iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 8 **Issue:** VIII **Month of publication:** August 2020

DOI: https://doi.org/10.22214/ijraset.2020.31031

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 8 Issue VIII Aug 2020- Available at www.ijraset.com

Performance Improvement of Existing Cache Replacement Policies

Siddhi R. Kadam[1], R. N. Awale[2]
[1, 2]*Department of Electrical Engineering, VJTI College, Mumbai*

Abstract: ***Modern processors have a clock speed in the range of GHz while the main memory (DRAM) has a read/write speed in the range of MHz, so the processor needs to halt till the memory completes its request. The halt period may seem to be very small, but when seen on a broad scale, we see that most of the processor's time is wasted in the halt cycles. Cache memory is intended to reduce the speed gap between the fast processor and slow memory. When a program needs to access data from the RAM (physical memory), it first checks inside the cache (SRAM). Replacement policies are methods by which the memory blocks are replaced in a filled cache. Cache replacement policies play a significant role in the memory organization of a processor and dictate how fast a processor will receive the block demanded. Various replacement policies such as RRIP, ABRRIP, AIRRIP, etc. have been designed but not been implemented, unlike LRU. LRU is predominantly used in most of the systems. The ABRRIP policy has two levels of RRPV – one at block and one at the core level. We observe that the performance of the ABRRIP policy improves as we increase the number of instructions during the simulation.***

Keywords: LRU, Replacement Policies, RRIP, ABRRIP, IPC, Cache memory

I. INTRODUCTION

Cache memory is designed to reduce the speed gap between the fast processor and slow memory. It is expensive compared to physical memory. When a program needs to access data from the DRAM (physical memory), it first checks inside the cache. A cache hit occurs if the required block is found in the cache, else a cache miss is said to occur [9], [10], [11]. The method by which these blocks are replaced inside the cache is known as Cache Replacement Policies. Today, we have Multicore architecture, and each core has its L1 (Instruction and Data Cache) and L2 (which is combined) [8]. All the cores share the Last Level Cache (LLC). As the number of cores is increasing day-by-day, the burden on LLC is increasing, and an efficient replacement policy needs to be implemented to reduce this burden. Cache memory performance is calculated based on Access latency, Hit ratio, and IPC. The access latency is the amount of time taken by the CPU to access the data or instruction [4], [6]. Hit ratio is the number of hits received by the Cache Memory w.r.t. the request. IPC (Instructions Per Cycle) is the inverse of CPI (Cycles Per Instruction) i.e. the number of cycles taken by the processor to execute an instruction. In a Multicore environment, multiple instructions are executed on different cores, and hence CPI cannot be used as a performance metric. Thus, we use IPC as a performance metric as it defines the number of instructions executed by all cores in a single clock cycle of a processor.

An access pattern is a way in which workload gives memory requests to the system. There are several cache access patterns commonly found in applications used to compare the performance of different replacement policies [1], [5]. Cache-friendly patterns have a near-immediate re-reference interval (example: a typical access pattern that repeats itself – a1 a2 a2 a1 a1 a2). Since the pattern is getting repeated, it receives hits more frequently, and do not pollute the cache with degrading blocks. Thrashing access pattern is a cyclic access pattern of length ‘k’ that repeats ‘N’ times (example: a1 a2 a3 a4 a1 a2 a3 a4). Streaming access patterns have a distant re-reference interval (example: a1 a2 a3 a4 b1 b2 b3 b4 a5 a6 a7 a8 b5 b6). A recency-friendly access pattern is a pattern which is accessed recently. LRU (Least Recently Used) policy evicts the block that has not been used for a long time, the block with distant re-reference interval. If the pattern length is greater than the number of blocks inside the cache (cache line size), it causes Cache Thrashing, and LRU receives no hits unless cache size increases to hold all the entries of the access pattern. LRU always predicts a block with near-immediate re-reference interval on a cache hit or miss. So, the applications with distant re-reference interval (example: Streaming applications), performs badly under LRU. Thus, LRU is not Scan-resistant (Scans are the blocks with distant re-reference intervals).

II. MOTIVATION

Over the years, the clock speed of the processor has increased drastically. Earlier it was 10MHz (1980’s), now we have a processor with 3-4GHz frequency. On the other side, there is not much improvement in the Memory speed (read/write speed). Today’s memory has a speed in the range of MHz, while the processors run on GHz scale. Cache memory has reduced this speed gap. Cache memory is made of SRAM and expensive to implement. Hence, we need to look at other parameters to improve the performance of the cache. While comparing two systems running on the same frequency, a program is executed on both systems, and the Time/Program is calculated. Time/Program is (Instructions/Program) X (Cycles/Instruction) X (Time/Cycle) [14].

Assuming L1 cache has an access latency of 1ns with a 100% hit-ratio, then CPU will take 100ns to load 100 blocks of data in a row. If the hit ratio of L1 is reduced by 1%, CPU will take 99ns to load the data from L1, and 10ns to load the missed data from L2. We see if the hit ratio is reduced simply by 1%, the CPU's performance degraded by 10%, earlier it was taking 100ns, and it took 110ns. Here, we assumed the missed data is sitting in the L2. As we move towards L3 and main-memory the access latency keeps on increasing. So, to improve the Cache performance access latency, hit-ratio and IPC are considered.

III. COMPARISION OF RRIP, AIRRIP AND ABRRIP

A. RRIP (Re-reference Interval Prediction) replacement policy

NRU (Not-Recently Used) used a 1-bit register (two possibilities 0 or 1) to identify whether the new block arrived has a near-immediate re-reference interval or distant re-reference interval [1]. It can lead to inefficient use of Cache, as the blocks get evicted as soon as they arrive if it doesn't receive any hit in the next cycle.

Thus, the distant blocks can evict cache-friendly blocks if a Cache-friendly block doesn't receive any hits, known as *scans*. RRIP is the extension of the NRU policy. RRIP makes use of the M-bit register giving us 2M-1 RRPV values to differentiate between different blocks present in the Cache.

M-bit register gives some time to the Policy to learn about the pattern to decide which block to evict, making it *Scan-resistant*. The given access pattern (a1 a2 a2 a1 b1 b2 b3 b4 a1 a2 a2 a1) shown in Fig.1 is a combination of the Cache-friendly and Streaming application.

In RRIP, a new block will have an RRPV value of 2M -2. Upon receiving a hit for the second time, the RRPV gets equal to '0', and on a cache miss the RRPV of all blocks increments by '1'. During the eviction, the block with maximum RRPV value (2M-1) is selected as a victim. If all the blocks have the same RRPV, then RRPV of all the blocks is incremented to max_RRPV, and the victim is selected randomly. We can see in the case of LRU 'a1' got evicted though it belonged to Cache-friendly application. In the case of RRIP, 'a1' and 'a2' were preserved until the end, thus giving more hits. For the given access pattern, we can say that RRIP outperforms LRU. Also, we can see that 'b2' got evicted first instead of 'b1' though 'b2' arrived later than 'b1'. Hence, justifying RRIP is not recency-friendly.

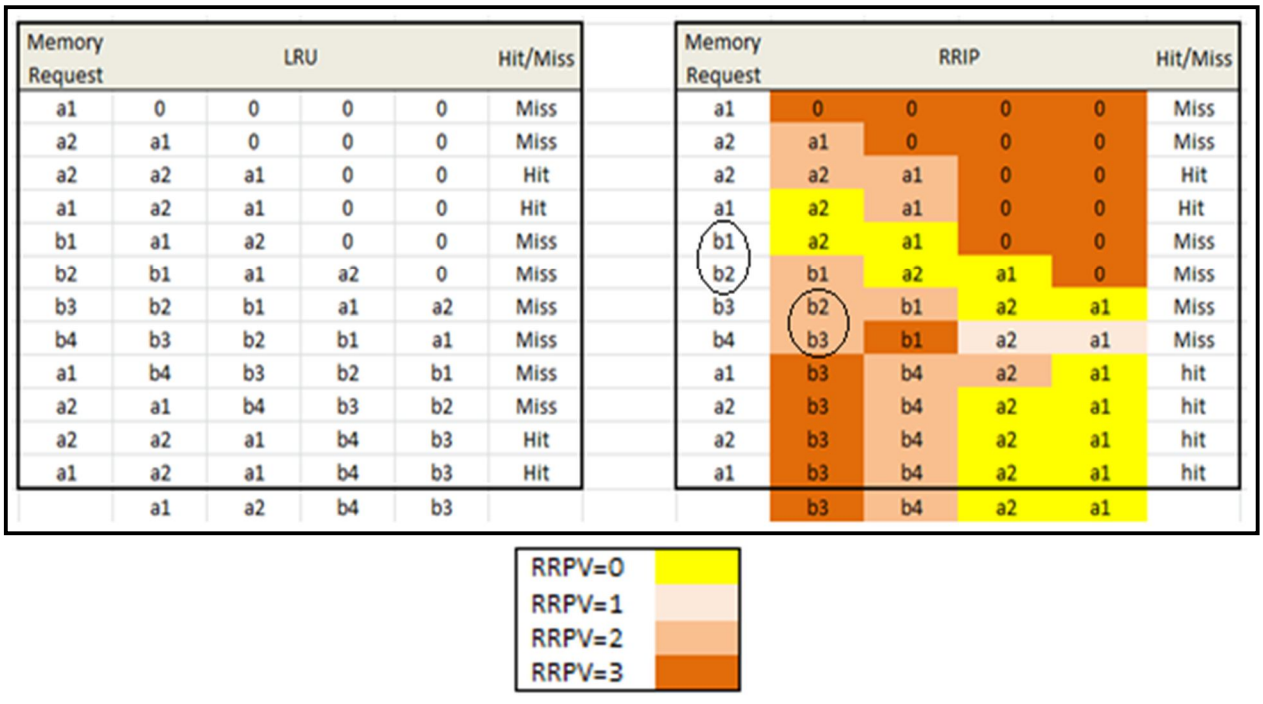


Fig. 1 LRU and RRIP comparison for the pattern a1 a2 a2 a1 b1 b2 b3 b4 a1 a2 a2 a1 [1]

B. AIRRIP (Adaptive Insertion Re-reference Interval Prediction) replacement policy

From Fig.2, we saw that RRIP overlooks the recency information of the block. To overcome this issue, an additional register is used to keep a track of the recency-information of the blocks [3]. Along with the M-bit RRPV register, the N-bit ARV register is used, which holds the information of the time the block got accessed.

For the given access pattern (a1 a2 a3 a4 a4 a3 a2 a1 b1 b2 a1 a2) shown in Fig.2, we can see that AIRRIP outperforms RRIP. In AIRRIP, during a cache hit, the ARV value gets updated to '0' similar to RRIP. But, if there is already a block with ARV = 0 ('a4') in the cache and another block receives a hit, then the ARV value will get incremented for the block with ARV = 0, leaving ARV of other blocks unaffected ('a1' and 'a2'). Upon a cache miss, ARV of all the blocks gets incremented by '1'. If the cache is full, the block with max_RRPV + max_ARV gets selected as a victim. As this policy needs to check both RRPV and ARV, it adds a little overhead.

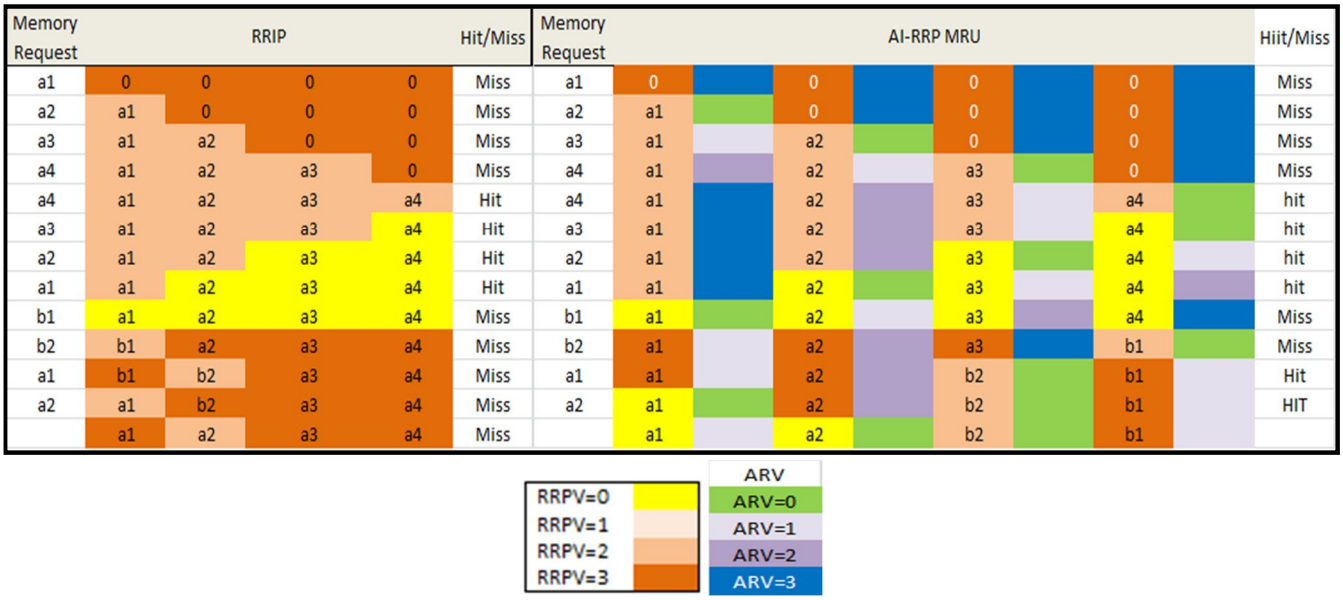


Fig. 2 RRIP and AIRRIP comparison for the pattern a1 a2 a3 a4 a4 a3 a2 a1 b1 b2 a1 a2

C. ABRRIP (Application Behaviour Aware Re-reference Interval Prediction) replacement policy

In modern multicore processors, LLC gets shared among all the cores running on different applications [8]. The data of the core with the Streaming application can interfere with the data of the core with Cache-friendly applications at LLC. Since the streaming applications need data frequently, it can evict Cache-friendly blocks leading to inefficient use of Cache memory. Hence, a policy is introduced which can differentiate the blocks not only at the block level but also at the application level, thus the name Application Behaviour Aware RRIP [2]. In ABRRIP, block-level RRPV (Br) and core level RRPV (Cr) are combined to get Application behavior aware RRPV value (ABr). To give more significance to the core-level 'α' parameter is multiplied with 'Cr' value. The given access pattern (a1 b1 b2 b3 a2 b4 b5 b6 a2 b7 b8 b9 a1) shown in Fig.3 is a mixed access pattern. We can see that RRIP receives no-hit, while ABRRIP receives some hits. On a cache hit, the ABr ('Br' and 'Cr') value gets updated to '0'. During a cache miss, 'α.Cr' value gets incremented along with RRPV ('Br'). If the Cache is full, the block with max_ABr gets selected as a victim. For a given access pattern, we can see that ABRRIP outperforms RRIP. In the case of RRIP, 'b2' stayed in the Cache polluting it, and caused the eviction of other blocks that have arrived recently. But in ABRRIP, 'b1'and 'b2' got evicted before 'b3' promoting recency-friendliness. Thus, ABRRIP promotes cache-friendliness and recency-friendliness better than RRIP.

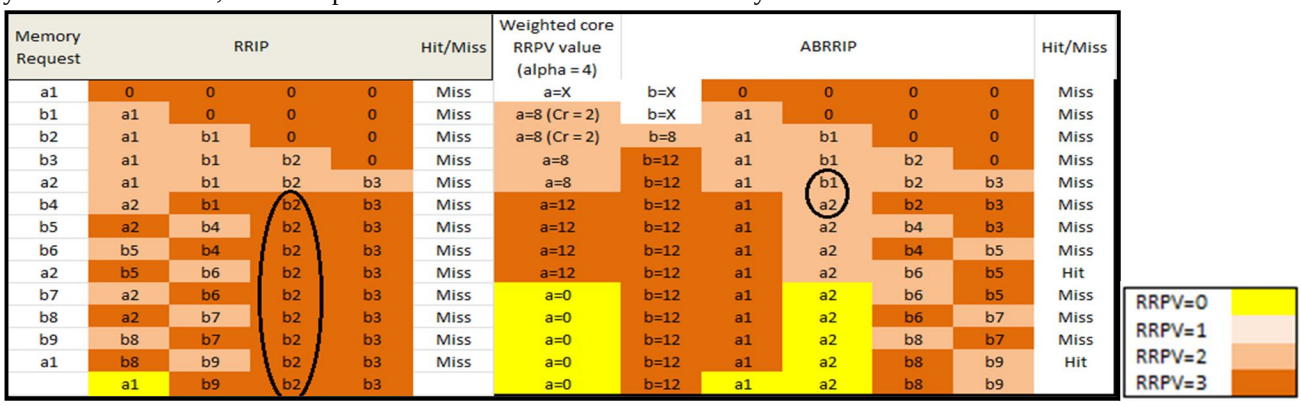


Fig. 3 RRIP and ABRRIP comparison for the access pattern a1 b1 b2 b3 a2 b4 b5 b6 a2 b7 b8 b9 a1

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 8 Issue VIII Aug 2020- Available at www.ijraset.com

D. ABRRIP, AIRRIP and RRIP comparison for two Cache friendly applications

From Fig.4, we see that for a combination of Cache-friendly + Cache-friendly running in a 2-core environment; all the three policies give the same number of hits. Since the main motive of all the three policies is to promote Cache-friendly block inside a workload, if all the workloads are Cache-friendly, then each workload will be promoted equally and hence all the cache-friendly workloads will behave the same as there is no Streaming / Thrashing / Recency un-friendly workload to be sacrificed.

Memory Request	RRIP				Hit/Miss	AI-RRP MRU								Hit/Miss	Weighted core RRPV value - a	Weighted core RRPV value - b	ABRRIP				Hit/Miss
a1	0	0	0	0	miss	0		0		0		0		Miss	a= X	b =X	0	0	0	0	Miss
a2	a1	0	0	0	miss	a1		0		0		0		Miss	a= 8	b=X	a1	0	0	0	Miss
b1	a1	a2	0	0	miss	a1		a2		0		0		Miss	a=12	b=X	a1	a2	0	0	Miss
b2	a1	a2	b1	0	miss	a1		a2		b1		0		Miss	a=12	b=8	a1	a2	b1	0	Miss
a1	a1	a2	b1	b2	Hit	a1		a2		b1		b2		Hit	a=12	b=12	a1	a2	b1	b2	hit
a2	a1	a2	b1	b2	Hit	a1		a2		b1		b2		Hit	a=0	b=12	a1	a2	b1	b2	hit
b1	a1	a2	b1	b2	Hit	a1		a2		b1		b2		hit	a=0	b=12	a1	a2	b1	b2	hit
b2	a1	a2	b1	b2	Hit	a1		a2		b1		b2		Hit	a=0	b=0	a1	a2	b1	b2	hit
a3	a1	a2	b1	b2	Miss	a1		a2		b1		b2		Miss	a=0	b=0	a1	a2	b1	b2	Miss
a4	a1	a2	b1	a3	Miss	a3		a2		b1		b2		Miss	a=4	b=0	a1	a2	b1	a3	Miss
b3	a4	a2	b1	a3	Miss	a3		a4		b1		b2		Miss	a=8	b=0	a4	a2	b1	a3	miss
b4	a4	b3	b1	a3	Miss	a3		a4		b3		b2		Miss	a=8	b=4	a4	b3	b1	a3	miss
a3	a4	b3	b4	a3	Hit	a3		a4		b3		b4		Hit	a=8	b=8	a4	b3	b4	a3	Hit
a4	a4	b3	b4	a3	Hit	a3		a4		b3		b4		Hit	a=0	b=8	a4	b3	b4	a3	hit
b3	a4	b3	b4	a3	Hit	a3		a4		b3		b4		Hit	a=0	b=0	a4	b3	b4	a3	hit
b4	a4	b3	b4	a3	Hit	a3		a4		b3		b4			a=0	b=0	a4	b3	b4	a3	hit
	a4	b3	b4	a3		a3		a4		b3		b4					a4	b3	b4	a3	

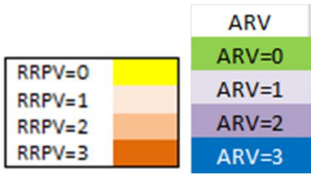


Fig. 4 Comparison of RRIP, AIRRIP and ABRRIP for Cache-friendly applications

IV. PERFORMANCE IMPROVEMENT OF ABRRIP

In the case of ABRRIP, as we increase the number of instructions or workload provided to the policy, an improvement in its Hit ratio and IPC is observed. This is because the ABRRIP policy is capable of handling a huge workload at a time. When we increase the number of instructions, the number of cycles will increase, but this increment is not linear, meaning the instructions executed per cycle increases drastically. From the given Fig.5, we can observe that both' a1' and 'a2' are kept in a cache by the algorithm untill the end. We can conclude that the higher the instruction count goes, the number of cache-friendly blocks inside the cache will increase. It eventually gives us a non-linear increase in IPC.

Memory Request	Weighted core RRPV value (alpha = 4)		ABRRIP				Hit/Miss
a1	a=8 (Cr = 2)	b=X	0	0	0	0	Miss
b1	a=8 (Cr = 2)	b=8(Cr=2)	a1	0	0	0	Miss
b2	a=8 (Cr = 2)	b=12(Cr=3)	b1	a1	0	0	Miss
b3	a=8	b=12	b2	b1	a1	0	Miss
a2	a=12	b=12	b3	b2	b1	a1	Miss
b4	a=12	b=12	a2	b2	b1	a1	Miss
b5	a=12	b=12	a2	b4	b1	a1	Miss
b6	a=12	b=12	a2	b4	b5	a1	Miss
a2	a=0	b=12	a2	b6	b5	a1	Hit
b7	a=0	b=12	a2	b6	b5	a1	Miss
b8	a=0	b=12	a2	b6	b7	a1	Miss
b9	a=0	b=12	a2	b8	b7	a1	Miss
a1	a=0	b=12	a2	b8	b9	a1	hit
b11	a=0	b=12	a2	b8	b9	a1	Miss
b12	a=0	b=12	a2	b11	b9	a1	Miss
a3	a=4	b=12	a2	b11	b12	a1	Miss
b13	a=4	b=12	a2	a3	b12	a1	Miss
b14	a=4	b=12	a2	a3	b13	a1	Miss
b15	a=4	b=12	a2	a3	b14	a1	Miss
a2	a=4	b=12	a2	a3	b15	a1	Hit
b16	a=0	b=12	a2	a3	b15	a1	Miss
b17	a=0	b=12	a2	a3	b16	a1	Miss
b18	a=0	b=12	a2	a3	b17	a1	Miss
a3	a=0	b=12	a2	a3	b18	a1	hit
	a=0	b=12	a2	a3	b18	a1	

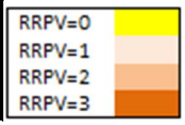


Fig. 5 ABRRIP's performance for more instructions

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 8 Issue VIII Aug 2020- Available at www.ijraset.com

V. EXPERIMENTAL METHODOLOGY AND IMPLEMENTATION OF REPLACEMENT POLICIES

A. Simulating Environment

To demonstrate the working of RRIP and ABRRIP policies, we have made use of the GEM5 simulator [1], [13]. It is an open-source Linux based platform where all the components emulated using C++ scripts, and the components configured using Python scripts. The simulator is used in System Emulation (SE) mode, as we want to emulate only a few parts such as the processor and memory and not the entire system. In SE mode, calls made to OS are also emulated. The architecture used is x86 with 512Kbytes of main memory (RAM), and DerivO3 (Out-of-order) CPU type.

The system simulation was done for a 4-core processor. The L1-D cache size of 64Kb, L1-I cache size of 32Kb, L2 cache size of 2Mb, and LLC cache size of 16Mb is used. The L1-D and L1-I had cache associativity of 2, L2, and L3 had cache associativity of 8 and 16 respectively.

B. Benchmarks

To test our design, we ran executable files from SPEC06 Benchmark Suite that has different types of benchmarks [1], [12]. These benchmarks emulate various applications such as C-code compilation, AI Game playing, AI Pattern recognition, etc. We run a combination of the benchmarks (GCC + MILC + LBM + GROMACS) that is one Cache-friendly and three Streaming for about 1 million instructions and fast-forwarding it by 1 billion instructions to prevent initial compulsory misses. For ABRRIP, we increase the instructions from 1 million to 10 million in steps.

VI. RESULTS

TABLE I
Read Hit Ratio and Write Hit Ratio For Different Policies

Mix type	CPU Number	ABRRIP		BRRIP		LRU		MRU	
		RHR	WHR	RHR	WHR	RHR	WHR	RHR	WHR
1 Cache Friendly + 3 Streaming	CPU 0	0.9769	0.9984	0.9408	0.9940	0.9402	0.9928	0.9000	0.9757
	CPU 1	0.8577	0.9961	0.6649	0.8932	0.6652	0.8932	0.6626	0.8921
	CPU 2	0.6619	0.4756	0.5888	0.6566	0.1194	0.2434	0.5538	0.6129
	CPU 3	0.6820	0.9775	0.8661	0.9967	0.8640	0.9962	0.8239	0.9918
	G-Mean	0.7842	0.8246	0.7515	0.8730	0.5039	0.6809	0.7222	0.8528

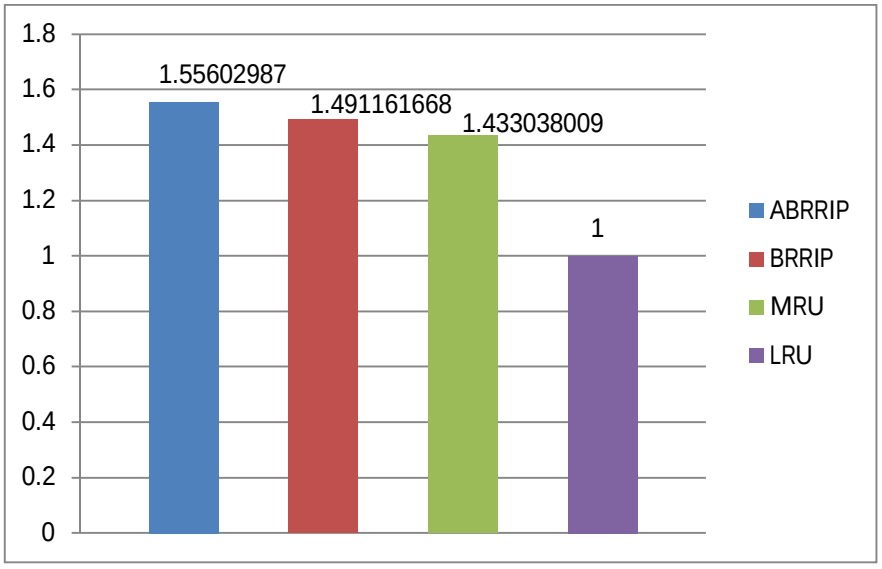


Fig. 6 Comparison of RHR for different policies

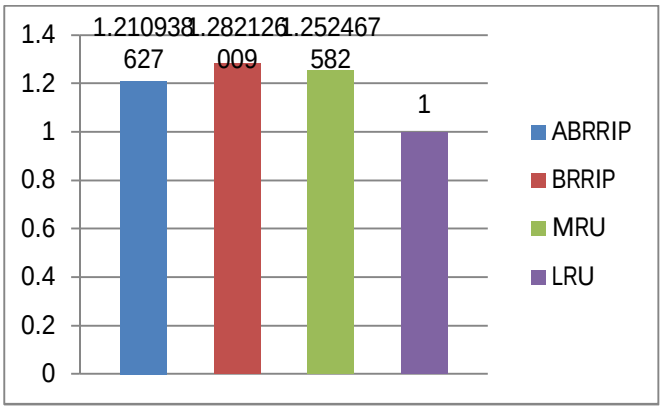


Fig.7 Comparison of WHR for different policies

TABLE II
IPC Comparison of Different Policies

Mix Type	CPU Number	ABRRIP	BRRIP	MRU	LRU
1 Cache Friendly + 3 Streaming	CPU0	1.0105	0.272403	0.153097	0.257109
	CPU1	0.8641	0.46403	0.43286	0.4612
	CPU2	0.0729	0.145112	0.124374	0.139855
	CPU3	0.5988	0.539563	0.410415	0.535875
	G-Mean	0.441853	0.31541	0.241166	0.307034

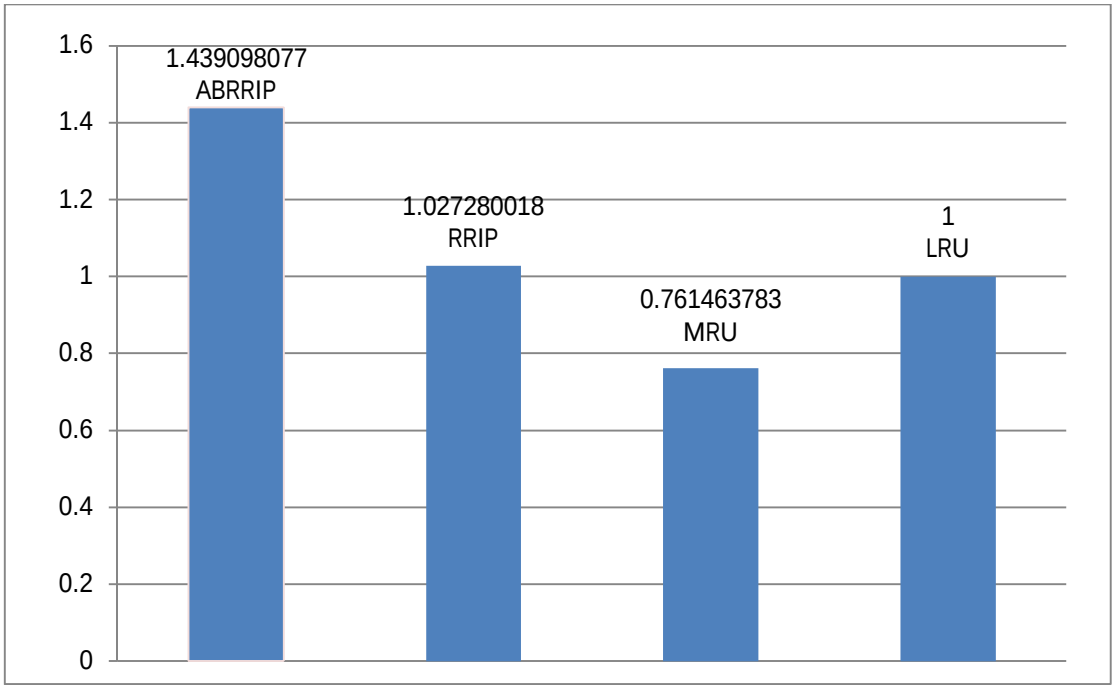


Fig. 8 Comparison of IPC for different policies

The IPC Comparison or improvement in IPC is shown in Fig.8. We see that as we increase the number of instructions from 1 million to 10 million, the IPC improves by 46.55% that is the higher the instruction count goes, the number of cache-friendly blocks inside cache will increase.

Table III
ABRRIP Comparison For Different Number Of Instructions

Mix Type	CPU Number	ABRRIP (1 Million Instructions)		ABRRIP (5 Million Instructions)		ABRRIP (10 Million Instructions)	
		RHR	WHR	RHR	WHR	RHR	WHR
1 Cache Friendly + 3 Streaming	CPU 0	0.9379	0.9757	0.9563	0.9932	0.9769	0.9984
	CPU 1	0.6626	0.8921	0.6639	0.8948	0.8577	0.9961
	CPU 2	0.5538	0.6129	0.6655	0.5455	0.6619	0.4756
	CPU 3	0.8239	0.9918	0.8738	0.9949	0.6820	0.9775
	G-Mean	0.7297	0.8528	0.7794	0.8333	0.7842	0.8246

TABLE IV
IPC Comparison Of ABRRIP For Different Number Of Instructions

Mix Type	CPU Number	ABRRIP (1 million Instructions)	ABRRIP (5 million Instructions)	ABRRIP (10 million Instructions)
1 Cache Friendly + 3 Streaming	CPU 0	0.251726	0.264859	1.0105
	CPU 1	0.454613	0.453193	0.8641
	CPU 2	0.13862	0.189329	0.0729
	CPU 3	0.520904	0.511264	0.5988
	G-MEAN	0.3015007	0.328315	0.441853

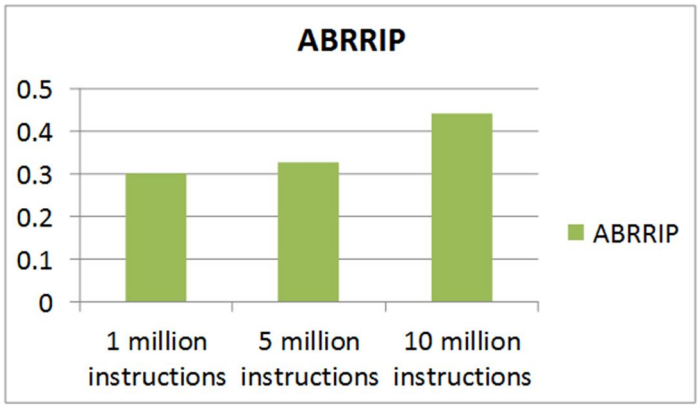


Fig. 9 IPC Comparison of ABRRIP for different number of instructions

VII. CONCLUSION AND FUTURE WORK

As we can see from Fig.6,7 and 8, the ABRRIP is better than other Replacement policies such as LRU, MRU, and RRIP. It is seen for one Cache-friendly and three Streaming applications running on a quad-core processor. Theoretically, it is observed from Fig.4 that for a combination of two Cache-Friendly applications running on a two-core processor, ABRRIP, AIRRIP, and RRIP behaves the same. It is observed that as we increase the number of instructions during the simulation for ABRRIP policy, its IPC increases by 46.55%. It is because the ABRRIP policy is capable of handling a huge workload at a time. When we increase the number of instructions, the number of cycles will increase, but this increment is not linear, meaning the instructions executed per cycle increases drastically.

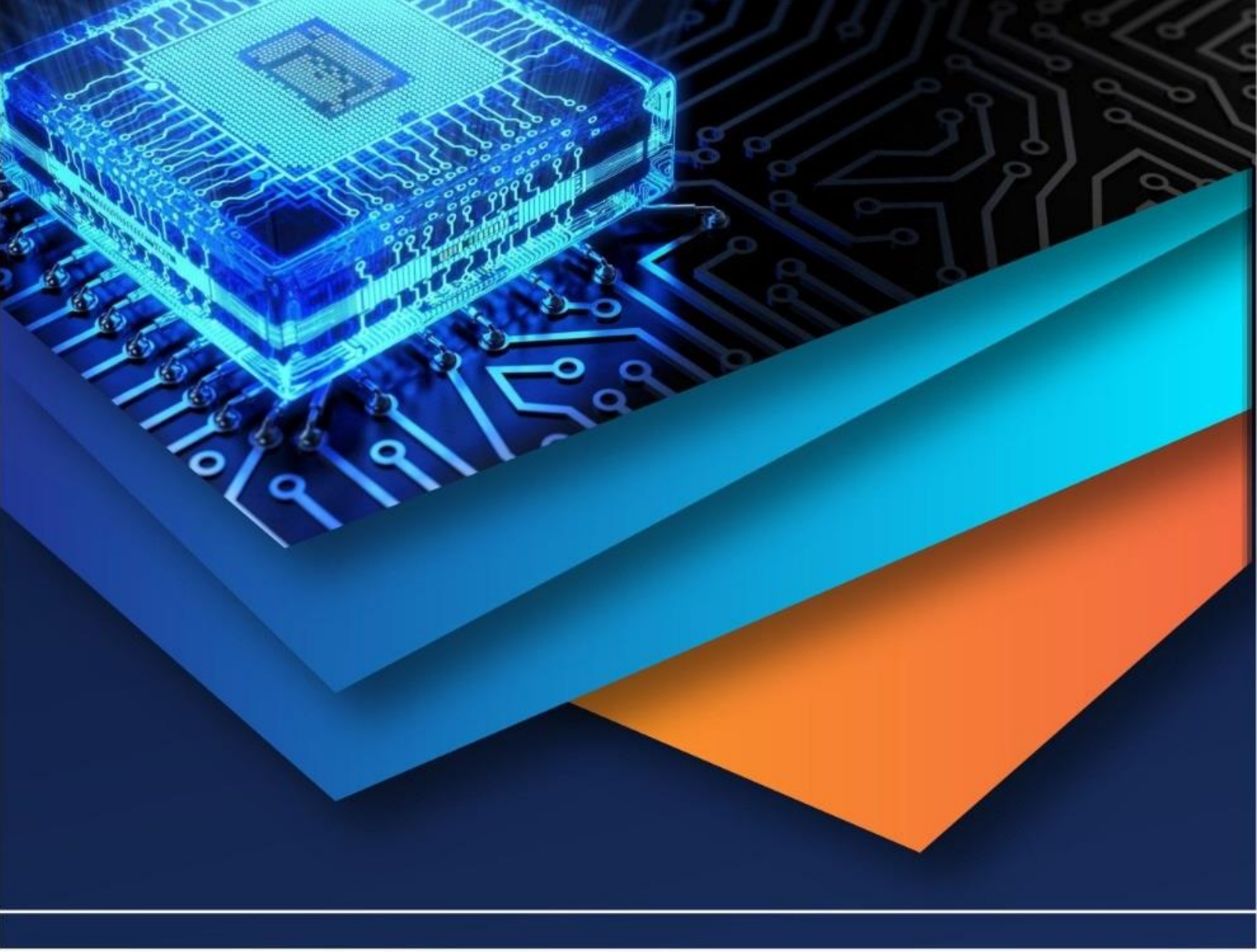
The project is observed on X86 architecture, it can be extended to ARM. The algorithm needs to be tested for a variety of different mixtures and benchmarks. Recency-friendly can be implemented in ABRRIP to boost its performance for mixtures other than CF. The projected can be tested and observed for multithreaded benchmark suites like the PARSEC benchmark suite.

VIII. ACKNOWLEDGEMENT

I thank all those people whose support and co-operation has been an invaluable asset during this project. I thank my guide Dr. R.N. Awale for guiding me throughout this project and sharing his valuable suggestions at every step. It would have been impossible to complete this project without their support, criticism, encouragement, and guidance. Also, I would want to thank Dr.Bhosle, lab-in-charge, and the Computer lab officials for providing me the technical support. I convey my gratitude also to Dr. Faruk Kazi**,** Head of Department for his motivation and providing various facilities, which helped me in the whole process of this stage of the project. I thank my family and friends for providing constant support and pushing me to conduct this research efficiently.

REFERENCES

[1] G. Jia, X. Li, C. Wang, X. Zhou and Z. Zhu, "Cache Promotion Policy Using Re-reference Interval Prediction," *2012 IEEE International Conference on Cluster Computing*, Beijing, 2012, pp. 534-537
[2] P. Lathigara, S. Balachandran and V. Singh, "Application behavior aware re-reference interval prediction for shared LLC," *2015 33rd IEEE International Conference on Computer Design (ICCD)*, New York, NY, 2015, pp. 172-179
[3] X. Zhang, C. Li, H. Wang and D. Wang, "A Cache Replacement Policy Using Adaptive Insertion and Re-reference Prediction," *2010 22nd International Symposium on Computer Architecture and High Performance Computing*, Petropolis, 2010, pp. 95-102
[4] S. Sreedharan and S. Asokan, "A cache replacement policy based on re-reference count," *2017 International Conference on Inventive Communication and Computational Technologies (ICICCT)*, Coimbatore, 2017, pp. 129-134
[5] Qaisar Javaid, Ayesha Zafar, Muhammad Awais, Munam Shah. Cache Memory: An Analysis on Replacement Algorithms and Optimization Techniques. Mehran University Research Journal of Engineering and Technology, Mehran University of Engineering and Technology, Jamshoro, Pakistan, 2017, 36 (4), pp.831-840
[6] S. Kumar and P. K. Singh, "An overview of modern cache memory and performance analysis of replacement policies," *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, Coimbatore, 2016, pp. 210-214
[7] Newton, S. K. Mahto, S. Pai and V. Singh, "DAAIP: Deadblock Aware Adaptive Insertion Policy for High Performance Caching," *2017 IEEE International Conference on Computer Design (ICCD)*, Boston, MA, 2017, pp. 345-352
[8] Nirmol Munvar, Shoba Gopalakrishnan, Arati Phadke, "Dynamic Non-Decaying ABRIP for Shared Level 3 Cache Memory Systems", International Journal of Innovative Science and Research Technolog, Volume 4, Issue 10, October – 2019
[9] W. Stallings, "Cache Memory," in Computer Organization and Architecture, 8 ed., Upper Saddle River, NJ, USA: Pearson Prentice Hall, 2006
[10] Carl Hamache, "Cache Memory Organization," in Computer Organization, 5 ed., McGrawHill Publications
[11] Cache replacement techniques on Wikipedia. [Online]. Available https://en.m.wikipedia.org/wiki/Cache_replacement_policies
[12] The SPEC2006 website [Online]. Available: https://www.spec.org/cpu2006/
[13] The Gem5 website. [Online]. Available: http://www.gem5.org/Main_Page
[14] L1 and L2 CPU Cache working. [Online]. Available: https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY