Managing Buffer Cache by Block Access Pattern

Reetu Gupta[1], Urmila Shrawankar[2]

[1] Project Student M.Tech (CSE), G.H. Raisoni College of Engineering, Nagpur, India

[2] G.H. Raisoni College of Engineering Nagpur, India

Abstract

As buffer cache is used to overcome the speed gap between processor and storage devices, performance of buffer cache is a deciding factor in verifying the system performance. Need of improved buffer cache hit ratio and inabilities of the Least Recent Used replacement algorithm inspire the development of the proposed algorithm. Data reuse and program locality are the basis for determining the cache performance. The proposed algorithm determines the temporal locality by detecting the access patterns in the program context from which the I/O request are issued, identified by the program counter signature, and the files to which the I/O request are addressed. For accurate pattern detection and enhanced cache performance re-reference behavior exploited in the cache block are associated with unique signature. Use of multiple caching policies is supported by the proposed algorithm so that the cache under that pattern can be best utilized.

Keywords: *replacement algorithms; block access pattern; program counter; reuse distance.*

1. Introduction

Technical advancements have widened the speed gap between the processor and storage devices. Buffer caches are used to keep blocks, which are likely to be accessed in the near future, thereby, enhancing the system performance on increasing workload.

More bandwidth and reduced memory access time are the demand of the modern multimedia and scientific applications. These demands require effective utilization of the available buffer cache space. By exploiting the patterns exhibited in the I/O request, the temporal locality of the blocks are determined by the replacement policies, which are decisive in determining the blocks to be replaced.

Due to the constant time and space complexity, and straightforward realization, LRU (Least Recently Used) or clock based approximation of LRU [1] has been widely used in real time system [2], [3] for managing the buffer cache. Conversely, recent studies revealed that there is considerable scope for performance improvement in LRU. Drawbacks as cache pollution and cyclic access to large working sets [4] lead to inefficient utilization of the buffer cache.

1.1 Causes of Inefficient Cache Utilization

Cache blocks with high temporal reuse, referenced after regular interval significantly enhance the system performance. Some problems identified in widely used LRU and prior work [5] that lead to degraded performance and increased cache miss ratios are discussed below.

Cache Pollution: The problem is identified in LRU and situations wherein blocks visited only once evict the cache blocks with high temporal reuse. This leads to cold blocks occupying the cache for a significant amount of time leading to the wastage of the memory resource.

Thrashing: When the cache is occupied by the blocks that are referenced regularly after certain intervals, blocks try to evict each other from the cache. Such situations are found in multimedia and gaming applications that demand high system performance.

Cyclic Access to Large Working Set: Such problem is recognized in scientific applications and LRU when the size of the frequently accessed blocks by the application is large than the cache size. The blocks to be referenced in near future are always evicted from the cache by LRU in such situations.

1.2 Characteristics of Pattern Based Replacement Schemes

Knowledge contained in the remarkable I/O request, made by the programs, was used by the efficient replacement schemes to overcome the above discussed problems. They made use of the patterns in which the buffer cache blocks were referenced. An automatic pattern based detection scheme should satisfy the following three requirements to efficiently exploit accessible the buffer cache space.

Accuracy: Precise detection of I/O access patterns is important as it determines the locality of accessed block,

IJCSI
www.IJCSI.org

decisive in concluding the block to be replaced and regulating the cache accordingly.

Responsiveness: Different behaviors demonstrated during the execution of a real time application reveal different transitions of the reference patterns, which are required to be adapted rapidly and sporadically by an efficient replacement policy.

Stability: A pattern based replacement schemes partitions the buffer cache based on the detected patterns and places the blocks in the corresponding cache partitions. A stable scheme should maintain the movement of blocks among the partition in a manner that reduces the maintenance overhead.

Section II gives a brief description about the types of replacement policies and the level at which the access patterns can be detected. Comparative study of the pattern detection algorithm is given in section III. Section IV introduces the program counter based technology in buffer cache management. Proposed algorithm is described in Section V followed by the expected result and conclusion.

2. Review of Existing Techniques

Replacement algorithms proposed to enhance the cache performance are grouped into following categories based on the phenomenon on which replacement decisions are made.

2.1 Reference History Based Replacement Schemes

To avoid the drawbacks of LRU, these algorithms examined the temporal locality of the cache blocks by maintaining a deeper past information than LRU about the accessed blocks.

LRU-K[6] replaced the block from the cache based on the K-th last distance. It handled the problem of cache pollution by adapting to the changing request of the applications. But the method adopted by LRU-K had logarithmic complexity.

To overcome the logarithmic complexity, 2Q [7] preserved cold blocks and reoccurring blocks in separate queues. Though it required tuning the parameters on per access it had constant complexity.

Frequency and recency of the cached blocks were combined by LRFU (Least Recently/ Frequently Used) [8]. The replacement decision was based upon the weights assigned to each cache blocks and a tunable parameter λ.

Distance between the last and second-to-the-last time references were preserved by LIRS [9] for estimating the probability of the blocks that could be re-referenced. Separate variable size LRU queues were used for discriminating the hot and cold blocks. Upon cache miss cold blocks were replaced.

Adaptive Replacement Cache (ARC) [10] and its variation CAR (Clock with Adaptive Replacement) [11] were the extensions to LRFU. Cache was divided into two queues one for holding cold and other for frequently accessed blocks. The partitions were managed as LRU in ARC and as clock based LRU in CAR

The inherent disadvantage of the above techniques was that they were not able to use the reference regularity information exhibited in the I/O request.

2.2 Replacement Schemes Based on Application / User Hints

Operating system makes use of the hints inserted by the programmer into the application to know well in advance the time and the blocks that would be accessed in future. This requires a careful and detailed understanding of the application behavior by the programmer. The replacement decisions are then taken based on the hints.

Upon cache miss the global policy decided the application and a local policy decided which block of the selected application would be removed from the cache in Application Controlled File Caching (ACFS) [12]. Hinted prefetching and caching blocks and unhinted caching blocks were placed in separate logical partitions allocated to file buffers dynamically according to the estimated cost in TIP [13].

Complier inserted hints [14] in the recompiled applications were used by the OS to take replacement decision and reduce the burden on the programmer. But I/O access patterns exhibited at runtime could not be explored by complier based techniques.

Neural networks and hidden markov models learning algorithms were proposed in [15] to classify the access patterns at runtime.

2.3 Access Pattern Based Replacement Schemes

Efficient algorithms that made replacement decisions, based on the patterns of the blocks, accessed by the I/O request, are classified into following three categories based on the level at which the patterns are detected.

Application Level: Detection based Adaptive Replacement (DEAR) [16], scheme proposed for buffer cache management, based on, the periodic and dynamic detection, of the reference pattern of the executing applications applied diverse replacement policies adaptable to diverse applications.

File Level: Unified Buffer Management (UBM) [17], surmounted the incapability of LRU by utilizing the regularities of block accesses made to files.

Program Context Level: By associating the Program Counters (PC) of the call instructions that generated I/O request with the program context in which they are issued, program counter based schemes, Program Counter Based Classification (PCC) [18] and Adaptive Multi Policy Caching (AMP) [19] accurately classified the access patterns.

Recent studies [20, 21] revealed that performance of buffer cache can be enhanced by using the knowledge enclosed in reference regularities of access pattern. In the technique proposed in [22] reuse pattern of each block was updated periodically to achieve high performance and take the replacement decision.

The applications having working set larger than the cache and those exhibiting mixed behavior by accessing some blocks in frequent interval while some after a long time were misclassified by the technique in [22].

Some of the techniques [20, 23, and 24] made changes in the cache structure incurring the hardware overhead to enhance the system performance and addressed the drawbacks of LRU. Conversely later techniques [21, 22] focused on analyzing the patterns on cache insertions avoiding changes in the cache structure and incurring less hardware overhead. Thus [20, 22] used reuse distance of accessed blocks, while [24] predicted dead block to make the replacement decision and enhance the performance.

3. Comparison of Pattern Detection Schemes

A file-level detection algorithm UBM [17] and two program context level detection algorithm PCC [18] and AMP [19] are discussed in the following section. The pattern detection process, advantages and the limitations of the techniques are described and compared.

3.1 Unified Buffer Management (UBM)

Working Steps:

- At file level it separates the I/O references according to target files.
- Automatically classifies the I/O access pattern into one of the three categories: sequential, looping and others.
- The UBM scheme stores the detected block in separate partitions of the buffer cache, managed by appropriate management scheme based on the detected pattern. Marginal gain function is used to allocate the block among partitions.
- For blocks in the sequentially referenced partition, MRU replacement policy is used. For periodic partition, block with the longest period is replaced first and MRU is used for the blocks with same period. LRU is used for blocks belonging to other reference pattern.

Advantages:

- The looping patterns are automatically detected and managed by a period based replacement policy and the buffer space is allocated, based on marginal gain.
- It also gave preferences to blocks belonging to sequential references when replacement was needed.

Limitations:

- It considered only the past access behavior of the block and worked at file level only.
- Significant amount of time was spent in training the access pattern of new file.

3.2 Program Counter Based Classification (PCC)

Working Steps:

- At the program context level, PCC technique exploited the virtual program counters exhibited in the application's binary execution codes to separate the I/O references into sub streams according to their Program Counter Signature (PCs).
- PCs are obtained by traversing the function stack backward through main.
- PCC then classified the pattern based on virtual program counter into the reference pattern categories similar to UBM with same replacement policies.

Advantages:

- It can accurately predict the reference pattern of new files before any access is performed, eliminating the training delay.
- It can differentiate among multiple concurrent access patterns in single file.

Limitations:

- Counters used to maintain block access information do not accurately reflect the statistical status of each PC process, resulting in the misclassification of the access pattern.

- It only considers the relation between the last PC and current PC that accessed the same data block.

3.3 Adaptive Multi Policy Caching (AMP)

Working Steps:

- It inherits the designs of PCC but measures the recency by using a mathematical expression
- AMP differentiates the I/O request based on the program contexts or code location according to the comparison between static threshold and average recency.

Advantages:

- Uses a new robust scheme for detecting looping pattern in access streams.
- Low over-head randomized way of managing the cache partition was adopted.

Limitations:

- It cannot detect the phenomenon of pattern sharing among multiple PCs.
- Moreover pattern conflicts reduce detection accuracy and increase management overhead.
- PC-based scheme cannot accurately distinguish locality strengths.

4. Program Counter Based Buffer Cache Management

Program counter based technique was proposed in [18] for optimizing the buffer cache. Program counters of the call instructions that issued the I/O request was utilized by the operating system to correlate the I/O operation with the program context in which they were issued.

Per PC-classification classifies the access patterns more accurately compared to per-file and per-application classification. Also as per-PC patterns are required to be learned once, their response time is earlier compared to prior classification technique.

By taking full benefit of temporal locality of the data blocks, used for taking the block replacement decisions, cache performance can be enhanced. Using this basic concept many pattern detection algorithms PCC [18] and AMP [19] examined the temporal locality obtained from the previous access information of the cached blocks.

Based on the program counter concept in computer architecture correlation between the previous accesses with the future reoccurrences were obtained. It was revealed that instructions identified by program counter performed uniquely and behaved in same manner in future. Thus these studies assumed that in future there is a high probability that program counter would access the block in the same manner as they did it in the past.

The working of PCC [18] and AMP [19] is depicted in the following call graph shown in the “Figure 1”. Run time relationships between program procedures are represented by call graph where node represents the function and the arcs describe the calling order.

Wrapper layer interprets the I/O instruction issued by some function in the application layer hiding the complexities and providing the flexible interface by invoking the system call to access the data. Program counter signatures, that are the sum of the entire program counter of all functions along the I/O call path, are defined for identifying the program context from which the I/O request was issued.

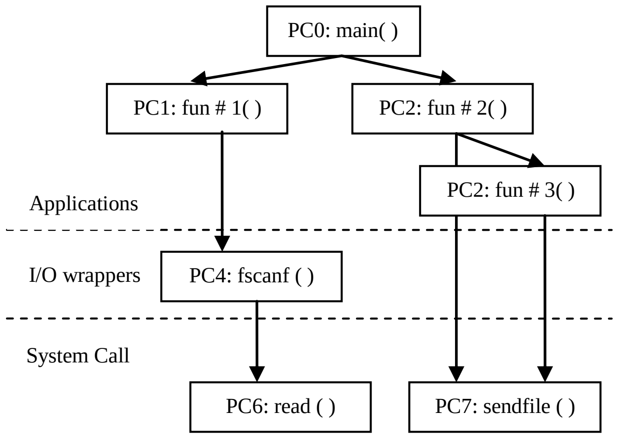


Figure 1.Call Graph of Application.

5. Proposed Algorithm

Traditionally predictions based on program counter (PC) were effectively and widely used in the field of computer architecture. The proposed algorithm uses the concept of PC for identifying the access pattern of buffer cache blocks. Benefits of the proposed algorithm compared to the previous pattern based algorithms are mentioned below in “Table 1”.

Table 1: Comparison with Existing Pattern Detection Algorithm

UBM	***PCC and AMP***	***Proposed Algorithm***
Works at File Level.	Works at Program Context Level.	Works at File and Program Context Level.
Classifies based on past access behavior.	Classifies based on Program Context	Classifies based on PC and past access behavior of current requested block

Following section provides a in depth explanation of proposed pattern based detection algorithm for managing the buffer cache that predicts the future behavior of the block by combing the significant features of file and program context level techniques.

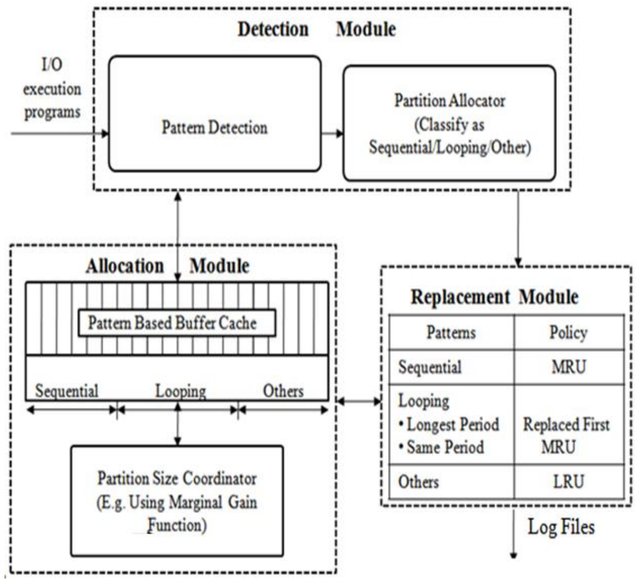
5.1 Modules in Proposed Scheme

The proposed pattern detection based algorithm is composed of three modules as shown in “Figure.2”.

Detection Module: Dynamic and periodic detection of the I/O access pattern is performed by this module, which classifies the I/O request into defined patterns.

Replacement Module: This module manages the cache under the respective partitions by applying specific replacement policies that best utilize the cache under that reference pattern.

Allocation Module: This module deals with efficient and dynamic management of the areas allocated to respective partitions and manages the movement of the blocks between the partitions in a manner that will reduce the cache maintenance overhead.


Detection Module
I/O execution programs
Pattern Detection
Partition Allocator (Classify as Sequential/Looping/Other)
Allocation Module
Pattern Based Buffer Cache
Sequential
Looping
Others
Partition Size Coordinator (E.g. Using Marginal Gain Function)
Replacement Module
Log Files

Patterns	Policy
Sequential	MRU
Looping • Longest Period • Same Period	Replaced First MRU
Others	LRU

Figure 2. Modular Design of the Proposed Algorithm.

5.2 Types of I/O Access Patterns

Access patterns could be classified into following categories

Sequential Reference: Blocks referenced only once and never revisited would be considered to exhibit sequential references.

Looping Reference: Blocks referenced at the regular interval are classified as looping.

Others: If none of the above patterns are found.

5.3 Phases of the proposed Algorithm

The Algorithm works in three phases.

Phase I: For each block reference, initially the algorithm updates the file table, containing the information about the blocks belonging to respective files. The flowchart in “Figure 3” explains the working of first phase. The block sequence is identified using file descriptors such as starting address and end block address. Loop time and period are used for finding out the block access pattern.

Phase II: The second phase verifies that whether the block is revisited or not by making use of signature based approach as depicted by “Figure 4”. It keeps a record of how many unique blocks each PC has accessed and how many references each PC has issued to access blocks that have been revisited previously in a program counter table. “Figure.5” explains the process of updating the PC table.

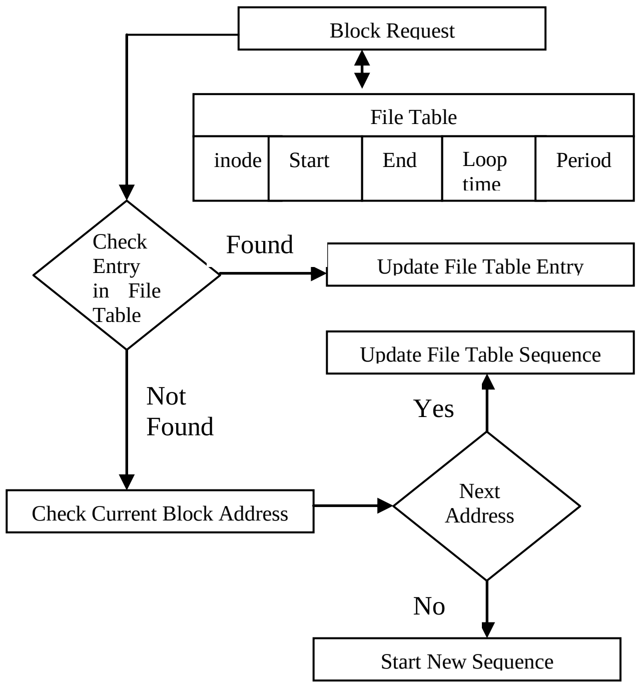


Figure 3. Phase I Update File Table.

Third Phase: Pattern detection is the task of third phase based on the results of file and program counter table. Looping pattern is returned if the file table returns that block is revisited. PC table is referred if file table fails to find the corresponding block entry. The values of fresh and reuse counter are used for determining the access pattern of revisited block.

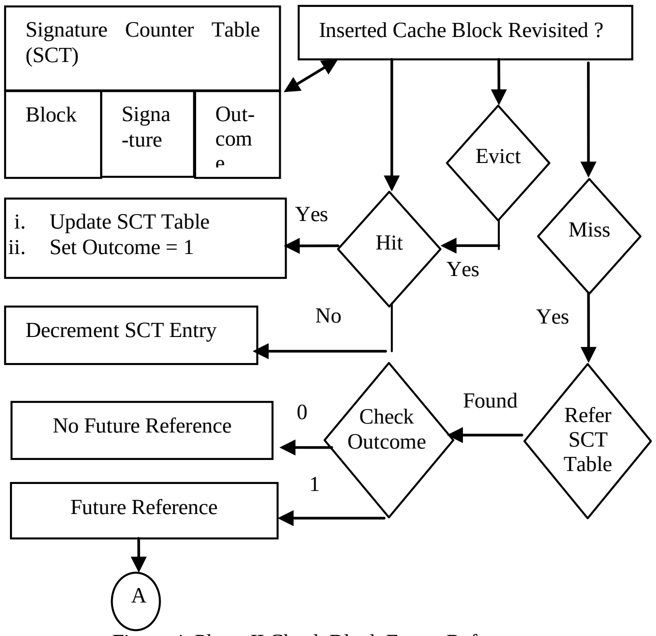


Figure 4. Phase II Check Block Future Reference.

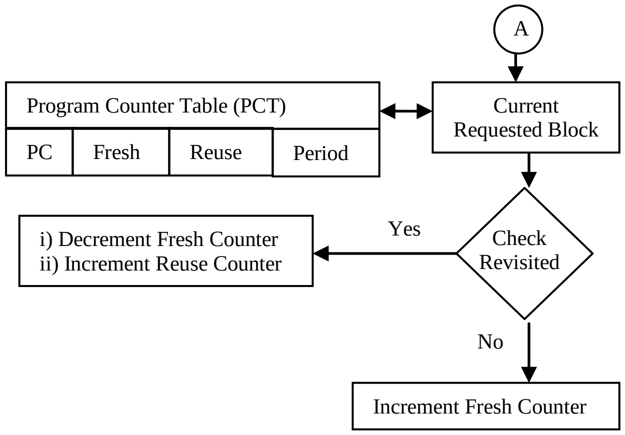


Figure 5. Phase III Update PC Table.

6. EXPECTED RESULT

The proposed algorithm would endeavor to satisfy the properties of accuracy, responsiveness and stability to the best of its ability. And by an attempt of precisely estimating the sequential, looping, mixed reference patterns the proposed algorithm would efficiently utilize the strength of data locality, decisive in determining the block to be replaced. The proposed algorithm is expected to improve the cache hit ration on an average of 10% to 15% compared to LRU.

7. CONCLUSION

Patterns detection at file level and program context level would enhance the performance of proposed algorithm compared to the receny based algorithm. By exploiting the reference regularities, such as sequential and looping references, in the block access pattern buffer cache performance shall be enhanced thereby significantly improving the application response time.

REFERENCES

[1] R. W. Carr and J. L. Hennessy, "WSCLOCK - a simple and effective algorithm for virtual memory management," in Proceedings of the eighth ACM symposium on Operating systems principles (SOSP). New York, NY, USA: ACM Press, 1981, pp. 87–95.
[2] M. J. Bach, The design of the UNIX operating system. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.
[3] A. S. Tanenbaum and A. S. Woodhull, Operating Systems Design and Implementation. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987.
[4] A. J. Smith, "Analysis of the optimal, look-ahead demand paging algorithms," vol. 5, no. 4, pp. 743–757, Dec. 1976.
[5] A.Jabeel, K.B. Theobald,"High performance cache replacement using reference interval prediction",in ISCA 2010.
[6] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," in Proceedings of the 1993 ACMSIGMOD international conference on Management of data. NewYork, NY, USA: ACM Press, 1993, pp. 297–306.
[7] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in Proceedings of the 20th International Conference on Very Large Data Bases (VLDB). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 439–450.
[8] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim,"LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," IEEE Transactions on Computer, vol. 50, no. 12, pp. 1352–1361, 2001.
[9] J. Choi, S. H. Noh, S. L. Min, and Y. Cho, "An implementation study of a detection-based adaptive block replacement scheme," in Proceedings of the 1999 USENIX Annual Technical Conference, Jun. 1999, pp. 239–252.
[10] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST), Mar. 2003, pp. 115–130.
[11] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," pp. 187–200, Mar. 2004.
[12] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling," ACM Transactions on Computer Systems, vol. 14, no. 4, pp. 311–343, 1996.
[13] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka,"Informed prefetching and caching," in Proceedings of the fifteenth ACM symposium on Operating

systems principles (SOSP). NewYork,NY, USA: ACM Press, 1995, pp. 79–95.

[14] A. D. Brown, T. C. Mowry, and O. Krieger. Compiler-based I/O prefetching for out-of-core applications. ACM TOCS, 19(2):111–170, 2001.

[15] T. M. Madhyastha and D. A. Reed, “Learning to classify parallel input/output access patterns,” IEEE Trans. Parallel Distrib. Syst., vol. 3,no. 8, pp. 802–813, 2002.

[16] J. Choi, S. H. Noh, S. L. Min, and Y. Cho, “An implementation study of a detection-based adaptive block replacement scheme,” in Proceedings of the 1999 USENIX Annual Technical Conference, Jun. 1999, pp. 239–252.

[17] J. Choi, S. H. Noh, S. L. Min, E.-Y. Ha, and Y. Cho, “Design, implementation, and performance evaluation of a detection-based adaptive block replacement scheme,” IEEE Trans. Comput., vol. 51, no. 7, pp.793–800, 2002.

[18] C. Gniady, A. R. Butt, and Y. C. Hu, “Program-counter-based pattern classification in buffer caching.” in Proceedings of 6th Symposium on Operating System Design and Implementation (OSDI), Dec. 2006, pp.395–408.

[19] F. Zhou, R. von Behren, and E. Brewer, “AMP: Program context specific buffer caching,” in Proceedings of the USENIX Technical Conference, Apr. 2005.

[20] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. In IEEE Trans. Comput., volume 57, April 2008.

[21] C.-J. Wu and M. Martonosi. Adaptive timekeeping replacement: Fine-grained capacity management for shared CMP caches. In ACM Trans. Archit. Code Optim., volume 8, February 2011.

[22] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In Proc. of the 38th International Symposium on Computer Architecture, 2010.

[23] H. Gao and C. Wilkerson. A dueling segmented LRU replacement algorithm with adaptive bypassing. In Proc. Of the 1st JILP Workshop on Computer Architecture Competitions, 2010.

[24] S. M. Khan, D. A. Jiménez, D. Burger, and B. Falsafi. Using dead blocks as a virtual victim cache. In Proc. of the 19th International Conference on Parallel Architecture and Compilation Techniques, 2010.

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.