Cache Access Pattern Based Algorithm for Performance Improvement of Cache Memory Management

REETU GUPTA[1] , URMILA SHRAWANKAR[2]
[1]Department of Computer Science and Engineering
Priyadarshini Indira Gandhi College of Engineering,
Nagpur (MS) INDIA
guptareetu.rs@gmail.com

[2]Department of Computer Science and Engineering
G.H. Raisoni College of Engineering,
Nagpur (MS) INDIA
urmila@ieee.org

Abstract: - Changes in cache size or architecture are the methods used to improve the cache performance. Use of a single policy cannot adapt to changing workloads. Non detection based policies cannot utilize the reference regularities and suffer from cache pollution and thrashing. Cache Access Pattern (CAP), is a policy that detects patterns, at the file and program context level, in the references issued to the buffer cache blocks. Precise identification is achieved as frequently and repeatedly access patterns are distinguished through the use of reference recency. The cache is partitioned, where each sub-cache holds the blocks for an identified pattern. Once-identified pattern is not stored, repeatedly identified patterns is managed by MRU, frequently identified and unidentified patterns are managed by ARC. Future block reference is identified from the detected patterns. This improves the hit ratio, which in turn reduces the time spent in I/O and overall execution.

Key-Words: -access pattern, program counter, reference recency, reuse distance, buffer cache, reference regularities, and replacement policies

1 Introduction

Buffer Caches are created in main memory and managed by the operating system to avoid the latencies associated with the accesses made by the system call to the secondary storage devices. There are various types of applications that exhibit different reference patterns. Access behavior is concerned with buffer hit ratio, one of the factors affecting the system performance. Non detection based buffer cache management schemes cannot utilize the knowledge exhibited in the request patterns and suffers from the problem of thrashing and cache pollution [1]-[2] Cache Access Pattern (CAP) based algorithms effectively and efficiently utilizes the available buffer cache space by identifying the patterns, exhibited in the accesses made by the buffer cache blocks.

Information contained in the I/O access patterns helps in understanding the application behavior. Single replacement policy employed for managing the cache cannot adapt efficiently to the changing workloads within applications, leading to degraded system performance [3]-[6].

Thus there exist an opportunity for applying multiple replacement policies, for managing the buffer cache. Now the choice of selecting the policy to be applied from multiple options is based upon the patterns observed in the applications. CAP exploits the patterns in the accesses made by the application as well as in the individual files that are accessed in the application. Pattern identification is achieved through the use of program counters. This leads to precise identification of locality strength, a crucial factor in determining the block to be replaced upon a cache miss.

CAP maintains a separate sub-cache partition for each of the identified patterns. The key idea behind partitioning the buffer cache is to increase the hit ratio of individual sub-cache partition. This is accomplished by selecting the policy to be applied based upon the patterns held by the sub-cache partition. Block allocation among the various partitions is managed by a dynamic cache management technique.

The paper is organized as follows; section 2 gives the overview of access pattern based techniques. CAP pattern detection algorithm, with its phases is explained in section 3. Section 4 deals with the experimental results and discussion. Paper concludes with the findings in section 6.

2 Overview of Pattern Based Techniques

The access patterns are utilized and identified by the techniques mentioned in Table 1 for enhancing the performances.

3 Cache Access Pattern (CAP) Design

Section 3.1 illustrates the components of CAP. Types of patterns detected are described in section 3.2. Section 3.3 gives the algorithm. The phases of the algorithm are discussed in detail in section 3.4.

Table 1. Overview of Access Pattern Based Techniques

Sr. No.	Technique	Use of Access Patterns
1	SHiP [1],	Associates the re-reference behavior with memory region, program counter and reference history
2	RACE [2]	Identifies patterns at the file and program context level.
3	Program Instruction [3]	Reuse distance predicted by program instruction.
4	Hybrid Storage [4],[5]	Random writes are predicted by access patterns on cache insertion and hits. [4] ,Block sharing, block access pattern and working set size at application level are identified.
6	Buffer Pre-fetching [6]	Pre-fetching is related with user access behaviour.
7	Access Frequency [7]	Access pattern were identified based on the frequency of user request
8	RRIP[8]	Cache pollution and thrashing was managed by using reference interval
9	Evicted Address Filter [9]	Filter based approach is used to distinguish between high and low reuse cache block.
10	Protection Distance [10]	Cache eviction was associated with reuse distance
12	UBM [14],[15]	Identified the access behavior in accesses made to individual files.
13	PCC [16],AMP [17]	Predicted the access behavior using program counter.
14	Data Locality[18] ,[19]	Data access locality was determined by using the complier based hints [18], software generated self test [19] approach.
15	Reuse Distance [20] – [21]	The reuse behavior of the cache block was determined using the program counter [20], by analyzing the loop transformation [20].

3.1 Components of CAP

CAP is composed of three components first the pattern detector, second the cache manager and third is replacement policy selector.

3.1.1 Pattern Detector

It identifies the patterns using the CAP detection algorithm.

3.1.2 Cache Manger

Cache manager dynamically adjust the size of each sub-cache partitions and manages the allocation of blocks among the partitions.

3.1.3 Policy Selector

Based on the identified pattern which replacement policy is to be applied from multiple available policies is decided by the policy selector. Most Recently Used (MRU) Adapative Replacement Cache (ARC) are the policies used by CAP for identified patterns.

3.2 Types of Pattern Identified

CAP identifies four types of pattern mentioned below.

3.2.1 Once Identified:

Blocks referenced only once and never visited again are classified under once identified pattern.

3.2.2 Repeatedly Identified:

Blocks those are being referenced after regular interval come under this category. These are the least referenced blocks that are likely to be accessed in near future.

3.2.3 Frequently Identified:

Blocks under this category are the ones that are referenced most recently and are likely to be referenced in the near future.

3.2.4 Un-identified:

The blocks that do not exhibit the above behavior are classified as un-identified.

3.3 CAP Pattern Detection Algorithm

Detail working of CAP based pattern detection algorithm is given below. Patterns are identified using file inode, program counter, block number and current referenced time. The algorithm makes use of three data structures Recency Table (RT), File Hash table (FHT), PC Hash Table (PCHT).

1. CAP (inode, blk, pc, currtime)
2. Begin
3. If (pc is not in RT) then
4. {pc = pc, totalblks = 1, validreceny = 0.0, avgrecency = -1.00,inode = inode,blk = blk}
5. Else if ((pc is in RT) & (inode & blk is not in RT)) then
6. totalblks = increment by 1,
7. Else if ((pc present in RT) && (inode and blk present in RT)) then
8. {if (totalblks > 1) then
9. {validreceny = +1,index=0,
10. do{ If(inode & blk at first place)then
11. {avgreceny=index/(totalblks -1)
12. Move inode,blk to end position found =1}
13. Else
14. index = + 1, found = 0}
15. } until (found == 1),}
16. Else {validreceny = increment by 1, avgrecency = 0.5,
Move ionde,blk to end position} }
17. End if
18. If (pc is not in PCHT) then
19. pc = pc, fresh = reuse = period = 0,
20. If (inode is not in FHT) then
21. inode = inode, start=end=blk, fresh = +1
22. Else if (inode in FHT)&(blk in inode seq)then
23. {reuse = +1, fresh = -1,
Pattern: Repeatedly Identified, from: FT
24. Goto Step 36}
25. Else if (inode in FHT) & (blk is next address in inode seq)) then
26. {end = blk, fresh = +1}
27. End If
28. If (reuse >= fresh) then
29. {Pattern: Repeatedly Identified, from : PCHT Goto Step 36}
30. Else if (fresh >= Sequential Threshold) then
31. Pattern: Once Identified, from: PCHT
32. Else Pattern: Un-Identified
33. End If
34. If (avgRecency for PC in RT table > -1)
35. {If (avgRecency <= Repeated Threshold)
36. Pattern: Frequently Identified
37. Else
38. Pattern = Repeatedly Identified}
39. End If
40. End

3.4 Phases of CAP

The algorithm works in four phases, first phase deals with the recency table; second and third phase updates the file and pc tables. Based on the search results of three tables, pattern is predicted in the fourth phase.

3.4.1 Phase I: Update Recency Table (RT)

The structure of the recency table is shown in Table 2. The flow chart for maintaining the rececncy table is shown in Fig 1. Basic processing step of this phase are:-

- **Receive Request:** Block request is processed by CAP by using the attributes (inode, block number, PC, current time).
- **Check PC, inode and block:** Requested block and inode are checked in recency table. If found the fields are updates else a new entry is made.CAP is composed of three components first the pattern detector, second the cache manager and third is replacement policy selector.

3.4.2 Phase II: Update File Hash Table (FHT)

File hash table maintains the record of each block reference. Its structure is shown in Table 3 and the process of maintain the table is explained in flowchart of Fig 2. The conditions checked on occurrence of block in file table are:-

- **Found:** If the referenced block occurs in the file table its access pattern is predicted in the manner described in the pattern detection phase.
- **Not Found:** If the reference block dose not exists in a sequence, then block address is verified. If it is next address the existing sequence is extended. Otherwise a new sequence is entered in the file table.

3.4.3 Phase III: Update PC Hash Table (PCHT)

The program counter associated with the block being referenced is tested for the following conditions:-

Check PC: For each block request, the programs counter (pc) accessing the block is checked in the PC table with the following conditions.

Table 2. Recency Table Structure

Rececny Table(RT)				
PC	TotalBlk	ValidRecency	AvgRecency	Blk(inode, blk)

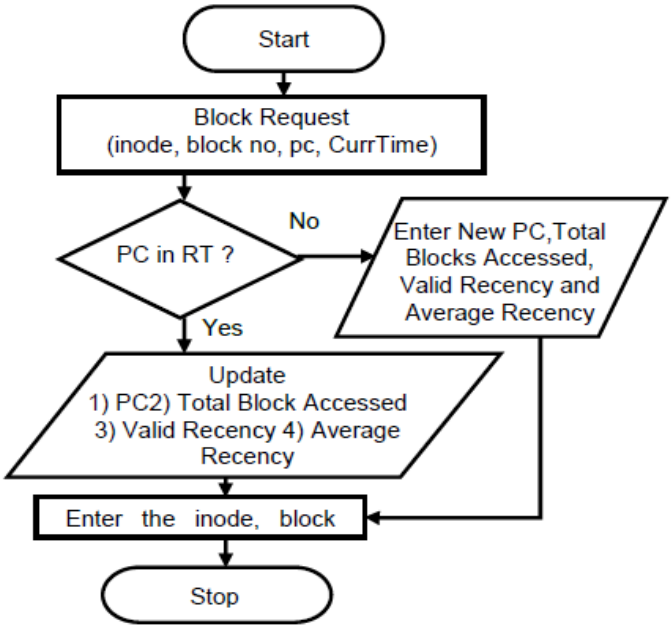

Figure 1. Maintaining Recency Table

Table 3. File Hash Table Structure

File Hash Table(FHT)		
Inode	Start	End

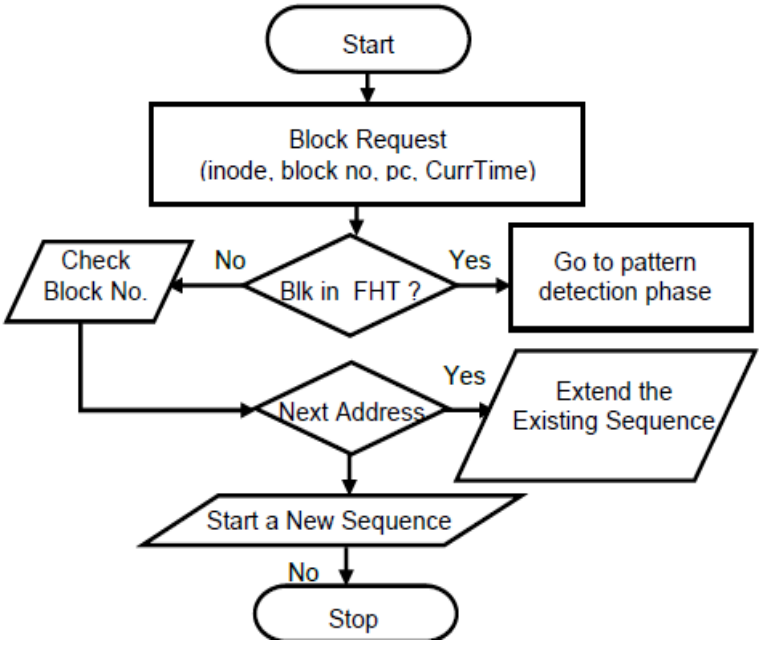

Figure 2. Maintaining File Hash Table

- **Found:** If the pc is found in PC table, block referenced by it is checked in the data structure table.
  - Present in FT: If file table contains the referenced block it means it is revisited by the program counter. Hence the fresh field is decremented, reuse field is incremented.
- **Not Present in FT:** If file table reports that block is not revisited before, it indicates that the block is referenced by the pc for the first time. Hence fresh field is incremented **Not Found:** If program counter is not found in the pc table then a new entry is made and fresh in incremented by 1 and reuse is set to zero.

3.4.4 Phase IV: Predicting Pattern

Block pattern is predicted by analyzing the entries made in the data file table, program counter table and recency table.

Check in FT: The requested block is first searched in the sequences of file hash table to verify if it revisited.

- If revisited, then reference is classified as repeatedly identified. Then reference recency is checked.

Check in PCHT: If not found in the file table, it is searched in the PC table. If block exist fresh and reuse fields are compared.

- If reuse field is greater than fresh counter then repeatedly identified pattern is returned by the PC table and the reference recency is checked.
- Otherwise if fresh field is found to greater than the once-identified threshold is checked. If threshold is crossed block reference is categorized as once identified, otherwise as un-identified.

Check Recency Table: When reference is categorized as repeatedly identified recency table is referred. If the average receny is found to be greater than or equal to the threshold set for frequently identified, then the block reference is said to exhibit frequently identified pattern.

The pattern identification phase is explained in two parts, A and B. Part A in Fig. 4, detects, repeatedly identified and un-identified pattern. Frequently identified pattern are determined once-identified from reference recency as in part B, shown in Fig 5.

Table 4. PC Hash Table Structure

PC Hash Table(PCHT)		
Pc	Fresh	Reuse

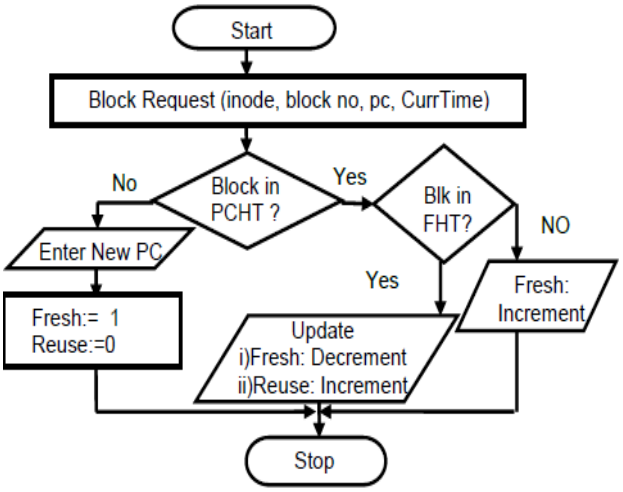

Figure 3. Maintaining PC Hash Table

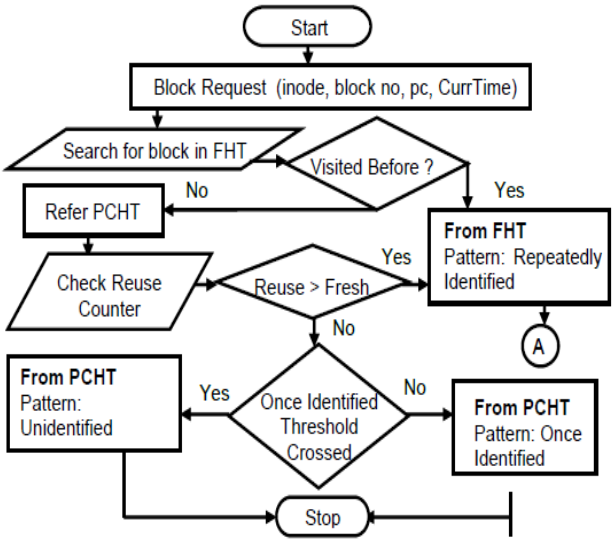

Figure 4. Predicting Pattern – Part A

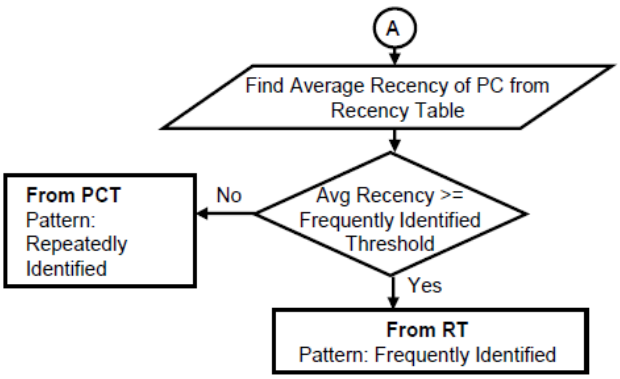

Figure 5. Predicting Pattern – Part B

4 Experimental Results

The buffer cache environment is created using Accusim [22] simulator. Modified Linux strace utility intercepts the system calls made by the processes and records the following information about the, file identifier (inode), request size, I/O triggering program counter, starting address. The program counters are obtained by tracing the function call stack backwards in the trace file.

4.1 Test Results

The applications used for evaluating the performance of CAP are listed below in Table 4. Table 5 lists the configuration when the applications were run concurrently. Details about total number of I/O references made by the applications, the total size of the application in MB whose traces are obtained by using modifying linux strace utility are mentioned in Table 4 and 5.

CAP has been compared with RACE pattern detection policy. The difference between the types of pattern identified and the replacement policy used is shown in Table 7.

Application	Number of References	Data Size
Cscope	1119161	260 MB
Viewperf	303123	495 MB
gcc	158667	41 MB

Table 6. Concurrent Application Statistics

Application	Number of References	Data Size [MB]
Combo1	gcc + cscope	300
Combo2	Cscope + gcc+ viewperf	755

Table 7. Comparison of Patterns and Policies

CAP		RACE	
Patterns Identified	**Policy Used**	**Patterns Identified**	**Policy Used**
Once -	Not	Sequential	LRU
Repeatedly	MRU	Looping	LRU/MR
Unidentified	ARC	Others	MRU
Frequently	ARC	------	-------

4.2. Hit Ratio Comparison

The future access request of the block is predicted by using the information contained in the I/O request. Thus when the block request arises the referenced block is present in the buffer cache. Thus buffer cache misses are reduced. The comparison of various application and hit ratio is discussed in the following section. CAP hit ratio is compared with RACE and other non detection based algorithms.

- **Cscope:** Table 8 shows the comparison of cscope hit ratio with CAP and various algorithms. And the comparison graph of hit ratio is shown in Fig 6. From the graph it can be seen that hit ratio is maximum with CAP compared to other algorithms.
- **Viewperf:** Table 9 shows the comparison of viewperf hit ratio with CAP and various algorithms. And the comparison graph of hit ratio is shown in Fig 7. From the graph it can be seen that hit ratio is maximum with CAP compared to other algorithms.
- **Combo 1:** Cscope and gcc is run in combination for identifying the access patterns. Table 10 shows the comparison of Combo1 hit ratio with CAP and various algorithm. And the comparison graph of hit ratio is shown in Fig 8. From the graph it can be seen that hit ratio is maximum with CAP compared to other algorithms.
- **Combo 2:** Combined application Cscope, Viewperf and gcc is run in combination for identifying the access patterns. Table 11 shows the comparison of Combo2 hit ratio with CAP and various algorithm. And the comparison graph of hit ratio is shown in Fig 9. From the graph it can be seen that hit ratio is maximum with CAP compared to other algorithms.

Table 8. Cscope Hit Ratio Comparison

Cache Size (MB)	RACE %	CAP %	ARC %	LRU %	LIRS %	MQ %	2Q %
8	44	54	43	43	44	43	44
16	45	54	44	43	44	43	44
32	48	54	45	44	45	44	44
64	50	54	46	45	45	45	45
128	59	97	46	46	77	54	75
256	87	97	97	97	97	97	97

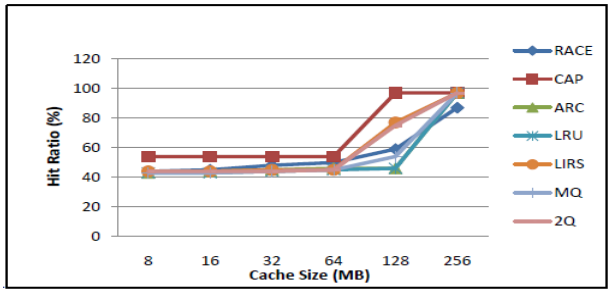

Fig 6. Cscope Hit Ratio

Table 9. Viewperf Hit Ratio Comparison

Cache Size (MB)	RACE %	CAP %	ARC %	LRU %	LIRS %	MQ %	2Q %
8	74	89	86	66	86	84	86
16	70	89	87	68	87	88	87
32	51	93	88	88	88	88	90
64	66	93	92	53	91	93	92
128	80	93	93	73	91	93	93
256	90	93	93	83	93	93	93

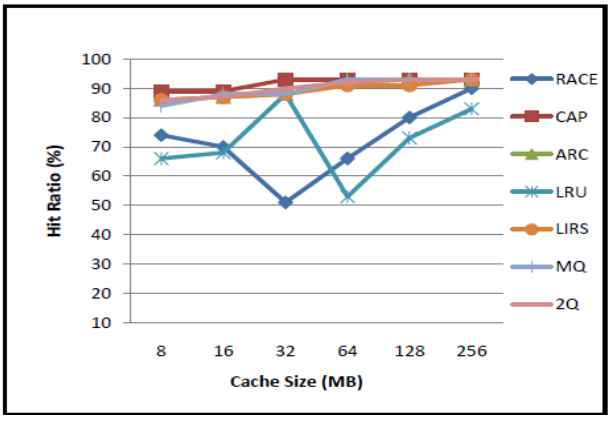

Fig 7. Viewperf Hit Ratio

Table 10. Combo1 Hit Ratio Comparison

Cache Size (MB)	RACE %	CAP %	ARC %	LRU %	LIRS %	MQ %	2Q %
8	53	59	48	47	50	47	50
16	55	60	51	48	50	48	50
32	56	60	51	49	51	49	51
64	52	60	52	51	52	51	52
128	57	96	53	52	79	59	77
256	82	96	96	96	96	96	96

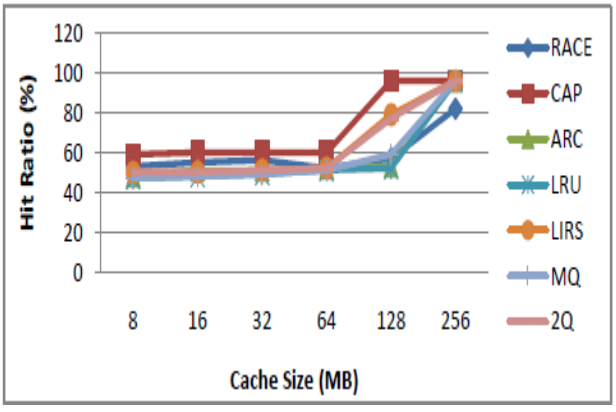

Fig 8. Combo1 Hit Ratio

Table 10. Combo2 Hit Ratio Comparison

Cache Size (MB)	RACE %	CAP %	ARC %	LRU %	LIRS %	MQ %	2Q %
8	62	65	53	54	56	54	56
16	53	65	57	55	58	55	57
32	64	66	58	57	58	56	58
64	62	66	60	59	59	59	59
128	70	81	61	60	81	80	53
256	74	96	96	66	95	96	96

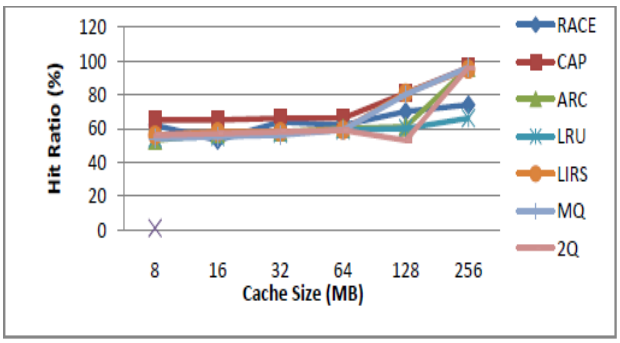

Fig 9. Combo 2 Hit Ratio

4.3. Total Execution Time Comparison

As the future access request is predicted by CAP the requested block is available in the buffer cache. As it is the main memory to main memory transfer the time spent in the I/O is reduced. The total time spent in the execution is minimum compared to the RACE and other non detection based scheme. The comparison with different application under varying cache size is shown below.

- **Cscope:** Table 11 shows the comparison of cscope execution time with CAP and various algorithms. And the comparison graph of of total execution time is shown in Fig 10. From the graph it can be seen that total execution time is minimum with CAP compared to other algorithms.

- **Viewperf:** Table 12 shows the comparison of total execution time with CAP and various algorithm. And the comparison graph of total execution time is shown in Fig 11. From the graph it can be seen that execution time is minimum with CAP compared to other algorithms.

- **Combo 1:** Cscope and gcc is run in combination for identifying the access patterns. Table 13 shows the comparison of Combo1 total

execution time with CAP and various algorithms. And the comparison graph of execution time is shown in Fig 12. From the graph it can be seen that execution time is minimum with CAP compared to other algorithms.

- **Combo 2:** Combined application Cscope, Viewperf and gcc is run in combination for identifying the access patterns. Table 14 shows the comparison of Combo2 total execution time with CAP and various algorithms. And the comparison graph of execution time is shown in Fig 13. From the graph it can be seen that execution is minimum with CAP compared to other algorithms

Table 11. Cscope Total Execution time Comparison

Cache Size (MB)	RACE %	CAP %	ARC%	LRU %	LIRS %	MQ %	2Q %
8	2188.94	1807.07	2245.43	2318.13	2217.33	2307.98	2242.77
16	2282.77	1798.74	2213.38	2296.01	2204.57	2289.5	223786
32	2459.04	1791.31	2195.12	2256.87	2180.16	2262.09	2219.95
64	2002.05	1781.72	2154.01	2209.33	2151.28	2209.43	2161 .86
128	1741.3	272.26	2096.03	2158.06	106716	1852.61	1068 .31
256	830.15	272.26	272 .28	272 .28	272.26	272.28	272 .5

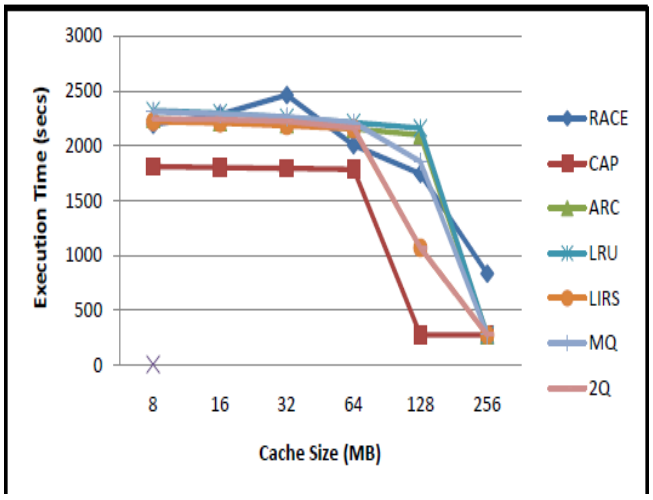

Fig 10. Cscope Total Execution Time

Table 12. Viewperf Total Execution time Comparison

Cache Size (MB)	RACE %	CAP %	ARC%	LRU %	LIRS%	MQ %	2Q %
8	842 .51	79 .74	912 .02	863 .77	861 .09	868 .77	864 .28
16	823.92	795 .75	844 .94	842. 79	842 .09	840 .11	845 .58
32	813.12	784.18	830.1	830 .02	839 .43	824 .52	825 .1
64	926.23	784 .18	805.69	797 .72	820 .7	800 .73	801 .13
128	788.09	784 .18	785.32	784 .28	809 .81	784 .28	784 .63
256	788.4	784 .18	784 .21	784 .21	784 .24	784 .21	784 .64

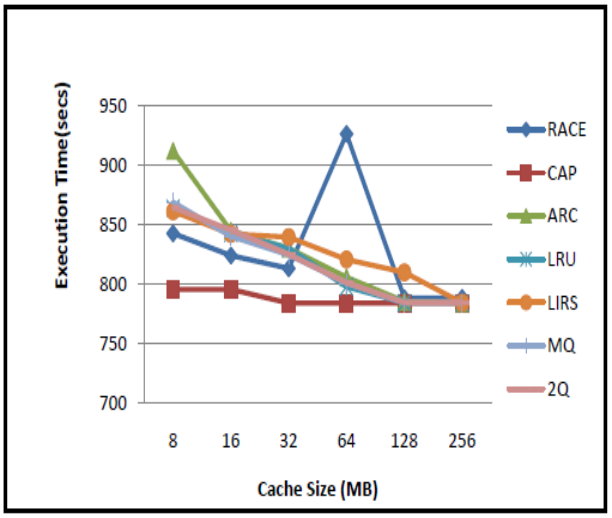

Fig 11. Viewperf Total Execution Time

Table 13. Combo1 Total Execution time Comparison

Cache Size (MB)	RACE %	CAP %	ARC%	LRU %	LIRS %	MQ %	2Q %
8	2766.44	1691.65	2526.7	2777.44	2503.47	2769.93	2166 .68
16	2661.96	2019.5	2457.9	2695.28	2446.82	2687.69	2477 .63
32	2423.77	2005.03	2423.56	2561.37	2406.12	2569.16	2448 .73
64	2566.19	1990.2	2371.47	2476.43	2369.77	2479.74	2393 .07
128	1339.05	494.19	2315.67	2409.16	1311.65	2109.95	1334 .2
256	446.25	494.19	494.29	494.3	494.31	494.3	495 .52

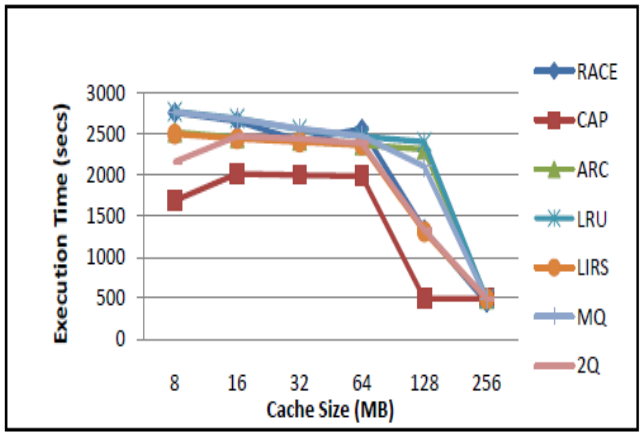

Fig 12. Combo1 Total Execution Time

Table 14. Combo2 Total Execution time Comparison

Cache Size (MB)	RACE %	CAP %	ARC%	LRU %	LIRS %	MQ %	2Q %
8	3463.15	2720.31	3334.17	3563.1	3266.72	3558.02	3282.37
16	3360.52	2697.91	3190.76	3466.49	3176.33	3459.02	3223.82
32	3312.56	2673.04	3147.11	3315.46	3138.95	3321.07	3170.58
64	3308.45	2659.98	3073.52	3182.6	3082.31	3188.79	3085.41
128	1151.15	3005.26	3005.26	3094.42	2028.16	2108.49	3515.46
256	1125.98	1151.15	1155.46	1156.1	1190.06	1156.1	1156.15

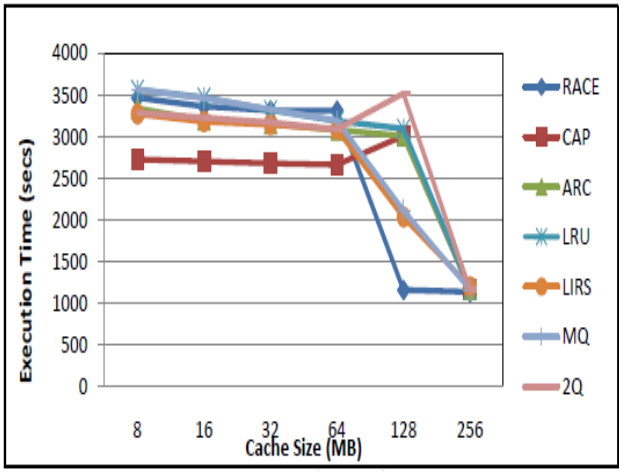

Fig 13. Combo2 Total Execution Time

4.4. I/O Time Comparison

The hit ratio of the application is improved by CAP as the miss are reduced. As block is available in the buffer cache, when the request arises the time spent in the I/O is reduced. The time required by CAP for performing the I/O with various applications under varying cache sizes are shown below.

- **Cscope:** Table 15 shows the comparison of cscope I/O time with CAP and various algorithms. And the comparison graph of I/O time is shown in Fig 14. From the graph it can be seen that I/O is minimum with CAP compared to other algorithms.

- **Viewperf:** Table 16 shows the comparison of I/O time with CAP and various algorithm. And the comparison graph of I/O time is shown in Fig 15. From the graph it can be seen that I/O is minimum with CAP compared to other algorithms.

- **Combo 1:** Cscope and gcc is run in combination for identifying the access patterns. Table 17 shows the comparison of Combo1 I/O time with CAP and various algorithms. And the comparison graph of I/O time is shown in Fig 16. From the graph it can be seen that I/O is minimum with CAP compared to other algorithms.

- **Combo 2:** Combined application Cscope, Viewperf and gcc is run in combination for identifying the access patterns. Table 18 shows the comparison of Combo2 I/O time with CAP and various algorithms. And the comparison graph of I/O time is shown in Fig 17. From the graph it can be seen that I/O is minimum with CAP compared to other algorithms.

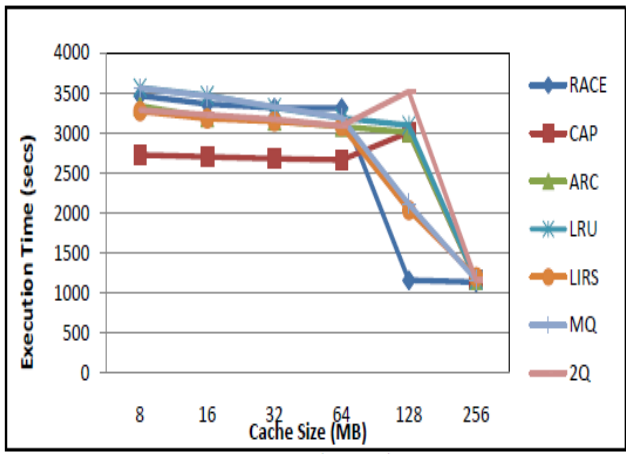

Fig 14. Cscope I/O Time Comparison

Table 16. Viewperf I/O time Comparison

Cache Size (MB)	RACE %	CAP %	ARC%	LRU %	LIRS %	MQ %	2Q %
8	152.96	101.19	217.47	169.22	166.55	174.23	169.74
16	137.37	101.2	150.4	148.24	147.55	145.56	151.03
32	126.46	89.64	135.55	135.47	144.89	129.97	130.56
64	101.68	89.64	111.14	103.18	126.15	394.54	106.58
128	85.54	89.64	90.78	89.73	115.27	89.73	90.08
256	85.85	89.64	89.67	89.66	89.7	89.66	90.101

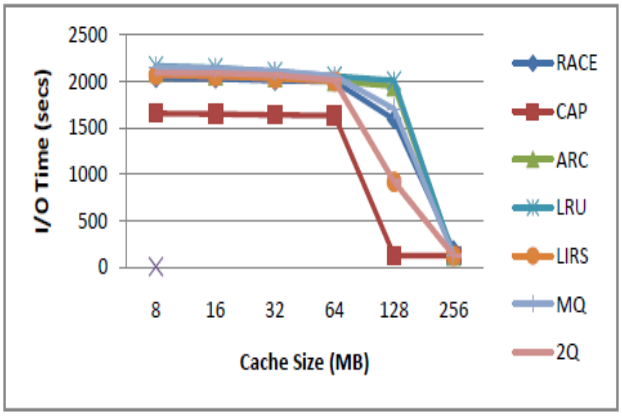

Fig 15. Viewperf I/O Time

Table 17. Combo1 I/O time Comparison

Cache Size (MB)	RACE %	CAP %	ARC%	LRU %	LIRS %	MQ %	2Q %
8	2224.63	2033.47	2184.88	2435.62	2161.26	2428.11	2508.49
16	2320.15	1677.68	2116.08	2353.47	2105.01	2345.87	2135.81
32	2181.95	1663.21	2081.74	2219.56	2064.3	2227.34	2106.91
64	2024.37	1648.38	2029.66	2134.61	2027.96	2137.93	2051.26
128	2207.24	152.38	1973.85	2067.34	969.83	1768.14	992.39
256	152.43	152.38	152.48	152.49	152.49	152.49	153.7

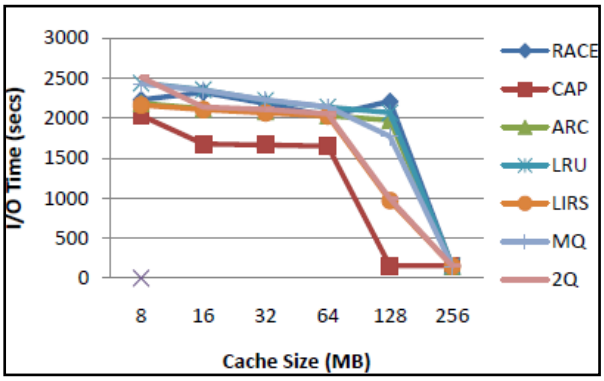

Fig 16. Combo1 I/O Time

Table 18. Combo2 I/O time Comparison

Cache Size (MB)	RACE %	CAP %	ARC%	LRU %	LIRS %	MQ %	2Q %
8	2247	1805	2418	2647	2351	2642	2367
16	2245	1782	2275	2551	2261	2543	2308
32	2317	1757	2232	2400	2223	2405	2255
64	2233	1744	2158	2267	2167	2273	2170
128	235 .4	235 .4	2089	2179	1112	1193	2600
256	238.2	235.4	239.7	240.3	274.3	240.3	240.4

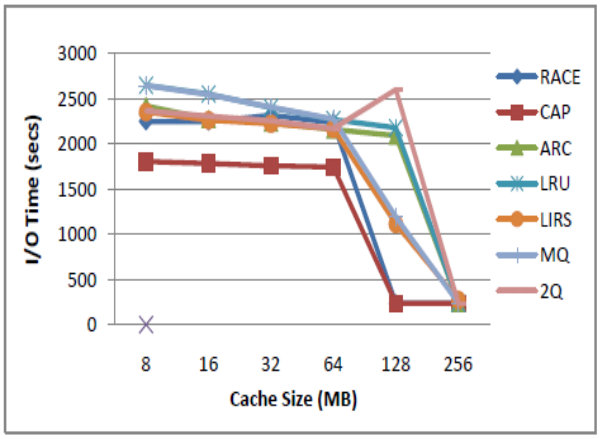

Fig 17. Combo2 I/O Time

4.5. Reference Recency

Reference recency is used to distinguish between frequently and repeatedly identified access patterns. An example of repeatedly and frequently identified pattern is shown below

- **Repeatedly Identified Pattern:**

Consider a repeatedly identified sequence as {0 1 2 3 4 1 2 3 4 }.From the calculation it can be seen that repeatedly identified always have their recencies 0. Table 19 shows the reference and average recency calculation for the sequence. The reference recency is calculated using the formula in column heading 4 where list of previously accessed blocks, p is block position and i is the ith access of the sequence.

- **Frequently Identified Pattern:**

Consider a temporarily clustered sequence from

(9021- 9032) for cscope at instance (65461, 9021, 31581990, 382). RACE classifies it as repeatedly identified. Through reference recency CAP correctly classifies it as frequently identified. The recency of frequently identified pattern is found to be greater than 0.4

Table 19. Repeatedly Identified Pattern Recency

i	blo c k	Li	Pi=(Pi /	Ri
1	9021	Empty	∞	∞
2	9022	9021	∞	∞
3	9022	9021, 9022	1	1
4	9023	9021, 9022	∞	∞
5	9026	9021, 9022, 9023	∞	∞
6	9027	9021, 9022, 9023, 9026	∞	∞
7	9028	9021, 9022, 9023, 9026, 9027	∞	∞
8	9029	9021, 9022, 9023, 9026, 9027, 9028	∞	∞
9	9030	9021, 9022, 9023, 9026, 9027, 9028, 9029	∞	∞
10	9031	9021, 9022, 9023, 9026, 9027, 9028, 9029,	∞	∞
11	9032	9021, 9022, 9023, 9026, 9027, 9028, 9029, 9030,	∞	∞
12	9023	9021, 9022, 9023, 9026, 9027, 9028, 9029, 9030,	0.22	0.6

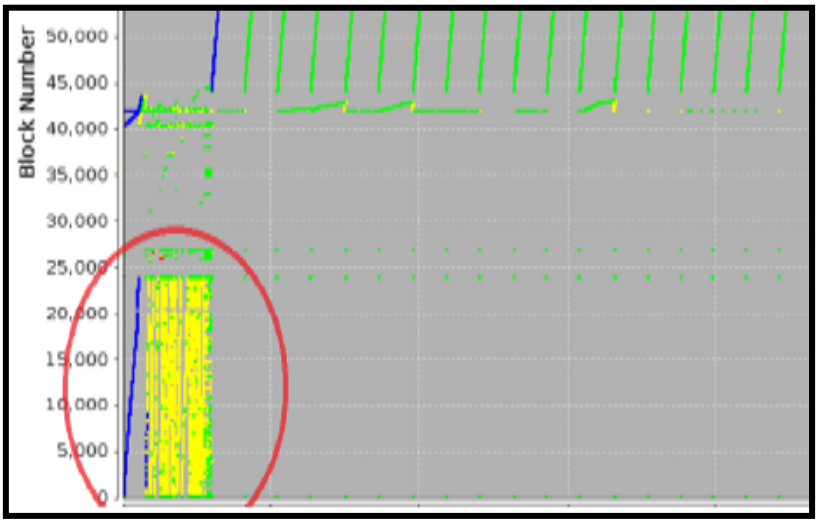

Fig 18. Frequently Identified Pattern by CAP in Yellow Color

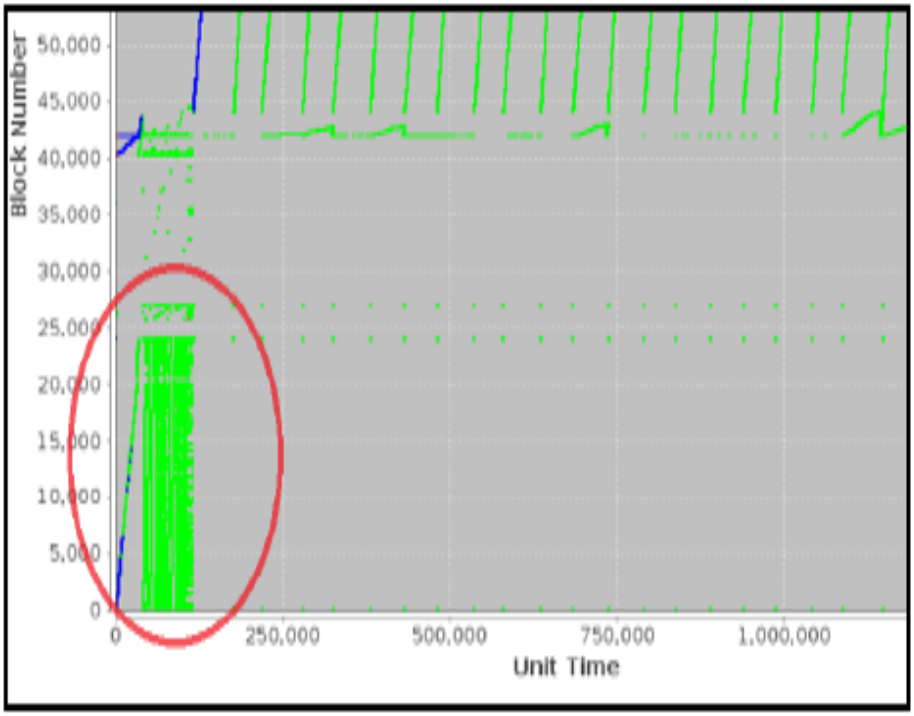

Fig 19. Misclassified as Repeatedly Identified Pattern by RACE in Green Color

5 Conclusions

The CAP pattern detection algorithm precisely identifies the pattern at the file and program context level through the use of program counter. The types of pattern detected are categorized as once-identified, repeatedly-identified, frequently identified, and unidentified. CAP shows improved performance compared to RACE pattern detection by improving the hit ratio, and reducing the time spent in performing the I/O and overall execution. CAP differs from RACE in the following manner.

- It detects an additional access pattern, frequently identified from the looping references.
- RACE uses MRU sequential, MRU/LRU for looping and LRU for other pattern. CAP does not store once-identified pattern, uses MRU for repeatedly identified ARC for frequently and unidentified pattern.
- RACE made use of a looping period based marginal gain policy for managing the cache partition. CAP made use of a probabilistic approach for managing the partition.

The algorithm automatically identifies the patterns and hence user and programmer are not required to understand the application behavior. Use of program counter helps in correctly identifying the references made to the small file.

As the policies to be applied are based on identified pattern, CAP can better adjust itself to the changing workload. Use of multiple policies helps to increases the hit ratio of each individual sub cache partition which in turn improves the overall hit ratio. This leads to effective and efficient utilization of the available limited buffer cache space.

The knowledge obtained in the I/O access pattern could be utilized to for hybrid structure and mobile storage devices to enhance the performance of on demand multimedia and gaming applications. It could also be used with the network to reduce traffic thereby reducing the latencies encountered in web. By observing the patterns on multi-core data sharing can be enhanced.

References:

[1] Wu, Carole-Jean, et al. "SHiP: Signature-based hit predictor for high performance caching." Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2011.

[2] Yifeng Zhu; Hong Jiang, "RACE: A Robust Adaptive Caching Strategy for Buffer Cache," Computers, IEEE Transactions on , vol.57, no.1, pp.25,40, Jan. 2008 doi:10.1109/TC.2007.70788

[3] Keramidas, G.; Petoumenos, P.; Kaxiras, S., "Cache replacement based on reuse-distance prediction," Computer Design, 2007. ICCD 2007. 25th International Conference on , vol., no., pp.245,250, 7-10 Oct. 2007.

[4] Seung-Ho Park; Jung-Wook Park; Shin-Dug Kim; Weems, C.C., "A Pattern Adaptive NAND Flash Memory Storage Structure," Computers, IEEE Transactions on , vol.61, no.1, pp.134,138, Jan. 2012 doi: 10.1109/TC.2010.212

[5] Miranda, Alberto, and Toni Cortes. "Analyzing Long-Term Access Locality to Find Ways to Improve Distributed Storage Systems." In Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on, pp. 544-553. IEEE, 2012.

[6] Joonho Choi; Reaz, A.; Mukherjee, B., "A Survey of User Behavior in VoD Service and Bandwidth-Saving Multicast Streaming Schemes," Communications Surveys & Tutorials, IEEE , vol.14, no.1, pp.156,169, 2012 doi:10.1109/SURV.2011.030811.00051

[7] Danqi Wang; Chai Kiat Yeo, "Exploring Locality of Reference in P2P VoD Systems," Multimedia, IEEE Transactions on, vol.14, no.4, pp.1309, 1323, Aug.2012

[8] Jaleel, Aamer, Kevin B. Theobald, Simon C. Steely Jr, and Joel Emer. "High performance cache replacement using re-reference interval prediction (RRIP)." In ACM SIGARCH Computer Architecture News, vol. 38, no. 3, pp. 60-71. ACM,2010.

[9] Seshadri, Vivek, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry. "The Evicted-Address Filter: A unified mechanism to address both cache pollution and thrashing." In 21st international conference on Parallel architectures and compilation techniques, pp. 355-366. ACM, 2012.

[10] Duong, Nam, et al. "Improving Cache Management Policies Using Dynamic Reuse Distances." 45th Annual IEEE/ACM International Symposium on Microarchitecture, Pages 389-400, 2012

[11] Gupta, Reetu, and Urmila Shrawankar. "Block Patten Based Buffer Cache Management." 8th International Conference on Computer Science and Education (ICCSE), pp 963-968, April 26-28, 2013

[12] Gupta, Reetu, and Urmila Shrawankar. "Managing Buffer Cache by Block Access Pattern." IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 6, No 2, November 2012

[13] Gupta, Reetu, and Urmila Shrawankar "A Methodology for Buffer Cache Block AccessPattern Based Policy Selection", Engineering and Systems, (SCES),Student Conference on, 12-14,April 2013.

[14] Kim, Jong Min, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. "A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references." Symposium on Operating System Design & Implementation-Volume 4, pp. 9-9. USENIX Association, 2000.

[15] Gniady, Chris, Ali R. Butt, and Y. Charlie Hu. "Program-counter-based pattern classification in buffer caching." Symposium on Opearting Systems Design & Implementation, vol. 6, pp. 395-408. 2004

[16] Zhou, Feng, Rob von Behren, and Eric Brewer. "AMP: Program context specific buffer caching." USENIX Technical Conference. 2005.

[17] Yong Li; Abousamra, A.; Melhem, R.; Jones, A.K., "Compiler-Assisted Data Distribution and Network Configuration for Chip Multiprocessors," Parallel and Distributed Systems, IEEE Transactions on , vol.23, no.11, pp.2058,2066, Nov.2012 doi: 10.1109/TPDS.2011.279

[18] Di Carlo, S.; Prinetto, P.; Savino, A., "Software-Based Self-Test of Set-AssociativeCache Memories," Computers, IEEE Transactions on , vol.60, no.7, pp.1030,1044, July 2011 doi: 10.1109/TC.2010.166

[19] Fensch, C.; Barrow-Williams, N.; Mullins, R.D.; Moore, S., "Designing a PhysicalLocality Aware Coherence Protocol for Chip-Multiprocessors," Computers, IEEE Transactions on , vol.62, no.5, pp.914,928, May 2013

[20] Albericio, Jorge, et al. "Exploiting reuse locality on inclusive shared last-level caches." ACM Transactions on Architecture and Code Optimization (TACO) Volume 9 Issue 4, January 2013

[21] hungsoo Lim; Seong-Ro Lee; Joon-Hyuk Chang, "Efficient implementation of an SVM-based speech/music classifier by enhancing temporal locality in support vector references," Consumer Electronics, IEEE Transactions on , vol.58, no.3, pp.898,904, August 2012 doi: 10.1109/TCE.2012.6311334

[22] Butt, Ali R., Chris Gniady, and Y. Charlie Hu. "The performance impact of kernel prefetching on buffer cache replacement algorithms." In ACM SIGMETRICS Performance Evaluation Review, vol. 33, no. 1, pp. 157-168. ACM, 2005

[23] Urmila Shrawankar, Ashwini Meshram et. al “Pattern Based Real Time Disk Scheduling Algorithms for Virtualized Environment," ICETET, Sixth International Conference on , Dec 2013.