A Study on Cache Replacement Policies in Level 2 Cache for Multicore Processors

Thesis submitted in partial fulfillment of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

(Specialization: Computer Science)

by

Priyanka Bansal

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela, Odisha, 769 008, India

June 2014

A Study on Cache Replacement Policies in Level 2 Cache for Multicore Processors

Thesis submitted in partial fulfillment of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

(Specialization: Computer Science)

by

Priyanka Bansal

(Roll- 212CS1087)

Supervisor

Dr. Bibhudatta Sahoo

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela, Odisha, 769 008, India

June 2014

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India.

Certificate

This is to certify that the work in the thesis entitled ***A Study on Cache Replacement Policies for Level 2 Cache in Multicore Procesors*** by ***Priyanka Bansal*** is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology with the specialization of Computer Science in the department of Computer Science and Engineering, National Institute of Technology Rourkela. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Place: NIT Rourkela
Date: 2-June 2014

Dr. Bibhudatta Sahoo
CSE Department
NIT Rourkela
Odisha

Acknowledgment

I am grateful to numerous local and global peers who have contributed towards shaping this thesis. At the outset, I would like to express my sincere thanks to Assistent Prof. Bibhudatta Sahoo for his advice during my thesis work. As my supervisor, he has constantly encouraged me to remain focused on achieving my goal. His observations and comments helped me to establish the overall direction of the research and to move forward with investigation in depth. He has helped me greatly and been a source of knowledge.

I am very much indebted to Prof. S.K Rath, Head-CSE, for his continuous encouragement and support. He is always ready to help with a smile. I am also thankful to all the professors of the department for their support.

I am really thankful to my all friends. My sincere thanks to everyone who has provided me with kind words, a welcome ear, new ideas, useful criticism, or their invaluable time, I am truly indebted.

I must acknowledge the academic resources that I have got from NIT Rourkela. I would like to thank administrative and technical staff members of the Department who have been kind enough to advise and help in their respective roles.

Last, but not the least, I would like to dedicate this thesis to my family, for their love, patience, and understanding.

Priyanka Bansal

Abstract

Cache memory performance is an important factor in determining overall processor performance. In a multi core processor, concurrent processes resides in main memory uses a shared cache. The shared cache memory reduces the access time, bus overhead, delay and improves processor utilization. The performance of the shared cache depends on the placement policy, block line size, associativity, replacement policy and write policy. Every application has a specific memory demand for execution. Hence the concurrent applications with a processor compete with each other for the shared cache. The traditional Least Recently Used (LRU) cache replacement policy considerably degrade the cache performance when the working set size is greater than the size of shared cache. In such cases the performance of the shared cache can be improved by selecting an appropriate shared cache size with an efficient cache replacement policy. Finding an optimal cache size and replacement policy for a multicore processor is a challenging task. For the shared cache management in a multicore processor, the cache replacement policy should be such that, it will make efficient use of available cache space and make some cache line available for the longest time. We have analyzed the variation of shared cache size and its associativity over hit rate, effective access rate and efficiency in single, dual and quad core processor using multi2sim with splash-2 benchmark. We have proposed a novel cache configuration for a single, dual and quad core system. This research also suggests a new Bit set insertion, replacement policy for thrashing access pattern for dual and quad core system. The Bit set insertion policy considering the miss rate with the shared cache of size = 128kb is reduced by 15 % for FFT application and 20 % for LU when compared with the Least Recently Used cache replacement policy in a dual core system. For quad core system for the shared cache of size=512 KB, the miss rate is reduced by 21 % for FFT application and 24 % for LU decomposition over Least Recently Used cache replacement policy using multi2sim with splash-2.

Keywords: *shared cache size; cache replacement policy; multicore; thrashing*

Contents

List of Figures

List of Tables

List of Abbreviations

SRAM	Static Random Access Memory
DRAM	Dynamic Random Access Memory
FIFO	First In First Out
CPU	Central Processing Unit
LRU	Least Recently Used
MRU	Most Recently Used
CMP	Chip Multiprocessor
IPC	Instruction per cycle
UAR	Unique Address References
l2	level 2 cache
l1	level 1 cache

Chapter 1

Introduction

Chapter 1
Introduction

1.1 Introduction

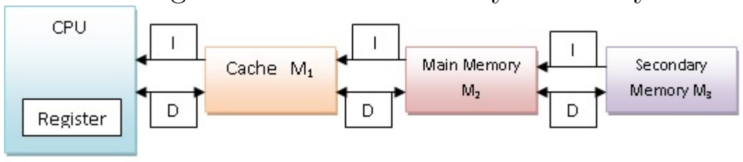
In the communication between CPU and main memory, the main issue is speed mismatch which results in poor processor utilization [1], [2]. Hence, to improve the processor utilization a fast memory cache is introduced [1]. Cache is an important component of the memory hirearchy. It *reduces the access time, bus overhead, delay and improves processor utilization* [2], [1]. There are five design parameter's for cache [3]: cache size, cache line size, placement policy, replacement policy, and write policy.

Importance of cache replacement policies Replacement Policy is an important parameter for the cache, it is the method of selecting the block to be deallocated and replaced with the incoming cache line.The basic replacement policy used are LRU, FIFO and Random [1], [3], [2]. The replacement policy is responsible for the efficient use of available memory space by making a place for the incoming line through deallocating one of the cache lines [1].It is called as primitive allocation [1]. It also reduces the miss rate or increases the hit rate for the cache. The performance of processor system is calculated in terms of hit or miss rate [1], this implies the replacement policy affects the overall performance of processor. Increase in cache size will increase the hit rate but if it is increased beyond a certain limit for a particular memory organisation, then it will degrade the performance as proved by amdhal's law in [4] and also increase the cost [1], [3] but by changing the cache replacement policy keeping the cache size fixed, we can increase the overall efficiency and hit rate of system.

Cache policy with multicore processors Multi-core processor has low power consumption, less heat dissipation and less space usage on the die which resuts in good performance in many application. But sharing of cache between more than one processing unit (core) present on the same die causes poor cache utilization, hence poor system performance. Therefore, to minimize that we are changing the cache replacement policy.Moreover, As the number of cores varies the cache size and replacement policy also varies [5], [6], [7].

Cache: A place to store data temporarily is called as cache [8] and the cache memory is a smaller, faster intermediate memory within a large memory hierarchy. Cache can be, a split cache means seperate cahce for data and instruction,unified cache for processor, browser cache of the internet, disk cache for disk storage, or local server cache for LAN servers. Cache is designed to speed up the processor by prioritizing the contents for quick access [9], [8]. The cache is organized in the memory hierarchy [1], [2] as shown:

Figure 1.1: Cache Memory Hierarchy



A different types of memory form a computer's memory hierarchy in which each memory unit is subordinate to its higher level memory unit. The goal for this memory organization is to have a good tradeoff between speed, cost, and storage capacity.The technologies used for these different memory units are [1], semiconductor SRAM's for cache, semiconductor DRAM's for main memory and magnetic disk for secondary memory.The reasons for introducing memory hierarchy, CPU speed is much faster than the memory [3], bus overhead, delay, less processor utilization.

he relation between two adjacent levels of memory hierarchy [1] i.e m_j and $m_j + 1$:

- *cost per bit:* $c_j > c_j + 1$
- *access time* : $t_j < t_j + 1$
- *storage Capacity:*$s_j < s_j + 1$

The central processing unit(CPU) directly communicate with the level 1 of memory hierarchy and so on. In order to increase the CPU utilization this memory hierarchy is introduced. To read the data from some memory level m_j the sequence followed by CPU is [1] as:

$$\begin{gathered} m_j - 1 := m_j; \\ m_j - 2 := m_j - 1; \\ \ldots\ldots ; \\ m_1 := m_2; \\ CPU := m_1 \end{gathered}$$

During program execution CPU generates a memory address. If the address is present at level $m_j (j \neq 1)$ then the address is reassigned to m_1. The reallocation of address means that the transfer of data between m_j and m_1.To improve CPU utilization the address must be present in m_1 and if not then reallocation of storage is made.

1.2 Literatue review

Cache performance is an important factor in determining overall system performance [1] and cache replacement policy is one of the main factor which affects cache performance [10].Replacement policy is the method of selecting the block to be deallocated and replaced with the incoming cache line. The basic replacement policy used are LRU, FIFO and Random [1], [3], [2].

Replacement policies with single core The most commonly used policy in cache management is LRU(Least recently used) [3] , [11] but it also suffers from some problems as discussed in [12]. The LRU policy shows less hit rate for particular application with significant increase in cache size and its inability to cope with different access pattern ,weak locality . [12].

In [13] Qureshi,Jaleel etal. discuss a solution to the problem of LRU policy to thrashing for memory intensive workloads that have a working set greater than the available cache size.They proposed LIP(LRU Insertion Policy) which places incoming line in LRU position instead of MRU position and BIP(Bimodal Insertion Policy) it accepts changes in working set while executing thrashing application and finally DIP (Dynamic Insertion Policy) which chooses dynamically between BIP and traditional LRU depending upon fewer misses.These polices does not require any hardware change to cache structure and require storage capacity less than 2 bytes.

In [14] Jaleel,Theobald,Steely and Emer attempt to implement optimal replacement policy by predicting the re-reference interval of cache block.LRU only deals with the near - immediate re-reference interval but Jaleel et.al consider distant and intermediate re-reference interval also .They proposed an algorithm SRRIP(Static Re-reference Interval Prediction) which is scan-resistant and DRRIP (Dynamic Re-reference Interval Prediction)i.e both scan and trash resistant .These policies requires 2-bits extra per cache block and can easily get integrated with LRU.In [14] they discussed about NRU(Not Recently Used) replacement policy which inserts the incoming line at LRU position instead of MRU.

Mazrouee and Kakoee [15] proposed a Modified pseudo LRU which is a novel block replacement policy and reduces the complexity of hardware implementation of LRU.

Chaudhuri [16] introduces a pseudo-LIFO(last in first out) , a new family of replacement heuristics for managing each cache set as a stack (opposed to the traditional access recency stack).It improves the performance of LLC(last level cache) with the increases in capacity and associativity.

There are some other polices which are implemented to improve the performance of cache [[17]] proposes LRFU (least recently frequently used)combines both LRU and LFU(least frequently policy) , [[18]] by its algorithm 2Q increases the performance of LRU by a constant additive factor , [[19]] proposes CAR(Clock with adaptive replacement) improves recency,constant time , scan-resistance ,frequency and Lock contention/MRU Overhead ,[[12]] proposes LIRS(Low inter-reference recency set) which deals with the inability of LRU for access pattern with weak locality and it uses recency to evaluates the inter-reference recency for making replacement decision.

Kedzierski and Moreto etal. [20] proposed a complete partitioning system using pseudo -LRU replacement policy. In this through there algorithm they overcome the complexity and area overhead of LRU policy implemented on LLC (last level cache) with high associativity.

In [21] Wong and Baer proposed an algorithm for the detection of temporal locality in the Level 2 cache with two new strategy profile based and on-line scheme.It basically deals with the high associativity in case of level 2 cache(last level cache).

Replacement policies with multi core Qureshi and Patt [5] proposed a new algorithm UCP(utility-based cache partitioning) which is a low overhead, runtime mechanism that partitions a shared cache between different applications on the basis of reduction in misses that each application is likely to get from available cache resources.It requires a hardware circuit of storage space less than 2kB.This algorithm overcome the limitation of LRU policy of demand based partitioning which may cause poor performance for some applications.

In [6] Jaleel etal. proposed an algorithm for the CMPs(Chip Multiprocessor) that allows different application run on single chip .Due to single shared cache between these applications it causes thrashing which gives poor result for LRU. Hence Jaleel proposed a new algorithm Thread -Aware Dynamic insertion policy i.e extension to DIP .It requires storage overhead less than 2 bytes per core.

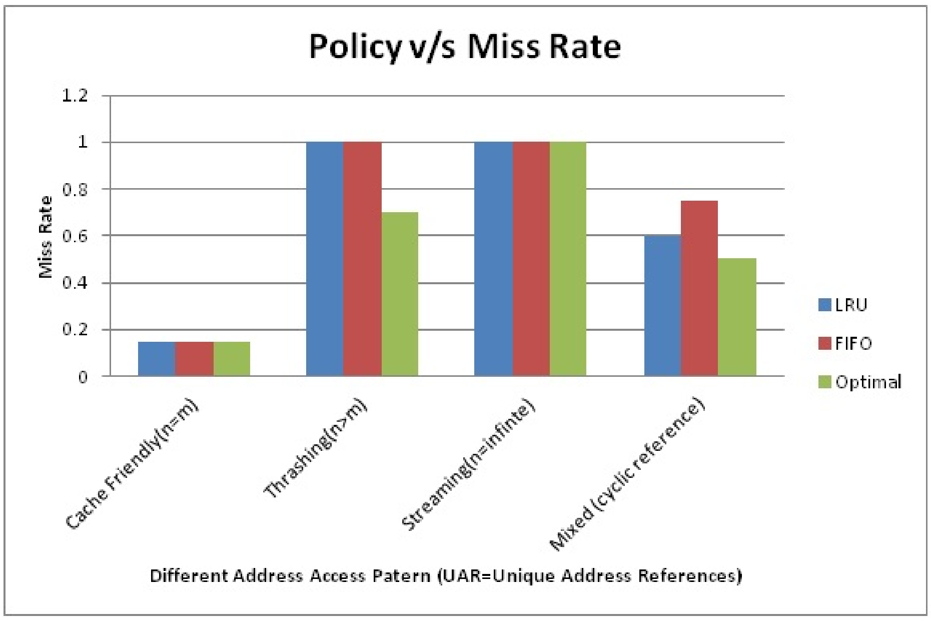
In [7] Xie and Loh proposed a approach that combines both dynamic insertion and promotion polices of cache partitioning , adaptive insertion and capacity stealing for cache management. It basically advantages to the shared last level cache for the multiple cores.

In [22] proposes TAP-TLP (TLP(thread level parallelism) aware cache management policy) and [23] proposes heterogeneous cache management policy for heterogeneous architecture for efficient utilization of shared cache.

1.3 Motivation

In multicore processor the performance of the shared cache can be improved by taking appropriate cache size with an efficient cache replacement policy which manages the shared cache in multi core processor. The analysis of the cache size and associativity on single and multi core processor is done to improve the processor utlization by reduceing its access time and miss rate. There are four access patterns [6, 13, 14]: *cache friendly, thrashing, steaming, mixed.*Thrashing occur in cache when the size of cache is less than working size of problem. To overcome the high miss rate problem in thrashing (cache size more than the working set size) application with traditional LRU policy [6, 13, 14] :

Figure 1.2: Access Patterns v/s Miss rate



As shown in the Fig 1.2 where *n represents the unique addresss references and m represents the number of cache line,* for the thrashing access pattern the LRU gives all misses where as the optimal cache replacement gives less miss rate. Hence, the gap between the LRU and optimal replacement policy can be bridge by changing the existing policy so that it works well for thrashing trashing access patterns. The performance of the processor can be improved by changing the existing replacement policy for multicore processors.

1.4 Objective

The objective of this thesis is to improve the performance of multicore processor by improving the efficieny of cache. By changing the cache size for single and multicore procesoors we have tried to find the optimum cache size with maximum hit rate and efficency for single core and multicore processors and to implement a new cache replacement policy, *Bit Set Insertion Replacement Policy(BSIRP)* for thrashing application in multicore processor so that it gives the lower miss(access) for thrashing access pattern and improves the performance of multicore processor.

.

1.5 Problem statement

In multicore processors, the efficiency of shared cache plays a vital role in deciding overall system performance. If s bits can store in a memory then the total memory space would be 2^s, this address can map to a cache of size 2^b with associativity 2^m. To improve the cache performance first we are finding appropriate *b and m* for single core, dual core and quad core system.
Secondly, when the shared cache size is less than the problem working size, then by manipulating the cache replacement policy the performance of cache is improved. In a particular cache, assume Level 2 cache is containing m block lines with 2^m associativity. When a program Y executes, it generates y_j, z_j Unique Address References(UAR) [1] where j=(1,2......n) and it represents block line address.
n= Number of distinct address lines.
The temporal sequences [6,13,14]with unique address references that repeats itself x times in cache are
$(y_1, y_2, ...y_n)^x, (z_1, z_2, ...z_n)^x$

Our problem is defined for thrashing accesss pattern in cache i.e $\underline{m < n}$ and $\underline{x >> n}$. We are considering the thrashing problem in cache and proposing a cache replacement policy for a thrashing access pattern. We have analyzed different cache sizes with different associativity over different organization for finding appropriate cache size.Here we have also used different replacement polices Least Recently Used(LRU) ,First In First Out(FIFO) ,Random to compare with a new replacement policy, for maximizing the Cache Utilization.

1.6 Research contributions

This research work contains two contributions :

1. Cache configuration for single and multicore processor.

2. A Cache replacement policy for thrashing access pattern in multicore processor.

1.7 Thesis organization

The rest of the thesis is organized as follows chapter-2 summarizes the cache configuration for the processors and cache replacemnt policies, Chapter-3 summarizes the differnt organisations, varriation of cache size with associativity , hitratio and efficency , chapter-4 provides the new approch named *Bit Set Insertion Policy* to solve the replacemnt problem in cache, chapter-5 provides the Conclusion and Futurework.

Chapter 2

Cache memory organisation for the processors

Chapter 2

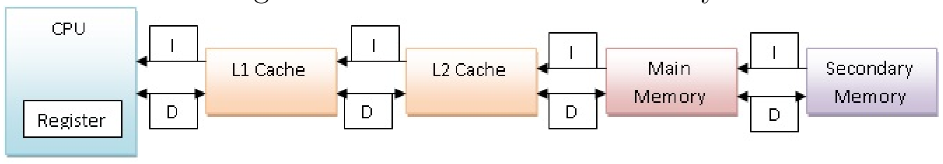
Cache memory organisation for the processors

2.1 Introduction

In memory hirerachy, level 2 cache is introduced between main memory and level 1 cache to reduce the access time (the time taken to retrieve data from storage (main memory) and make it available to computer).Level 1 cache is a split cache and Level 2 cache is unified cache. Access time may be called as delay or latency.A computer memory hierarchy for level 2 cache is as shown in fig: [1]

The level 2 cache is introduced to improve the system performance. This is in-

Figure 2.1: Level 2 Cache Hierarchy



troduced between the main memory and level 1 cache [9], [24].For any request the processor will first check level 1 cache if it data is not there than it will access the level 2 cache and finally the main memory .The block size of the level 2 cache must be either large or equal to the block size of level 1 cache as if miss occur in level 1 cache than one or more second level cache block can accommodate into level 1 cache block but i.e is not feasible.

Access time for level 2 cache We will consider an example to compute average access time for level 2 cache and will prove that the access time for level 2 cache is much lower than the access time for main memory.

Why level 2 cache is considered: Assume T_h(access time of level 1 cache) $= 2ns$,
P_hc(hit rate for level 1 cache) $= 75\%$,
T_h(access time of level 2 cache) $= 10ns$,
P_hc(hit rate for level 2 cache) $= 80\%$,

T_m(access time of main memory) $= 60ns$,
P_mc(miss rate for level 1 cache)$= (100 - 80 = 20\%)$,
P_mc(miss rate for level 1 cache)$= (100 - 75 = 25\%)$,
P_hm(hit rate for main memory)$=99\%$,
P_hm(miss rate for main memory)$=1\%$,
T_s(access time for secondary memory)$= 10ms(millisecond)$

Hence, the average access time for the request that reach to main memory main is:
$EAT\_mainmemory = (60ns * 0.99) + (0.01 * 10ms)$
$EAT\_mainmemory = (100059.4ns)$
The average access time for level 1 cache with main memory:
$EAT\_level1cache = (2ns * 0.75) + (0.25 * 60ns)$
$EAT\_level1cache = (16.5ns)$
The average access time for level 2 cache with level 1 cache and main memory:

$EAT\_level2cache = (2ns * 0.75) + (0.25 * ((10ns * 0.80) + (0.20 * 60ns))$
$EAT\_level2cache = (6.5ns)$

Hence , from above example it is clear that the access time for main memory

is approx 100% more than the access time for level 1 cache and for Level 2 cache. The access time level 2 cache is further reduced. Hence,level 2 cache is added to memory hirerachy.
The level 1 cache ,level 2 cache and main memory also differ is size and cost per bit i.e the size of level 1 cache is lower than size of level 2 Cache and both are much lower than the size of main memory and cost per bit is lower for the main memory than the level 1 cache and Level 2 cache [1], [3].

2.2 Performance metrics

Some important definitions

- *Throughput:* Throughput is defined as the amount of work done by computer in given period of time. In determines the performance.It is measured in terms of IPC(Instruction Per Cycle).
  For single core it is calculated as the geometric mean of the improvement of IPC from baseline system to new system [13].
  For multicore it is calculated as given in [5], [6], [7] :

  $$\text{Throughput} = \Sigma\ IPC_i$$

  IPC_i = Represents IPC of the ith application when it concurrently executes with other application.

- *Cache size:* The size of cache implies amount of data it can store. [1].Larger the size of cache ,more data it will store but with more cost.

- *Associativity:* Associativity of cache determines that how many locations with in the given cache are occupied by particular memory address [1].

- *Number of cores:* In CMP's (chip multi processors) the number of cores is an important parameter in determining performance of replacement policy.

- *Weighted speed up:* This speed is calculated as given in [6], [7], [5] :

  $$\text{Speedup} = \Sigma\ (IPC_i/SingleIPC_i)$$

  $SingleIPC_i$ = It is the IPC of the same ith application when it executes in isolation.

- *Harmonic speed up:* It balances the fairness and performance [25] .It is calculated as[[6], [7], [5]] :

  $$HSP = \frac{N}{\Sigma(IPC_i/SingleIPC_i)}$$

  HSP=Harmonic Speed Up.
  N=Threads executed concurrently.

- *Hardware overhead:* Hardware overhead for replacement polices is considered as the storage overhead [13] , [5] required by any policy if we change the design of base policy.

- *Misses per Kilo Instruction:* It is defined as the ratio of number of misses and the sum of total number of instruction. It is calculated as :

  $$MPKI = \frac{\Sigma missesofallCores}{\Sigma Instructionofallcore * 1000}$$

  MPKI=Miss per Kilo Instruction .

- *Block line size:* It is the size of the chunks of the data that are brought in and thrown out of the cache in response to miss in the cache [1].

- *Hit rate:* It is defined as the probability that the address generated by CPU refers to the information currently stored in faster cache memory [1].It is calculated as:

$$H = \frac{N_1}{N_1+N_2}$$

H =Hit Rate.

N_1=Number of references hit in cache.

N_2= Number of references hit in main memory or cache .

- Miss Rate: It is the probability of miss in cache when referred by CPU.It is calculated as:

$$H = \frac{N_1}{N_1+N_2}$$

H =Miss rate.

N_1=Number of references miss in cache.

N_2= Number of references miss in main memory or cache .

- *Effective Access Time:* It is the time to access the data from lower level cache. It is caclulated as [3], [1] :

$$T_a = (T_h P_h) + (T_m P_m)$$

  - T_h = The time taken to access request that is hit in the level,
  - P_h = The rate of hit in the level (expressed in terms of probability)
  - T_m = The average access time of the all the levels below this level in the hierarchy, and
  - P_m = The miss rate of the level

- *Efficency:* It is calculated as a ratio of:

$$Efficency = \frac{t_c}{t_m}$$

  - t_c = Cache access time

  - t_m = Main memory access time

- *Combined hit ratio:* For more than one cache the combined hit ratio is calculated as:

Combined hit ratio=Hit rate of Level 1 cache + Miss rate of Level 1$*$ Hit rate of Level 2 cache

Table 2.1: Performance Parameters for Cache and Cache Replacement Policy

Policy	Author	**Throughput**	**Cache size**	**Associativity**	**Number of Cores**	**Hardware Overhead**	**Weight Speed Up**	**Harmonic Speed Up**	**MPKI**	**Miss rate**	**Block Line Size**	**Hit Ratio**
LRU	Asit Dan(1990) [1,3,11]	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
NRU	AamerJaleel etal. (2010) [14]	✔	✔	✔	✖	✖	✔	✖	✔	✔	✖	✖
SRRIP	AamerJaleel etal. (2010) [14]	✔	✔	✔	✔	✖	✔	✖	✔	✔	✖	✖
DRRIP	AamerJaleel etal. (2010) [14]	✔	✔	✔	✔	✖	✔	✖	✔	✔	✖	✖
LIP	Moinuddin K. Qureshi etal. (2007) [13]	✔	✔	✔	✖	✖	✔	✖	✔	✖	✖	✖
DIP	Moinuddin K. Qureshi etal. (2007) [13]	✔	✔	✔	✖	✖	✔	✖	✔	✖	✖	✖
Modified LRU	Wayne A,Wong etal.(2000) [21]	✔	✔	✔	✖	✔	✖	✖	✔	✔	✔	✖
Modiefied Pseudo LRU	Hassan Ghasemzaden etal. (2006) [15]	✖	✔	✔	✖	✔	✖	✖	✖	✔	✖	✖
Psedo LIFO	Mainak Choudary(2009) [16]	✔	✔	✔	✔	✔	✔	✔	✔	✖	✖	✖
UCP	Moinuddin K.Qureshi etal.(2006) [5]	✔	✔	✔	✔	✔	✔	✔	✔	✖	✖	✖
PIPP	Yuejian Xie etal. (2009) [7]	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✖
TADIP	Aamer Jaleel etal. (2008) [6]	✔	✔	✔	✔	✔	✔	✔	✔	✖	✖	✖
Pseudo LRU	Kamil Kedzierski etal. (2010) [20]	✔	✖	✖	✔	✔	✔	✔	✖	✖	✖	✖

In the Table 2.1 we have summarized thirteen replacement policies in level 2 cache over different performance parameters. These policies considering both single and multicore system. In case of single core throghput, miss rate and associativity are varied to analyze the performance of cache. With multicore they have considered miss rate, throghput and speed ups to analyze its performance. For our research work, we have considered following parameters: *cache size, efficency, access time, hit ratio, associativity,Block line size* to analyze the cache configuration for single core, dual core and multi core and for cache replacement policy we have considered **throughput,miss rate ,number of cores, hit rate, cache size** to analyze its performance.

2.3 Access patterns for cache

During a program execution a memory is accessed in a particular sequence called as access pattern. There are four access patterns [14], [6] cache friendly, thrashing, streaming and mixed.
Assume Level 2 cache's is containing m blocks. When a program Y executes, it generates Y_j Unique Address References(UAR) [1] where j=(1,2......n) and it represents block address.
n= Number of distinct address references.
The above four access pattern are:

- **Cache friendly:** When UAR is less than or equal to the given cache size($n \leq m$).With this condition the access pattern for all the policies will give minimum and same number of misses as the size of the cache(compulsory misses [[3]]). The illustration of particular cache pattern is as shown in example with LRU,FIFO and optimal.

- **Thrashing:** When UAR is greater than the cache size($n>m$). If this condition is true the LRU and FIFO receives zero hits(i.e all miss) [14] but optimal shows variation and receives less misses. As shown below in example.

- **Streaming:** When UAR is much greater than cache size (n= ∞). With this condition the access pattern shows no hits and number of misses equal to n as shown in graph. These type of pattern has no locality and have infinite re-reference interval [14].
- **Mixed:** UAR may be less than or greater than cache size but there is a cyclic re-reference pattern i.e UAR will repeat itself in the distant future. This type of pattern is used in most of the application containing both near and distant re-references interval [14].For this pattern LRU is showing best results in comparison to FIFO but less than optimal as shown in below example.

2.4 Existing cache replacement policies

Cache replacement policy is one of the main factor which effects cache performance [10]. In placing the cache line these cache replacement policies plays a vital role [1], [2], [3].

LRU LRU replacement policy is widely used policy. In this policy,the incoming data is sorted by ageing factor [1]. In case of cache miss, the data at the LRU position is evicted and if it is a cache hit the data is moved to the head of linklist.

Algorithm 1 Least Recently Used cache replacement algorithm

```
1: $tag \leftarrow$ tag of new $cache\ block$
2: $way = 0$
3: while $way < cache- > assoc$ do
4:     $k \leftarrow (tag == cache- > block.tag)$ or $(cache- > set- > way.way - prev)$
5:     if $k$ then
6:         move cache block to head
7:         break
8:     end if
9:     $way \leftarrow way + 1.$
10: end while
11: if $way == cache- > assoc$ then
12:     replace cache block at tail and insert the incoming block at head
13: end if
```

But LRU policy does not consider the frequency of data, it only focus on the most recently used data which degrades the system performance in case of thrashing application. LRU policy can be expensive when the set associativity is high [21]. Hardware overhead is more for LRU policy [14]. Hence, we are going to improve the LRU policy for thrashing application.

Random Random policy is a low cost technique [1]. In this policy, a block to be evicted is selected randomly. Unlike, LRU this replacement policy does not require any prior access information.

Algorithm 2 Random cache replacement algorithm

```
1: if cache miss then
2:     replace the bloock at $(random()\%cache- > assoc)$
3: end if
```

This policy suffers from very less delay and hardware overhead [1]. In case of thrashing application works better than LRU.

MRU In this policy the most recent block is evicted for a cache miss. This policy is good when older data is more likely to be accessed in future.

This policy is good for thrashing application when the old data is expected to be accessd in the distant future.

2.5 Tool and benchmark

We have used multi2sim for simulation work.Multi2sim [26] is a heterogenous open source Simulator.It is capable to model superscaler pipelined processor,

Algorithm 3 Most Recently USED cache replacement algorithm

```
1: $tag \leftarrow$ tag of new *cache block*
2: $way = 0$
3: **while** $way < cache- > assoc$ **do**
4:     **if** $tag == cache- > block.tag$ **then**
5:         *move cache block(tail)*
6:         ***break***
7:     **end if**
8:     $way \leftarrow way + 1.$
9: **end while**
10: **if** $way == cache- > assoc$ **then**
11:     *replace cache block at tail and insert the new block at head*
12: **end if**
```

GPUs(Graphical Processing Unit) and multithreaded and multicore architecture. It supports the most common real time benchmarks. Memory hierarchy configuration and interconnection network is highly flexible. We can define as many cache level as needed and cache cohrence is maintained using a protocol MOESI(Modiefied, Owner, Exclusive, Shared, Invaild). Write back policy is used. Cache memory can be split in to data and instruction cache memories.Due to its flexibility towards the memory configuration we have choosen multi2sim.

We have used splash- 2 benchmark. It is a suite of parallel applications. It is used to provide the studty of address-space replated to multiprocessors [27]. We have used Baren, FFT(Fast Fourier Transform) and LU (Lower Upper) application from splash-2 benchmark suite. Baren simulates the intraction of number of bodies in three dimension over number of time stamp, FFT is used to optimize the interprocessor communication and the input is (sqrt(n) * sqrt(n)) matrix for the dataset of n data points, LU kernal factors a dense matrix (n*n) i.e the product of lower and upper triangular matrix.

2.6 Summary

In this chapter, we have seen different performance parameters and access patterns for cache and cache replacement policy. Cache are used to improve the procesoor utlization and to increase its efficency. Here we have considered cache and cache replacement policy. Efficency of processor can be improved by reducing the access time for CPU. Different Cache replacement policies have explained.

Chapter 3

Cache configuration for single and multicore processors

Chapter 3

Cache configuration for single and multicore processors

3.1 Introduction

Cache Configuration plays a vital role in designing any processor. The performance of the processor depends on the three factors [1, 2]speed,cost and capacity. For a good processor there must be a balance in all these three factors. As we go from cache to secondary memory in memory hierarchy the cost reduces and access time increases(speed decreases). Hence, for a particular processor we must take a cache configuration which is less in cost and gives maximum hit rate(speed of a processor depends on hit rate [1]).

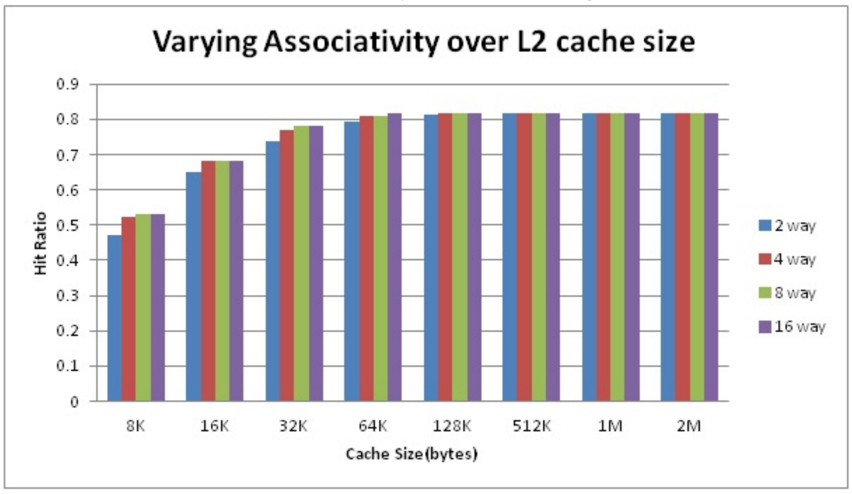
For simulating different cache for single , 2 and 4 core syatem we have used Multi2sim [26] and Splash-2 [27] benchmark is used. Splash-2 benchmark suite containing real time parllel application. In order to analyze the different cache configuration we have taken Baren and FFT application of splash-2 suite. All t he experiments were run on system with 32 bit linux (Operating system) on intel core i3 processor.

3.2 Cache configuration for single core processor

In deciding the cache configuration for processor we have considered Hit ratio to analyze the performance of processor.

1. We have analyzed the variation associativity over cache size as follow as:

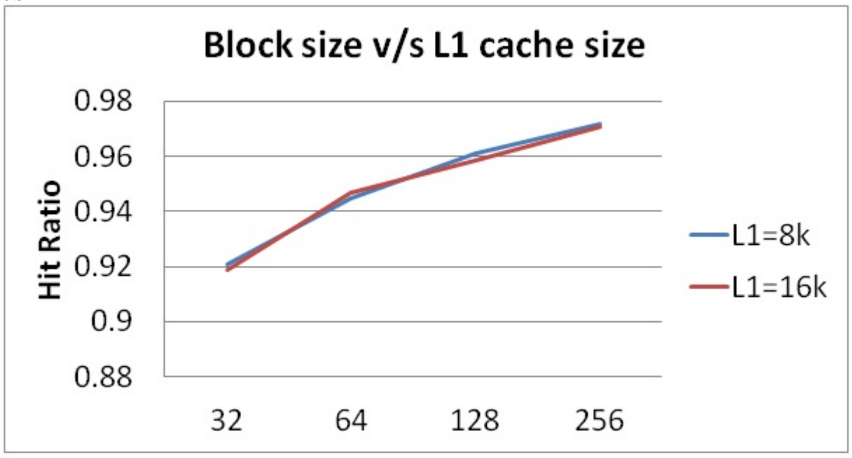
Figure 3.1: Hit ratio with associativity on executing baren for different cache size



In the figure 3.1 by varying hit ratio on y axsis and x axsis is representing L2 cache size with associativity, we observed that there is no change in hit ratio after 128kb for single core and not much improvement from 4 to 8 or 16 way associativity. Hence, we have considered 4 way associativity for L2 cache.

2. The variation of block line size with L1 cache size in single core system :

Figure 3.2: Hit ratio with block line size on executing baren for different L1 cache block lines

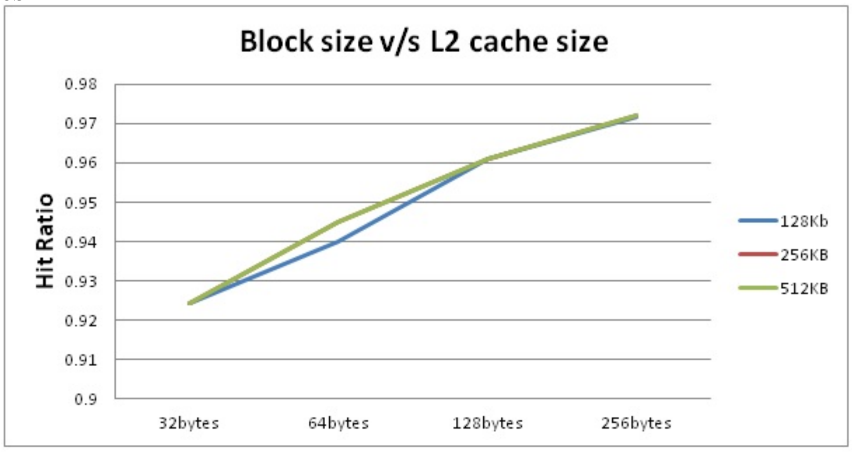


In the figure 3.2 by varying hit ratio on y axsis and x axsis is representing

L1 block line size in bytes with 8kb and 16kb L1 cache sizes, we observed that the hit ratio increases the with increase in block line size and maximum at 256b. Hence, we have considered the block line size 256b for L1 cache.

3. The variation of block line size for L2 cache size in single core system:

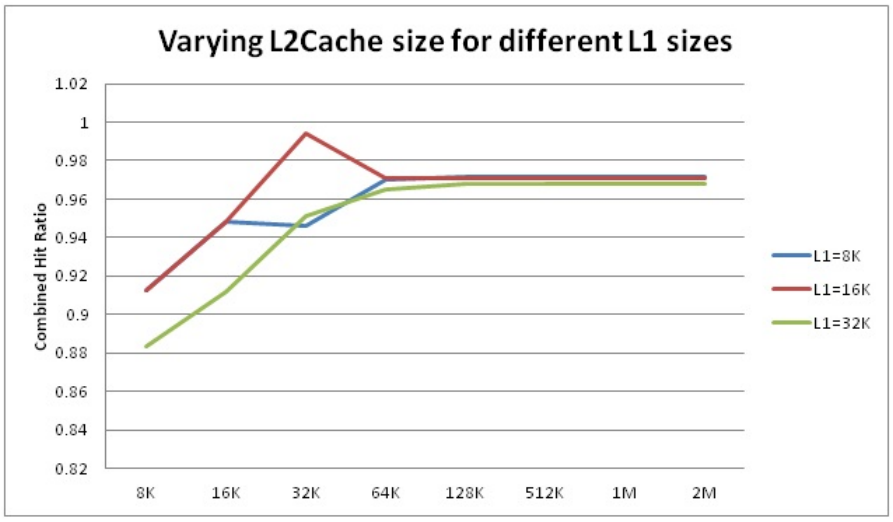
Figure 3.3: Hit ratio with block line size on executing baren for different L2 cache block lines



In the figure 3.3 by varying hit ratio on y axsis and x axsis is representing L2 block line size in bytes with 128kb, 256kb and 512kb L2 cache sizes, we observed that the hit ratio increases the with increase in block line size and maximum at 256b. Hence, we have considered the block line size 256b.

4. The analysis of combined hit ratio on increasing the cache sizes in single core system.

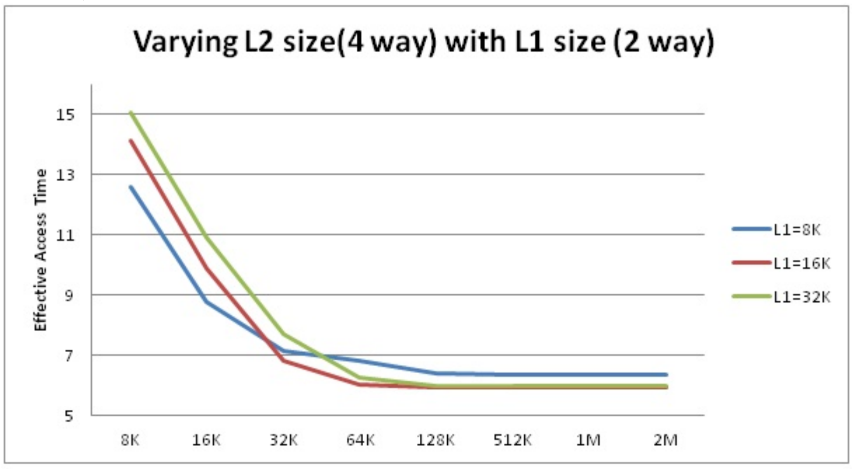
Figure 3.4: Combined hit ratio on executing baren for L1 cache sizes (8kb,16kb,32kb) in single core system



In the figure 3.4 by varying combined hit ratio on y axsis and x axsis is representing L2 cache size in kilo and mega bytes with 8kb, 16kb and 32kb L1 cache sizes, we observed that the Combined hit ratio is approx same for L1-8,16 or 32 kb after 128 kb L2 cache. Hence, there is conflict for the apropriate size for L1 of cache.

5. To decide the apropriate size for L1 cache we have considered effective access time. The variation of access time with different cache sizes:

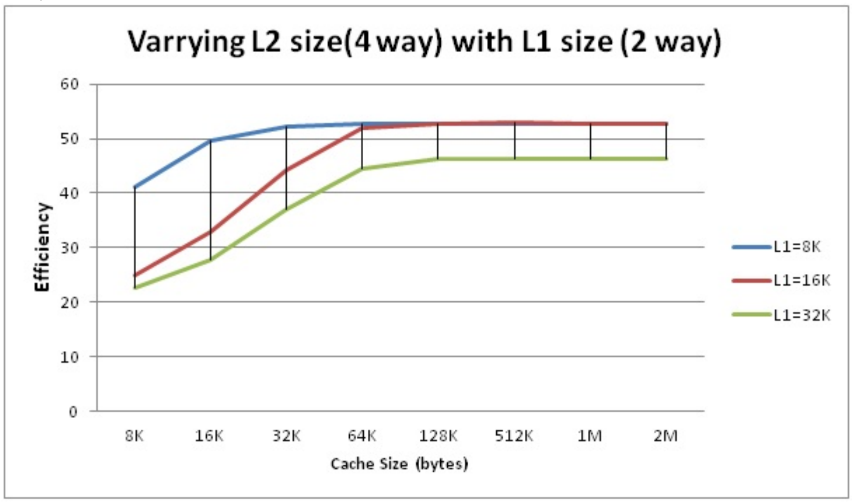
Figure 3.5: Effective access time on executing baren for L1 cache sizes (8kb,16kb,32kb) in single core system



In the figure 3.5 by varying effective access time on y axsis and x axsis is representing L2 cache size in kilo and mega bytes with 8kb, 16kb and 32kb L1 cache sizes, we observed that the effective access time is less for the L1 = 16k or 32k than 8k but it is approx same for both of them as the cache memory is very costly [1] so we go for L1= 16k and L2= 128kb.

6. Atlast we have analyzed the varriation of L2 cache size with efficency :

Figure 3.6: Efficiency on executing Baren for L1 cache sizes (8kb,16kb,32kb) in single core system



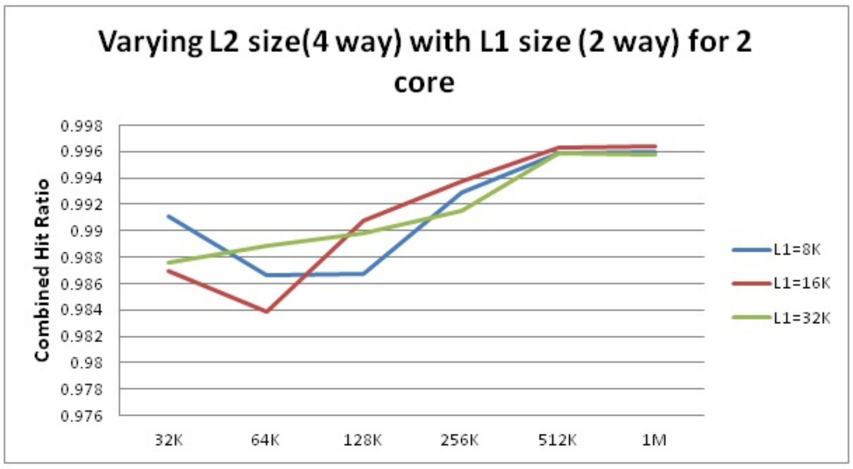
In the figure 3.6 by varying efficiency on y axsis and x axsis is representing L2 cache size in kilo and mega bytes with 8kb, 16kb and 32kb L1 cache sizes, we observed that the efficiency is maximum with L1=16kb and L2=128kb same as in case of effective access time.

3.3 Cache configuration for dual core processor

In deciding the cache configuration for a processor we have considered Hit ratio, effective access time and efficency to analyze the performance of a dual core processor system.

1. The analysis of combined hit ratio on increasing the cache sizes in dual core system.

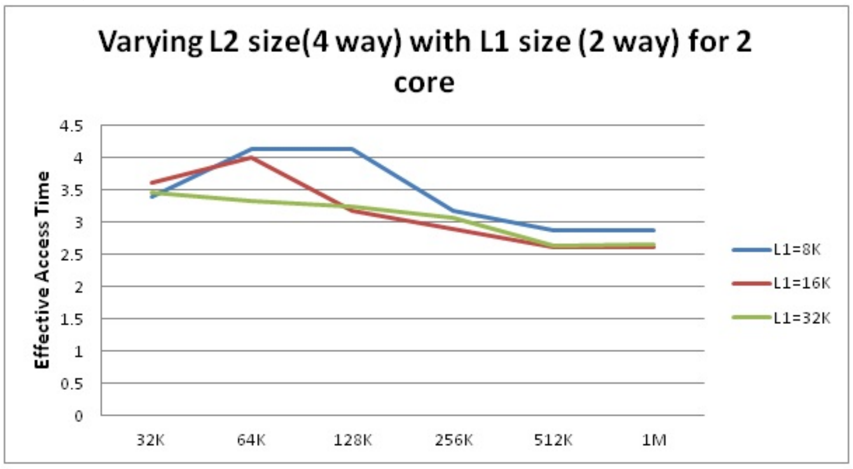
Figure 3.7: Combined hit ratio on executing FFT for L1 cache sizes (8kb,16kb,32kb) in dual core system



In the figure 3.7 by varying combined hit ratio on y axsis and x axsis is representing L2 cache size in kilo and mega bytes with 8kb, 16kb and 32kb L1 cache sizes, we observed that the combined hit ratio is approx same for L1-16 or 32 kb after 512 kb L2 cache. Hence, there is conflict for the apropriate size for L1 of cache.

2. To decide the apropriate size for L1 cache we have considered effective access time. The variation of access time with different cache sizes:

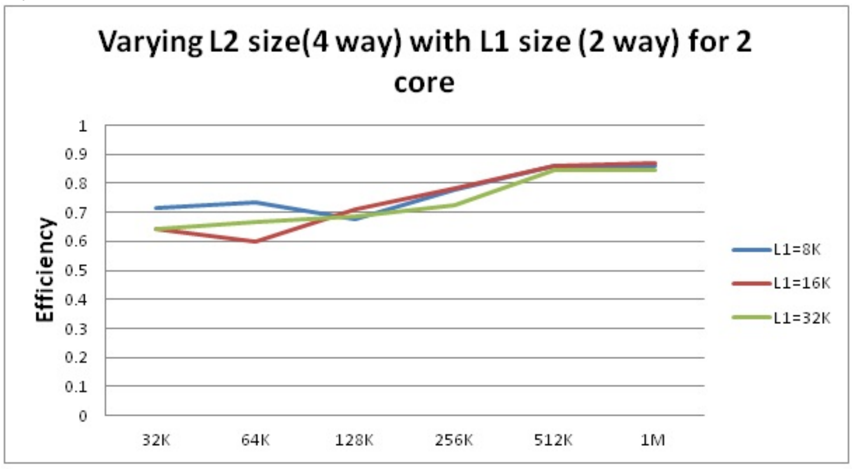
Figure 3.8: Effective access time on executing FFT for L1 cache sizes (8kb,16kb,32kb) in dual core system



In the figure 3.8 by varying effective access time on y axsis and x axsis is representing L2 cache size in kilo and mega bytes with 8kb, 16kb and 32kb L1 cache sizes, we observed that the effective access time is least and approx same for both the L1 = 16k and 32k but as the cache memory is very costly [1] so we go for L1= 16k and L2= 512kb.

3. Finally, we have analyzed the varriation of L2 cache size with efficency :

Figure 3.9: Efficiency on executing FFT for L1 cache sizes (8kb,16kb,32kb) in dual core system



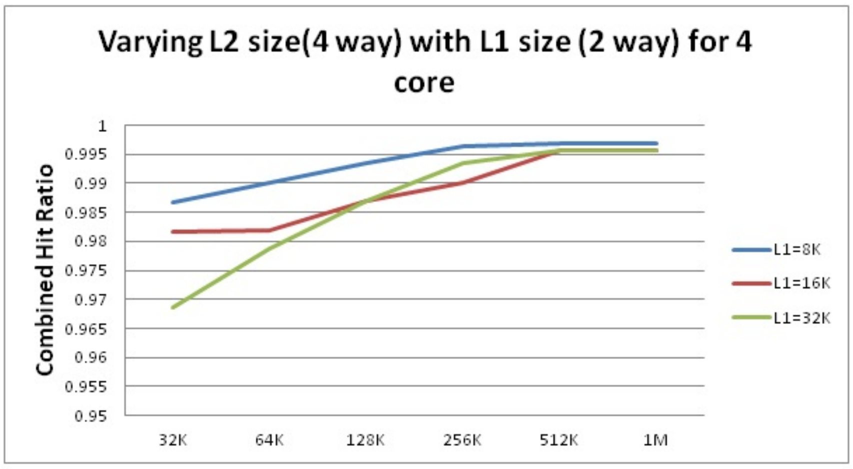
In the figure 3.9 by varying efficiency on y axsis and x axsis is representing L2 cache size in kilo and mega bytes with 8kb, 16kb and 32kb L1 cache sizes, we observed that the efficiency is maximum with L1=16kb and L2=512kb.

3.4 Cache configuration for quad core processor

In deciding the cache configuration for a processor we have considered Hit ratio, effective access time and efficency to analyze the performance of a quad core processor system.

1. The analysis of combined hit ratio on increasing the cache sizes in quad core system.

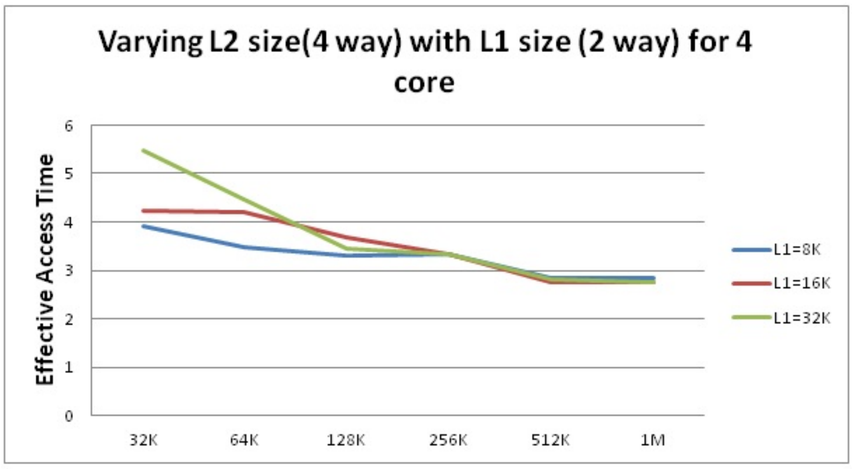
Figure 3.10: Combined hit ratio on executing FFT for L1 cache sizes (8kb,16kb,32kb) in quad core system



In the figure 3.10 by combined hit ratio on y axsis and x axsis is representing L2 cache size in kilo and mega bytes with 8kb, 16kb and 32kb L1 cache sizes, we observed that the effectiveThe Combined hit ratio is maximum for L1-8kb and after L2-1Mb cache size is approx same.

2. To decide the apropriate size for L1 cache we have considered effective access time. The variation of access time with different cache sizes:

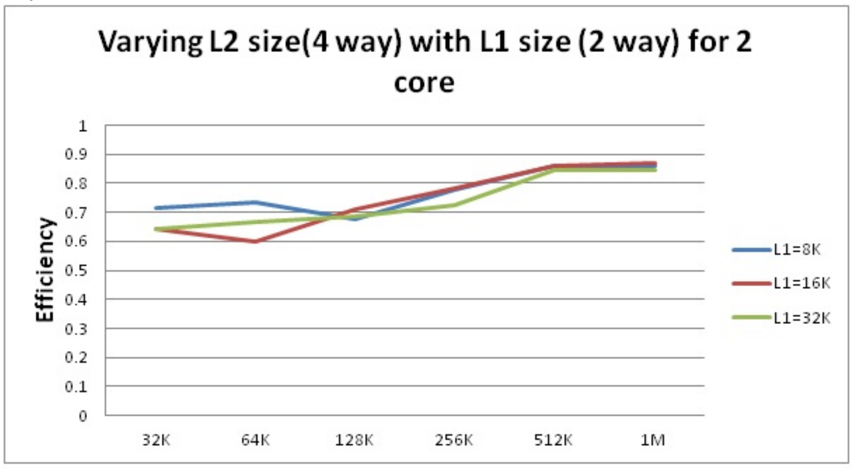
Figure 3.11: Effective access time on executing FFT for L1 cache sizes (8kb,16kb,32kb) in quad core system



In the figure 3.11 by varying effective access time on y axsis and x axsis is representing L2 cache size in kilo and mega bytes with 8kb, 16kb and 32kb L1 cache sizes, we observed that the effective access time is also showing the same behaviour as shown by combined hit ratio i.e is is good to consider L1= 8k and L2= 1Mb for 4 core system.

3. Finally, we have analyzed the varriation of L2 cache size with efficency :

Figure 3.12: Efficiency on executing FFT for L1 cache sizes (8kb,16kb,32kb) in quad core system



In the figure 3.12 by varying efficiency on y axsis and x axsis is representing L2 cache size in kilo and mega bytes with 8kb, 16kb and 32kb L1 cache sizes, we observed that the efficiency is maximum with L1=8kb and L2=1Mb.

3.5 Proposed cache configuration for single, dual and quad core system

Table 3.1: Proposed cache configuration for single and multicore system

Number of cores	L1 Data Cache	L1 Instruction Cache	L2 shared cache
Single core	size- 16kb	size- 16kb	size- 256kb
	assoc- 2way	assoc- 2way	assoc- 4way
	latency- 2	latency- 2	latency- 10
	policy- LRU	policy- LRU	policy- LRU
2 core	size- 16kb	size- 16kb	size- 512kb
	assoc- 2way	assoc- 2way	assoc- 4way
	latency- 2	latency- 2	latency- 10
	policy- LRU	policy- LRU	policy- LRU
4 core	size- 8kb	size- 8kb	size- 1Mb
	assoc- 2way	assoc- 2way	assoc- 4way
	latency- 2	latency- 2	latency- 10
	policy- LRU	policy- LRU	policy- LRU

In table 3.1, we have summarized all the simulation results. Level1 cache we have taken 2 ways associative and level2 4 way associative as it give maximum hit rate in single core, dual core and quad core system. 256 bytes block line size give maximum hit rate in single core, dual core and quad core system. For single core system the l1 cache size =16kb and l2 cache size =256kb gives optimum result. For dual core system the l1 cache size =16kb and l2 cache size =512kb gives optimum result. For quad core system the l1 cache size =16kb and l2 cache size =1 Mb gives optimum result.

3.6 Summary

In this chapter, we have seen analyzed different cache configuration for single and multicore processors. On varying cache size with combined hit ratio,effetive access time and efficiency in single core system we observed that it performs better for *L1 size= 16kb and L2 size = 128kb* than other configuration. On varying cache size with combined hit ratio,effetive access time and efficiency in dual core system we observed that it performs better for *L1 size= 16kb and L2 size = 512kb* than the other configuration. On varying cache size with combined hit ratio,effetive access time and efficiency in quad core system we observed that it performs better for *L1 size= 8kb and L2 size = 1Mb* than other policies.

Chapter 4

A cache replacement policy for thrashing application in multicore processors

Chapter 4

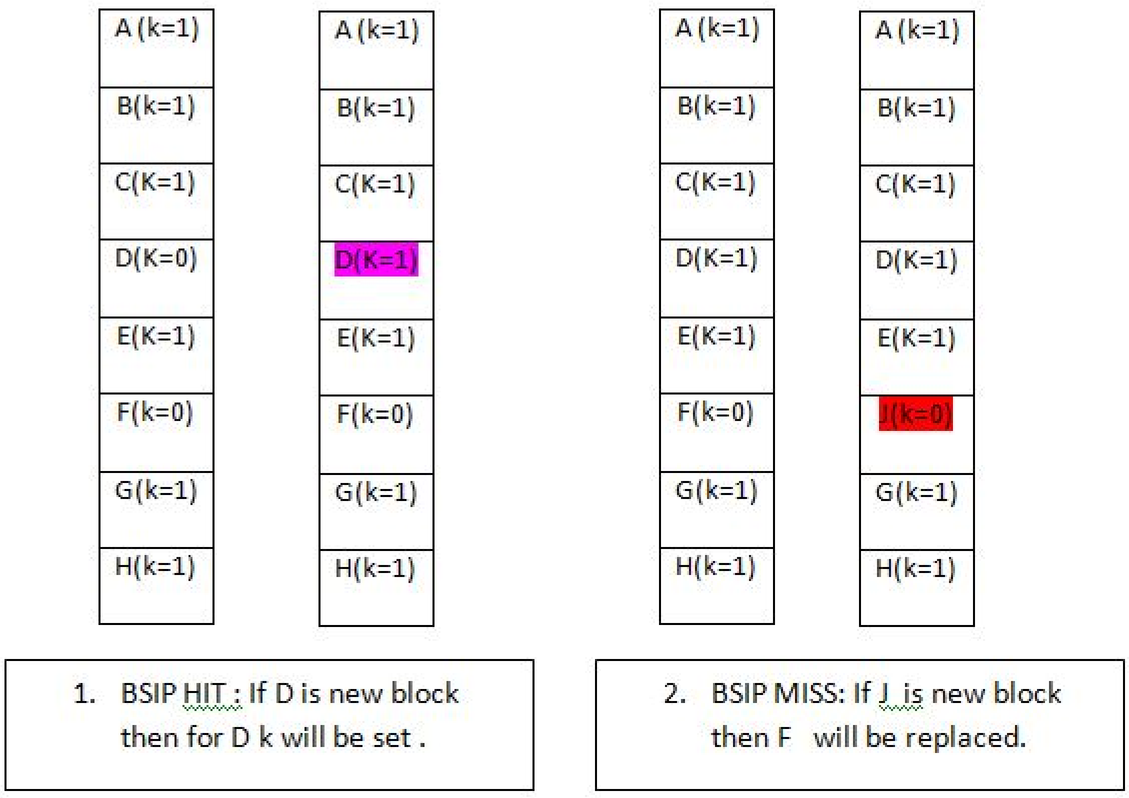
A cache replacement policy for thrashing application in multicore processors

4.1 Introduction

Replacement Policy is an important parameter for the cache, it is the method of selecting the block to be deallocated and replaced with the incoming cache line.The basic replacement policy used are LRU, FIFO and Random [1], [3], [2]. The replacement policy is responsible for the efficient use of available memory space by making a place for the incoming line through deallocating one of the cache lines [1].

In the new replacement policy we are addressing the thrashing problem i.e when the cache size is less than the working size of application. As discussed in chapter we have considered: $n > mandn << x$. In Bit Set Insertion Policy (BSIP) our concept is to make some of the cache line stay for longer time so, that they would be hit in distant future. In this policy we have tried to overcome the drawback of LRU policy with thrashing application. In BSIP we have taken one extra tag bit **k** per cache block line. Our concept in BSIP is that if there is a hit in cache then set the bit k for that cache line this implies that this block may get hit in distant future hence, it will stay in cache for longer time. If miss occur for particular access in the cache than search for first reset bit i.e(k=0) and replace the corresponding cache line with incoming block line and if all the cache lines are set in a particular set then replace block at LRU position and reset its k bit. As shown in fig 4.1, if there is a hit than the corresponding bit will be set and if miss will occur than starting from MRU position the first k=0 will be replaced and if all the bit are set for all cache lines than for next 50% of n cache line k bit will be reset and block at LRU position is replaced..

Figure 4.1: BSIP Policy HIT and MISS



4.2 Proposed Algorithm

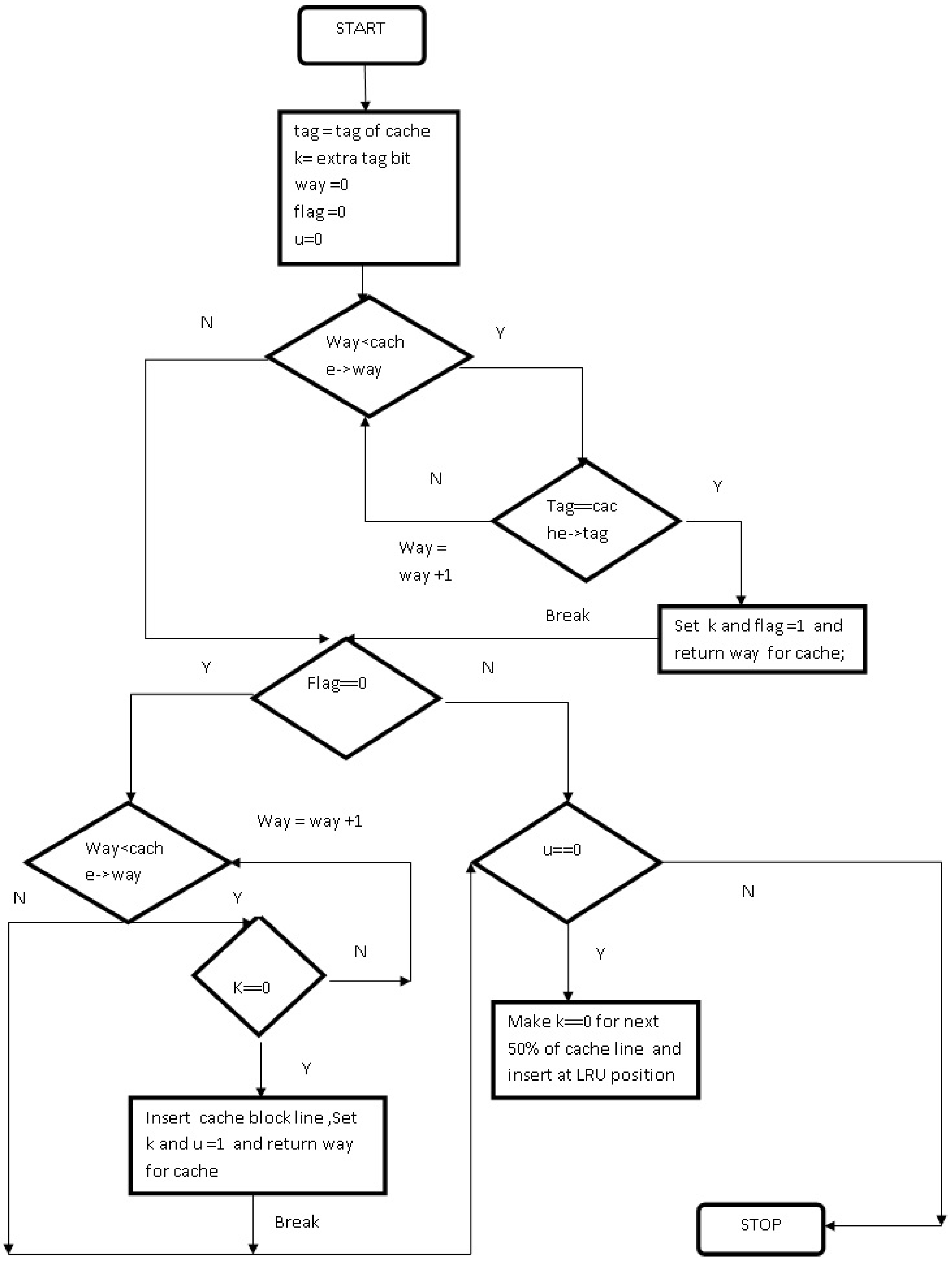
Our aim for this algorithm is to make cache efficient in case of thrashing access pattern i.e (when cache size is less than the working size of application).

Algorithm 4 BIR SET INSERTION cache replacement algorithm

```
1: $tag \leftarrow$ tag of new *cache block*
2: $k \leftarrow$ extra tag bit for each *cache line*
3: $way = 0$
4: $flag = 0$
5: $u = 0$
6: **while** $way < cache- > assoc$ **do**
7:    **if** $tag == cache- > tag$ **then**
8:        *set k for that cache line*
9:       *flag=1 and return the way for that cache line*
10:       ***break***
11:    **end if**
12:    $way \leftarrow way + 1$
13: **end while**
14: **if** $flag == 0$ **then**
15:    $way = 0$
16:    **while** $way < cache- > assoc$ **do**
17:       **if** ( $(k == 0)$ **then**
18:          *insert cache block to that position, set k, u=1 and return the way for*
             *that cache line*
19:          ***break***
20:       **end if**
21:       $way \leftarrow way + 1$
22:    **end while**
23:    **if** ( $(u == 0)$ **then**
24:       *then for the next 50 % of n cache line make k=0 and replace the cache*
          *block line at LRU position*
25:    **end if**
26: **end if**
```

The flow for the algorithm 4 is repsented as shown for level 2 cache in dual and quad core system:

Figure 4.2: Flow chart for bit set insertion policy


START
tag = tag of cache
k= extra tag bit
way =0
flag =0
u=0
N
Way<cach e->way
Y
N
Tag==cac he->tag
Y
Way = way +1
Break
Set k and flag =1 and return way for cache;
Y
Flag==0
N
Way = way +1
Way<cach e->way
N
Y
K==0
N
Y
Insert cache block line ,Set k and u =1 and return way for cache
Break
u==0
N
Y
Make k==0 for next 50% of cache line and insert at LRU position
STOP

In the alogritm 4 and figure 4.2 we have represented the proposed algorithm, in this technique if the cache block is already there in cache then we will set the value of k for the corresponding cache else if it is a miss than we search for first cache line from MRU position for which k==0 and replace that cache line with incoming cache line and if all the bits are set than we will reset the 50 % of the cache lines in a particular set and will insert the incoming line at LRU position. With this algorithm we have tried to improve the shared cache performance in dual and quad core system. Hardware Overhead for LRU is *O(mlogm)* [14] but for BSIP it is *O(m)*.

4.3 Observation with dual core system

Cache replacement policies are implemented and observed using multi2sim [26] in dual core system. We have considered Splash-2 benchmark [27] as it contains all real time applications. In splash-2 benchmark we have considered FFT and LU application.

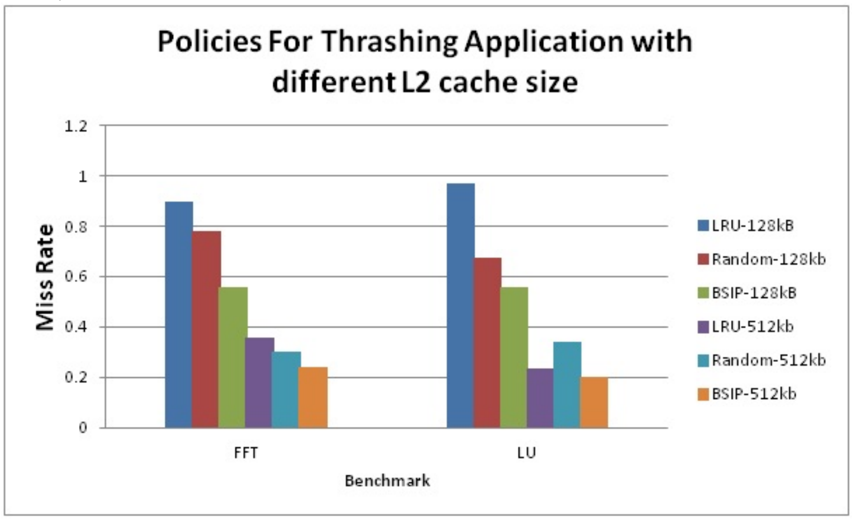
Table 4.1: Simulation model for analysing replacement policies in dual core system

Number of cores	L1 instruction Cache	L1 data cache	L2 shared cache
2	size- 16kb	size -16kb	size - 128kb
	assoc- 2way	assoc- 2way	assoc- 4way
	Policy- LRU, MRU, Random, BSIP	Policy- LRU, MRU, Random, BSIP	Policy- LRU, Random, BSIP

In table 4.1 we have given the basic cache configuration for L1 and L2, for that we have varied the cache replacement policies such as LRU,Random and BSIP. The simulations with above considerations are:

1. Miss rate has been observed with different cache sizes over different polices(LRU,Random and BSIP) in order to analyze the performance of BSIP in dual core system.

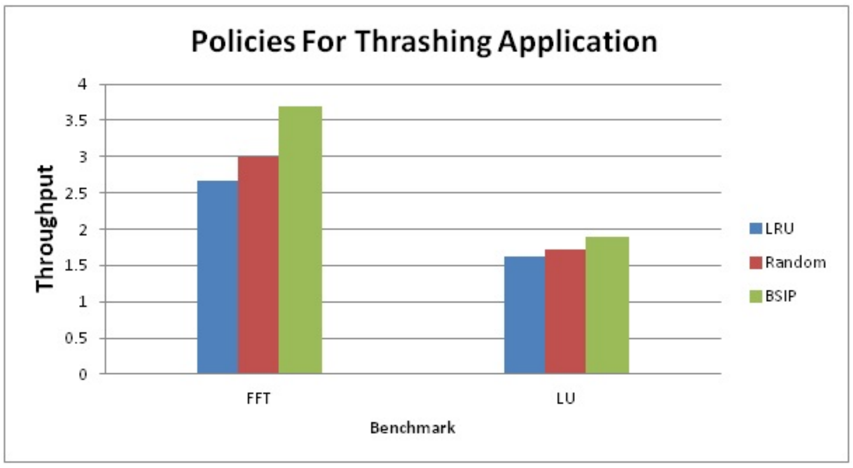
Figure 4.3: Miss rate on executing FFT and LU for L2 cache block (128kb,512kb) in dual core system



From figure 4.3 we can observe that it is clear that FFT and LU are forming a thrashing access pattern when we are taking the cache size less than the working size of problem, for cache size 128kb the miss rate is reduced by 15 % for FFT and by 20% for LU and for cache size 512 kb, as the cache size is appropritе for the application then the miss rate is low with LRU policy and it is further reduced by some fraction when executed with BSIP.

2. Throughput has been observed over different polices(LRU,Random and BSIP) in order to analyze the performance of BSIP in 2 core system over thrashing access pattern.

Figure 4.4: Throughput on executing FFT and LU for L2 cache block (128kb,512kb) in dual core system



From figure 4.4 we can observe that it is clear that FFT and LU are forming a thrashing access pattern when we are taking the cache size less than the working size of problem, for cache size 128kb the throughput is maximum for BSIP as if miss rate is low then there will be less access time. Hence, more instructions will be executed per cycle.

4.4 Observation with quad core system

Cache replacement policy is implemented using multi2sim [26] in quad core system environment. In splash-2 [27] benchmark we have considered FFT and LU application.

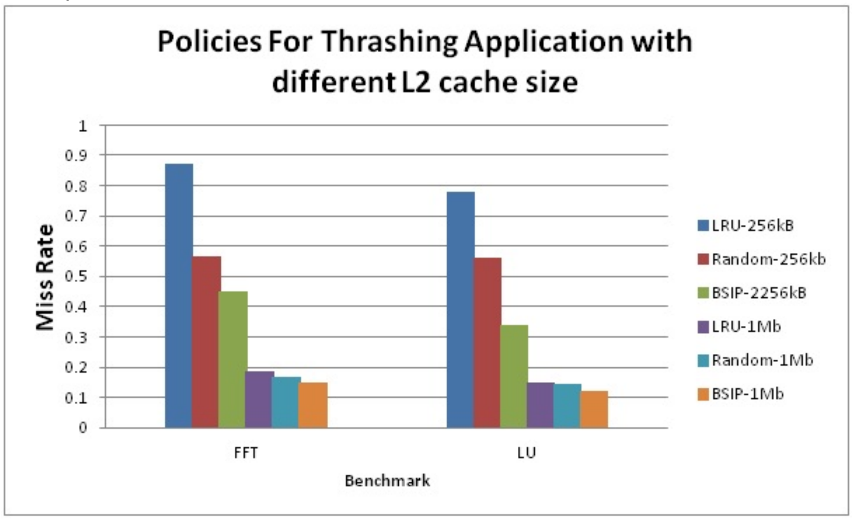
Table 4.2: Simulation model for replacement policy in quad core system

Number of cores	L1 instruction Cache	L1 data cache	L2 shared cache
4	size- 16kb	size -16kb	size - 256kb
	assoc- 2way	assoc- 2way	assoc- 4way
	Policy- LRU, Random, BSIP	Policy- LRU, Random, BSIP	Policy- LRU, Random, BSIP

In table 4.2 we have given the basic cache configuration for L1 and L2, for that we have varied the cache replacement policies such as LRU,Random and BSIP in quad core system. The simulations with above considerations are:

1. Miss rate has observed with different cache sizes over different polices(LRU,Random and BSIP) in order to analyze the performance of BSIP in quad core system.

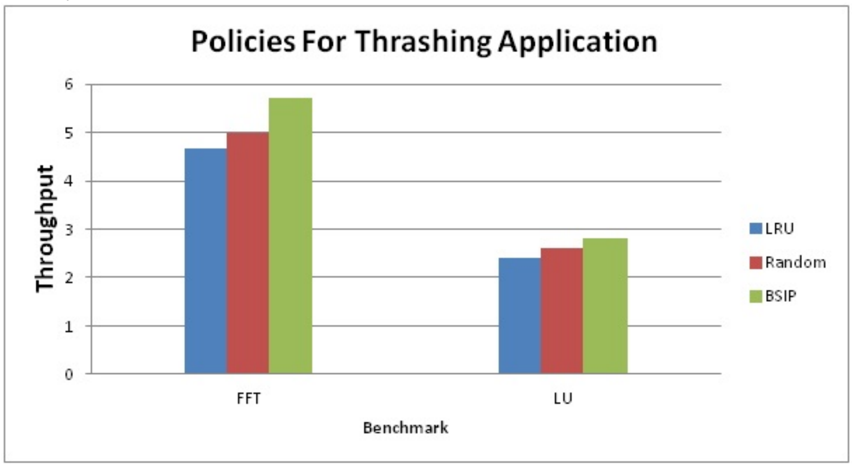
Figure 4.5: Miss rate on executing FFT and LU for L2 cache block (128kb,1Mb) in quad core system



From figure 4.5 we can observe that it is clear that FFT and LU are forming a thrashing access pattern when we are taking the cache size less than the working size of problem, for cache size 256kb the miss rate is reduced by 21 % for FFT and by 24% for LU and for cache size 1 MB, as the cache size is approprite for the application then the miss rate is low with LRU policy and it is further reduced by some fraction when executed with BSIP.

2. Throughput has been observed over different polices(LRU,Random and BSIP) in order to analyze the performance of BSIP in qyad core system.

Figure 4.6: Throughput on executing FFT and LU for L2 cache block (128kb,1Mb) in quad core system



From figure 4.4 we can observe that it is clear that FFT and LU are forming a thrashing access pattern when we are taking the cache size less than the working size of problem, for cache size 256kb the throughput is maximum for BSIP as if miss rate is low then there will be less access time. Hence, more instructions will be executed per cycle.

4.5 Summary

In this chapter, we have disscused new cache replacement policy *BSIP*. Impact of cache replacement policy on dual and quad core system have been observed by varying the cache size and analysing the miss rate and throughput on thrashing access pattern. On varying cache size with different benchmarks we observed that it works best for BSIP policy and worst for LRU as **miss rate** is maximized. On observing the **throughput** with different benchmarks we get that it works best for BSIP policy and worst for LRU and random is fluctuating. On increasing the number of cores miss rate is reduced and the throughput is increased, but not as prominent as with replacement policy.

Chapter 5

Conclusions and future works

Chapter 5
Conclusions and future works

5.1 Conclusions

In this thesis we have proposed a cache configuration to improve the efficiency of Level 2 cache in single and multicore system. Efficiency of cache means with minimum cache size it must give us maximum hit rate i.e least effective access time. Here we have observed that the for L2 cache with 4 way associativity gives better performance than the 8 or 16 way and for single core processor the cache combination of L1 size= 16kb and L2 size= 128kb and for dual core processor the cache combination of L1 size= 16kb and L2 size= 512kb and for quad core processor the cache combination of L1 size= 8kb and L2 size= 1Mb gives optimum performance with less miss rate. In case of applications with thrashing access pattern i.e cache size is less than the working size of problem, we have optimized the cache performance by changing the existing cache replacement policies. In this thesis we have used LRU and Random for thrashing access pattern to compare the efficency of our replacement policy. Simulation results have shown that LRU shows worst result for thrashing access pattern and Random performs better than the LRU but it is not fixed. The new replacement policy, BSIP gives much better results with thrashing access pattern when compared to LRU and Random. Hardware Overhead for LRU is *O(mlogm)* [14] but for BSIP it is *O(m)*.

5.2 Future works

- By using a central limit theoram [13] a **Dynamic Set Insertion Policy** can be implemented by considering Least Recently Used(LRU) cache replacement policy, if a application is cache friendly.
- A **Dynamic Set Insertion Policy** can be implemented by considering Bit Set Insertion Policy (BSIP) cache replacement policy, if a application is thrashing.
- Cache replacement policies can be compared using *power consumed as a performance parameter.*
- Cache replacement policies can be implemented by considering *thread level parallelism* in multicore processors.

Bibliography

[1] John P Hayes. *Computer Organization and Architecture.* McGraw-Hill, Inc. Newyork,USA, 1978.

[2] William Stallings. *Computer organization and architecture: designing for performance.* Pearson Education India, 1993.

[3] Behrooz Parhami. *Computer architecture: from microprocessors to supercomputers.* Oxford University Press New York, NY, 2005.

[4] Xian-He Sun and Yong Chen. Reevaluating amdahls law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010.

[5] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432. IEEE Computer Society, 2006.

[6] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely Jr, and Joel Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 208–219. ACM, 2008.

[7] Yuejian Xie and Gabriel H Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 174–183. ACM, 2009.

[8] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[9] James R Goodman. Using cache memory to reduce processor-memory traffic. In *ACM SIGARCH Computer Architecture News*, volume 11, pages 124–131. ACM, 1983.

[10] Fei Guo and Yan Solihin. An analytical model for cache replacement policy performance. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 228–239. ACM, 2006.

[11] Asit Dan and Don Towsley. *An approximate analysis of the LRU and FIFO buffer replacement schemes*, volume 18. ACM, 1990.

[12] Song Jiang and Xiaodong Zhang. Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30, pages 31–42. ACM, 2002.

[13] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 381–391. ACM, 2007.

[14] Aamer Jaleel, 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*, volume 38, pages 60–71. ACM, 2010.

[15] Hassan Ghasemzadeh, Sepideh Mazrouee, and Mohammad Reza Kakoee. Modified pseudo lru replacement algorithm. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, pages 6–pp. IEEE, 2006.

[16] Mainak Chaudhuri. Pseudo-lifo: the foundation of a new family of replacement policies for last-level caches. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 401–412. ACM, 2009.

[17] Donghee Lee, Jongmoo Choi, Honggi Choe, Sam H Noh, Sang Lyul Min, and Yookun Cho. Implementation and performance evaluation of the lrfu replacement policy. In *EUROMICRO 97.'New Frontiers of Information Technology'. Short Contributions., Proceedings of the 23rd Euromicro Conference*, pages 106–111. IEEE, 1997.

[18] Theodore Johnson and Dennis Shasha. X3: A low overhead high performance buffer management replacement algorithm. 1994.

[19] Sorav Bansal and Dharmendra S Modha. Car: Clock with adaptive replacement. In *FAST*, volume 4, pages 187–200, 2004.

[20] Kamil Kedzierski, Miquel Moreto, Francisco J Cazorla, and Mateo Valero. Adapting cache partitioning algorithms to pseudo-lru replacement policies. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[21] Wayne A Wong and Jean-Loup Baer. Modified lru policies for improving second-level cache behavior. In *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pages 49–60. IEEE, 2000.

[22] Jaekyu Lee and Hyesoon Kim. Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[23] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai. Managing shared last-level cache in a heterogeneous multicore processor. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 225–234. IEEE Press, 2013.

[24] Francesc Alted. Why modern cpus are starving and what can be done about it. *Computing in Science & Engineering*, 12(2):68–71, 2010.

[25] Kun Luo, Jayanth Gummaraju, and Manoj Franklin. Balancing thoughput and fairness in smt processors. In *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, pages 164–171. IEEE, 2001.

[26] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: A simulation framework for cpu-gpu computing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 335–344. ACM, 2012.

[27] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36. ACM, 1995.

[28] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared l2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, 2007.

[29] Alan Jay Smith. *Cache evaluation and the impact of workload choice*, volume 13. IEEE Computer Society Press, 1985.

[30] Mark D Hill and Alan Jay Smith. Evaluating associativity in cpu caches. *Computers, IEEE Transactions on*, 38(12):1612–1630, 1989.

[31] Alan Jay Smith. Line (block) size choice for cpu cache memories. *Computers, IEEE Transactions on*, 100(9):1063–1075, 1987.

[32] Paul Sweazey and Alan Jay Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *ACM SIGARCH Computer Architecture News*, volume 14, pages 414–423. IEEE Computer Society Press, 1986.

[33] Erlin Yao, Yungang Bao, Guangming Tan, and Mingyu Chen. Extending amdahl's law in the multicore era. *ACM SIGMETRICS Performance Evaluation Review*, 37(2):24–26, 2009.

[34] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122. IEEE Computer Society, 2004.

[35] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 293–304. ACM, 2012.

[36] Nitin Chaturvedi and S Gurunarayanan. Study of various factors affecting performance of multi-core processors. *International Journal*, 2013.

[37] Yi Yang, Ping Xiang, Mike Mantor, and Huiyang Zhou. Cpu-assisted gpgpu on fused cpu-gpu architectures. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[38] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[39] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: A simulation framework for cpu-gpu computing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 335–344. ACM, 2012.