NATIONAL INSITITUTE OF TECHNOLOGY, ROURKELA

An Effective Cache Replacement Policy For Multicore Processors

by

Suraj Sharma [109cs0490]
Babu Lal [109cs0597]

A thesis submitted in partial fulfillment for the degree of
Bachelor of Technology

under the guidance of
Prof. Ashok Kumar Turuk
Computer Science & Engineering

May 2013

NATIONAL INSITITUTE OF TECHNOLOGY
CERTIFICATE

This is to certify that the work in the thesis entitled **"An Effective Cache Replacement Policy for Multicore Processor"** submitted by **Suraj Sharma and Babu Lal** is a record of an authentic work carried out by them, under my supervision and guidance in partial fulllment of the requirements for the award of the degree of Bachelor of Technology in Computer Science and Engineering at National Institute of Technology, Rourkela.

Prof. Ashok Kumar Turuk
Head Of The Department,
Computer Science & Engineering Department,
NIT Rourkela.

Date: May 20, 2013

Acknowledgement

We would like to articulate our profound gratitude and indebtedness to those persons who helped us in the project.

First of all, we would like to express our obligation to our project guide Prof. Ashok Kumar Turuk for his motivation, help and supportiveness. We are sincerely thankful to him for his guidance and helping effort in improving our knowledge on the subject. He has been always helpful to us in all aspects and we thank him from the deepest of our heart.

An assemblage of this nature could never have been attempted without reference to and inspiration from the works of others whose details are mentioned in reference section. I acknowledge my indebtedness to all of them.

At the last, my sincere thanks to all my friends who have patiently extended all sorts of helps for accomplishing this assignment.

(Suraj Sharma)
(Babu Lal)

NATIONAL INSITITUTE OF TECHNOLOGY, ROURKELA

Abstract

Cache management is increasingly important on multicore systems since the available cache space is shared by an increasing number of cores. Optimal caching is generally impossible at the system or hardware. The goal of cache management is to maximize data reuse. In this paper, a collaborative caching system is implemented that allows a program to choose different caching methods (generally LRU or MRU) for its data. We have developed a LRU MRU caching replacement policy for multicore processor which is simulated using Multi2sim simulator. We have compared the value of the following parameters (Instruction committed per cycle and branch predication Accuracy) of the multicore processor when different caching algorithm is used for cache management: replacement policies are LRU, MRU, FIFO and LRU-MRU. We use 2 benchmarks suits: Mediabench and Splash-2 to see how our LRU-MRU replacement is behaving and what is the value of IPC, is it less than LRU, MRU, FIFO and Random or vice versa. We get, LRU is performancing far better than MRU and our LRU-MRU performance is better than LRU for some of the benchmarks. Here performance refers to the value of IPC, More the value of IPC better the performance of the replacement policy used.

Contents

List of Figures

Abbreviations

LRU	**L**east **R**ecently **U**sed
MRU	**M**ost **R**ecently **U**sed
FIFO	**F**ast **I**n **F**ast **O**ut
IPC	**I**nstructions **C**ommitted **P**er **C**ycle

Chapter 1

Introduction

1.1 Introduction

A multicore processor is a single computing component with more than one central processing units (called cores), which read and execute program instructions. In the single core processor, a series of requests for memory locations is send to the cache, where each request appears after the last one has been served. The time by the cache to serve the request depends on whether the memory location is present in the cache (a cache hit) or not (a miss occurs and the location is fetched from the main memory. If the cache is full,the memory location already in the cache gets evicted to make space for the new memory location. Which memory location(or block) is to be evicted is decided using the cache replacement policy, mainly LRU.We tried to design a new replacement policy known as collaborative LRU-MRU policy.

1.2 Cache Replacement Policies

1.2.1 LRU

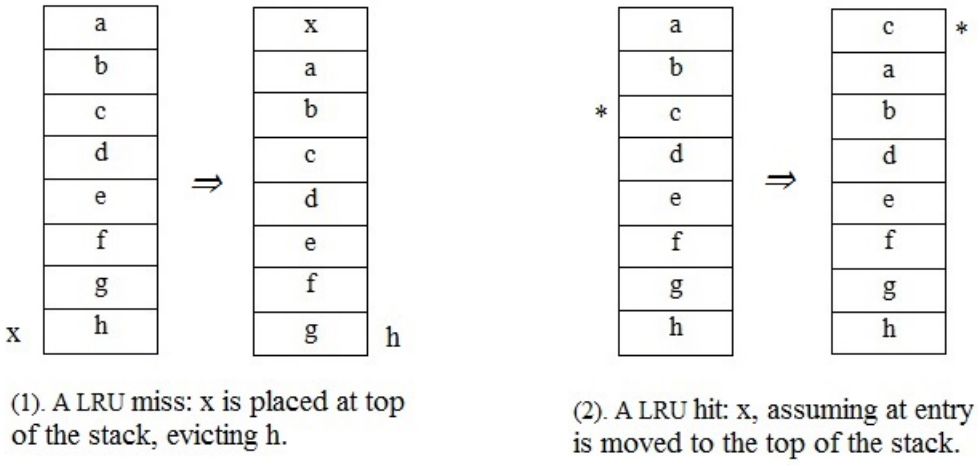

FIGURE 1.1: LRU Replacement Policy

This replacement policy is the most widely used. In this policy,the data in cache is sorted by the last use time. If the data is not present in the cache and the cache is full, the data in the LRU position is evicted. But the LRU replacement algorithm just considers the most recently accessed information of data block, and doesn't care about the frequency of data blocks that are accessed. LRU policy can be expensive when the set associativity is high.When the capacity of cache is less than the work set of program,the cache can

present the jitter phenomenon. This phenomenon will lead to computer performance decline.The disadvantage of LRU policy is that it cannot predict whether the data is be used frequently.

In case of cache hit(data is already present in the cache), the data move to the top of the stack(i.e., MRU position) and in case of cache miss(data is not present in the cache), the data present at the bottom of the stack is evicted and the new data is placed at the top of the stack, as shown in the figure 1.1.

1.2.2 MRU

In this replacement policy,the most recently data is evicted from the cache whenever there is a cache miss.Iat is most useful in situations where the older a data is, the more likely it is to be accessed.

In case of cache hit(data is already present in the cache), the accessed block is moved to the bottom of the stack and hence most recently used blocks are placed at the bottom of the stack. In case of cache miss(data is not present in the stack),the most recently used data(block at the bottom of the stack) gets evicted and the new data is placed at the bottom of the stack, as shown in the figure 1.2.

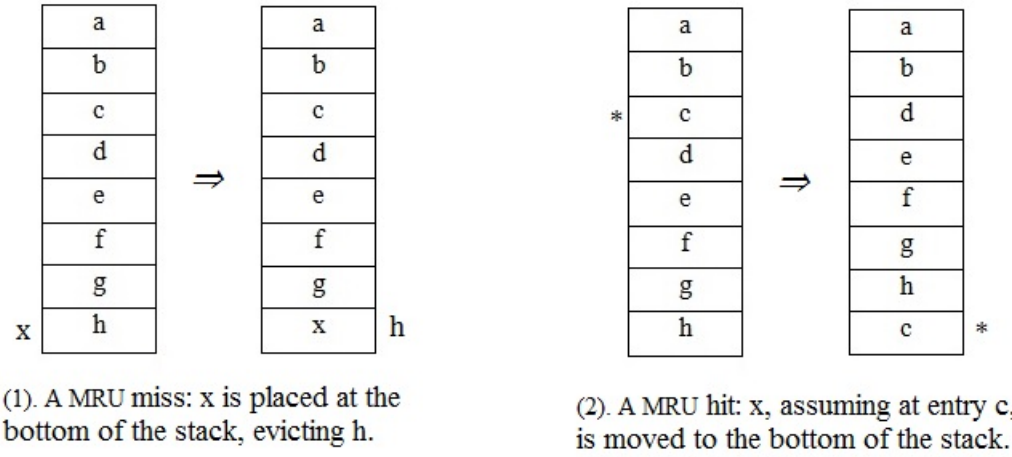

FIGURE 1.2: MRU Replacement Policy

1.2.3 FIFO

In this replacement policy, data which entered into the cache first is evicted.It has low overhead and runs faster, but not good for practical use.

In case of cache miss,data is evicted from the bottom of the stack(data entered into the cache first) and the new data is placed at the top of the stack,as shown in the figure 1.3.

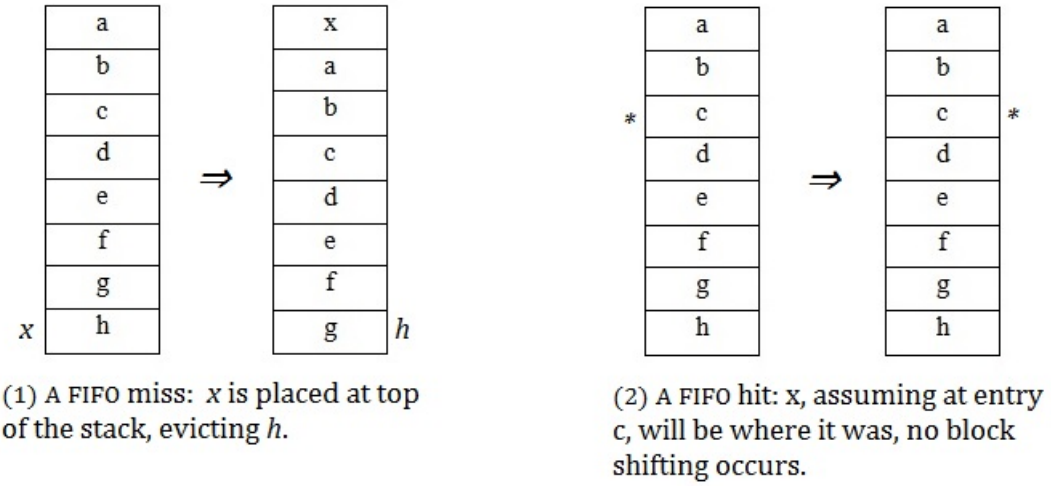

FIGURE 1.3: FIFO Replacement Policy

1.2.4 Random

In case of cache miss , data is randomly selected from the cache and is evicted from the cache to make space for the new data. This replacement policy does not require keeping any information about the access history of the blocks.the probability of data eviction is same for each block. It is simple to use and hence it has been used in ARM processors.

Chapter 2

An attempt: LRU-MRU Cache Replacement Policy

2.1 Introduction

Our concept in LRU-MRU cache replacement policy is to choose LRU or MRU policy anytime a cache miss or cache hit occurs.Choosing between them every time a block is accessed. In both LRU and MRU blocks are replaced from the bottom of the stack but in case of LRU, new data is placed at the top of the stack and in MRU, new data is placed at the bottom of the stack.In MRU, most recently used data is placed at the bottom of the stack and in LRU, least recently used data is placed at the bottom of the stack.

Which replacement policy is to be chosen is done by checking the value of N(bit_tag) which is either 1 or 0, N equals to 1 when previous accessed block's tag different than that of the current accessing block's tag and N equals to 0 if both tags are same. In case of cache miss(data is not present in the cache), which replacement policy is to be chosen for evicting the data is done by checking the value of N, if N equals to 0, MRU policy is selected and data present at the bottom of the stack gets evicted and new data is placed at the bottom of the stack and if N equals to 1, LRU policy is selected and data present at the bottom of the stack gets evicted and new data is placed at the top of the stack, as shown in the figure 2.1.In case of cache hit(data is already present in the stack), where to place the current accessed blocke(top or bottom of the stack) is decided by checking whether N is set or unset. If N is set, LRU policy is selected and the current accessed data is placed at the top of the stack else, the current accessed data is at the bottom of the stack, as shown in the figure 2.2.

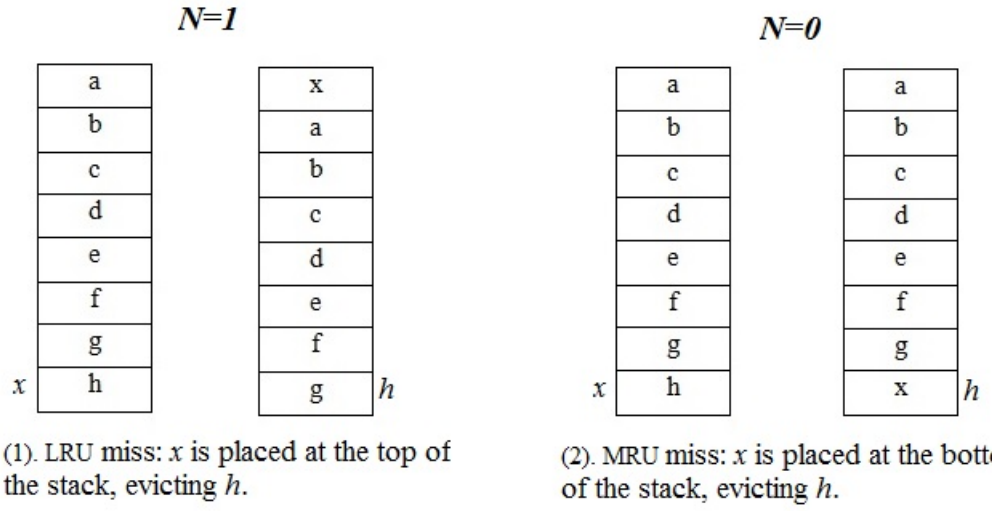

FIGURE 2.1: LRU-MRU Replacement Policy(Cache Miss)

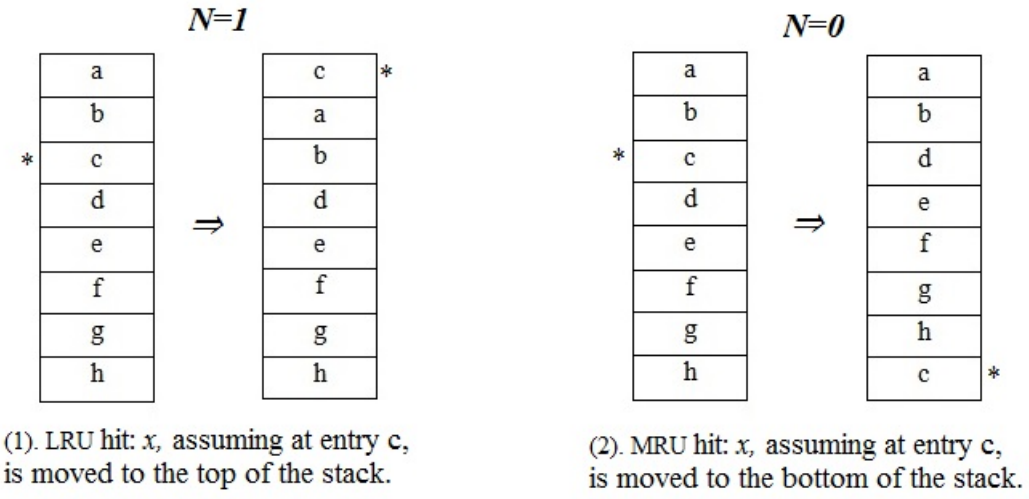

FIGURE 2.2: LRU-MRU Replacement Policy(Cache Hit)

2.2 Proposed Algorithm

Our aim of proposing LRU-MRU replacement algorithm is to increase the value of IPC and it should be greater than that for the LRU.

LRU-MRU Replacement Algorithm

1. prev_tag := 0;
2. bit_tag := 0;

// initially both previous tag and value of N equals to 0

3. current_tag := tag of the block being accessed ;
4. if current_tag == prev_tag then

N := 0;

select MRU replacement policy;

5. else N:= 1;

select LRU replacement policy;

6. End if

In this algorithm we are keeping track of previous accessed block's tag and comparing it with that of current accessed block's tag. In case of cache miss, current accessed block is the new data which want to enter the cache.

This proposed algorithm has outperformed LRU for most of the benchmarks programs we have used, as shown in the figure 2.3.

Benchmark	LRU	MRU	LRU-MRU
Splash2-Radix	0.6192	0.5862	0.6925
Splash2-Cholesky	1.087	0.8468	1.085
Mediabench-Jpeg-dec	0.8493	0.6662	0.8539
Mediabench-Epic-dec	0.7289	0.528	0.7306

FIGURE 2.3: LRU-MRU performance against MRU and LRU

Chapter 3

Results and Discussion

3.1 Introduction

The Multi2sim simulator framework was used to compare the performance of LRU,MRU and LRU-MRU replacement policies.Multi2Sim is a simulation framework for CPU-GPU heterogeneous computing written in C. It includes models for superscalar, multithreaded, and multicore CPUs, as well as GPU architectures.
To compare the performance of the replacement policies we are comparing the instructions committed per cycle(IPC) by the multicore processor. We have used two benchmarks suits Mediabench and splash2 for comparing the performance of the replacement policies.

3.2 Configuration

3.2.1 CPU-configuration

No. of cores = 3,
No. of threads per core = 1.

3.2.2 Memory-configuration

The memory configuration stated below, is used for comparing the performance of the replacement policies (or comparing the value of IPC).Memory configuration state the main memory architecture,L1 and L2 cache architecture.

[CacheGeometry geo-l1]
Sets = 128
Assoc = 2
BlockSize = 256
Latency = 2
Policy = (LRU or MRU or LRU-MRU)
Ports = 2

[CacheGeometry geo-l2]

Sets = 512

Assoc = 4

BlockSize = 256

Latency = 20

Policy = (LRU or MRU or LRU-MRU)

Ports = 4

[Module mod-l1-0]

Type = Cache

Geometry = geo-l1

LowNetwork = net-l1-l2

LowModules = mod-l2-0 mod-l2-1

[Module mod-l1-1]

Type = Cache

Geometry = geo-l1

LowNetwork = net-l1-l2

LowModules = mod-l2-0 mod-l2-1

[Module mod-l1-2]

Type = Cache

Geometry = geo-l1

LowNetwork = net-l1-l2

LowModules = mod-l2-0 mod-l2-1

[Module mod-l2-0]

Type = Cache

Geometry = geo-l2

HighNetwork = net-l1-l2

LowNetwork = net-l2-mm

LowModules = mod-mm

AddressRange = BOUNDS 0x00000000 0x7FFFFFFF

[Module mod-l2-1]

Type = Cache

Geometry = geo-l2

HighNetwork = net-l1-l2

LowNetwork = net-l2-mm

LowModules = mod-mm

AddressRange = BOUNDS 0x80000000 0xFFFFFFFF

[Module mod-mm]

Type = MainMemory

BlockSize = 256

Latency = 200

HighNetwork = net-l2-mm

[Network net-l2-mm]

DefaultInputBufferSize = 1024

DefaultOutputBufferSize = 1024

DefaultBandwidth = 256

[Network net-l1-l2]

DefaultInputBufferSize = 1024

DefaultOutputBufferSize = 1024

DefaultBandwidth = 256

[Entry core-0]

Arch = x86

Core = 0

Thread = 0

DataModule = mod-l1-0

InstModule = mod-l1-0

[Entry core-1]

Arch = x86

Core = 1

Thread = 0

DataModule = mod-l1-1

InstModule = mod-l1-1

[Entry core-2]

Arch = x86

Core = 2

Thread = 0

DataModule = mod-l1-2

InstModule = mod-l1-2

3.3 Analysis

3.3.1 IPC vs Associativity, keeping No. of sets equal to 512

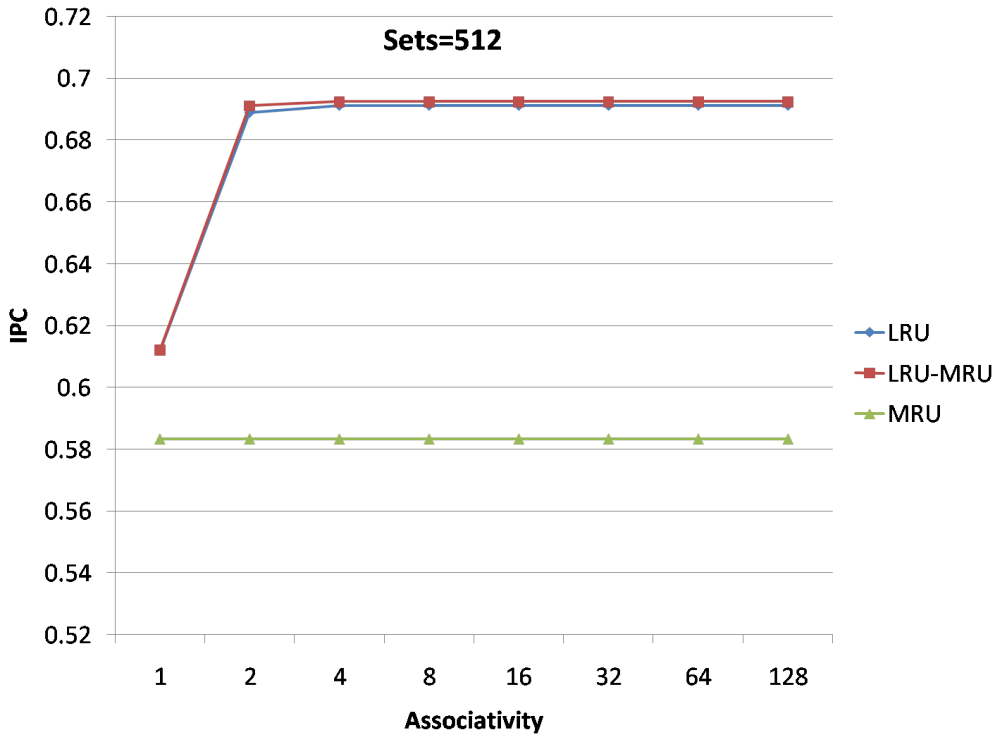

FIGURE 3.1: Graph showing the variation of IPC vs Associativity for different replacement policies, keeping no. of sets = 512

Associativity means number of cache blocks(cache lines) in each set inside the cache memory. For example, in 2-way set associative cache,there will be 2 cache lines in each set. In Direct mapping, associativity equal to 1.

Cache Size = (No. of sets * associativity)* block_size

if we consider the above memory configuration of L2 cache then

cache size = (512 * 4) * 256 bits = 64 Kb,

hence, the size of L2 cache is 64 kb and that of L1 cache is 8kb (128*2*256)

We are changing the associativity and comparing the value of IPC for different replacement policies, the benchmark program we are using for this is radix sort. IPC remains constant(IPC=0.5833) for all values of associativity,thus show that IPC is independent of associativity in MRU, but cfor LRU and LRU-MRU both, IPC becomes constant when associativity is greater than equal to 4. For low associativity i.e., when assoc = 1 or 2 , LRU outperforms LRU-MRU but not at high associativity, as shown in the figure 3.1.

3.3.2 IPC vs No. of sets, keeping associativity equal to 4

We are changing the value of number of sets in the cache keeping associativity equal to 4 and comparing the value of IPC for different replacement policies,the benchmark program we are using is radix sort. IPC for MRU is always less than that for LRU and LRU-MRU for all values of sets. For low values of sets, IPC for MRU is almost equal to 0 i.e., IPC = 0.0888 and increases as no. of sets increases but becomes constant (IPC=0.6447) for sets greater than equal to 4096. LRU outperforms LRU- MRU for low values of sets and reaches it maximum value(IPC = 0.6946) at sets = 128 but decreases little to IPC = 0.6912 and after that it becomes constant for sets greater than equal to 512, LRU-MRU outperforms LRU at sets equal to 512(IPC=0.6925) and reaches its maximum(IPC=0.6981) at sets equal to 2048 remains constant after that, as shown in the figure 3.2.

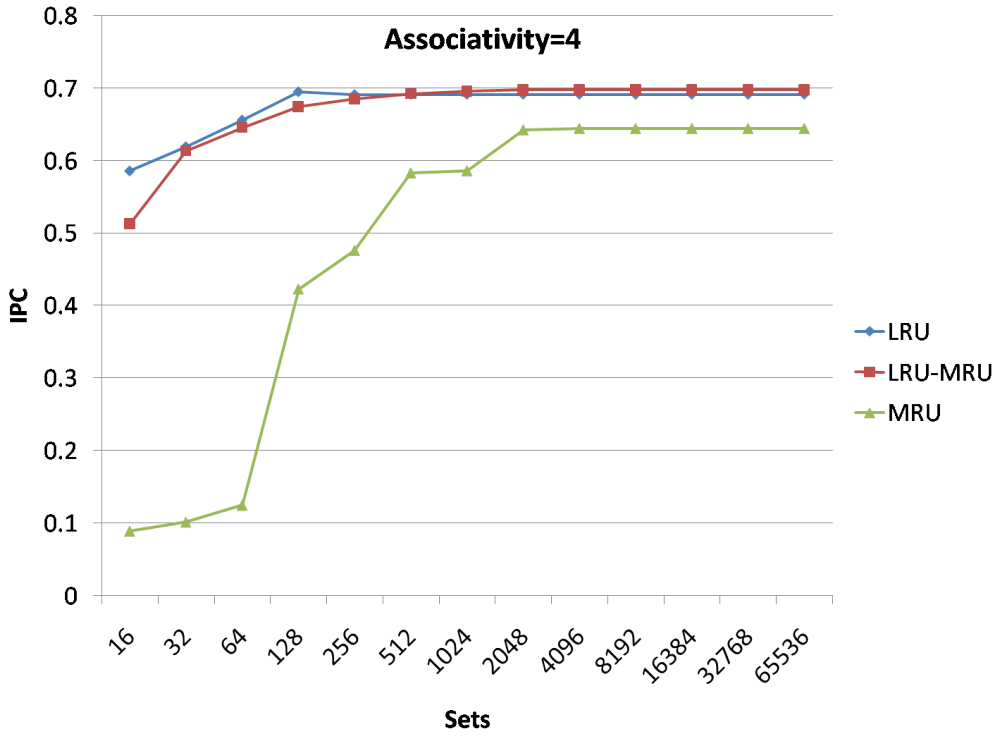

FIGURE 3.2: Graph showing the variation of IPC vs No. of Sets for different replacement policies, keeping Associativity=4

3.3.3 IPC vs No. of sets, keeping associativity equal to 2

We are changing the value of number of sets in the cache keeping associativity equal to 2 and comparing the value of IPC for different replacement policies,the benchmark program we are using is radix sort. IPC for MRU is always less than that for LRU and LRU-MRU for all values of sets. For low values of sets, IPC for MRU is almost equal to 0 i.e., IPC = 0.0888 and increases as no. of sets increases but becomes constant (IPC=0.6447) for sets greater than equal to 4096. LRU-MRU outperforms LRU for all values of sets and reaches its maximum value (IPC = 0.6981) at sets equal to 2096 and remains constant after that, as shown in the figure 3.3.

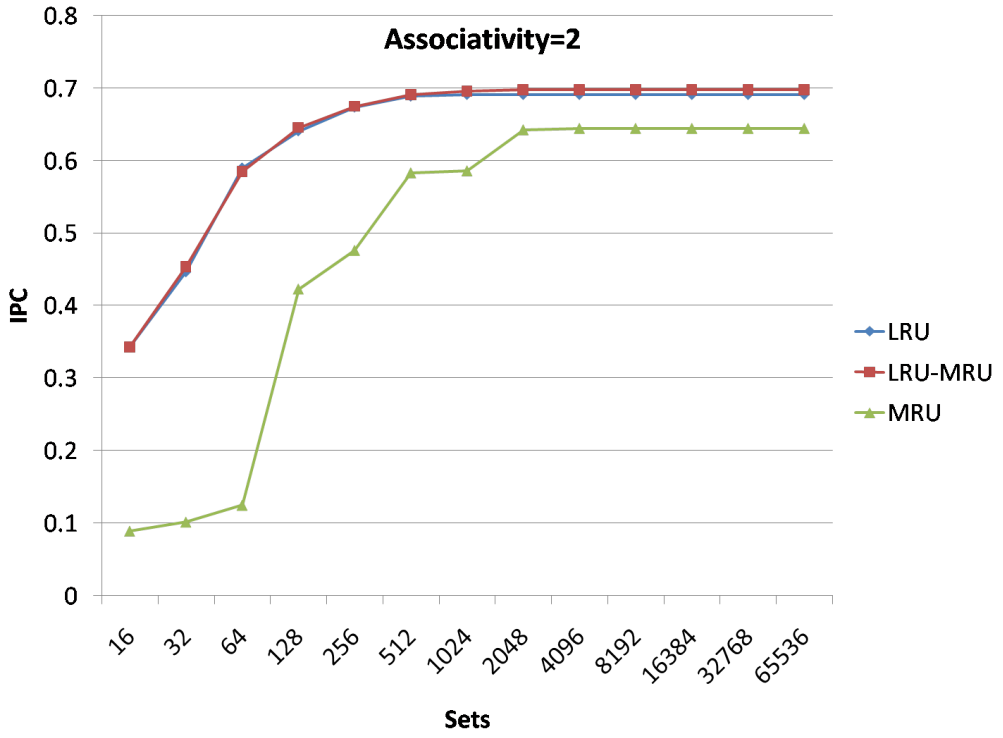

FIGURE 3.3: Graph showing the variation of IPC vs No. of Sets for different replacement policies, keeping associativity=2

3.3.4 Performance of LRU-MRU Against LRU

LRU-MRU have outperforms LRU for most of the benchmark programs we used, the benchmarks suits we used are Mediabench and Splash2. The instructions committed per cycle for MRU is least for all benchmarks programs but LRU-MRU has shown the statisfied result and hence it can be used as an effective cache replacement policy for multicore processors.

The results which we got are shown in the figure 3.4.

If we observe the figure 3.4 carefully, we will see that for splash2-cholesky, LRU-MRU

Benchmark	LRU	MRU	LRU-MRU
Splash2-Radix	0.6192	0.5862	0.6925
Splash2-Cholesky	1.087	0.8468	1.085
Mediabench-Jpeg-dec	0.8493	0.6662	0.8539
Mediabench-Epic-dec	0.7289	0.528	0.7306

FIGURE 3.4: Instructions committed per cycle for different benchmarks programs.

performance is not better than LRU but for other three benchmarks programs LRU-MRU has outperformed LRU.Hence the above result shows that, for predicting which replacement policy to be use for evicting the data from the cache can be done by comparing the tag of the previous accessed block and the current accessed block, if they are equal, choose MRU cache replacement policy else choose LRU cache replacement policy. In the screenshots shown in the figure 3.5,3.6 and 3.7, we have shown the simulation statistics summary which we get while executing the benchmarks program (radix sort)
command entered : m2s –ctx-config ctx-config-radix –x86-sim detailed –x86-config x86-config –mem-config mem-config

FIGURE 3.5: Simulation Statistics Summary for LRU replacement policy.

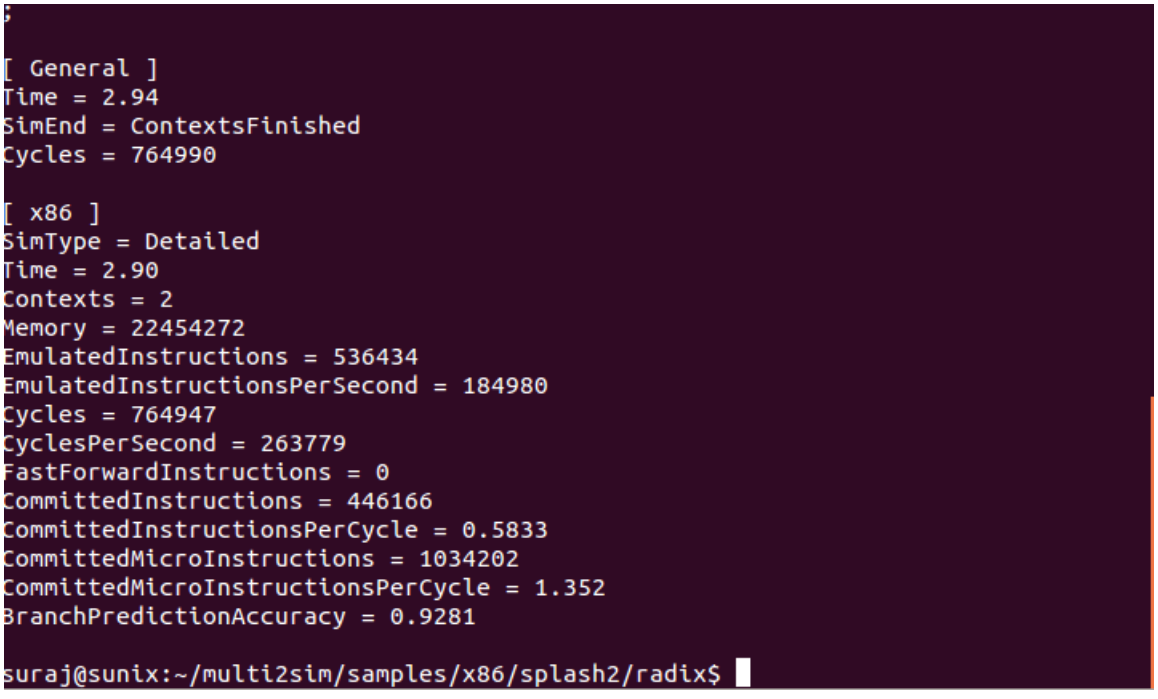

FIGURE 3.6: Simulation Statistics Summary for MRU replacement policy.

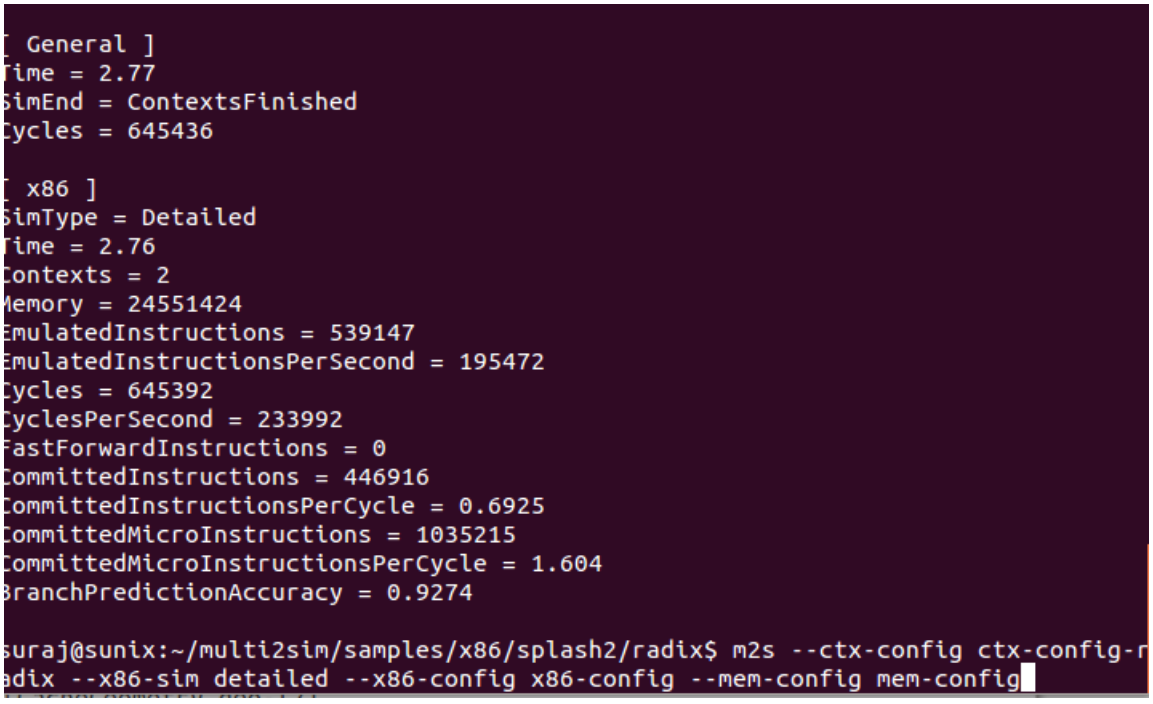

FIGURE 3.7: Simulation Statistics Summary for LRU-MRU replacement policy.

Chapter 4

Conclusion

What we have done in our project is proposed an algorithm which can outperforms LRU replacement policy for most of the benchmarks programs which is simulated using Multi2sim simulator.The results we got was better than that for LRU policy.According to our result, MRU performs worst in multicore processor and IPC remains constant for all values of associativity if no of sets is kept constant.IPC for LRU and LRU-MRU policy increases for low value of associativity and reaches a constant value for high associativity, and LRU-MRU outperforms LRU at high associativity while keeping no of sets constant. If we change the no. of sets keeping associativity, all the three replacement policy(LRU, MRU, LRU-MRU) IPC increases with increase in no. of sets (value of IPC for MRU is least) and reaches a constant value for sets greater than equal to 2048. When no of sets is less(32,64) LRU outperforms LRU-MRU but after it reaches 512 and above LRU-MRU starts to outperforms LRU.

Bibliography

[1] *Shared cache simulation for Multicore system with LRU-MRU collaborative cache replacement algorithm* by Shan Ding, Shiya Lui, Northeastern University, College of Information science and engineering, Shenyang city, China.

[2] *On the Theory and potential of LRU-MRU Collaborative cache management* by Xiaoming Gu and Chen Ding, Department of Computer science, University of Rochester, NY, USA.

[3] *A generalized theory of collaborative caching* by Xiaoming Gu and Chen Ding, Department of Computer Science, University of Rochester, NY, USA.

[4] *Design of CPU Cache Memories* by Alan Jay Smith, University of California, USA.

[5] *Cache coherence techniques for multicore processors* by Micheal R. Marty, University of Wisconsin, Madison.

[6] *Multi2sim: A Simulation framework to evaluate multicore-multithreaded processors*, http://www.multi2sim.org/files/multi2sim-r311.pdf

[7] *Cache Replacement Policies for Multicore Processors* by Avinatan Hassidim, Massachusetts Institute of Technology, Cambridge, USA.

[8] *Cache memories* by Alan Jay Smith, University of California,USA.

[9] *Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors* by R. Ubal, J. Sahuquillo, S. Petit and P. Lopez, Universidad Politecnica de Valencia,Spain