used Filebench to perform comparison with other policies. The results show about 31 and 53 per cent improved I/O throughput than FIFO and LRU respectively while reducing number of erase operations by about 17 per cent.

Motivated to decrease the garbage collection overhead in flash memory based SSDs, authors have proposed Random First Flash Enlargement (RFFE) algorithm in [20]. This algorithm makes performance improvements in write operation by employing sequence detection mechanism and presenting three novel techniques: spatial locality buffering, varied write enlargements and write random ahead. The main complexity of the proposed algorithm is designing efficient data structure for searching customary pre-write contexts (PWCs) and removing outdated PWCs. The random ahead characteristic of this algorithm is beneficial for interleaved and slow sequential wires. The application of this algorithm on random as well as sequential write queues brings down the number of merge operations in garbage collection thereby improving write performance in SSD. Furthermore, the frequent write feature of sequential stream reduces wait time of buffer data and thus enhances data reliability. The simulation results of RFFE depict that it outperforms Block Padding Least Recently Used (BPLRU), Recently-Evicted-First (REF) and Fully-Associative Sector Translation (FAST) algorithms for random as well as sequential write patterns.

Authors in [21] have proposed a model for rule caching that is based on traffic as well as the path of flows for optimizing the switch cache replacement. The proposed algorithm called Flow Driven Rule Caching (FDRC) is an attempt to deal with unpredictable flows and the size constraint of cache in software defined networks. Particularly, a low complexity optimized algorithm has been designed by authors for achieving considerably high cache-hit ratio by making use of prefetching together with a special replacement policy for predictable as well as unpredictable flows. Furthermore, the performance of the algorithm in terms of cache hit ratio has been evaluated against popularly used replacement algorithms namely LRU and FIFO.

Authors in [22] have proposed an algorithm for cache replacement Least Error Rate (LER) for minimizing error rate in case of L2 caches. This study is known to be the first of its kind to address the contribution of cache replacement on error rate of Spin Torque Transfer RAM (STT-RAM). In this algorithm, the block to be accommodated is placed in a line which has the least write-operation error rate. This has been achieved by performing a comparison of incoming block contents with cache set lines. The efficiency of this algorithm is dependent on the data value patterns of workloads in cache lines. For evaluation of LER, gem5 simulator has been used with SPEC CPU2006 workload. The simulations which were carried out on a billion instructions showed a reduction in error rate by 2 times with approximately 3.6 and 1.4 per cent dynamic energy consumption and performance overheads respectively on comparison with LRU. It is to be noted that authors have used the method of Early Write Termination (EWT) in the simulation.

The conventional schemes that have been proposed for cache replacement like LRU, LFU and other utility-based schemes prove to be unsuitable for caching video stream. The authors in [23] have presented Optimized Cache Replacement (OCR) scheme that caches video stream by taking into consideration the arrival patterns of user request. After grouping the users in all the request intervals possible, the density of users is computed in every request interval and those groups are cached which have the maximum user density. In this way, the groups having comparatively lower user density are replaced to accommodate the high user density groups. When simulated, this cache replacement scheme is shown to increase the hit ratio by 2 times in comparison to LRU scheme. This scheme has been further extended from a single cache to cooperative caches and is known as Cooperative Cache Replacement (CCR). Performances of both OCR and CCR have been verified through simulation that evidently shows the reduction in server load and the improvement in hit-ratio when compared to LRU.

The following Table I highlight some of the state-of-the-art studies introduced by various researchers in this area

TABLE I. SIGNIFICANT STUDIES CARRIED OUT TOWARDS DESIGNING OPTIMAL CACHE REPLACEMENT ALGORITHMS

AUTHOR	CONTRIBUTION	RESULTS OBTAINED	LIMITATIONS
(Xu et al., 2004) [11]	Presented Min-SAUD gain based cache replacement policy for wireless data dissemination	• Outperforms the previously proposed LRU and SAIU in terms of better access cost when evaluated for diverse system configurations	• Location dependent services and object transcoding have not been considered • Prefetching should be combined with this scheme to further increase performance • Scope for further enhancing parameter estimation method
(Li et al., 2006) [12]	Presented an effective algorithm for cache replacement meant for proxy transcoding based on an aggregate cost-saving function	• Showed better performance than some of the existing algorithms like DSR, RRR, OHR and SR besides cache consistency • Combined cache consistency and additional emerging factors in transcoding proxies which was not done in the previously proposed algorithms	• Can incur huge cost theoretically if large number of different objects are to be removed
(Zhansheng et al., 2008) [13]	Proposed CRFP – a novel adaptive cache replacement strategy that brings together LRU and LFU strategies	• Self-tuning policy that is capable of switching among various cache replacement policies dynamically and adaptively • Higher bit ratio than other algorithms in most of the simulations • Applicable to majority of applications like TPC-H and TPC-C workloads • Simple and easy implementation besides less computational overhead and space consumption	• Further investigation needs to be done for optimizing the CRFP by tuning cache directory size and SWITCH_TIMES value
(Sheikh and Kharbutli, 2010) [14]	Proposed LACS algorithm combining the principle of locality with cost sensitivity	• Boosted L2 cache performance by a considerable percentage • Performed robustly when demonstrated on various cache configurations • Outperformed the other state-of-art cache replacement algorithms that claimed to be cost-sensitive	• Certain issues arise in case of shared L2 caches • Maintenance of LRU stack information expensive and difficult • More comprehensive evaluation of LACS in a multi-threaded environment required
(Arunkumar and Wu, 2014) [15]	Presented ReMAP which considers memory access behaviour and reuse characteristics to make	• Reduced number of misses and IPC performance gain as compared to MLP-aware replacement, LRU and DRRIP • Provides superior performance than prior	• Extra hardware requirement despite the substantial performance gain • Negligible logic overhead which is the result of effective cost calculation

	eviction decisions at LLC(last level cache)	works by combining recency, memory access cost information and PERD	• Can misguide reuse behaviour if there is a very large number of entries in victim buffer thereby degrading performance
(Anandkumar et al., 2014) [16]	Proposed hybrid algorithm – RFR for cache replacement combining LRU and LFU	• Improved cache hit-to-miss ratio than LRU, FIFO and LFU when simulated on a variety of cache sizes together with associativity • Increases overall system performance by combining frequency and recency of the cache block • Simple implementation and requirement of minimal hardware	• Degree of accuracy can be increased by considering extra parameters such as application complexity, library execution status and dependencies
(Sun and Wang, 2015) [17]	Put forward an algorithm for light-weight caching management that maximizes volume of traffic handled by caches besides reducing bandwidth usage	• Provides better system performance in terms of server bandwidth consumption and access latency as compared to existing strategies • Easily implementable scheme that yields 2 times better hit ratio than LRU for a small sized cache	• Evaluated only in homogenous cache environment neglecting heterogenous one
(Ye et al., 2015) [18]	Presented an algorithm namely RPAC that improves lifetime as well as I/O performance of SSD	• Prolongs SSD lifetime by reducing unnecessary cache replacements and erase operations • Enhanced I/O throughput by about 53% and minimized cache replacements by about 98.5% • Implemented successfully in real systems and evaluated for many workloads that revealed its extra-ordinary performance than conventional algorithms • Memory efficient and adoptable for a range of applications	• Increases the replacements with the increase in cache size therefore more suitable for small caches • I/O throughput shows weak sensitivity to statistic cycle • Performance can be enhanced only when degree of spatial locality of workloads is high
(Ramasamy and Karantharaj, 2015) [19]	Put forward a novel page replacement algorithm - RFFE for improving write-operation performance in flash memory based SSD	• Outperformed the existing schemes like FAST, BBLRU and REF in terms of erase, merge and write count • Provides improved write-response time besides no upsurge in log block area of SSD • Minimizes the overall page replacement cost on write by improving and bringing up the best schemes proposed in the past	• Small overhead is involved by the process of indexing and purging of PWCs • Restricted to NAND based flash memories only • Negative effect on write performance on frequent issue of flush command by host file system
(Li et al., 2015) [20]	Proposed algorithm called FDRC – flow driven rule caching for optimizing caache replacement in software defined networks	• Low complexity algorithm with higher cache-hit ratio thereby improving network performance • Performs better than LRU and FIFO under diverse network conditions	• A meagre improvement of 3.2% in performance of FIFO and LRU for a large cache size
(Sheu and Chuo, 2015) [21]	Proposed wildcard rules caching and rule cache replacement algorithms to solve rule dependency problem and cache important rules to TCAM using cover-set approach	• Improved cache-hit ratios as compared to prior works viz. LR, random replacement and ARC • Usage of wildcard rule cache algorithm shows about 10% improvement in comparison to cover-set cache method	• Still a scope for improving cache hit ratio • RCR algorithm can be further refined for calculating weight value which is conforming to traffic locality
(Monazzah et al., 2016) [22]	Presented cache replacement algorithm namely LER (Least Error Rate) for minimizing error rate in L2 caches	• No area overhead imposed on system • In comparison to LRU, 2 times reduction in error rate with 1.4% performance overhead and 3.6% overhead in dynamic energy consumption	• Indirectly imposes dynamic energy performance overhead with additional L2 cache misses because it evicts cache lines

III. Theoretical Analysis of Conventional Memory Hierarchy

A. *Ideal Cache Heap Model*

Define In this model, there are two different levels of hierarchy comprising of cache heap size of N bytes along with the cache heap lines length of B bytes as shown in Fig. 1, and the calculation of total cache heap lines is done using (1).

$$\text{Total cache heap lines: } \frac{N}{B} \qquad (1)$$

An ideal cache heap is fully associative. It implies that any line can go anywhere in the cache heap memory [24]. The most significant aspect of an ideal cache heap is that it incorporates an optimal omniscient replacement model for page allocation in cache heap blocks. It figures out which page needs to be replaced or eliminated from the cache heap blocks if it is required.

B. *Performance Measurement*

In this model, the performance evaluation is done by computing the running time of instruction while executing.

Work ← W (Ordinary serial running time during execution of instruction when you run your code in processor)

C_{miss} ← Cache heap Misses

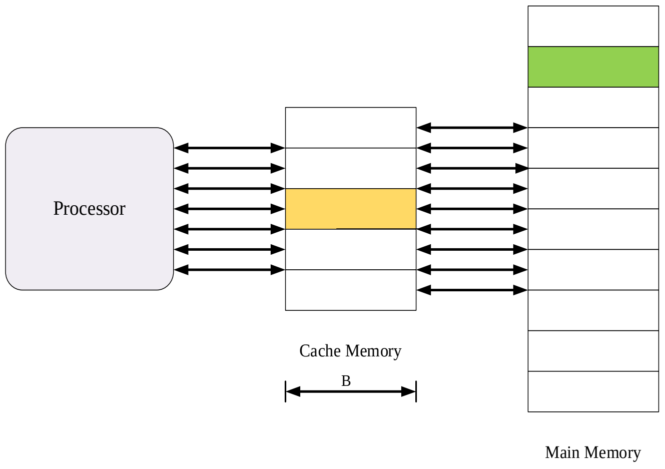

Fig. 1. A cache heap memory hierarchy

C. *How Reasonable the Ideal Cache Heaps are?*

Suppose an algorithm is processed through, and it incurs Q cache heap misses on an ideal cache heap of size *n*. The algorithm will be running in the machine, and the least recently used (LRU) replacement policy will be used. Instead of using ideal cache heap, the cache heap will be fully associative with size two *m*. Basically, instruction of a block which has been referenced longest ago in the past will be eliminated using least recently used algorithm. The algorithm incurs 2Q cache heap misses; hence, it indicates that LRU with the same constant factors depicts a similarity with optimal solution [25][26].

D. *Choosing LRU or the Ideal Cache with the Omniscient Replacement*

For most of the asymptotic analysis, LRU and optimal page replacement policies are considered as convenient. The upper bounds of an algorithm depict its efficiency regarding optimal page replacement whereas the lower bound conveys that an algorithm is not that efficient on its LRU. The above-stated statement can bring an idea to get the reason behind the internal memory management. Therefore, it is intended to use both optimal and LRU policy for upper bounds and lower bounds.

The proposed system theoretically assumes that the cache heaps which are taken into account are not fully associative and incur a different cost on bandwidth and latency (load and store). Miss on a load and miss on a store have their different impacts.

E. *Design of a Theoretically Good Algorithm*

1) Tall Cache Heap Assumptions: Tall cache heap assumption is made based on an ideology of considering the cache heap lines in ideal mode but in the real-time configuration, the cache heap lines are assumed to be not ideal. The concept of tall Cache heap line is theoretically derived by (2) below:

$$B^2 < \eta \times N \text{ where the constant } \eta \text{ is} \leq 1 \quad (2)$$

Equation (2) represents the fact that the cache heap integrated with the system should always be tall [27]. For example, in the modern computing systems, the cache line length (Intel Core i7) usually considered is 64 bytes where the L1 cache size is considered 32 KB. It can be easily interpreted that the L2 and L3 cache heaps should be much bigger in size as it could have 64K cache heap lines. Thus, more lines associativity can increase the chances of cache heap replacement. It also ensures that more items can be placed into the cache.

2) Disadvantage of Using Small Cache Heaps: It can be seen that a matrix of dimension D × D may not be fitted in a small cache heap even if it satisfies the criteria of tall cache heap assumption which is $D^2 < \eta \times N$. Hence, it is said that a matrix always fits into a tall cache heap.

The asymptotic analysis of the above stated test case shows that if $D = \Omega(B)$, then the cache heap miss that occurs while loading in D^2 data into the B cache heap is $\Theta(D^2/B)$.

Fig. 2 shows a tentative representation of the above stated test scenario.

After the text edit has been completed, the paper is ready for the template. Duplicate the template file by using the Save As command, and use the naming convention prescribed by your conference for the name of your paper. In this newly created file, highlight all of the contents and import your prepared text file. You are now ready to style your paper; use the scroll down window on the left of the MS Word Formatting toolbar.

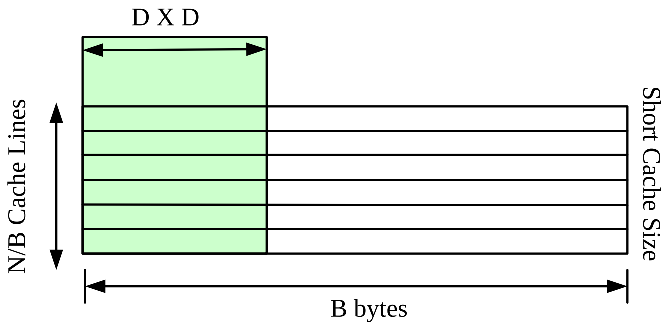

Fig. 2. Short cache heap over matrix

The analysis of the multiplication of D×D matrix depicts the cache heap misses from the theoretical aspect. The analysis of cache heap misses was performed by considering row-major layout of arrays with two different test cases.

a) Test Case 1

The test case scenario 1 set the parameters in a way where D > N/B. The Least Recently Used (LRU) policy has been assumed to derive the computational asymptotic notations. It is also used for computing the cache heap misses which is denoted as μ(D) here.

The cache heap misses in test case 1 is represented with $\mu(D) = \Theta(D^3)$. The row-major layout conventionality defines that matrix B misses in every access on cache heap lines. A tentative diagram for the analysis of cache heap misses is given in Fig. 3.

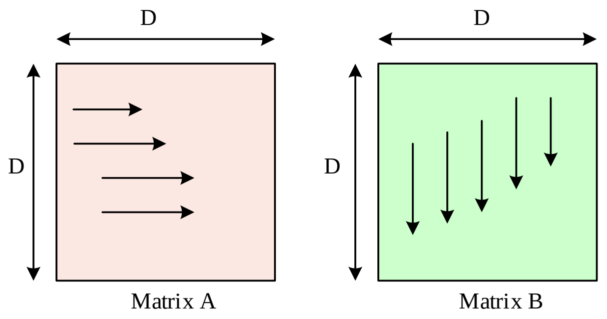

Fig. 3. Row major layout of arrays during the matrix multiplication

b) Test Case 2

The test case scenario 2 defines the parameters in such a way where $N^{(1/2)} < D < \eta \times (N/B)$. After the execution of LRU, the cache heap misses is represented as $\mu(D) = D.\Theta(D^3/ B)$. It also shows that matrix B evaluates the spatial locality. The work analysis associated with tiled matrix multiplication by considering the tall cache heap for sub-matrix, given in (3) shows that

$$W(D) \rightarrow \Theta((D / S)^3(S^3)) \rightarrow \Theta(D^3) \tag{3}$$

It implies that a tuning technology is introduced to organize the sub matrix in a way thus all the sub matrices can fit into the Cache heap. The tuning of the sub metrics are defined by $s \rightarrow \Theta(M^{1/2})$. In case of a tall cache, the misses will be $\Theta(S^2/B)$.

IV. Proposed System

The proposed study aims to design an efficient and optimal least recently used algorithm based on Java's garbage collection mechanism. The advantage of using the proposed system is, it incorporates Java's garbage collector to perform an indirect object allocation, while making use of an optimal least recently used policy to replace or eliminate the object which has been referenced long back during the program execution. It also incorporates a mechanism that allows an object which has been referred long back to get out of the scope of a program execution state. Thus, there will be no need of allocating a cache heap line for referring the least recently used object. The proposed system incorporates an indirect way of object allocation performed by the garbage collector that ensures lesser execution time and higher efficiency in cache heap memory. The above stated fact depicts that due to less execution time, it requires fewer iteration during implementation process while ensuring less time and space complexity from a theoretical point of view. The proposed model represents an approximation of the pseudo-tree based and MRU based algorithm for the optimal page or referenced object replacement from the cache heap lines [28][29]. The associativity of these two algorithms improves the cost of operations and also reduces the complexity of the implementation process. As per the theoretical interpretations, it can be said that the least recently used objects addressed in a cache heap line are not always only the entities to be replaced whereas it can be replaced on the most recently used object reference lines also. The proposed system utilizes an indirect way of object allocation (JVM Indirect Method) in the cache heap blocks which save a lot of processing time as compared to the shift method of JVM for object allocation. The proposed page replacement heuristic for all the cache heap line accessibility usually references objects, which are tracked down by the n-way-1 bits. The n-way usually denotes the number of cache heap lines associated with each block of memory. The proposed model is tested on 4-way cache heap memory, and it also considers the pseudo LRU method to track the bits which are B_0, B_1, and B_2 from a decision binary engine. The track control signal flag value when set to true (i.e., when $B_1 = 1$) indicates that the objects residing on the lower cache heap blocks CL0 and CL1 are recently used whereas the flag value of B_1 when set to false (i.e., when $B_1 = 0$) means that two other higher cache heap blocks CL2 and CL3 objects are recently used. The proposed model initiates the bit B_2 block for keeping the access track in between CL2 and CL3 object entities. The cache heap scheduler is programmed in such a way that it looks for CL0, CL1, CL2 and CL3 cache heap line information. In the proposed work, each cache heap line contains the information like which is Significant_Flag_Bit, Main_memory_word, etc. The significant flag bit contains the value 1 which indicates that the line cache heap object is not similar to the correspondent object of the RAM (Main Memory) whereas the value 0 represents that the cache heap line contains the exact copy of object value referenced in main memory line. The above stated fact corresponds to the situation when a CPU request for an object referenced line from the main memory. Hence, a translation script using MTL enables the accessing of the cache heap line address which is supposed to hold the object value. Cache heap miss can arise when no address information is found on the cache heap line block and variable B_1 which is also associated with the JVM's garbage collector, looks for the least recently used objects from the 2 lower cache heap lines or the two higher cache heap lines. B_0 and B_2 identify that and replace the cache heap line along with the least recently used object reference. This study also deals with the introduction of a new analytical concept of computationally efficient P-LRU based optimal cache replacement algorithm using Java's garbage policy from all the technical perspective. The proposed heuristic cache heap object replacement policy set the binary tree bits accordingly on a cache heap hit. The following Fig. 4 represents the conceptual overview of the proposed algorithm from a theoretical point of view.

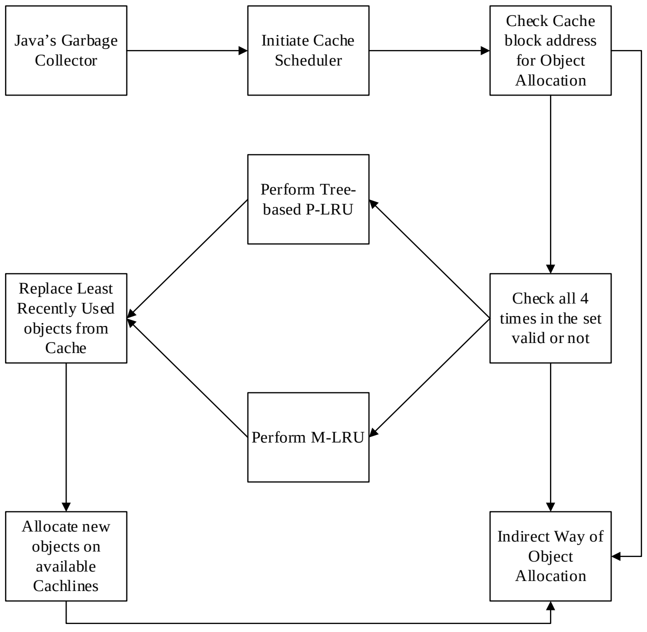
Java's Garbage Collector
Initiate Cache Scheduler
Check Cache block address for Object Allocation
Perform Tree-based P-LRU
Replace Least Recently Used objects from Cache
Check all 4 times in the set valid or not
Perform M-LRU
Allocate new objects on available Cachlines
Indirect Way of Object Allocation

Fig. 4. A tentative architecture of the proposed system

The following sections highlight an in-depth description of A. Tree based Cache Heap Object Replacement and B. MRU bits based Cache Heap Object Replacement respectively. Both algorithmic procedures are further combined for approximation of optimal page replacement using Java's garbage collector.

A. *Tree based Cache Heap Object Replacement*

In this case, the pseudo-LRU heuristic incorporates a binary tree based structure for localizing or de-localizing the object references from cache heap memory lines. SED heuristic study considers three levels of cycles in CL2, CL3, and CL1. In cycle 1, the algorithm checks for a valid bit i.e., if (B_1=1); if it is there, it will check for B_0 and B_2. If B_2 contains the flag bit 0, then it will search for the least recently used object from higher level cache heap blocks and replace that according to the cache heap hit. In cycle 2, it replaces cache heap line object from the lower level cache heap blocks (hit on CL3). The following diagram represents the concept of the tree based pseudo LRU policy. The object referencing and replacing the cache heap lines are performed by JVM's garbage collector.

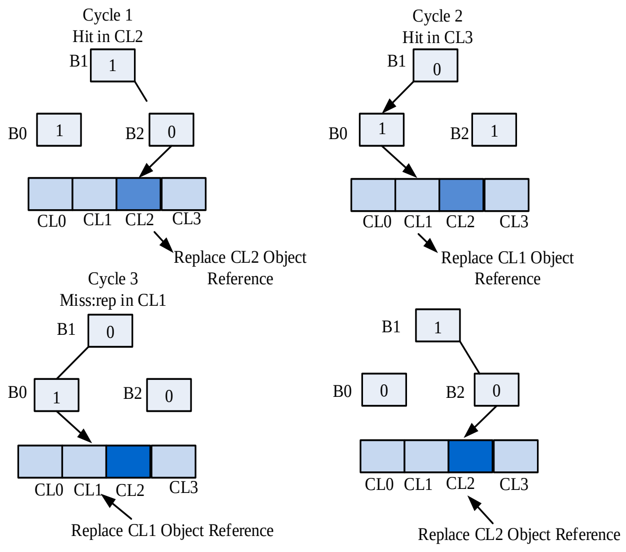

Fig. 5. Tree based cache heap object replacement

The above given Fig. 5 also highlights how cache heap line referenced objects are replaced or reallocated in cycle 3 and 4 based on the valid flag bits initiated by the tree based pseudo least recently used algorithm.

B. *MRU bits based Cache Heap Object Replacement*

The proposed system also incorporates another concept which is based on pseudo least recently used policy in turn based on most recently used cache heap reference bits for Java's object allocation. The concept is namely denoted as LRUm. In this case, each of the cache heap blocks is assigned with an MRU bit which is referenced with a tag table. The tag table usually maintains and updates the flag values i.e., from 0 to 1 or from 1 to 0. If the MRU flag value is set to 1, it indicates that a cache hit occurred in the cache heap block. It also represents that the cache block is most recently used. The cache controller is configured in a way that when it examines the MRU flag bits which indicate '0', it replaces the cache heap line address and sets the flag value as 1. MRU flag bit set to '1' for each cache object reference line indicates that it is most recently used. An example of this concept is illustrated in the Fig. 6.

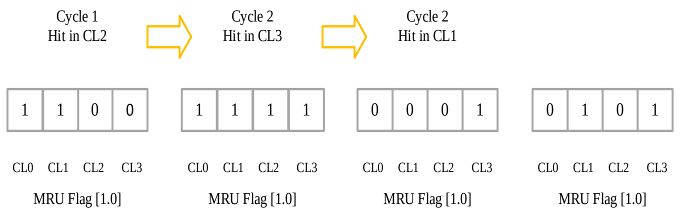

Fig. 6. JVM's cache heap object replacement based on MRU

The implementation process sometimes incurs a problem, if the MRU flag bits for all cache memory are set to 1 which usually depicts a deadlock situation on unavailability of all cache heap address lines. Therefore, to overcome this uncertain and problematic situation, a principle has been introduced. It says that all the MRU flag bits in the set should be cleared except the MRU flag bit which is being accessed simultaneously with a current program execution. The above-stated deadlock situation also may arise during any potential overflow situation.

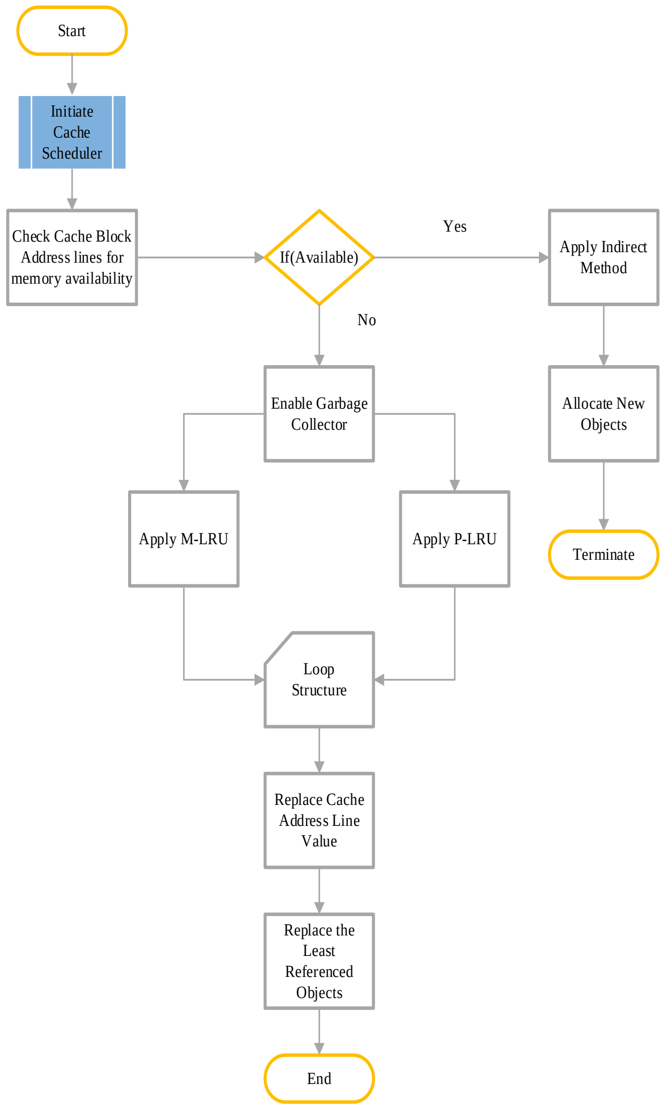

Fig. 7. Flow diagram of the proposed system

Fig. 7 represents a tentative flow chart of the proposed P-LRU based optimal cache heap object replacement policy. The asymptotic analysis of the proposed model evaluation depicts the complexity comparison of the above mentioned two different algorithms. The replacement heuristics also state that MRU based replacement updates the MRU flag bit(s) on cache hit and cache miss whereas Tree based replacement strategy updates the tree bit(s) during cache hit or cache miss.

V. Experimental Analysis

This section represents the experimental analysis carried out considering the cache miss rates and performance improvement (speed up) for a CMP architecture. It shows that the proposed model reduces the L2 cache miss rate very effectively as compared to the conventional LRU, OPT, FIFO, and random algorithms for cache replacement. The performance metric associated with different cache configuration is highlighted in Table II.

TABLE II. UNITS FOR MAGNETIC PROPERTIES

CONFIGURATION	MINIMUM	AVERAGE	MAXIMUM
256KB, 4-way	0%	4%	10%
512KB, 4-way	0%	15%	85%
1MB, 4-way	-3%	8%	48%
2MB, 4-way	-3%	19%	195%

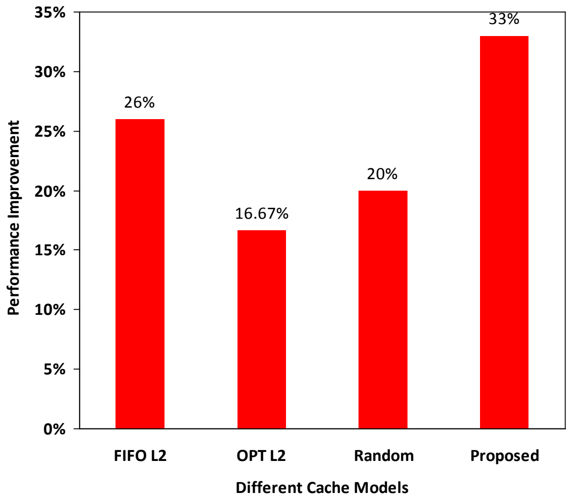

Fig. 8. Performance analysis of different cache configurations

Fig. 8 shows the performance analysis of different cache models which shows that the proposed JVM heap L2 cache achieves 33% performance improvement rate (Speed up) during garbage collector's object fetching and replacement phase. The proposed system has been evaluated using SPEC CPU2000 benchmarks [30][31]. It uses a uniprocessor architecture which speeds up the JVM L2 cache performance up to 33%. Another performance parameter which is taken into consideration is L2 cache miss rates. Fig. 9 shows the results obtained from the L2 cache miss rates. To further estimate the L2 cache misses for JVM, the correlation between blocks evicted by the OPT and LRU replacement mechanisms are thoroughly studied.

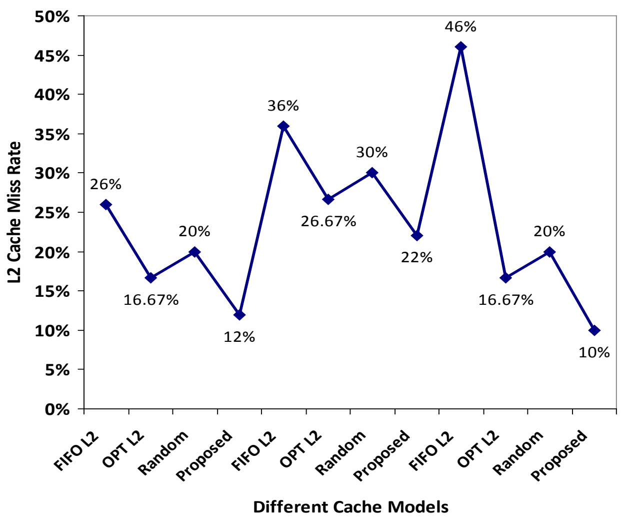

Fig. 9. L2 cache misses

The asymptotic analysis shows that our proposed algorithm achieves Θ(D^2/B) cache misses for loading the D^2 object into B bytes cache heap. The upper bound asymptote conveys that the proposed algorithm achieves very less L2 cache miss rates as well as high performance ratio in comparison to the conventional methods i.e. FIFO L2, OPT L2 and Random Replacement Policy. Fig. 9 also highlights the cache miss rates for different cache configurations.

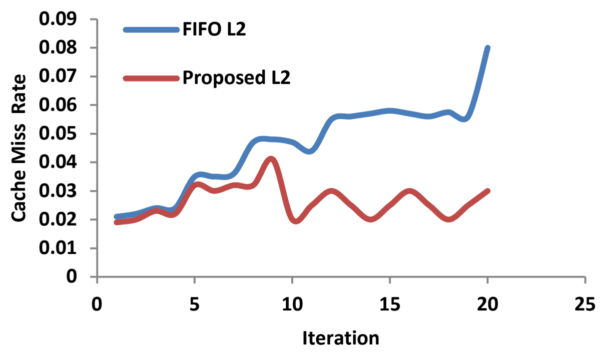

Fig. 10. L2 cache miss rate Vs Iteration

A performance evaluation in between FIFO L2 Cache and proposed JVM L2 has been carried out by obtaining the simulation results explicitly during processing of each and every instruction considering a constant rate of task arrival (depicted in Fig. 10). It shows that the proposed algorithm achieves very less cache miss rate while increasing the iteration as compared to FIFO L2 conventional cache replacement algorithm. Therefore, the simulation results in display of the interface design considering all the parameters also exhibit that the proposed algorithm executes jobs very faster and achieves very less amount of cache miss rate, which is considered as the significant contribution of the proposed study.

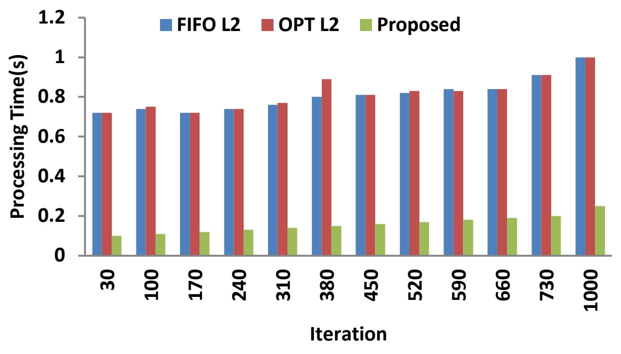

Fig. 11. Evaluation of processing time (s)

The graph in Fig. 11 represents the values (i.e. processing/computation time in seconds) obtained after processing the proposed cache heap object replacement policy in an experimental and iterative test-bed configuration. The above highlighted comparative study shows that our proposed algorithm achieves very less processing time as compared to the conventional mechanisms (OPT-L2 and FIFO-L2 Cache) during cache heap object replacement. In other words, it is computationally very much efficient and optimal in between both upper and lower time bound.

VI. Conclusion

The proposed study developed a Pseudo LRU based optimal cache object replacement policy to enhance the performance of Java's garbage collector and the cache scheduler. It incorporates two different types of page replacement policies i.e., Tree based Cache heap Object Replacement and MRU bits based Cache heap object replacement policies which improve the performance of the computation and execution of instructions on very less cache miss rates. The proposed algorithm has been configured with JVM's intermediate levels to enable the garbage collector during the instruction fetching and executing time. A significant theoretical analysis of the conventional memory management is highlighted in section III that depicts how an efficient cache replacement algorithm can be designed from efficient asymptotic aspects. The performance evaluation of the proposed system ensures its effectiveness in future research direction of cache memory design and development.

References

[1] R. F. Olanrewaju, A. Baba, B. U. I. Khan, M. Yacoob and A. W. Azman, "An Efficient Cache Replacement Algorithm for Minimizing the Error Rate in L2-STT-MRAM Caches," presented at Fourth International Conference on Parallel, Distributed and Grid Computing(PDGC), 2016

[2] M. Kharbutli and R. Sheikh, "LACS: A Locality-Aware Cost-Sensitive Cache Replacement Algorithm", *IEEE Transactions on Computers*, vol. 63, no. 8, pp. 1975-1987, 2014.

[3] Z. Wang, K. S. McKinley, A. L. Rosenberg and C. C. Weems, "Using the compiler to improve cache replacement decisions," in *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, Charlottesville, Virginia, 2002, pp. 199-208.

[4] S. Kumar and P.K. Singh, “An overview of modern cache memory and performance analysis of replacement policies,” in *Engineering and Technology (ICETECH), IEEE International Conference on*, 2016, pp. 210-214.

[5] C. C. Kavar and S. S. Parmar, “Improve the performance of LRU page replacement algorithm using augmentation of data structure,” in *Computing, Communications and Networking Technologies (ICCCNT), Fourth International Conference on*, Tiruchengode, 2013, pp. 1-5.

[6] R. F. Olanrewaju, A. Baba, B. U. I. Khan, A. W. Azman, and M. Yacoob, “A Study on Performance Evaluation of Conventional Cache Replacement Algorithms: A Review,” presented at Fourth International Conference on Parallel, Distributed and Grid Computing(PDGC), 2016

[7] Y. Xue and Y. Lei, “LRU-MRU with physical address cache replacement algorithm on FPGA application,” in *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on*, Chengdu, 2014, pp. 1302-1307.

[8] S. Ding, S. Lui and Y. Li, “Shared-cache simulation for multi-core system with LRU2-MRU collaborative cache replacement algorithm,” in *Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD), 2012 13th ACIS International Conference on*, Kyoto, 2012, pp. 127-131.

[9] A. Valero, J. Sahuquillo, S. Petit, P. Lopez and J. Duato, “MRU-tour-based replacement algorithms for last-level caches,” in *Computer Architecture and High Performance Computing (SBAC-PAD), 23rd International Symposium* on Vitoria, Espirito Santo, 2011, pp. 112-119.

[10] J. Jeong, P. Stenstrom, and M. Dubois, “Simple penalty-sensitive cache replacement policies,” *Journal of Instruction-Level Parallelism*, vol. 10, pp. 1-24, 2008.

[11] J. Xu, Q. Hu, W. C. Lee, and D. L. Lee, “Performance Evaluation of an Optimal Cache Replacement Policy for Wireless Data Dissemination,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 4, pp. 125-139, 2004.

[12] K. Li, H. Shen, K. Tajima and L. Huang, “An effective cache replacement algorithm in transcoding-enabled proxies,” *The Journal of Supercomputing*, vol.35, no. 2, pp.165-184, 2006.

[13] L. Zhan-sheng, L. Da-wei, and B. Hui-juan, **“**CRFP: A Novel Adaptive Replacement Policy Combined the LRU and LFU Policies,” in *Computer and Information Technology Workshops, IEEE 8th International Conference on,* 2008, pp. 72-79.

[14] R. Sheikh and M. Kharbutli, “Improving cache performance by combining cost-sensitivity and locality principles in cache replacement algorithms,” in *Computer Design (ICCD), 2010 IEEE International Conference on*, 2010, pp. 76-83.

[15] A. Arunkumar and C. J. Wu, “ReMAP: Reuse and Memory Access Cost Aware Eviction Policy for Last Level Cache Management,” in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, 2014, pp. 110-117.

[16] K. M. AnandKumar, A. S., D. Ganesh, and M S. Christy, “A Hybrid Cache Replacement Policy for Heterogeneous Multi-Cores,” in *Advances in Computing, Communications and Informatics (ICACCI), 2014 International Conference on*, 2014, pp. 594-599.

[17] X. Sun and Z. Wang, “An optimized cache replacement algorithm for information-centric networks”, in *Smart City/SocialCom/SustainCom (SmartCity), 2015 IEEE International Conference on*, 2015, pp. 683-688.

[18] F. Ye, J. Chen, X. Fang, J. Li and D. Feng, “A regional popularity-aware cache replacement algorithm to improve the performance and lifetime of SSD-based disk cache,” in *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*, 2015, pp. 45-53.

[19] A.S. Ramasamy and P. Karantharaj, “RFFE: A buffer cache management algorithm for flash-memory-based SSD to improve write performance,” *Canadian Journal of Electrical and Computer Engineering*, vol. 38, no. 3, pp. 219-231, 2015.

[20] H. Li, S. Guo, C. Wu, and J. Li, “FDRC: Flow-Driven Rule Caching Optimization InSoftware Defined Networking”, *IEEE ICC 2015 - Next Generation Networking Symposium*, 2015, pp. 5777-5782.

[21] J-P. Sheu and Y-C. Chuo, “Wildcard rules caching and cache replacement algorithms in software-defined networking,” *IEEE Transactions on Network and Service Management*, vol. 13, no. 1, pp.19-29, 2016.

[22] A. Monazzah, H. Farbeh and S. Miremadi, “LER: Least error rate replacement algorithm for emerging STT-RAM caches,” IEEE *Transactions on Device and Materials Reliability*, vol. 16, no. 2, pp. 220-226, 2016.

[23] X. Sun and Z. Wang, “Optimized cache replacement scheme for video on demand service,” in *Dependable, Autonomic and Secure Computing, 2013 IEEE 11th International Conference on,* 2013, pp. 192-199.

[24] R. Hemani, S. Banerjee, and A. Guha, “On the Applicability of Simple Cache Models for Modern Processors,” in *2016 2nd International Conference on Green High Performance Computing (ICGHPC)*, 2016.

[25] M. Frigo, C. Leiserson, H. Prokop and S. Ramachandran, “Cache-Oblivious Algorithms,” *ACM Transactions on Algorithms*, vol. 8, no. 1, pp. 1-22, 2012.

[26] E. Peserico, “Paging with dynamic memory capacity,” *arXiv preprint arXiv:1304.6007*, 2013.

[27] E. Demaine, "Cache-oblivious priority queue and graph algorithm applications", MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139, USA, 2002.

[28] K. Kamil, M. Moreto, F. J. Cazorla, and M Valero, “Adapting cache partitioning algorithms to pseudo-lru replacement policies,” in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, IEEE, 2010, pp. 1-12.

[29] X. Gu and C. Ding, “On the theory and potential of LRU-MRU collaborative cache management,” In *ACM SIGPLAN Notices*, vol. 46, no. 11, pp. 43-54. ACM, 2011.

[30] S. Sair. and M. Chamey, “Memory behavior of the SPEC2000 benchmark suite,” IBM Thomas J. Waston Research Center Technical Report RC-21852, 2000.

[31] M. Qureshi, A. Jaleel, Y. Patt, S. Steely and J. Emer, “Adaptive insertion policies for high performance caching”, *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, p. 381, 2007.