Australian Journal of Basic and Applied Sciences, 9(11) May 2015, Pages: 781-789

ISSN:1991-8178

Australian Journal of Basic and Applied Sciences

Journal home page: www.ajbasweb.com

A Dynamic Ratio Based Cache Replacement Algorithm

[1]Muthukumar S., [2]Anirudhan Sudarsan, [3]Balakumaran Palanivel and [4]Chandrasekar Rajasekar

[1]Professor, Department of Electronics and Communication Engineering, Sri Venkateswara College of Engineering,Sriperumbudur, India,
[2,3,4]UG Scholar, Department of Computer Science and Engineering, Sri Venkateswara College of Engineering, Sriperumbudur, India

ARTICLE INFO

Article history:
Received 25 March 2015
Accepted 28 May 2015
Available online 14 June 2015

Keywords:

ABSTRACT

Caching is a fundamental feature found in modern microprocessor architectures. It is imperative to design effective cache replacement algorithms as they play an important role in the overall performance of the system. In this paper, we propose a new cache replacement policy for multicore processors. The proposed algorithm combines two existing algorithms called LRU and LFU. In LRU replacement, a line once used stays in the cache for a very long time until it becomes the LRU line. Such a line which may not be used again reduces cache capacity available for other lines. While, LFU policy solves this problem to an extent by replacing the block that is least frequently accessed, it faces its own share of problems. In a situation where a particular block may be accessed repeatedly for a short duration of time and is then no longer accessed for long time periods, a situation may be encountered where the block may not be replaced for a long time despite other blocks being accessed more often within the recent time period. This paper is an attempt to propose a method that improves the hit rate while reducing the impact of the aforementioned problems. The proposed algorithm is intended to be implemented in the last level cache of the processor. This algorithm functions by logically segmenting each block in the cache into two halves and implementing LRU and LFU in a dynamic ratio. The dynamic ratio varies throughout the life of the input programs by taking into account the ratio of hits in each logical half of a set in the cache. Upon implementation, significant performance improvement was observed in the last level cache.

To Cite This Article: Muthukumar S., Anirudhan Sudarsan, Balakumaran Palanivel and Chandrasekar Rajasekar. ***Aust. J. Basic & Appl. Sci.,*** *9(11): 781-789, 2015*

INTRODUCTION

Parallel processing can be defined as one in which several calculations are carried out simultaneously. The importance of parallel applications in the present day scenario cannot be understated. There is a pressing necessity to develop algorithms and methods that exploit the parallelism in these applications.

Parallelism can be seen in several forms-

a) Bit Level Parallelism
b) Instruction Level Parallelism
c) Task Level Parallelism

Historically, programs were written for sequential or serial processing/ computation. The solution to a problem was written as a sequence of instructions that had to be executed one after the other in a predefined order.

Parallel programming on the other hand processes multiple elements simultaneously. This is accomplished by sub dividing the problem into multiple independent pieces or subtasks so that these tasks can be executed simultaneously. Subtasks in parallel programming are called threads. Parallel applications are usually considered to be multithreaded (Parallel Computing, 2015).

Parallel processing has grown steadily in popularity in recent years as a response to the constraints in frequency scaling. In the time period between 1980 and 2005, frequency scaling was the dominant method used to achieve performance gains in microprocessors. Frequency scaling is a method by which a processor's frequency is increased to cause a performance gain by reducing the time taken to execute one instruction. One disadvantage of frequency scaling was that, it consumed more power. The end of frequency scaling as the prevailing reason for performance gains led to a shift of focus towards parallel computing (Frequency Scaling, 2015).

The rise of the multi core paradigm was a direct response to the growth in parallel applications. A multicore processor is a single chip with two or more processing units called cores which can execute instructions simultaneously leading to an increase in throughput. The true capability of a multi core

Corresponding Author: Muthukumar S., Professor, Department of Electronics and Communication Engineering, Sri Venkateswara College of Engineering,Sriperumbudur, India,
E-mail: smuthukumar.65@gmail.com

processor can be observed only when it processes parallel applications (Multi-core Processor, 2015).

The need for faster processing continues to grow at an exponential rate. Today, programmers and developers desire an unlimited amount of fast memory to execute their programs. Such a desire is however misplaced as a processor containing large amount of fast memory is economically unfeasible. A prudent solution that was adopted was to structure the memory hierarchically so that it would contain some amount of fast but expensive memory and some amount of slow but inexpensive memory. The main aim is to provide a memory system which has speeds as much as that of the fastest memory and cost as low as that of the cheapest memory (Hennessy, L. 2007).

The first or the highest level of memory is called the cache. In the multicore processors, cache itself is implemented as a hierarchy with the first level of cache being private to each core and the last level cache being shared among all the cores. There are potential physical limitations on the size of cache memory and hence an important aspect of cache memory performance is the cache replacement policy. A properly defined cache replacement policy can lead to an improvement in performance through an increase in hits or reduction in misses. Hence, study of cache replacement policies assumes significant importance and is a widely researched area.

Overview Of Cache Memory:

CACHE-The need for cache memory arose from the startling difference in the speeds of the processor and the main memory. The time taken to retrieve data by a processor from its own registers is considerably lesser than the time taken to retrieve the same data from the main memory. One may think that having more number of registers can solve this problem. However, this issue presents a catch-22 situation as there are both physical and financial limitations on processor memory. In the absence of a bridging mechanism, a processor will spend much of its time retrieving data from the main memory, potentially wasting several cycles on this. A processor that aims to provide superior performance cannot afford to get caught up in such a situation.

The most efficient solution to this situation is using a cache memory which basically bridges the speed gap between the processor and the main memory by making the main memory to appear faster than it actually is.

The efficiency of cache mechanisms is grounded on the concept called locality of reference. A thorough investigation of programs has demonstrated that most of a program's execution time is spent on repeatedly executing a small subset of instructions. These instructions may form a part of an iterating block such as a loop or a set of functions that call each other. The remaining instructions on the other hand are accessed quite infrequently. The aforementioned phenomenon is what is called locality of reference (Hamacher, V., 2011).

When the processor requires a word, it first searches for the word in the cache. If the required word is found, it is termed a *cache hit*. If the data word is not found, then it is termed a *cache miss*. When a cache miss occurs, data is searched for and retrieved from the main memory and then placed in the cache. The following diagram is one of a multicore processor with multiple levels of cache. The diagram shows a processor with four cores. Each core has its own private cache (specified by the words *Individual memory* in the diagram) and all four cores share a cache (specified by the words *shared memory* in the diagram).

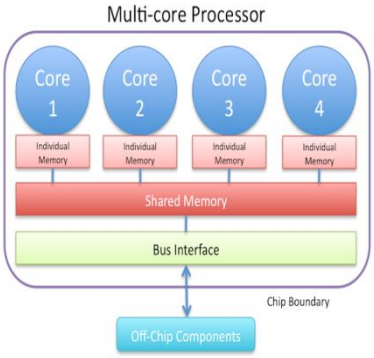

Fig. 1: Multi core processor.

The size of the application will usually be much higher than the size of the cache itself. Hence the cache will not be able to accommodate all the items (instruction and data) of the application simultaneously. Hence, if the required item is not present in the cache, it has to be retrieved and then placed in the cache.

The quantity of data that is retrieved from the main memory is called as block or line. The hardware handles a cache miss. A cache miss causes processors using in order execution to stall until the required data is available. When out of order execution is used, other instructions can proceed but the instructions needing the result will stall.

The processor issues read and write requests utilizing addresses that reference locations in memory. If the requested word exists on the cache itself, the read and write operation is performed on that location. This scenario terms the hit as either read hit or write hit. If the word is in the cache for a read operation, the main memory will not be involved. However, the write operation presents a different challenge. If a write involves a change to the cache word, then the corresponding word in the main memory has to be updated sooner or later.

There are two techniques that are used for this purpose. The first technique is called the write through protocol. This technique updates both the

cache memory location and main memory location simultaneously. The second technique is called the write back or the copy back protocol. This technique initially updates only the cache location and marks it with a flag called the dirty bit or modified bit. The main memory location corresponding to this cache block is updated when the block is evicted by the replacement policy. The write through protocol is easier to implement but usually results in a number of unnecessary writes that can be avoided by the use of the write back protocol.

1. Cache Performance Metrics:

One common performance measure is the average memory access time which is calculated as Hit time + (Miss rate * Miss penalty) where hit time is the time required for a hit in the cache, miss rate is the fraction of access that result in a cache miss and the miss penalty is the cost of a miss. There is yet another metric called the hit rate which is defined as the ratio of the total number of hits to the total number of accesses.

There are three kinds of misses:

a) Compulsory miss-The first access to any block will lead to a miss as the block will not be present in the cache. Such a miss cannot be avoided even if the size of the cache is infinity.

b) Capacity miss-This miss occurs because of the finite size of the cache. The cache may not be able to contain all the blocks needed for execution and hence misses will occur leading to new incoming blocks replacing old blocks in the cache.

c) Conflict miss-This occurs because the block placement strategy may not be fully associative. This miss occurs because a block may be discarded and later retrieved if conflicting blocks map to its set.

Compulsory misses can be reduced through the use of prefetching techniques. Prefetching is a technique where the processor fetches an instruction from the main memory before it is actually needed. The retrieved instruction is placed in the cache. It would be practical to prefetch instructions in program orders as programs are likely to execute sequentially at least in the initial stages of execution. The role of the pre-fetcher involves determining which blocks of data will be required in the initial stages of an applications execution.

Capacity misses can be reduced by the use of well-defined cache replacement policies. Several methods are used in addition to replacement algorithms to reduce the misses. These include using a larger block size, bigger caches and merging write buffer to reduce miss rate, using multilevel caches and giving priority to read misses over writes to reduce miss penalty, using small caches, trace caches and way prediction to reduce the hit time. These optimizations are detailed in the following subsection.

When a capacity miss occurs, a new block has to be brought into the cache and an existing block has to be removed from the cache. The block that is removed from the cache is called the victim. The role of cache replacement policies is to optimize this function. The most common cache replacement policies are Least recently used (LRU) and Least Frequently Used (LFU). As the same suggests, LRU considers the least recently accessed block in the cache as the victim and LFU considers the least frequently accessed block as the victim. One of the major disadvantages of LRU algorithm is that a block or a line once accessed stays in the cache for a long time until it reaches the least recently used position. If the line has not been used since its first access, then it has effectively reduced the capacity available for other blocks/ lines. The algorithm we propose attempts to reduce this ill-effect of LRU(Hennessy, L. 2007; Cache Algorithms, 2015).

2. Related Work:

This paper proposes two advancements to the base algorithm under consideration called SLRU algorithm. The SLRU algorithm divides the cache lines into two lists termed the reference and non-referenced list. The victim is usually chosen from the referenced list. However, if one is not present, then the victim is selected from the global LRU cache line. The proposed advancements are as follows:

a) Random promotion of lines from non-referenced to referenced list.

b) Ageing mechanism that allows lines in the referenced list to be aged and removed.

The proposed method uses adaptive bypassing. When a cache miss occurs, a decision on whether to bypass the cache and keep the selected victim or whether to replace the victim is taken. The algorithm keeps changing based on the effectiveness of the decision. If the decision has been deemed effective, then the probability of bypassing is doubled. If the decision has not been found to be effective, then the probability is halved. Simulations show that the proposed mechanism attains significant performance speedups when compared to LRU for most traces (Gao, H., C. Wilkerson, 2010).

This paper proposes a replacement mechanism for the last level cache that attempts to overcome the cache thrashing problem. The method divides each set in the cache into multiple subsets of which only one is active at any given point of time. When a cache hit occurs, the corresponding line is promoted by moving it to MRU position within that subset. When a miss occurs, the victim is chosen from the present active subset by using LRU policy. The incoming line is inserted into the LRU position of the active subset.

The miss also causes a counter to increment. If the counter exceeds a threshold value, the active subset is changed. The threshold value of the counter is the performance determinant for the algorithm.

Maximum performance was found to be attained when the threshold value was around 4 or 8. The threshold value was selected from a sampling set. The policy attained an average speed up of 4.5% over LRU across 28 SPEC CPU benchmarks (He, L., 2010).

This paper proposes a replacement policy called the Dueling CLOCK which has a low overhead cost and captures recency information in memory accesses. It attempts to exploit the frequency patterns of accesses and is also scan resistant. The method dynamically adapts between the CLOCK algorithm and the scan resistant version of the algorithm through set dueling. The CLOCK algorithm uses a reference bit called the hitBit for each cache block that is set to 1, each time the block is accessed. This reference bit is also used to exploit the frequency feature of the memory accesses.

The algorithm uses a circular queue and a replace pointer to indicate the victim block. When a hit occurs, the algorithm sets the reference bit to 1. Upon a cache miss, CLOCK searches for a cache block that has the reference bit set to 0. The algorithm is made scan resistant by not allowing the pointer to advance to the next cache block whenever the hit bit of the current cache block is 0. Experimental results indicated that the number of misses per thousand instructions was lower than LRU by an average of 10.6% (Janapsatya, A., 2010).

This paper proposes a technique called the adaptive time keeping replacement (ATR) that assigns a lifetime to cache lines according to OS assigned process priority and application memory reference pattern. The ATR mechanism solves the shared contention issue by identifying the high priority threads. A modified *cache decay* method is used. A decay counter is assigned to each cache line based on process priority and is decremented over time. When a hit occurs, the decay counter is reset to decay interval. When the counter hits zero, the corresponding cache line will be considered for replacement i.e. victim. The decay counter for the high priority processes receive a longer decay interval so that low priority processes do not overshadow higher priority ones. ATR shows considerable performance improvement under all workload (WU, C., M. Martonosi, 2011).

This paper proposes a method that seeks to take into account that data accessed by multiple threads need to be given more priority to stay in the cache as a miss on such a data can lead to a bottleneck by stalling multiple threads. The proposed methods accounts for the sharing status of data in the cache. Every element in the cache is associated to a two bit counter value called the sharing degree counter which indicated the extent to which a given element is shared.

The counter takes four values corresponding to four states- not shared, lightly shared, heavily shared and very heavily shared. To keep track of the threads that access a block, a thread track filter which is a dynamic array is associated with every cache block. The priority value for a cache block is determined by using both hit count and sharing degree. Eviction takes place based on increasing priority values. If there is a tie, the first encountered cache block is taken as the victim. In addition, the priority value of all cache blocks is decremented by one after an eviction to ensure the cache is not polluted by stale data. Initial incoming block priority is 10 and the sharing degree is 0. The method yielded an improvement of 5% when simulated for various PARSEC benchmarks (Muthukumar, S., P. Jawahar, 2014).

This paper proposes a replacement algorithm for the last level cache. A victim is selected based on some priority values. The algorithm seeks to ensure that a line that is evicted is not immediately needed by another core by taking shared nature into consideration. The blocks from which victims are chosen are divided into four groups- shared but clean, private and exclusive, private and modified, other shared lines.

The cache lines having highest reuse register value is first determined for the purpose of eviction. A block that is continually accessed is given a higher reuse register value. The value is high, if the block is private. Otherwise it decreases to a value close to zero. In terms of eviction, lowest priority is assigned to shared dirty lines as their eviction is assumed to be more costly. This logic is also applicable for private modified lines which are however given a higher priority as they are present only in private caches of a single core.

Highest priority is accorded to the clean lines which are evicted first. However, in some sets the highest priority is assigned to private exclusive lines. The first time any block is accessed, it is in private mode. The second access will determine whether it remains in private mode or shared mode based on whether or not the accessing thread is the same as the first. Eviction is performed based on calculations involving reuse- register value and the priority. The block with the highest value is evicted first. Upon simulation, the WARP algorithm showed a better performance than LRU (Balaji, S., 2013).

In direct contrast with LRU, the most recently used policy discards the most recently accessed cache block. This algorithm is known to give strong performance when a file is repeatedly scanned in a looping reference pattern. Also, it has been noted that MRU generates more number of hits than LRU for random access patterns and repeated scans over large data sets. MRU algorithms usually have more hits than LRU due to their proclivity towards retaining older data.

This algorithm is theoretically supposed to be the best performing cache replacement algorithm. It is considered a clairvoyant algorithm as it is supposed to evict the blocks that will not be required

in the near future. This is called Belady's optimal algorithm. This algorithm is considered to be non-implementable as it is not possible to determine the time that will be consumed before a word is accessed again.

This algorithm selects a victim randomly when a capacity miss occurs. The victim is then evicted to make way for a new word. There is no requirement to keep track of accesses in this policy.

3. *Proposed Mechanism:*

Multi-core architectures play a predominant role in the performance of real time parallel applications as such applicants are usually multi-threaded in nature. In the shared memory architecture, where the last level cache is shared, there exists a contention for the memory in the cores. Hence, a good cache replacement policy is required to improve the performance through an increase in hits or by reducing the miss rate.

5.1 *Motivation:*

Our motivation arises from the nature of the behavior of LRU and LFU. The default characteristics lead to behaviours which were not to our liking from a performance perspective. One central issue that plagues LRU replacement policy is that a block or a word that is accessed once is shifted to the MRU position. This block may not be accessed again in the near future or may be even for a long duration in time. However, this block or line will stay in the cache for a very long time until it becomes the LRU block. Only then will it be evicted. Such a line, which may not be accessed, again in the near future reduces the cache capacity that could be potentially available for other lines.

LFU replacement policy solves this problem through its default behaviour of evicting the blocks that are accessed most infrequently. However, the usage of LFU brings in its own set of problems. In a situation where a particular block may be accessed repeatedly for a short period of time and is then no longer accessed in the future, a situation may be encountered where the block may not be replaced for a long time despite other blocks being accessed more often within the recent time period. This paper is an attempt to propose a method that improves the hit rate while reducing the impact of the aforementioned problems.

5.2 *Implementation:*

The proposed work considers the mapping mechanism to be 8 way set associative. In our method, we segment each set in the cache into two equal parts i.e. 4 blocks or lines in each segment. In one of the segments we implement the LRU algorithm, while in the other half we implement LFU algorithm. For the LFU algorithm, we maintain a special counter which keeps track of how many times a cache block is accessed. The uniqueness of our work lies in victim finding process. The victim block may be in either of the two segments. The segment to search to find the victim block is determined by a dynamically self-changing ratio. This ratio is initialized at 3:1 i.e. for every three victims found in the LRU segment, one victim will be found in the LFU segment.

This ratio will vary dynamically based on the ratio of hits in both the segments. The rule to vary the ratio is as follows- *The number or proportion of times a victim block is searched for in a segment is inversely proportional to the number of hits in the segment. The amount of times a victim is searched for cannot fall below 25% or go above 75%.*

The ratio changes dynamically every ten thousand hits. A counter is checked to determine which half of the set has more number of hits. The proportion for that half is reduced correspondingly. There are only three possible ratios that have been defined in the algorithm. 3:1, 2:2 and 1:3 for LRU segment: LFU segment. The counter is reset to zero along whenever the ratio is changed. The following is an illustration of the structure of the cache.

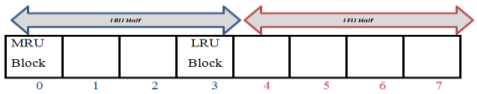

Fig. 2: Cache structure.

The block indexed by '0' is the Most Recently Used (MRU) position of the cache set. The order of the blocks with increasing values of the indices defines, how recently a particular block was requested by the processor. Thus, the block indexed by the greatest value '3' is the Least Recently Used (LRU) block in the LRU segment or half.

Every cache replacement policy has three phases-

1) Insertion
2) Promotion
3) Eviction

The three phases of our algorithm are explained below with illustration.

1) Insertion:

Initially all the sets in the cache are empty. This leads to compulsory misses when a block is accessed. Whenever a block must be retrieved from the main memory, it is always inserted into the MRU position of the LRU segment. This insertion strategy was adapted directly from that of LRU algorithm. This insertion strategy is applicable irrespective of whether a capacity miss occurs or not. The insertion process is explained below with the help of illustrations.

Insertion of a new block always occurs at the MRU position of the cache set.

When insertion occurs at the MRU position of the cache set which is previously not empty, all the

blocks shift one position towards the right (towards LRU position) , thus making MRU position empty.

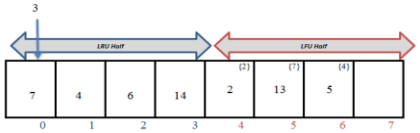

Fig. 3: Insertion process for proposed method.

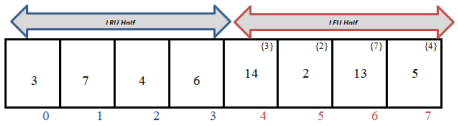

Fig. 4: Structure of set post insertion.

2) Promotion:

The promotion technique varies for both the segments. The promotion for the LRU segment is again adapted from that of the LRU algorithm- A block or line which is accessed by the processor is sent to the MRU position and the other blocks are shifted towards LRU position.

The promotion strategy implemented for the LFU segment is-*no promotion*. The accessed block is not shifted anywhere. The promotion process is illustrated below.

The set in a cache before CPU access is shown below. The frequency or no. of times a block is accessed is shown at top right corner of the block.

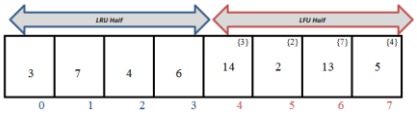

Fig. 5: Structure of cache set prior to promotion.

Case 1:

When the requested block is in LRU half- when block 4 is accessed, block 4 is moved to MRU position adjusting all the blocks in between.

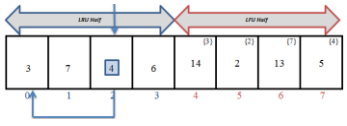

Fig. 6: Promotion in LRU half.

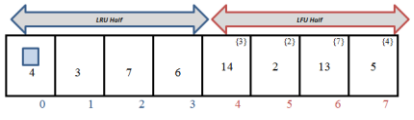

Fig. 7: Structure of set post promotion.

Case 2:

When the requested block is in LFU half- when block 5 is accessed, the frequency of the block 5 is incremented by 1.

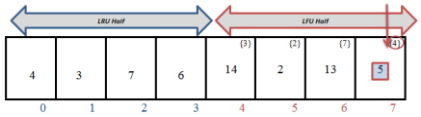

Fig. 8: Promotion in LFU half.

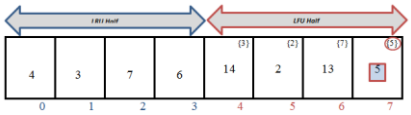

Fig. 9: Structure of set post promotion.

3) Eviction:

Eviction is the process of determining which block to remove from the cache when a capacity miss occurs. First, the segment in which the eviction has to be carried out is identified- i.e. whether we should evict from LRU half or LFU half. This is determined by the dynamically self-changing ratio. This is illustrated below.

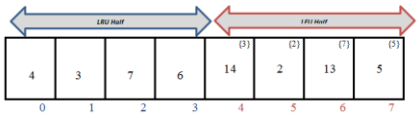

Fig. 10: Set before eviction.

When Cache miss occurs to this set, either the victim to be evicted is chosen from LRU Half or LFU Half based upon dynamic self-changing ratio.

Case 1:

The block is evicted from LRU Half. We choose the Least Recently Used Block from LRU Half as the candidate for replacement. I.e. the Block at Position 3(LRU).

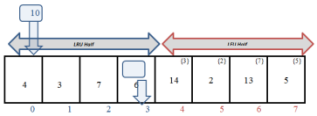

Fig. 11: Eviction from LRU half.

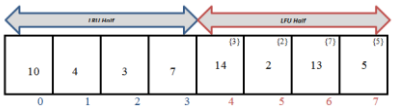

Fig. 12: Set post replacement.

Case 2:

When the block is evicted from LFU Half. We choose the Least Frequently Used Block from LFU Half as the candidate for replacement .i.e Block with least no. of access.

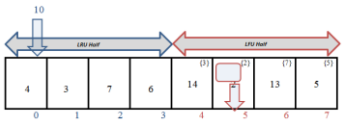

Fig. 13: Eviction from LFU half.

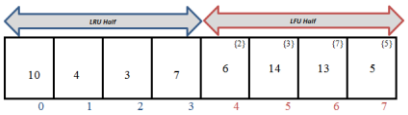

Fig. 14: Set post replacement.

4. Simulators And Benchmarks:

The efficacy of the proposed method is tested by comparing its performance against the performance of LRU for some standard inputs. These standard input programs are called benchmarks. Wikipedia defines a benchmark as "*the act of running a computer program, a set of programs or other operations in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it.*"

The simulator we use to test our algorithm's performance is the gem5 simulator which m5sim.org defines as a modular discrete event driven computer system simulator platform. Gem5 was designed in python and C++ and most of its constituents are delivered under a licence along the lines of BSD.

It is possible to simulate Alpha, ARM, MIPS, Power, SPARC, and 64 bit x86 architectures in Gem5. GEM5 can simulate PARSEC, SPLASH and SPEC benchmarks (Main Page, 2015; Introduction, 2015).

According to http://parsec.cs.princeton.edu/, the Princeton Application Repository for Shared-Memory Computers (PARSEC) is a benchmark suite composed of multithreaded programs (PARSEC Benchmarks, 2015; The PARSEC Benchmark Suite, 2015). It has a collection of programs that are quite diverse and characteristic of modern parallel applications. There are a total of thirteen workloads (ten applications and three kernels) and all of them have been written in either C or C++ (Christian Bienia, 2011).

5. Implementation Results:

Our algorithm was tested against five of the PARSEC benchmarks. We measured the performance of the proposed algorithm against LRU for a standard cache performance parameter- number of hits. The graphs in the following figures show the performance of our algorithm against that of LRU. In addition to the above, there were also two other parameters that were considered- the average cache occupancy and the average number of references to a valid block.

The proposed method yielded a decrease in average cache occupancy values. This is an improvement over LRU because one of the aims of our research was to reduce the possibility of cache pollution occurring. A reduced cache occupancy value validates the assumption that cache pollution would be reduced. The graphs in figures 17 and 18 show the performance and the improvement in performance of our algorithm with respect to LRU for cache occupancy.

The other parameter that was considered- average number of reference to a valid block is an indication of how long an invalid block is present in the cache. In a multi core processor, one of the cores may change the value of some data leading to invalidation in the cache. The longer a block stays in the cache, the greater the possibility of it being invalidated. Such invalidated data may remain in the cache too long if they are not accessed at all. Our algorithm reduces the possibility of this situation. This assumption was also validated by the increase in the average number of references to valid blocks. This performance parameter is represented graphically in Figures 19 and 20. .

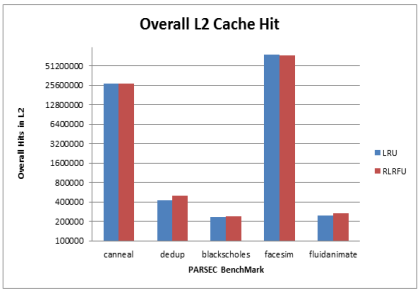

Fig. 15: Comparison of Overall Hits in L2 cache.

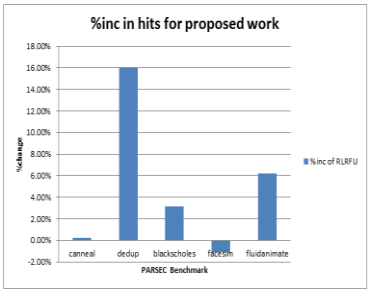

Fig. 16: Percentage change in number of hits in L2 cache for proposed mechanism.

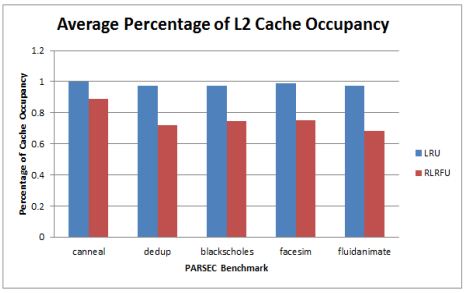

Fig. 17: Comparison of Average cache occupancy.

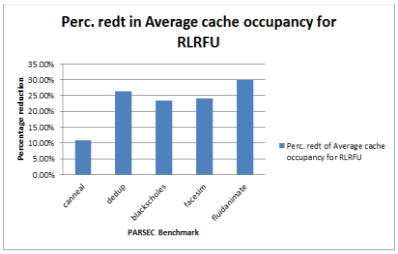

Fig. 18: Percentage improvement (reduction) in average cache occupancy for RLRFU.

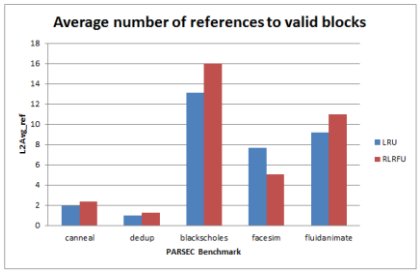

Fig. 19: Comparison of average number of references to valid blocks.

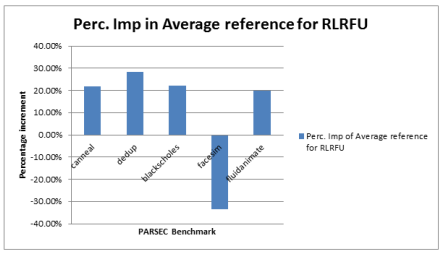

Fig. 6.20: Percentage difference in references to valid blocks.

6. Discussion:

The importance of improving the performance of cache memories can never be overstated. The advent of multi core processors has led to an increase in the number of parallel applications in the market. Even though parallel applications have not become ubiquitous yet, they are expected to in the future. Hence, it is vital that research in the area of computer architecture focus on improving the algorithms that would lead to faster execution and computation of parallel programs. One such specific area is cache replacement policies. The traditional algorithms like LRU and LFU suffer from several drawbacks which lead to them, not utilizing the inherent parallel-ness in application programs to improve the hit rate in the LLC (last level cache). Hence, in this paper, we proposed a new cache replacement policy that combines the best features of both LRU and LFU and ends up with a much better performance on an average in terms of the hit rate.

To simulate our algorithm we used the Gem5 simulator and the PARSEC benchmarks were used for characterizing the performance of the algorithm. The main measure of comparison was the total number of hits for a parallel execution of the applications. For the purpose of evaluating our algorithm's efficacy, it was tested against LRU which is the most common cache replacement policy. The testing was conducted for five of the benchmark programs.

The following is a tabulated representation of the results presented in Figures 6.15-6.20.

Table 7.1: Total number of hits in L2 cache for various benchmarks.

Benchmark	LRU	RLRFU	Percentage difference
Dedup	428575	496981	15.96127
Fluid Animate	250390	265970	6.222293
Canneal	26993057	27056607	0.235431
Facesim	75810144	75002072	-1.06592
Blackscholes	233204	240559	3.153891

Table 7.2: Average Cache Occupancy.

Benchmark	LRU	LFU	Percentage Difference
canneal	0.997595	0.88957	10.82854
dedup	0.975944	0.719437	26.28296
blackscholes	0.972434	0.745492	23.33752
facesim	0.990515	0.752204	24.0593
fluidanimate	0.9732	0.680957	30.02908

TABLE 7.3: Number of references to valid blocks.

Benchmark	LRU	LFU	Percentage Difference
canneal	1.955353	2.379102	21.67123
dedup	0.973383	1.250637	28.48355
blackscholes	13.10335	16.01105	22.1905
facesim	7.654171	5.086966	-33.5399
fluidanimate	9.182315	10.99761	19.76942

7. Future work:

An improvement in performance obtained is of great value because even a very small improvement can make a huge difference in real time. In future we expect to develop a modified version of our algorithm in which the ratio changes dynamically

based on the characteristics of the application in the cache.

Conclusion:

Cache memories are one of the most fundamental features of all microprocessors. They are the primary determinants of the overall performance of the microprocessor and it is the replacement policy which determines the impact of the cache on the microprocessor performance. Existing cache replacement policies such as LRU are not ideal for all workloads as they have their own disadvantages which may hinder the performance for some workloads. In this project we proposed a new cache replacement policy called the Ratio Based algorithm which logically partitions the cache and implements LRU and LFU in a dynamically self-changing ratio. Since our aim was to increase the hit rate while reducing the possibility of pollution in the cache, we focused on three parameters- total number of hits in L2 cache, the average cache occupancy and number of references to valid blocks. Following simulations these three parameters were found to yield an average improvement of 4.9%, 22.9% and 11.7% respectively for the tested benchmarks.

REFERENCES

"Parallel Computing, 2015." Wikipedia. Wikimedia Foundation. Web, 29.

"Frequency Scaling, 2015." Wikipedia. Wikimedia Foundation. Web. 29.

"Multi-core Processor, 2015." Wikipedia. Wikimedia Foundation. Web. 29.

Hennessy, L. John and A. David, 2007. Patterson. Computer Architecture a Quantitative Approach. 4th ed. Amsterdam: Elsevier/Morgan Kaufmann, Print.

Hamacher, V., Carl, Zvonko G. Vranesic and S. Zaky, 2011. Computer Organization. 5th ed. New York: McGraw-Hill, Print.

"Cache Algorithms, 2015." Wikipedia. Wikimedia Foundation. Web, 29.

Gao, H., C. Wilkerson, 2010. "A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing". 1st JILP Worshop on Computer Architecture Competitions: Cache Replacement Championship., Saint Malo, France.

He, L., Y. Sun, C. Zhang, 2010. "Adaptive Subset Based Replacement Policy for High Performance Caching." 1st JILP Worshop on Computer Architecture Competitions: Cache Replacement Championship, Saint Malo, France.

Janapsatya, A., A. Ignjatovic, J. Peddersen, S. Parameswaran, 2010. "Dueling CLOCK: Adaptive Cache Replacement Policy Based on The CLOCK Algorithm." Design, Automation & Test in Europe Conference & Exhibition (DATE), 920-925. Dresden

WU, C., M. Martonosi, 2011. "Adaptive Timekeeping Replacement: Fine-Grained Capacity Management for Shared CMP Caches." ACM Transactions on Architecture and Code Optimization (TACO), 8-1.

Muthukumar, S., P. Jawahar, 2014. "Sharing and Hit based Prioritizing Replacement Algorithm for Multi-Threaded Applications." International Journal of Computer Applications, 90(12): 34-38.

Balaji, S., R. Gautham Shankar, P. Arvind Krishna, 2013. "WARP: Workload Nature Adaptive Replacement Policy." International Journal of Computer Applications, 70(12): 35-38.

"Main Page, 2015." Gem5. Web, 29.

"Introduction, 2015." - Gem5. Web, 29.

"PARSEC Benchmarks, 2015." - Gem5. Web, 29.

"The PARSEC Benchmark Suite, 2015." The PARSEC Benchmark Suite. Web, 29.

Christian Bienia, 2011. "Benchmarking modern multiprocessors." PhD dissertation, Department of Computer Science, Princeton University.