Ph.D. Thesis

An Algorithmic approach for optimizing Page Fault to increase Processor Performance along with Thread-Level Speculation

Jisha P.Abraham

Department of Computer Science
Faculty of Technology
Cochin University of Science and Technology
Kochi- 682022, INDIA

Computer Science

Ph.D. Thesis

Jisha P.Abraham

An Algorithmic approach for optimizing Page Fault to increase Processor Performance along with Thread-Level Speculation

Under the guidance of
Prof.Sheena Mathew

October 2017

Department of Computer Science
Faculty of Technology
Cochin University of Science and Technology
Kochi- 682022, INDIA

An Algorithmic approach for optimizing Page Fault to increase Processor Performance along with Thread-Level Speculation

Thesis submitted to
Cochin University of Science and Technology
In partial fulfilments for the requirements of the degree of
Doctor of Philosophy
under the
Faculty of Technology

By

Jisha P. Abraham

Department of Computer Science
Cochin University of Science and Technology
Cochin 682022, Kerala, India

October 2017

An Algorithmic approach for optimizing Page Fault to increase Processor Performance along with Thread-Level Speculation

Ph.D. Thesis in the field of Computer Science

Author
Jisha P. Abraham
Associate Professor
Department of Computer Science & Engineering
Mar Athanasius College of Engineering
Kothamangalam – 686691, Kerala, India
Email: jishaanil@gmail.com

Research Advisor
Dr. Sheena Mathew
Professor
Division of Computer Science & Engineering
School of Engineering
Cochin University of Science & Technology
Cochin 682022, Kerala, India
Email: sheenamathew@cusat.ac.in

Department of Computer Science
Cochin University of Science and Technology
Cochin 682022, Kerala, India

October2017
Dr. Sheena Mathew
Professor
Division of Computer Science &Engineering
School of Engineering
Cochin University of Science & Technology

Cochin 682022, Kerala, India
Ph: +919446509508 Email: sheenamathew@cusat.ac.in

Certificate

Certified that the research work proposed in the thesis entitled "**An Algorithmic approach for optimizing Page Fault to increase Processor Performance along with Thread-Level Speculation**" is based on the original work done by **Mrs. Jisha P. Abraham** under my supervision and guidance at Department of Computer Science, Cochin University of Science and Technology, Kochi and has not been included in any other thesis submitted previously for the award of any degree.

Kochi - 682 022

Dr. Sheena Mathew
(Supervising guide)

Certificate

This is to certify that the relevant corrections and modifications by the audience during the pre-synopsis presentation and recomended by the Doctral Committee of the candidate have been incorporated in the thesis entitled "**An Algorithmic approach for optimizing Page Fault to increase Processor Performance along with Thread-Level Speculation**" .

Kochi - 682 022

Dr. Sheena Mathew
(Supervising guide)

Declaration

I hereby declare that the work presented in this thesis entitled "**An Algorithmic approach for optimizing Page Fault to increase Processor Performance along with Thread-Level Speculation**" is based on the original research work done by me under the supervision of **Dr. Sheena Mathew**, Professor, Division of Computer Science & Engineeering, School of Engineering, Cochin University of Science and Technology, Kochi and has not been included in any other thesis submitted previously for the award of any degree.

Kochi - 682022 Jisha P.Abraham

Acknowledgements

While submitting this thesis report with a profound sense of gratitude, I would avail this opportunity to thank those who have helped me generously in the completion of this work. First and foremost, I am grateful to the God Almighty for having been my lodestar throughout my life and showered all blessing upon me.

It is my proud privilege to express sincere gratitude to my supervisor and guide, **Dr.Sheena Mathew** for her inspiration, sustained encouragement, timely advice, constructive criticisms and concrete suggestions in the course of this dissertations.

I am much greatful to **Dr.B.Kannan** during the timely advice he had given to me during tough times. Also my greatfulness to my students who helped me with suggestions during the testing phase of my work. I also express my sincere thanks to all my colleagues at Department of Computer Science & Engineering, Mar Athanasius College of Engineering especially to Ani Sunny, Jiso George and Siji Eldhose for their support they have extended to me for the completion of this work.

Mere words would not suffice to express my gratitude to my parents, family members and my childrens for their support and prayers. I express my heartfelt gratitude to my beloved husband who was always my strength and support, standing by me especially during various stages of the thesis.

Jisha P. Abraham

Abstract

The processor performance is highly dependent on the delivery of the correct instruction or data at the appropriate instance. If the data miss is occurring in the cache memory, the processor has to spend more cycles for fetching data from the lower memory levels. One of the methods used to reduce cache miss is instruction prefetching , which in turn will increase the number of instructions readily available for the processor. Instruction prefetching is a technique used by the central processing unit to speed up the execution of a program by reducing the wait states.

Bandwidth is the amount of data that can be transferred from one memory component to the other. Prefetchers trade bandwidth for latency. If the prefetched address is correct, more bandwidth is required to reduce latency. Whereas if incorrect prediction is made, the bandwidth is reduced and latency increased. Processor idle condition during program execution also may lead to system performance degradation. Current instruction cache prefetching schemes have higher memory acces latencies. Data prefetching has been used as one of the mechanism to reduce memory access latencies. Even though a number of prefetching methods are available, none of them proved to be accurate. To resolve the various issues an elaborated study on the memory management module of the various operating systems is done, which includes, Open VMS, Microsoft windows, Unix-like operating systems (including Mac OS X,

Linux, Solaris) and z/OS are carried out.

Along with branch handling, system performance greatly depends up on disk scheduling, page replacement algorithm, and compiler level parallization. In case of multiprocessing environment more than one process is going to access the disk at the same time. Currently, Complete Fair Queuing (CFQ) based disk scheduling techniques are used in various operating systems. In Complete Fair Queuing the time slots are assigned to requested process ie, it works in time domain. This time interval allowed for the disk access of each application includes access latency which consequently reduces the system throughput. In order to overcome the above problem a new disk scheduler, Budget Fair Queuing (BFQ), is considered and which is based on service domain instead of time domain. But the results show that the application with smaller file size will lead to starvation, ie they have to wait in the request queue, for a long time. To overcome this problem an another method is suggested called BFQ+. In this work, two new methods were suggested, first method Modified Budget Fair Queuing Version1 (MBFQV1) and second method Modified Budget Fair Queuing version2 (MBFQV2) . The implementation is differ based on the how the budget allocation is done for the schedilder. And the results shows that MBFQV2 is giving a better result.

A page replacement algorithm considers the limited information about access to the pages provided by the hardware. Least Recently Used (LRU) is the page replacement algorithm used in

most operating systems. The number of times the page is referred is not considered for replacement. A new algorithms is developed, Least Recently Used and Least Frequently Used (LRU-LFU) which will give a better performance than LRU page replacement method.

Ensuring the availability of required pages in cache is one of the major tasks of kernel in memory management. Any failure of kernel in this functionality may lead to cache miss instead of cache hit. CaMMEBH is a new algorithm developed on branch handling statements to reduce the fault rate in memory. The property of locality of reference allows the system to prefetch the pages from one memory level to the next one. CaMMEBH designed to load all the probable destination pages into cache while loading from main memory, so that maximum hit ratio is found when searching in the cache itself.

Today all operating systems are designed in a multithreaded environment. But the compilers used are not considering this fact. Here the loops which are executable in parallel are identified and they are executed using various threads. On the other hand, automatic parallelization offered by compilers only extracts parallelism from loops and the compiler can assure that there is no risk of dependence violation at runtime. Only a small fraction of loops falls into this category, by leaving many potentially parallel loops unexploited. Thread-Level Speculation (TLS) techniques allow the extract parallelism from fragments of code that cannot be analyzed at compile time. Automated Code Parallelizer using Open Multi-Processing(OpenMP), which automates the insertion of compiler directives

to facilitate parallel processing on shared memory machines with multiple cores. It converts an input sequential program into a multi- threaded program for multi-core shared memory architectures. This work focuses on loops and speculatively parallelizes the different iterations of a loop while taking care of data dependency between the different iterations. While executing the various iterations the window size is varied and found the optimal window size.

Different mechanisms for processor performance improvement are introduced in this work. A combination of disk scheduling, page replacement, branch handling and parallel programming are used to ensure better processor performance under different situations. The result obtained from these methods shows reduction in the fault rate along with reduced waiting time.

Contents

CONTENTS

List of Figures

LIST OF FIGURES

List of Tables

Abbrevations

BFQ	Budget Fair Queuing
BTB	Branch Target Buffer
C-LOOK	Circular Look
CaMMEBH	Cache Memory Management with Efficient Branch Handling
CFQ	Complete Fair Queuing
CMPs	Chip multiprocessors
CPF	Cache Probe Filtering
FCFS	First Come FirstServe
FDP	Fetch Directed Prefetching
FIFO	First in First Out
FTQ	Fetch Target Queue
I-cache	Instruction Cache
LFU	Least Frequently Used
LIFO	Last in First Out
LRU	Least Recently Used
LRU-LFU	Least Recently Used and Least Frequently Used
LRU-MFU	Least Recently Used and Most Frequently Used
MBFQV1	Modified Budget Fair Queuing Version1
MBFQV2	Modified Budget Fair Queuing version2

ABBREVATIONS

MFU	Most Frequently Used
MRU	Most Recently Used
MRU-LFU	Most Recently Used and Least Frequently Used
MRU-MFU	Most Recently Used and Most Frequently Used
OpenMP	Open Multiprocessing
PC	Program Counter
TLS	Thread-Level Speculation

1

Introduction

1.1 General

In the current world, it is almost impossible to imagine a life without computers. They play many important roles in society, like to promote communication and interaction between users, provide a way to shop, play games and have access to education etc., as well as provide a convenient way to create and store valuable information along with media and files. Various types of softwares are used for these activities. Traditionally, software has been designed for sequential computation, that is one instruction may be executed at a time. The major problem of sequential computation is the increased response time due to the long execution time. In most cases, serial programs run on modern computers and "waste" potential computing power. To overcome this problem, the concept of parallel processing is introduced.

Nowaday the word 'parallel processing' is very commonly used. It is used extensively even outside the realm of computer world. In computer field [Computer Architecture and parallel processing, Kai Hwang, Faye A Briggs, Tata McGraw-Hill] parallel processing is an effi-

cient form of information processing which emphasizes the exploitation of concurrent events in the computing process. Concurrency implies parallelism, simultaneity and pipelining. The word Parallelism is not only applicable to the multiprocessing environment, but to uniprocessor environment as well. In a uniprocessor environment parallelism can be obtained through methods like multiplicity of functional units, parallelism and pipelining within CPU, overlapped CPU and I/O operations, use of hierarchical memory system, balancing of subsystem bandwidths and multiprogramming and timesharing. Among these the first five methods are related to hardware technology and the last one is related to software technology. Almost all the modern operating systems are incorporated with multiprogramming, multiprocessing and even multi threading capabilities.

Implementation of parallelism needs special hardware and software support. As mentioned earlier hardware parallelism is defined by the machine architecture and hardware multiplicity [Computer Architecture A Quantitative Approach, 4th edition, John L Hennessy and David A Patterson, Elsevier]. The level of hardware parallelism is a trade-off between cost and performance. It depends on the performance of the processor and resource utilization patterns of the multiple parallely executed processes. Software parallelism relies on algorithm, programming style and program design. Extent of software parallelism can be identified by varying the program flow graph, that lists the sequence and flow of CPU operations in a program.

Under programmatic level,parallelism can be achieved at job or program level, task or procedure level, inter-instruction or intra-instruction level. The highest level of parallel processing is achieved through multiprogramming and multiprocessing where multiprogramming requires a parallelly processable algorithm. Quality of parallel computing can be expressed as a measure of granularity, the ratio of computation to communication. It can be either coarse-grained, medium-grained or fine-grained [http://www.cs. utexas.edu]. Coarse-grained category includes processes with relatively large amount of computational work between communication events. Medium and fine-grained categories

includes process with medium and small amount of computational work respectively. Multiprogramming provides coarse-grained parallelism.

The processor performance is highly dependent on the delivery of the correct instruction or data at the appropriate instance. If the requested data is missing (not available at the particular instance) the searching of the missed data will be done based on the principle of inclusion. Principle of inclusion states that [Computer System Architecture, 3rd edition, Morris Mano, PHI] when ever a data miss occurs, a search of the data takes place as per the memory hierarchy. Memory at each level is a subset of its higher level. Data can be transferred between various memory levels in a parallel manner. If the data miss is occurring in the cache memory, the processor has to spend more cycles for fetching data from the lower memory levels. One of the methods used to reduce cache miss is instruction prefetching, which in turn will increase the number of instructions readily available for the processor. Branch target buffer of modern processors handle target address of branches to fetch ahead an instruction stream for increasing the performance of the processor [A.J. Smith,1982].

Instruction prefetching is a technique used by the central processing unit to speed up the execution of a program by reducing the wait states. Wait state is a delay experienced by a processor when accessing the external memory. When the instruction or data block is actually needed, it can be fetched much more quickly from the cache than other memories. Thus, prefetching reduces memory access latency. It is a useful technique for addressing the memory wall issue. Memory wall is the growing disparity of speed between CPU and off-chip memory [http://istc-bigdata.org/index.php/memory-wall-what-memory-wall/]. Main reason for this disparity is the limited communication bandwidth beyond chip boundaries known as bandwidth wall.

Generally programs are executed sequentially, so the instructions are prefetched in program order. Whenever a non sequential execution comes, the prefetch may be part of a complex branch prediction

algorithm, where the processor tries to anticipate the result of a calculation and fetch the right instructions in advance. Most of the high-performance processors use some type of prefetching techniques.

Though the processor speed has increased drastically over the years, throughput of the processor has not increased proportionally. This lag is due to unavailability of efficient mechanisms for proper delivery of data to the processor. Branch predictors play a crucial role in achieving effective performance in many modern pipelined microprocessor architecture [https://web.njit.edu/ rlopes/Mod5.3.pdf]. Commonly used methods for branch predictions are software prefetching and hardware prefetching. In software prefetching the compiler will insert a prefetch code in the program. In this case since actual memory capacity is not known to the compiler, it will lead to some harmful prefetches. In hardware prefetching, instead of inserting the prefetch code, an additional hardware examines memory access sequences for common patterns. Using the behaviour of the stored commonly accessed patterns, address of the next instruction stream is generated by the prefetcher. The guessed addresses are placed into the prefetch queue. If there are no pending access requests, the request from the prefetch queue is placed for access.

Bandwidth is the amount of data that can be transferred form one memory component to the other. Prefetchers trade bandwidth for latency. If the prefetched address is correct, more bandwidth is required to reduce latency. Whereas if incorrect prediction is made, the bandwidth is reduced and latency increased. Processor idle condition during program execution also lead to system performance degradation. In a pipelined system the time that is wasted with a branch misprediction is equal to the number of stages in the pipeline, starting from fetch stage to execution stage. All the prefetching methods concentrate only on the fetching of the instruction for execution, and not on the overall performance of the processor.

Current instruction cache prefetching schemes have higher memory acces latencies. Data prefetching has been used as one of the mecha-

nisms to reduce memory access latencies. To be effective in practice, prefetching requires accurate timing. This timing issue adversely affects multi-processor systems. Study shows that timing and scheduling of prefetch instructions [R. S. Chappell, et al., SSMT] is a critical issue in software data prefetching and prefetched instructions must be issued in a timely manner for them to be useful.

If a prefetch is issued too early, there is a chance that the prefetched data will be replaced from the cache before it is referred by the processor. It may also lead to replacement of other useful data from the higher levels of the memory hierarchy. If the prefetch is issued too late, the requested data may not arrive before the actual memory reference is made, thereby introducing processor stall cycles. This may pre-empt the current running sequence from the processor.

Design of an effective prefetching algorithm should consider minimizing the prefetching overhead. This is a big challenge and it needs more thought and effort. Even though a number of prefetching methods are available, none of them proved to be accurate. It is also observed that in both hardware and software implementations of the prefetching techniques, branch prediction and handling takes place during runtime. A better idea would be to use this technique during compile time instead of runtime. This can be done by introducing a new branch prediction table created during the initial phase of compilation.

Tokens identified during lexical analysis are classified into branch and non branch codes. Branch prediction table contains entries for each branch related tokens. The entry in the table are current address and the branching address along with the token, which can be used for handling the prefetching. Also the look ahead distance for branching is considered for better utilisation of the available memory capacity as in the case of memory management section.

Along with branch handling, system performance greatly depends up on disk scheduling, page replacement algorithm, and compiler level parallization. In case of multiprocessing environment more than one

process is going to access the disk at the same time. Memory management unit of the operating system can handle this situtation by using a round robin scheduling. In this method the requests are queued as per their order of arrival time and equal time slots are given to each of the process. If the process is not able to complete the transfer with in the time slot that process will undergo the pre-emption. Here the scheduling of the process is done based on time domain which is also known as Complete Fair Queuing (CFQ) [Paolo Valente and Fabio Checconi, 2010]. The main problem of this method is low throughput. Hence, the concept of scheduling based on service instead of time was proposed. In this method a budget value is assigned to each of the process. The budget stands for the number of sectors occupied for the storage of that process. This type of scheduling is known as Budget Fair Queuing (BFQ) [Paolo Valente and Fabio Checconi, 2010]. Here, when the budget value is set high the throughput is low. By making some changes on BFQ the throughput can be increased.

Page replacement algorithm decides which memory pages to be swap out or write to disk, when a new page of memory needs to be allocated. Paging happens when a page fault occurs and a free page cannot be used to satisfy the allocation. If the page that is selected for replacement and swapped out is referenced again, then it has to be paged in which involves waiting for I/O completion. The quality of the page replacement algorithm depends on lesser waiting time for page-ins, then better the algorithm. A page replacement algorithm considers the limited information about access to the pages provided by the hardware. It tries to guess which pages should be replaced to minimize the total number of page misses while balancing this with the cost (primary storage and processor time) of the algorithm. Least Recently Used (LRU) is the page replacement algorithm used in most operating systems. This makes use of only time of usage of the page for the replacement operation. Here an attempt is made to make use of frequency of the usage of page along with the time for page replacement.To resolve the various issues an elaborated study on the memory management module of the various operating systems is done, which includes, OpenVMS, Microsoft windows, Unix-like operating systems

(including Mac OS X, Linux, Solaris) and z/OS are carried out [M G Sobell, A Practical Guide to solaris]. In case of windows operating system'page fault' is refered as 'hard fault'.

Today all operating systems are designed in a multithreaded environment. But the compilers used are not considering this fact. Here the loops which are executable in parallel are identified and they are executed using various threads. On the other hand, automatic parallelization offered by compilers only extracts parallelism from loops and the compiler can assure that there is no risk of dependence violation at runtime. Only a small fraction of loops falls into this category, by leaving many potentially parallel loops unexploited. Thread-Level Speculation (TLS) [Cosmin E Oancea, Alan Mycroft, 2008], [Paraskevas Yiapanis, Gavin Brown, Makel Lujan, 2016] techniques allow to extract parallelism from fragments of code that cannot be analyzed at compile time.

Many technological advancements usually bring new hardware which replaces the old ones and leading to the obsolete of the existing ones by generating E-waste. This is one of the biggest environmental problems that threaten the entire humanity. This work mainly focuses on how to increase the processor performance (uniprocessor environment) with out adding any extra hardware. Instead of introducing new hardware, performance improvement by design modification of system software is considered.

Different mechanisms for processor performance improvements are introduced in this work. A combination of disk scheduling, page replacement, branch handling and parallel programming are used to ensure better processor performance under different situations.

In round robin scheduling only the arrival time of the processor is considered for scheduling. BFQ is suggested as an improvement of CFQ by Paolo Valente and Fabio Checconi [Paolo Valente and Fabio Checconi, 2010], [Nandhini Sivasubramaniam, Palaniammal Senniappan, 2014]. A new version of BFQ+ is implemented to overcome the draw back of BFQ. Two revised versions of BFQ is proposed in this

work to improve the throughput further. Among these one is similar to BFQ+ [P. Valente, M. Andreolini, 2014].

The refinement of page replacement algorithm mainly concentrates on modifying LRU algorithm by incorporating frequency of page references along with the time of reference.

Prefetching module of the operating system is modified to fetch pages corresponding to branch false condition also in addition to branch true condition.

In parallel programming whenever a dependency between the variables are detected inside the loop statement then that loop will be executed in the sequential order only. TLS is a method in which even if the dependency is found between the variables inside the loop the execution is done in parallel. TLS is used to introduce compiler level parallelism. Performance was studied by varying the number of cores, number of threads within the core as well as number of windows used within the cores. The conditions that lead to higher performance were then identified in this phase.

The experiments were done by editing the Linux kernel. Linux is one of the most popular open source operating system. The Linux kernel is a monolithic kernel, which means that the whole operating system is on the RAM reserved as kernel space [https://www.inso.tuwien.ac.at /uploads /media.pdf]. The kernel owns that space on the RAM until the system is shutdown. In contrast to kernel space, there is user space. User space is the space on the RAM that the user's programs own. The Linux kernel is also a pre-emptive multitasking kernel. This means that the kernel will pause some tasks to ensure that every application gets a chance to use the CPU. Portability is one of the best features that make Linux popular. Portability is the ability for the kernel to work on a wide variety of processors and systems that permits the editing of the kernel as per our requirements.

Implementation of Linux kernel is a collection of modules. Each module has specific actions to be performed. The entire kernel source

code is partitioned into a number of subsections which incorporates millions of code lines. Kernel code area where all memory management operations are performed is mm module. Both kernel space and user space are taken into consideration while managing memory. The page replacement policies, allocation and deallocation of cache memory regions etc are handled in 'mm module'.

TLS part of the work is done on OpenMP . OpenMp is compiler directive based and it is portable and multi-platform which includes Linux.

1.1.1 Layout of the Thesis

Chapter 1 points out the need for parallel processing in todays world along with the various methods that can be adopted to reduce the page fault rate inside the system.

Chapter 2 takes up the study on BFQ disk scheduler. Comparative study is done with CFQ. The design of two new disk scheduling methods such as MBFQV1 and MBFQV2 are also included.

Chapter 3 describes the study of four page replacement methods and their implementations. A new set of replacement methods are introduced and their performance evaluation has been carried out.

Chapter 4 illustrates the development of a new branch handling method CaMMEBH and compares its performance with the existing method.

Chapter 5 deals with the integration of MVBFQV2, LRU-LFU and CaMMEBH. Comparative study is performed on each level of integration.

Chapter 6 explores the use of TLS method in parallel programming. The performance evaluation is also done.

Chapter 7 concludes by summarizing the results in the work and the possible developments in future.

The performance evaluations of different algorithms are done on various systems with different configuration. The result shown in the thesis is taken from the system with the following configuration: Intel I core processor, 4 GB RAM, and a 700 GB IBM-DTLA- 307030 SATA IDE hard drive. Linux Operating System kernel 3.14.30. Parallel programming is done using OpenMp in GCC compiler with version 4.6.2.

2

Disk Scheduling with Equivalent Bandwidth Sharing

2.1 Abstract

This chapter concentrates on the enhancement of data transfer speed between the secondary memory and primary memory. Currently, Complete Fair Queuing (CFQ) based disk scheduling techniques are used in various operating systems. CFQ is a timestamp based scheduler, where high throughput is obtained by just idling the disk for a short time interval after the completion of each request. Applications such as file copy or transfer, Web, DBMS or video streaming make use of synchronous disk requests in order to access the data from the secondary storage devices. In Complete Fair Queuing the time slots are assigned to requested process ie, it works in time domain. This time interval allowed for the disk access of each application includes access latency which consequently reduces the system throughput. In order to over-

come the above problem a new disk scheduler, Budget Fair Queuing (BFQ), is considered and which is based on service domain instead of time domain. In BFQ a budget will be allotted to each process with a maximum budget value. In order to study the performance variation, the analysis of CFQ scheduler and the implementation and analysis of BFQ scheduler were done and it was observed that performance of BFQ scheduler is better. But the results show that the application with smaller file size will lead to starvation, ie they have to wait in the request queue, for a long time. To overcome this problem two new methods were suggested. The first method Modified Budget Fair Queuing Version1 (MBFQV1), the budget is calculated based on half of the maximum budget of the number of requests present in the queue initially. In the second method Modified Budget Fair Queuing version2 (MBFQV2), the budget is modified each time, based on the average of the budgets of the processes present in the request queue. The result of MBFQV2 shows better performance compared to MBFQV1 and BFQ.

2.2 Introduction to Disk Scheduling

Disk scheduling is a method by which data is transferred from secondary memory to primary memory. Performance of the disk scheduler depends on access time parameter. Access time is the average time required to search for a particular storage location in the memory to access its contents. Access time consists of seek time and transfer time. Seek time is the time required to position the read-write head to a location where data is residing. Seek time is usually much longer than the transfer time (time required to transfer data to or from the device) [Computer System Architecture, 3rd edition, M. Morris Mano, PHI]. Transfer rate is the number of characters or words that the device can transfer per second after positioning the head to the beginning of the record. In a multiprocessing environment, more than one process may access the disk at the same time. Memory management unit of the operating system can handle this situation by using a round robin

scheduling method. In this method the requests are queued according to their arrival time and equal time slots are given to each of the process. If the process is not able to complete the transfer with in allotted slot, it will undergo pre-emption. Complete Fair Queuing (CFQ) [Paolo Valente and Fabio Checconi, 2010] is the scheduling method used for this purpose. The low throughput is the main drawback of this method. As a solution to this problem, a new approach is suggested by Paolo Valente and Fabio Checconi [Paolo Valente and Fabio Checconi, 2010] in which scheduling is done based on service instead of arrival time. In this method a budget value is assigned to each of the process. The budget stands for the number of sectors occupied for the storage of the given process. This type of scheduling is known as Budget Fair Queuing (BFQ) [Paolo Valente and Fabio Checconi, 2010].

2.3 Complete Fair Queuing

The purpose of the scheduler is to schedule the processes which are requesting for accessing the disk in the computer system. Most of the applications, such as file transfer, Web, DBMS, Video on Demand or Internet TV etc., need to transfer large amount of data to or from disk devices. The time needed to serve a request will depend on the seek time, rotational latencies, and the variation of the transfer rate. In a multiprocessing environment multiple number of requests are generated inside the system and they are kept in a request queue. Request queue is an internal queue inside the system to store the pending requests for disk access by various applications. Majority of applications usually issue synchronous [S.layer and P. Druschel, 2001] request. Synchronous means one request or batch of requests at a time. When one request is in service the other requests are blocked till the ongoing process has been completed. This type of request or batch of requests is called synchronous. In order to obtain effective utilisation of resources, when multiple requests are activated, disk scheduler divides the resource of disk I/O among the pending requests in the system.

Pending requests are requests which are waiting in the request queue. Pending requests are selected based on the current position of the disk head, to reduce the access latency. When multiple requests arrive to the system a timestamp is given to the processes based on the arrival time. The dispatch of the request to the disk is done based on the ascending order of timestamp. This is the basic working concept of timestamp based schedulers.

Some processes can be initiated only when the previous process is completed. The problem faced here is that due to the delay of arrival of first process, the dispatching of the second process is delayed. Delayed synchronous request may get higher timestamp only because of the delayed invocation. If the preceding request is activated earlier they may get better time stamp. Due to this higher time stamp it has to wait longer time in the request queue getting dispatched to the disk. It delays service and hence the completion of the request will delay the arrival of the successive synchronous request of the same application, and so on. Due to the delay in servicing of the request, scheduler may force the application to issue requests at a deceptively lower rate. If this occurs, the scheduler just fails to guarantee the reserved bandwidth or the assigned request completion time to the application.

In addition to the above problem a minimum amount of time is needed for an application to handle a completed request and to submit the next synchronous request which will prevent the scheduler from achieving a high disk throughput. This minimum amount of time, is the time taken to read the data and locate the position of next set of data to transfer. Until the next synchronous request is issued the application will be deceptively idle. During such idle time, the disk head may be moved away from the current position. Thus loosing the chance of a close access for applications which are near to the disk head, and that will result in increased rotational latency. Even if all these problems are there CFQ [http://mirror.linux.org.au/pub/linux.conf.au] is the most widely used disk scheduling technique in current operating systems.

2.4 Budget Fair Queuing

Budget Fair Queuing (BFQ) is a service based disk scheduling method. It uses a proportional share disk scheduling algorithm, which relies on the slice-by-slice service scheme of CFQ. In BFQ, budget is assigned to each requested process. This budget is measured in terms of number of sectors used by that process for storage. The disk is assigned to the active process until it has exhausted its assigned budget. During that time no other process is allowed to access it. In CFQ, allotment is done based on the duration of time slice but in case of BFQ it based on the budget value. In CFQ the variation in the workload as well as the number of requested processes, will affect the duration of time slot given to active process. But in the case of BFQ, the disk bandwidth can be distributed among processes without any distortion due to workload variations or other factors, such as in and out movement of the processes to the request queue. To overcome this problem a new method called BFQ+ is introduced [P. Valente, M. Andreolini, 2014].

BFQ uses an internal scheduler (BFQ scheduler) to schedule tasks according to their budgets. This BFQ scheduler is responsible for the budget calculation and the assignment of the budget value to the various process that require less seek time, which boosts the throughput compared to CFQ and guaranties low latency. The Figure 2.1 shows the diagrammatic representation of BFQ scheduler.

2.4.1 Working of BFQ Scheduler

BFQ scheduler handles requests from application queue. The application queue is the same as the request queue present inside the computer system. When a request for a process is generated by the processor, it is enqueued in the application queue or request queue. BFQ scheduler will calculate the budget value of each of the process present in the application queue and select the maximum value and

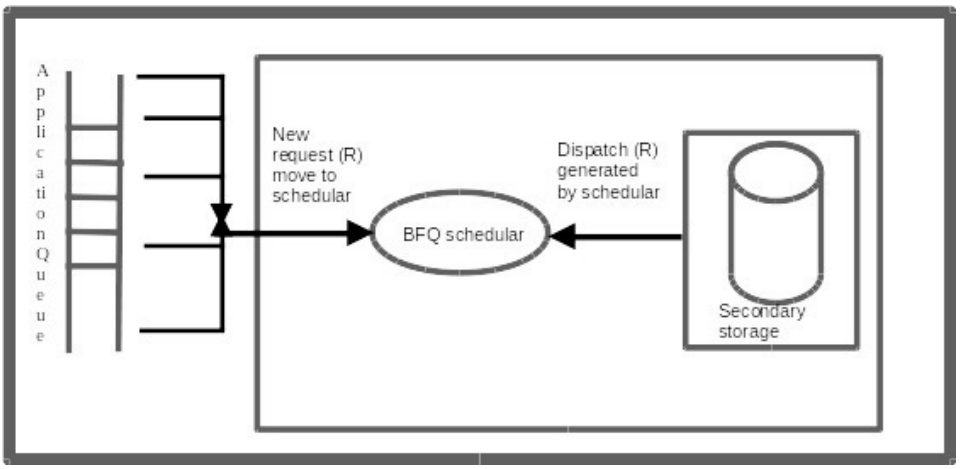

Figure 2.1: BFQ system model

assign it as the budget (B_max). The selection of the process is done based on the first come first serve (FCFS) order and it gets exclusive access to the disk. During the service of the process disk idling is performed to wait for the arrival of synchronous requests only if these requests are deemed sequential. The process will continue the access from the disk until the budget value is reached or data transfer is completed. In BFQ the budget value will be fixed to B_max always. Due to this the job with small budget value has to wait for a long time in the request queue. Even if BFQ starts the scheduling process based on the arrival of the request later all enqueued request will be arranged based on the Circular Look (C_LOOK) [Operating system concepts, 5th edition, Abraham Silberschatz, Wiley], [Operating system Design and implementation, A S Tanenbaum and A S Woodhull, PHI] disk scheduling algorithm. C-LOOK is a disk scheduling algorithm used to determine the order in which disk read and write requests are processed. C-LOOK basically scans in only one direction. Either sweep from inside to out, or outside to in within the disk surface. When the disk head reaches the end, the head is moved all the way back to the beginning. This actually takes advantage of the fact that many drives can move the read_write head at high speeds if it's moving across a large number of tracks , which will reduce the seek time. Caches (disk

caching) are usually tuned to reduce disk access and achieve feasible response times.

To guarantee a controllable maximum queuing time, the order in which the requests are extracted from the queue of the active process depends on a user-configurable TFIFO parameter [Paolo Valente and Fabio Checconi, 2010], [http://www.lohninger.com/helpcsuite/how_to_use_fifos.htm]. TFIFO is the queuing time after which the queued requests must be served in FIFO order. When the next request of the i-th process is to be dispatched at time 't' if 'ai'(arrival time of i-th application) + TFIFO>t holds for all the queued requests Rj. Once it is dispatched the next request is chosen in C-LOOK order, otherwise the oldest request is picked.

As the application work is in the service domain and not in the time domain, they are scheduled by the BFQ scheduler as a function of their budgets. Regardless of the time interval, workload and disk physical parameter, BFQ guarantees to each application a minimum possible delay, achievable by serving applications budget-by-budget. In order to perform the analysis of CFQ with BFQ the implementation is done on the various versions of Linux kernel and experimentally evaluated its single disk performance with file transfer time, transfer rate and throughtput with several system files and other application programme.Table 2.1 will give specification of the testing environment.

Table 2.1: Specification of the testing environment

Specification
Processor:Intel i5 4 GB RAM 700 GB IBM-DTLA- 307030 SATA IDE hard drive. Linux kernel 3.14

2.4.2 Implementation

The new disk scheduler is added to the kernal as a header file using the #include "bfq.h". The algorithm used for the implementation of BFQ is given in ALGORITHM 2.1 :

ALGORITHM 2.1:-

Step 1: Input: Application index, request issued by the application
Step 1.1 Insert the request R in the request queue
addrequest(int i, request R)
appl = applications[i]; / / reference to the i-th application
Step 1.2 Activate the timer for waiting time calculation
Step 2: Generate tree stucture for the storage of the request
Step 3: Generate weighted tree corresponding to the request tree
Step 3.1 Activate the sector calculation function
(budget calculation)
Step 3.2 Assign the corresponding budget to various requests
Step 4: Select the request for service
Step 4.1 Calculate the sector distance from the current
position to the request.
Step 4.2 Fix the budget value
Step 4.3 Dispatch the request requestdispatch( )
Step 5: Activate data transfer function
step 6: Activate the timer function
Step 7: Check the status of the dispatch request
Step 7.1 If the request is servised completely
Step 7.1.1 The request should be removed from the application queue
Step 7.1.2 The request should be removed from the tree
Step 7.1.3 The request should be removed from the weghted tree
Step 7.2 If it is not serviced completely
Step 7.2.1 Calculate the remaining budget value
Step 7.2.2 Store the information about how much is serviced
Step 8: Extract the next active application from queue

Step 8.1 go to Step 4.3

All the request comes to the scheduler is stored in a tree sturucture using the function 'static void bfq_rq_pos_tree_add(struct bfq_data *bfqd, struct bfq_queue *bfqq)'. From this tree a weighted tree is generated based on the budget value using the function 'static void bfq _weights _tree_add(struct bfq_data *bfqd, struct bfq_entity *entity, struct rb_root *root)'. For the calculation of the weight paramerter the size of the application and the number of sectors used are made use of. With the help of budget allocation function the generation of the weighted tree will be completed.

Once the requests are comming to the application queue the selection of the request should be done using the function 'static struct request *bfq_choose_req (struct bfq_data *bfqd, struct *rq1, struct request *rq2, sector_t last)' [www.akhilnarang/ kernal_bullhead]. For selection process the distance between current posion of the head and the destination sector should be calculated using the function 'static inline sector-tbfq-dist-from(sector-tpos1, sector-tpos2)'. After the selection of application calculated budget will be assigned. The selected request should be dispatched using the function 'static inline void bfq_schedule_dispatch (struct bfq_data *bfqd)'.

Once it is dispatched and serviced it should be removed using 'static void bfq_weights_tree_remove(struct bfq_data *bfq, struct bfq_entity *entity, struct rb_root *root)' function. The request should be removed from the all the locations such as the application queue and from the tree sructures .

If the requested application is not serviced completely it should be maintained inside the system for further service. At that time information like, how much is serviced and how much remaining, should be stored in the corresponding data structures. Along with this information the time function is also get activated for the calculation of waiting time. For these purpose the function 'static inline void

bfq_bfqq_save_state(struct bfq_queue *bfqq)'
is used.

The selection of the application is done based on TFIFO along with this the result returned by the sector distance calculation function 'static inline sector-tbfq-dist-from(sector-tpos1, sector-tpos2)' is also considered. Using this information the function 'static bfq_queue * bfqq_close(struct bfq_data *bfqd, sector_t *sector)' selects the next application. Using this function we are able to reduce the seek time parameter, which reduces the waiting time.

The timing functions should be activated in all the required places for the calculation of the various time parameters such as system time, user time , and waiting time. Flow chart representation of Algorithm 2.1 is given in Figure 2.2.

2.4.3 Need to Modify BFQ

As mentioned earlier in BFQ the budget (B_max) will be always set as the maximum value of the budget in the request queue and kept it as static value. Which will lead to the starvation of the smaller processes which are waiting in the application queue. To overcome this problem some modification is done on the above method and two new versions are generated, Modified Budget Fair Queuing version1(MBFQV1) and Modified Budget Fair Queuing version2(MBFQV2). In MBFQV1, budget (B_max) is reduced to half of the Maximum Budget. MBFQV1 is able to increase the average throughput when compared to BFQ. One of the problems encountered with the MBFQV1 was that the calculation of B_max was done only in the beginning. As a solution to this, in MBFQV2, the value of the budget (B-max) is modified in real time, as the average of the budget values, of the processes in the application queue. When a process is completed or when ever a new entry is made to the application queue, the new budget value is calculated with out considering the budget value of the completed processes and taking only currently active processes in the request queue.

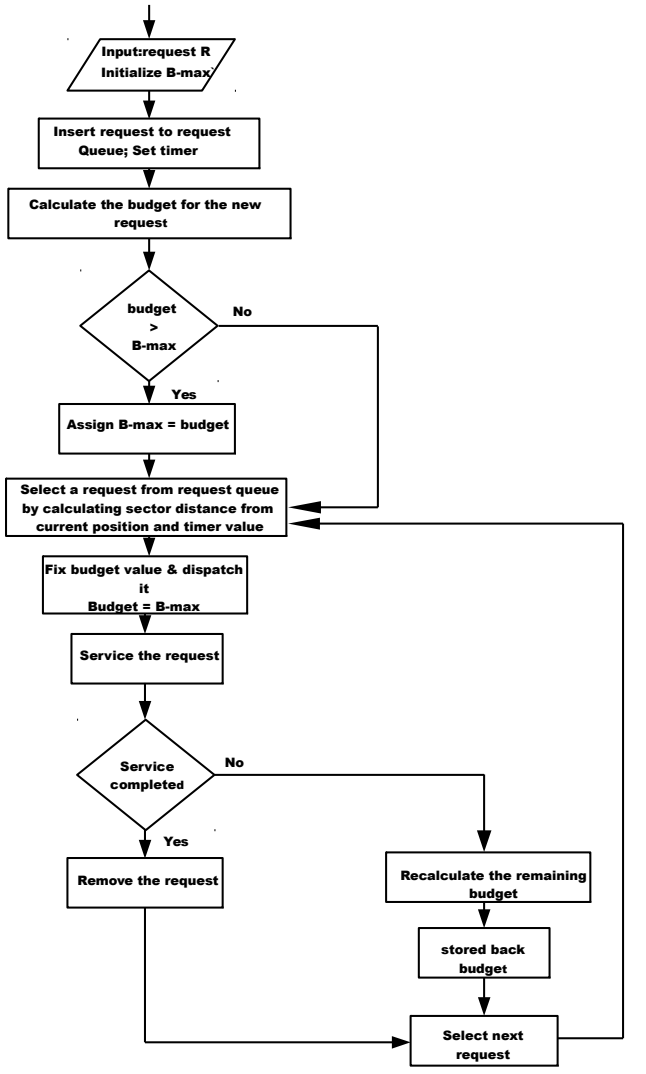

Figure 2.2: Flow chart of BFQ

2.5 MBFQV1

Performance of an operating system is measured in terms of responsiveness. Responsiveness means the number of applications that are getting disk access within a time period. System will give more importance for loading and execution of a program than copying data. Here importance is given to improve the responsiveness of the system further. The processes with a small budget value will improve the responsiveness of the system. But in case of BFQ the application with small budget value will have to wait for long time. For small applications to complete quickly it is necessary to switch between the active applications rapidly. Hence instead of keeping maximum budget value as the B_max it changed to half the value. By this the switching between the processes will take place speedily and the smaller process will complete the transferring because it has only small budget. The size of the application queue will be considered for B_max fixing, when the number of applications in the application queue is more than a specified value (set threshold value as 5). When the number of application in the application queue is less than the threshold value, this method does not give any improvement in performance. Different threshold values were tested on the algorithm and it was observed that a better performance is obtained with a threshold value of 5. If the number of applications for disk access is more than the threshold value, B_max is reduced to half its value. Since the budget is reduced, the switching between the active applications becomes faster. So more number of applications can access the disk within less time which improves the system responsiveness further. If number of application is less than 5 then changing B_max will not affect the responsiveness of the system, which is similar to BFQ. The implementation of MBFQV1 is done similar to BFQ, the selection of the budget is changed. The algorithm which is used for fixing the budget is shown in ALGORITHM 2.2.

ALGORITHM 2.2

```
update_maximum_budget()
{
If(application_queue_length>5) //application_queue_length indicates the
                                  number of requests currently waiting
                                  in the request queue.
{
B_max = max(Application.Budget )/2
                               //Setting the maximum budget value to its
                                  half. }
else
{
B_max =max(Application.Budget) //Setting Bmax to its default max
                                  imum value }
}
```

The change in B_max will be done only when the number of requests is more than the threshold value. When the number of requests gets increased the waiting time for requests also increases. So in order to give access to more number of requests within a particular time period the maximum budget value is reduced to half its value. Here small files get disk access much faster which results in improving the responsiveness of the system. The results of experiments were satisfactory. The system responsiveness was increased further after the reduction in the maximum budget when the number of applications in the queue becomes higher.

2.6 MBFQV2

In MBFQV1 once the B_max is calculated which will be maintained in the system for the entire scheduling process. To make B_max allocation in a dynamic form a new B_max is calculated by considering all the applications which are waiting in the request queue.

2. DISK SCHEDULING WITH EQUIVALENT BANDWIDTH SHARING

Average budget of all the applications present in the application queue is calculated each time and is set as the B_max value. If the average value is greater than the defaultmaxbudget then B-max is set with defaultmaxbudget. This is done because if B-max is very large the system is not able to maintain the basic feature of the multiprocessing environment[Operating systems concepts, 4th edition, Abraham Silberschatz, Wiley], [Operating system Design and Implementation, Andrew S Tanenbaum, PHI]. Along with this, the problem which we found in BFQ will occur here also. That is the applications with small budget have to wait for long time period. Whenever a process is completed, the budget calculation will be reactivated by removing the serviced one from the application queue. Along with these steps C-LOOK scheduling concepts are also considered for the scheduling process, in order to reduce the rotational latency. The implementation of MBFQV2 is done similar to MBFQV1 and BFQ. During the implementation of MBFQV2, consider the algorithm given in ALGORITHM 2.1, in Step 8.1 instead of going to Step 4.3, the go to statement should be given to Step 4.1. Because the budget calculation should be done on current application queue. The algorithm which is used for fixing the budget is shown in ALGORITHM 2.3.

ALGORITHM2.3

```
defaultmaxbudget =16*1024
update_maximum_budget()
{
Budget =0
For(i=0.....maximum_queue_size)
{
Budget =Budget+Application.Budget
Application++
}
B_max =Budget/queuesize     //queuesize is the number application
                              currently in queue
If (B-max>defaultmaxbudget)
{
```

B_max =defaultmaxbudget
}

2.7 Performance Evaluation

The performance evaluation was done based on the existing system with CFQ scheduler, then using BFQ scheduler, MBFQV1 scheduler and MBFQV2 scheduler. Implementation was done on the various Linux kernel versions. The experiment result shown below are obtained when the above methods were implemented in Linux kernel 3.14, PC equipped with an Intel i5 core processor, 4 GB RAM, and a 700 GB IBM-DTLA- 307030 SATA IDE hard drive. The BFQ implementation code was inserted into the kernel. After inserting the BFQ codes into the kernel, the kernel was compiled with new configuration parameters.

To compare the performance evaluation of BFQ with CFQ, applications with varying file size were activated and the results were observed. Transfer time, transfer rate and throughput were the various parameters considered for this. As mentioned earlier the major problem that identified in this method was the waiting time, which leads to the respons parameter. As a solution for this problem MBFQV1 and MBFQV2 were implemented and the analysis was done.

MBFQV1 codes were inserted in to the kernel. And the same set of applications were Specification of the testing environmentactivated again to study about the performance. In MBFQV1, the allocation of the process was done in round robin manner based on the budget value. The pre-emption of the task will be based on the allocated budget. Budget allocation will take place only once as in the case of BFQ. Hence if application queue contains more number of applications with less budget value, it may lead to an increase in the waiting time of the shorter jobs. As a solution to this dynamic allocation of budget

was considered and implemented using MBFQV2. In the implementation of MBFQV2 the maximum budget is set as the average of all the budgets based on the request at that time. In the case of MBFQV2 the budget allocation is completely based on the current working scenario, dynamic assignment of the B_max will take place. During the completion of each task, allocation of the budget is redefined by the calculation of budget once again and is fixed as B_max. Due to this waiting time of the various task gets reduced, which will reduce the data transfer time of the various tasks. The process selection was done based on round robin method but not based on the time slots. Pre-emption process was activated based on the B_max value which was generated by the system.

Once the module was generated, Linux kernel was configured using the commands:
sudo make menuconfig
sudo make oldconfig
then build and install the kernel by using the commands:
sudo make
sudo make modules
sudo make modules_install
sudo make install
Module for BFQ, modified MBFQV1 and MBFQV2 schedulers were installed and tested.

All these applications only issue synchronous requests on a Linux system. The set of experiments were aimed at estimating the aggregate throughput, transfer speed and time for transferring process for each scheduler. Specification of the testing environment Table 2.2 shows the comparison of the file transfer time of various disk schedulers with respect to the various file sizes. When compared with the existing CFQ scheduler, BFQ will consume less amount of time to complete the scheduling and transferring of file. It shows that in CFQ the number of pre-emption is high and which leads to increase in waiting time. In case of BFQ the pre-emption is done based on the maximum budget value. It may also increase the waiting time of the various process those

are having less budget value, and hence it leads to the degradation of the performance. In the case of MBFQV1 this problem is overcomed by fixing the budget as the half of the maximum MBFQV1 also once the B_max is calculated which will be maintained in the system for the entire scheduling process. To make B_max allocation in a dynamic form new B_max is calculated by considering all the applications which are waiting in the request queue. Average budget of all the applications present in the appplication queue is calculated each time and is set as B_max of the entire task which are present in the request queue. Once the scheduler starts the operation the newly arrived request budget is not considered for the fixation of the B_max, and the pre-emption is handled with reference of the budget. In MBFQV1 the time taken by the scheduler is almost in the same level as of the BFQ, with a slight increase in certain cases. But MBFQV1 is not able to produce a better performance when compared to the BFQ. This leads to the implementation of MBFQV2, where B_max value is varied based on the working scenario. The budget allocation is always kept as a varying parameter and the newly arrived request to the queue is also considered for the B_max fixation. Hence the shorter task will be able to complete first and it leads to increase the response parameter. Experimental result shows that MBFQV2 gives a better performance when compared to BFQ and MBFQV1. Except in a file of size 2700MB, in all other cases the time taken is less or same as that of BFQ. In the real stream application the size of the request will always be in the medium range and the result shows that if the request comes under the range then the performance will be better.

Figure 2.3, Figure 2.4 and Figure 2.5 will give a comparison of the BFQ, MBFQV1 and MBFQV2 with the existing CFQ method. In order to make a comparison on the different methods with CFQ, a diagramatic representation is shown in Figure 2.6.

Along with transferring time the transfer speed is also measured for the evaluation. Because it depends on how fast the data is located in the storage media. Table 2.2 shows the information about the transferring speed of the schedulers. Transfer rate is measured in Mega

Table 2.2: Time taken by various schedulers to transfer files

File	Size(MB)	Time in Sec. for various scheduling methods			
		CFQ	BFQ	MBFQV1	MBFQV2
FILE1	2700	854.53	765.64	772.92	768.33
FILE2	1900	820.21	750.91	740.64	738.2
FILE3	1600	794.36	714.92	725.51	714.92
FILE4	1300	733.73	655.83	662.71	633.83
FILE5	995.5	655.24	577.47	578.67	510.49
FILE6	887.1	621.44	558.72	551.26	529.8
FILE7	735.7	552.23	491.58	489.01	481.01
FILE8	734.6	458.53	376.96	400.88	367.35
FILE9	729.7	548.58	489.58	487.06	472.09
FILE10	407.8	322.12	289.12	295.92	293.2
FILE11	6.3	4.74	4.52	3.81	4.61
FILE12	4.3	3.24	3.06	2.76	3.06
FILE13	3	2.46	2.37	1.95	2.37

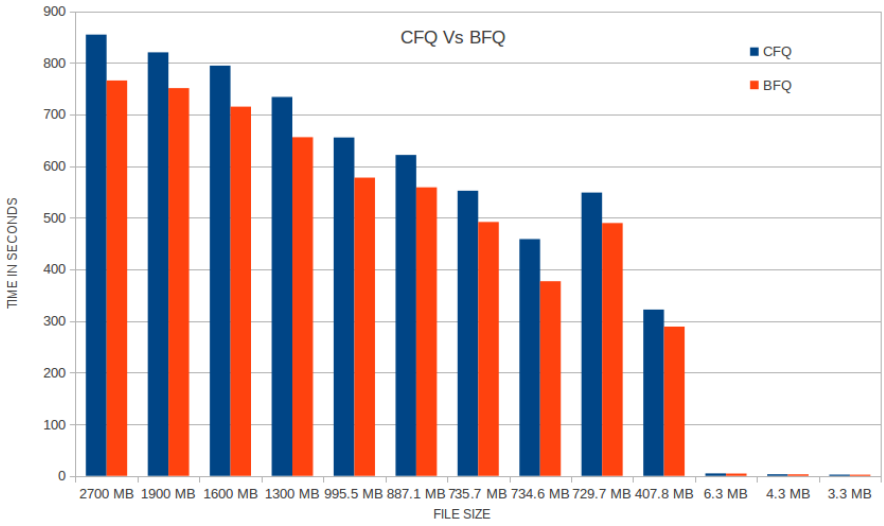

Figure 2.3: Comparison between CFQ and BFQ

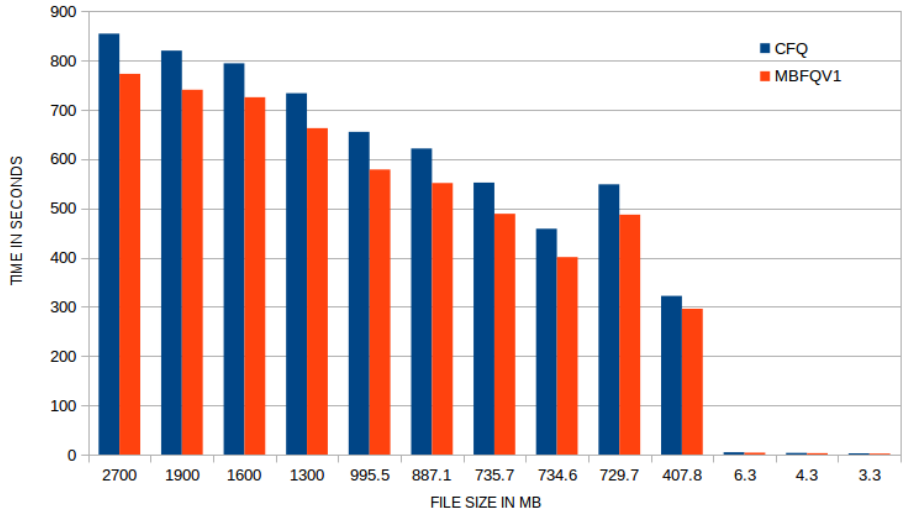

Figure 2.4: Comparison between CFQ and MBFQV1

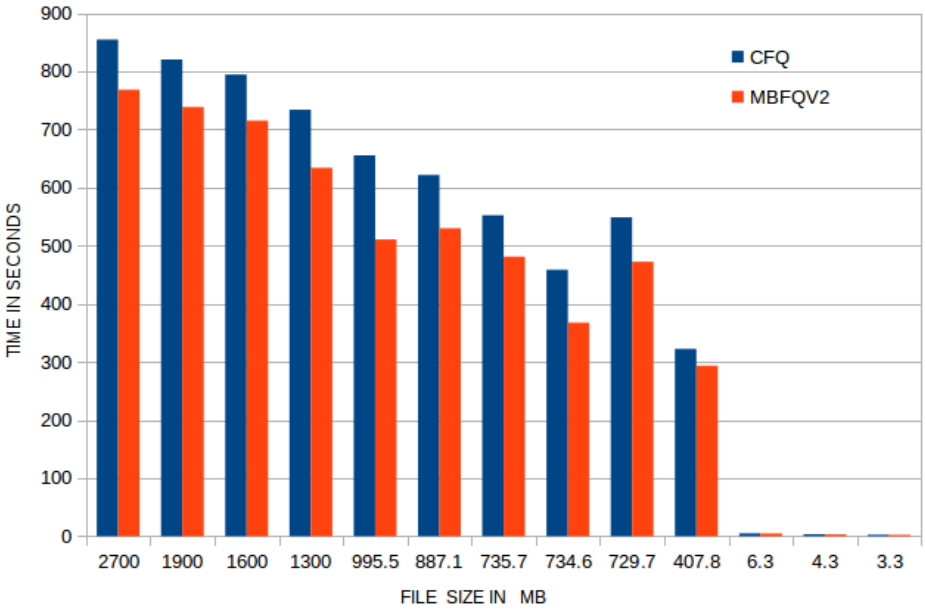

Figure 2.5: Comparison between CFQ and MBFQV2

Bytes/Sec. It gives the comparison of the transfer rate of the various scheduler. Figure 2.7, Figure 2.8 and Figure 2.9 show comparative study of various schedulers with respect to CFQ. From the Table it is clear that the transfer rate of the BFQ is higher than that of the CFQ. After applying the MBFQV1 and MBFQV2 the variations in the rate of transfer is also shown in the Table. The time taken for the schedulers in Table 2.3 includes the waiting time of the process in the requested queue along with the overhead of the pre-emption process.

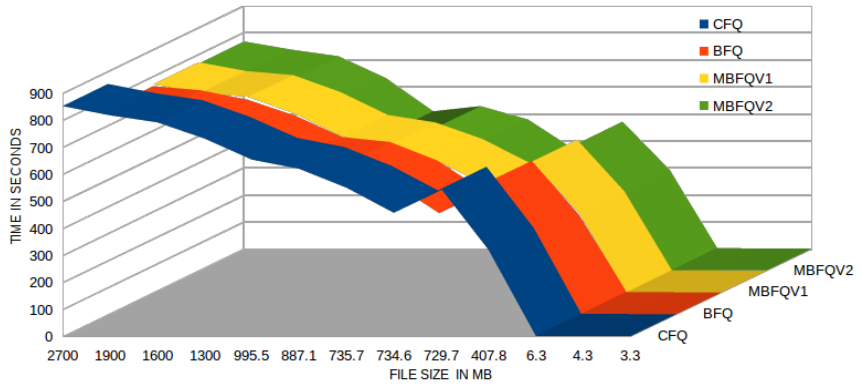

Figure 2.6: Time to transfer the files in various scheduling methods

Hence MBFQV1 is not able to give a better result when compared with BFQ. But in the case of MBFQV2 it is observed that except the file of size 2700MB and smaller size files we are able to get a higher transfer rate. As mentioned earlier size of real stream applications will be normally in medium range and experiment results show that the request comes under that range will give a better transfer rate as shown in Figure 2.11.

Figure 2.10 shows the diagramatic form of Table 2.3, which will give a comparative study of the various methods along with CFQ which is existing in the current system.

Table 2.3: File transfer speed of various schedulers

File	Size(MB)	File transfer speed(MB/Sec)			
		CFQ	BFQ	MBFQV1	MBFQV2
FILE1	2700	3.1596316	3.526461522	3.493246	3.5141
FILE2	1900	2.3164799	2.530263281	2.565349	2.5738
FILE3	1600	2.0142001	2.238012645	2.205345	2.238
FILE4	1300	1.7717689	1.982221002	1.961642	2.051
FILE5	995.5	1.5192906	1.723899077	1.720324	1.95
FILE6	887.1	1.427491	1.587736254	1.609223	1.6744
FILE7	735.7	1.3322348	1.496602791	1.504468	1.5294
FILE8	734.6	1.6020762	1.948747878	1.832469	1.9997
FILE9	729.7	1.3310107	1.490461212	1.498173	1.5456
FILE10	407.8	1.2659878	1.410486995	1.378075	1.4208
FILE11	6.3	1.3291139	1.39380531	1.653543	1.3925
FILE12	4.3	1.3271605	1.405228758	1.557971	1.45607
FILE13	3.3	1.3414634	1.392405063	1.692308	1.3924

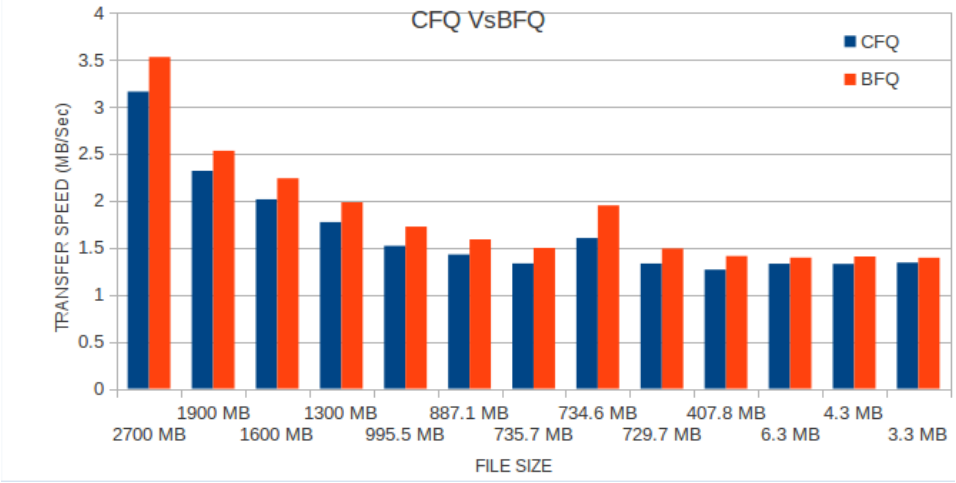

Figure 2.7: Comparison between CFQ and BFQ

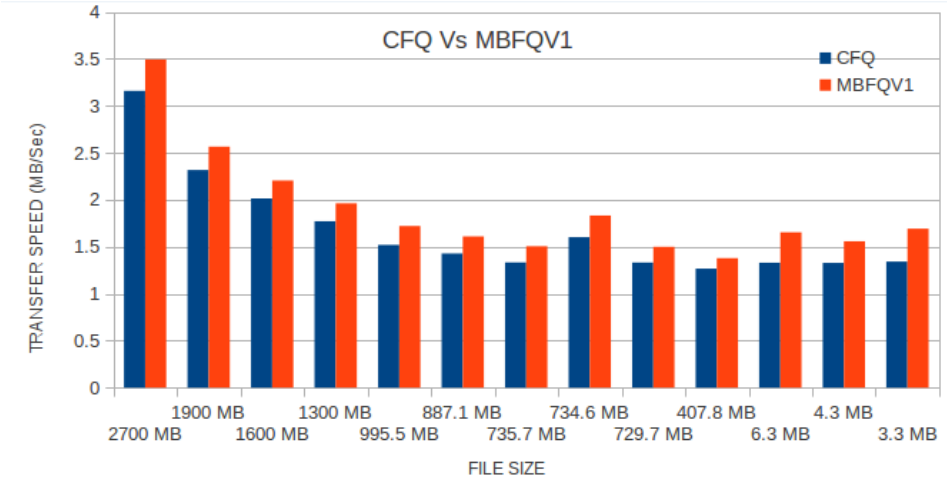

Figure 2.8: Comparison between CFQ and MBFQV1

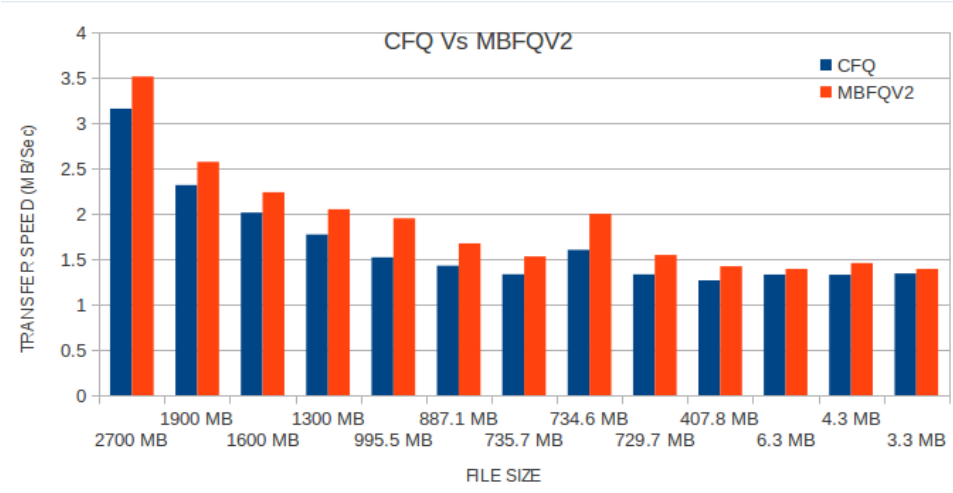

Figure 2.9: Comparison between CFQ and MBFQV2

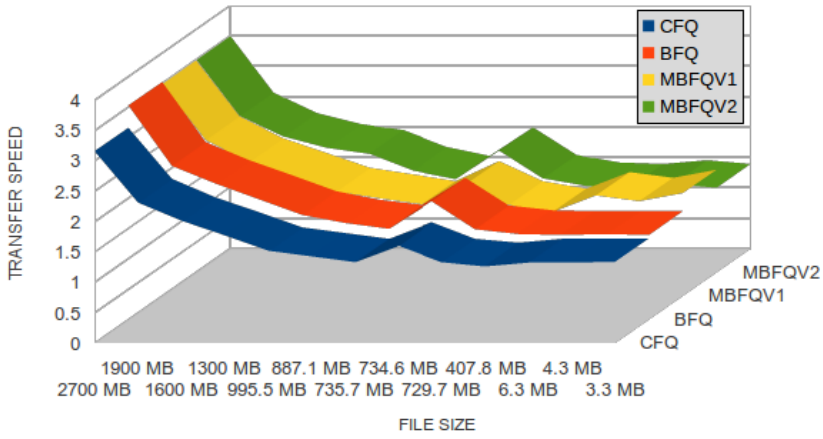

Figure 2.10: Transfer rate of files in various scheduling methods

Table 2.4: Throughput parameter

Throughput(MB/Sec)			
CFQ	BFQ	MBFQV1	MBFQV2
1.67215	1.8559	1.8979	1.9029

The next parameter which can be used for the analysis is throughput. Throughput means the number of jobs which are completed at a particular time period. The results were promising, BFQ displayed a substantial increase in throughput and responsiveness with low latency for applications. The throughput difference between CFQ and BFQ is more compared to BFQ and MBFQV1. Here the reduction in throughput is due to the frequent switching between active applications with the header movement. Hence MBFQV2 is considered and which produced better result. The results obtained from the four schedulers CFQ, BFQ, MBFQV1 and MBFQV2 were analysed and Shown in Figure 2.11. The results clearly indicate that MBFQV2 provides a high throughput in almost all conditions as shown in the Table 2.4.

It is clear from Figure 2.11 that the throughput is maximum in MBFQV2 compared to other schedulers. Hence it give a better performance inside the system

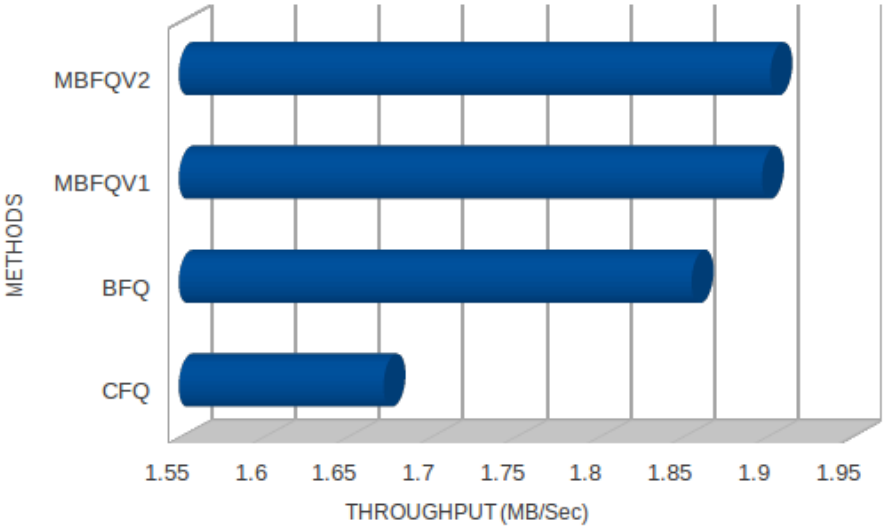

Figure 2.11: Throughput

2.8 Conclusion

This Chapter mainly deals with reduction of waiting time on various synchronous requests. Here instead of existing CFQ, BFQ and modified versions of BFQ are considered. From the experimental results it is observed that MBFQV1 gives a better performance compared to BFQ. Further better throughput is achieved using MBFQV2, where each time new budget value is calculated based on the processes present in the request queue.

3

Handling of Various Page Replacement Techniques

3.1 Abstract

Virtual memory is a concept used in the memory management unit of operating system. Even if the program is only partially available in primary memory the modern operating system will permit the execution of the program. It provides an illusion to the user that a very large memory is available and makes him free from the concern of large program size. As technology grows, the cost of memory gets reduced and its capacity gets increased. The primary memory size has increased by multiple orders of magnitude. As the size becomes several gigabyte, algorithms that are used for periodic check of each and every memory frame for the presence of data is practically difficult. Nowaday the programming strategy is shifting from structured programming to object oriented programming. Due to this, the locality of reference of user software has weakened. Current page replacement methods are based on the locality of reference concept, hence some new methods need to be introduced for page replacement. This chapter mainly fo-

cused on the improvement of processor performance by modifying the page replacement methods. To increase the processor performance, the page faults should be decreased. In order to achieve this a number of page replacement techniques such as LRU, MRU, LFU and MFU are considered. Among the various page replacement methods, Beladys optimal page replacement is the best method [D Stephens, et al., 1999]. But to implement this, the remaining required pages should be known in advance which is not practical at all the times. Modern operating systems mainly make use of the Least Recently Used (LRU) page replacement method, where only the arrival time of the pages to the page frame along with usage of the pages are considered. The number of times the page is referred is not considered for replacement. In this work both the time and the number of reference done on the page is considered. By considering these two parameters, the new algorithms is developed as Least Recently Used and Least Frequently Used (LRU-LFU), which will give a better performance than LRU page replacement method

3.2 Page Handling

Paging is a memory management technique where the memory is divided into fixed size pages. Paging is used for faster access to data. If a page is not available in the main memory, when the processor needs it, the operating system copies pages from secondary storage device to main memory. Paging allows the physical address space of a process to be non-contiguous.

Page replacement algorithm decides which memory page is to be swap out when a new page is to be allocated. Paging happens when a page fault occurs and the available free page space cannot be used to satisfy the allocation [Computer organization and architecture designing for performance, 10th edition,William Stalling, Pearson, 2017], [A practical Guide to solaris, M G Sobell]. When the page that was selected for replacement is paged out and is to be referenced again then

it has to be paged in, and this involves waiting for I/O completion. The quality of the page replacement algorithm is based on the time spent for paging. Lesser the waiting time, better the algorithm. A page replacement algorithm considers the limited information about access to the pages provided by hardware, and tries to guess which pages should be replaced to minimize the total number of page misses, while balancing this with the costs (primary storage and processor time) of the algorithm itself

Algorithms can be of online or offline in nature. An online algorithm is a strategy which, at each point of time, decides what to do based only on the past information and with out any knowledge about the future. Where in offline a full information about the requirement of the pages is available. An online algorithm A is presented with a request sequence $\sigma = \sigma(1)$;$\sigma(2)$; ..... ;$\sigma($ m). The algorithm A has to serve each request online , i.e., without future requests. More precisely, when serving request $\sigma($ t), $1 \geq$ t $\geq$ m , the algorithm does not know any requests σ (t^1) with $t^1 >$ t [Susanne Albers, Germany]. Serving requests incurs cost, and the goal is to serve the entire request sequences so that the total cost is as small as possible. Paging problem is an example of the online problem. In paging problem, consider a two-level memory system that consists of a small fast memory and a large slow memory. Here, each request specifies a page in the memory system. A request is served if the corresponding page is in the fast memory. If a requested page is not in the fast memory, a page fault occurs. Then a page must be moved from fast memory to slow memory so that the requested page can be loaded into the vacated location. A pageing algorithm specifies which page is to be evicted on fault. If the algorithm is online, then the decision as to which page is to be evicted must be made without the knowledge of any future requests. The cost can be minimized if the total number of page faults incurred on the request sequence is less. Performance of an online algorithm is done using competitive analysis [D.D. Sleator and R.E. Tarjan, 1985]. In competitive analysis, an online algorithm A is compared to an optimal offline algorithm. An optimal offline algorithm knows the entire request sequence in advance and can serve it with minimum cost. Given

a request sequence σ , let C A (σ) denote the cost incurred by A and let C OPT (σ) denote the cost paid by an optimal offline algorithm OPT. The algorithm A is called c-competitive if there exists a constant 'a' such that

$$\text{C A } (\sigma) < \text{c .C OPT}(\sigma) + \text{ a}$$

for all request sequences . Here we assume that A is a deterministic online algorithm. The factor 'c' is also called the competitive ratio of A .

Marking algorithms is a general class of paging algorithms [M.Chari kar, 2013]. The algorithm processes a request sequence in phases. At the beginning of each phase, all the pages in the memory system are unmarked. Whenever a page is requested, it is marked. On a fault, a page is chosen uniformly at random from among the unmarked pages (if there exists no unmark pages then unmark all pages) in fast memory, and these pages are evicted. A phase ends when all pages in the fast memory are marked and a page fault occurs. Then, all marks are erased and a new phase is started.

If ALG is a marking algorithm [http://algo2.iti.kit.edu / vanstee / courses / caching.pdf] with a cache of size 'k', and OPT is the optimal algorithm with a cache of size 'n', where n≥k, then ALG is

$$\frac{k}{k-n+1}$$

-competitive. So every marking algorithm attains

$$\frac{k}{k-n+1}$$

the competitive ratio. The section 3.3 deals with the various page replacement algorithms.

3.3 Page Replacement Algorithms

As per the definition of the virtual memory, it provides an illusion to the user that a huge amount of primary memory is available to the user. This illusion will allow us to execute a program even if it, partially resides in main memory [Operating system a concept based approch, D M Dhamdhere, Tata McGraw-Hill] . For the implementation of the virtual memory concept memory management module of the operating system makes use of demand paging. Demand paging means, whenever a request is generated for a page, only at that time the page will make an entry to the primary memory. Availability of the data in the cache memory will depends on the availability of the data in primary memory. The overall performance of the system is affected by the choice of page replacement technique used in virtual memory. Page replacement is one of the active research areas where many techniques have been suggested by different researchers. There are a number of page replacement methods available such as First in First Out(FIFO), Last in First Out(LIFO), Least Frequently Used(LFU), Most Frequently Used(MFU), Least Recently Used(LRU), Most Recently Used(MRU) and Belady's optimal method. All these methods will make use of either the arrival time of the page to the page frame or the number of times the page is referred by the processor for their implementation, which in turn referred as the time stamp or the frequency stamp in further explanation [B Juurilink, 2004], [S Khanjoueinejad, et al., 2007]. The Beladys optimal page replacement method, cannot be implemented efficiently because the order of page requirement (reference string) is not known in advance. LRU is one of the most popularly used page replacement methods among various operating systems [E J O'Nelill, et al., 1993]. Whenever a new page is moving towards the higher memory level the first step is searching for free space in the memory. If there is no free space available then replacement activity should be initiated. For that identify a victim page (page which is taken out from the memory). In LRU, selection of victim page is done based on the page that has not been accessed

for a long period of time among the pages those are present in the current memory. The selection of the victim page is done on the basis of two conditions; the page that program does not need to access further or the program is facing page faults because it is not able to access the page. In LRU page replacement implementations the pages are not discriminated and are treated equally [SongJianga and, Xiaodong Zhangb, 2004]. In this work a combination of the concept of time stamp and frequency stamp is used.In Combined the LRU and LFU Policies(CRFP) method an adaptive replacement policy is used, which is self-tuning and can switch between either LRU or LFU [Zhan-sheng Li, Da-wei Liu, Hui-juan Bi, 2008] [N Dafre, D Kapgate,2014].

3.3.1 First in First Out(FIFO)

The simplest page-replacement algorithm is First in First Out algorithm [Operating system, Willam Stallings, Pearson]. The FIFO page replacement algorithm is a low-overhead algorithm that requires little bookkeeping on the part of the operating system. As the name indicates, the operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the oldest arrival in front. When a page needs to be replaced, the page at the front of the queue is selected.

3.3.2 Last in First Out (LIFO)

This algorithm works just opposite to the FIFO concept. It also requires little bookkeeping on the part of the operating system. In this method, operating system keeps track of all the pages in memory in a queue, with the oldest arrival at the back, and the most recent arrival in front. When a page needs to be replaced, the page at the front of the queue is selected [Operating system a concept based approch, D M Dhamdhere, Tata McGraw-Hill].

3.3.3 Least Frequently Used (LFU)

When ever a page is referred inside the system a counter is incremented to count the number of times that page is referred. In this method, the page with the smallest count is replaced. Here the assumption is that an actively used page will have a large reference count [Operating System concepts, 6th edition, Silberschatz, et al., Wiley]. But this will create a situation where if a page is used heavily during the initial phase of the process, and is never used again, it will have large count due to which it remains in memory even though it is no longer needed. One solution to this problem is to reduce the count at regular intervals.

3.3.4 Most Frequently Used (MFU)

This is based on the argument that the page with the smallest count is more likely to be used in the future. By taking this in consideration the page with maximum count will undergo the replacement. The implementation of these algorithms are expensive [Operating System concepts, 5th edition, Silberschatz, et al., Wiley].

3.3.5 Least Recently Used (LRU)

This method makes use of the time when the page is to be used. If the recent past is an approximation of the near future, then replace the page that has not been used for the longest period of time. LRU replacement associates with each page, the time of the page's last use. When a page must be replaced, LRU chooses that page which has not been used for the longest period of time. This strategy is the optimal page replacement algorithm, looking backward in time, rather than forward [Operating system a concept based approch, D M Dhamdhere, Tata McGraw-Hill].

3.3.6 Most Recently Used

This method also makes use of the time when the page is used. If the recent past as an approximation of the near future, then replace the page that has been used for the longest period of time [Operating system a concept based approch, D M Dhamdhere, Tata McGraw-Hill]. MRU replacement associate with each page the time of its first use. When a page is to be replaced, MRU chooses that page which has been used for the longest period of time.

3.3.7 Belady's Optimal

An optimal page-replacement algorithm has the lowest page-fault rate of all the algorithms. Replace the page that will not be used for the longest period of time. Use of this page replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. It is difficult to implement, because it requires future knowledge of the reference pages. Hence it is used for comparitive studies [Operating System concepts, 6th edition, Silberschatz, et al., Wiley].

In the above set of all algorithms except Belady's optimal all others are coming under online algorithms. All the online algorithms make use of any one of the parameters either time or frequency for implementing the page replacement policies [A Meyerson, 2004]. Here an attempt is made to consider more than one parameter, time of usage along with its frequency for the design of the new algorithms.

3.4 Handling of Page Fault

Whenever the processor requests for data, as per the principle of inclusion, first the data will be checked in the registers and if there

is a miss then the request reaches the cache memory, after that primary memory area and finally to the secondary storage area. Inside the system, data search always follows the memory hierarchy. Whenever a miss condition arises the search will extend towards the lower memory levels. To handle this miss condition, the data should make a move towards the higher memory level from the lower one. Memory management module of the operating system tries to handle the miss condition by making the required page available in higher memory levels or by terminating the program in case of an illegal access. This miss condition is called as page fault in primary memory and as cache miss in the case of cache memory. Fault handling is done in the same way in both primary memory as well as cache memory.

Primary memory is divided into number of page frames and cache is divided in to a number of cache lines. The technique mentioned here is applicable to both secondary memory to primary memory transfer and primary memory to cache memory. The only difference is that whenever a fault occurs in primary memory, it is referred as major fault and a fault occurred in cache is referred as minor fault. The memory management unit is the one that detects the page fault in the operating system [Operating system design and implementation, Andrew S Tanenbaum and A S Woodhull, PHI]. If a minor fault is detected inside the system it can be handled by the memory management unit. If there is a major fault present, exception handler gets activated. The software used for exception handling in case of page fault is generally part of the operating system. Page fault is not an error condition. Rather, it is a method used to increase the amount of memory available in each level by the operating system.

Minor page fault is created when a process attempts to access a portion of memory before it has been initialized. That is, a potential source of memory latency is called a minor page fault [http://blog.scoutapp.com/articles/ 2015/04/10]. When this occurs, the system needs to perform some operations to fill the memory maps or other management structures. The existence of a minor page fault depends on system load and other factors, but they are usually short and have

a negligible impact. The fault which occurs in the cache memory will come under this fault level.

Major fault occurs when the system has to synchronize memory buffers with the disk, swap memory pages belonging to other processes, or undertake any other Input or Output activity to free memory. When the processor references a virtual memory address that doesn't have a physical page allocated to it, the reference to an empty page causes the processor to execute a fault, and instructs the kernel code to allocate a page and return, all of which increases latency dramatically. [[http://blog.scoutapp.com/articles/ 2015/04/10], [Programming Manual,Logix5000 Controllers Major, Minor, and I/O Faults] This is the major fault that occurs in the primary memory level. Due to this reason, major fault is more expensive than minor fault.

Major fault is the mechanism used by an operating system to increase the amount of program memory available on demand. The loading of the program from the disk to the memory is delayed by the operating system until a page fault is generated. During page fault, the requested page should make an entry to the corresponding memory level to make it a page hit in the next request handling time. If there is no free space available in the respective memory area, replacement algorithms should be activated. In order to handle this problem the page fault handler in the operating system needs to find a free space in the memory to load the requested page. For that one of the currently existing page should be swapped out from memory (victim page) to generate room for the incoming page [Operating system a concept based approch, D M Dhamdhere, Tata McGraw-Hill]. There may be some pages that are shared by various processes. If the handler select such a page as the victim, the operating system needs to write out the data in that page (write back policy) and mark that page as not loaded in memory in its process page table. To mark the page as not available the corresponding entry in the page table should be marked as invalid. Whenever a page is loaded in memory the corresponding entry will be done in the process page table which indicates that the page is loaded. During each swapping process a number of updates have to be done

by the handler, only then the accessing of the data by the processor can take place. Hence major fault and minor fault inside the system play a big role in the evaluation of the performance of the system.

Along with the page fault the performance of a system can be measured in terms of execution time of a program. The execution time of a program is measured based on the following parameters user time, system time and elapsed time. The user time is the time in 'user mode' (outside the kernel) within the process by the operating system. User time is the time spent for doing calculations. The system time is the time spent by the kernel executing in system mode or kernel mode or supervisor mode on behalf of the process. System time is how much time the operating system spends responding to process requests. Elapsed time is the time taken by the system in order to execute the program including the pre-emption condition in a multiprogramming environment. Elapsed time is the sum of user time, system time and over all waiting time of the process.

These various parameters can be extracted from the system by using the system command, time./a.out in Linux. Sum of user time and system time will give the actual processing time of the current process, but the time taken by the machine will include the elapsed time parameter. Elapsed time depends on the type of process scheduling that is implemented inside the system along with the pre-emption condition. The change in the page replacement method doesn't have much influence in the user time and system time. The page replacement mainly affects the elapsed time. Here the pre-emption due to page fault will be reflected in the result. Hence when the execution of a single process is considered elapsed time doesn't have much importance. But nowaday all the operating systems are designed based on multiprocessing and multithreading concept, hence the elapsed time becomes an important parameter.

3.5 Implementation of Various Page Replacement Methods

In order to perform the analysis of various replacement methods the module is implemented in Linux kernel version 3.13. Advanced version of Least Recently Used (LRU) page replacement policy is made use in Linux kernel version 3.13. The LRU algorithm is based on the assumption that pages that are not used recently will not be needed frequently in the immediate future. Such pages are therefore taken as the victim page by the page replacement method for swap-out, when there is a scarcity of memory. The fundamental principle used in LRU may be simple but the implementation is difficult. The following sessions will deal with the implementation of various replacement methods except LRU because it is the page replacement method present in the experimenting platform.

The implementations of various algorithms are done on Ubuntu 14.04. By modifying the current kernel of operating system the following algorithms are implemented, MRU, LFU, MFU, LRU-LFU, MRU-LFU, LRU-MFU and MRU-MFU. The page replacement section of the memory management is done under the modules vmscan.c, swap.c which comes under the mm_types.h header file in Ubuntu.

Each physical page in the system has a struct page associated with it to keep track of the reference of the page at the current time. There is no way to track all the tasks those are using a page. 'rmap' structure can tell who is mapping the page at the current time. Here the pages are arranged in circular queue which consist of a doubly linked list that contains both prev and next pointers, prev points to the previous page and next points to the next page. The two variables inside the 'struct page' will give the information about the usage of the pages. 'atomic_t_count' variable will return the number of count of page and 'atomic_t_mapcount' variable shows when the page was referred. These two variables are made use for the implementation of various algorithms. Current LRU implementation is done based on

the time parameter, 'atomic_t_mapcount'. If more than one page with same count value is found prefetch schedular will select the page according to FIFO.

When we make use of both the variables the frequency parameter inside the page structure is assigned to 'compound' variable and then it is left shifted by six bits, to make it a large value. This will be summed up with time parameters. This compound variable associated to each page is used for LRU-LFU, MRU-LFU, LRU-MFU and MRU-MFU implementations. In these algorithms first the frequency parameter is considered for selection of victim pages and if more than one page has the same frequency value, then time parameter is used for the selection. If both these parameters are same, the system follows the existing method (LRU).

For implementing these codes corresponding '#define prefetch_prev_ lru _page()', '#define prefetchw _prev_lru_page()', 'static unsigned long isolate_lru_pages()' functions are edited, which come under 'vmscan.c'. Function 'void mark_page_accessed()' should be modified, which comes under swap.c. struct page and a number of functions which are part of mm_type.h, should be modified as per the requirement.

3.5.1 MRU

In this method time stamp is used. The selection of the victim page is based on the assumption that the most recently used page is not likely to be used in the future. The MRU algorithm is based on the assumption that pages used recently will not be needed frequently in the immediate future. In the existing system, for the implementation of Least Recently Used algorithm, reference is made to each of the pages with the help of a pointer.

3.5.2 LFU

Here instead of using time stamp the frequency parameter is used to indicate how many times the pages residing in the respective memory are referred. For the implementation of LFU a parameter frequency is added as unsigned long data type to the struct page. This is used as a counter or as a frequency parameter. Whenever a new page is referenced the value of 'freq' variable is incremented. Whenever a replacement has to be activated the function 'isolate_lru_pages', will identify the victim page. The pages are sorted with the help of an inbuilt sorting mechanism depending on the frequency parameter 'freq'. Merge sorting is the built in sorting method available inside the system whenever a sorting operation is required. Victim page is selected with smaller frequency value by taking the assumption that the pages which are referred few number of times are not likely to be referred in recent time. Once the victim page is identified it is replaced with the new requested page.

3.5.3 MFU

In MFU, the frequency parameter is used as in the case of LFU. To implement this, a parameter 'freq' is added as the unsigned long data type to the struct page which is used as the counter or a frequency parameter. Whenever a new page is referenced the value of 'freq' is incremented. Whenever a replacement has to be activated the function 'isolate_lru_pages', will identify the victim page. The pages are sorted with the help of an inbuilt merge sorting mechanism depending on the frequency parameter 'freq'. The page with maximum frequency value will give room for the upcoming page. Here the assumption is that the more frequently referred pages are not likely to be referred in recent future.

3.5.4 LRU-LFU

Instead of using one parameter, this method makes use of both the time and frequency paramerters. The LRU-LFU, needs to consider the existing parameter available in LRU along with new parameter for frequency 'freq'. For the implementation, another data field is used in the struct page and is named as 'compound'. 'compound' is used to take care of the above combination in a well defined manner. Majority of the operations are carried out in the function 'mark_page_accessed', which adds a page to the list whenever the page is referred.

The time parameter is obtained by using the 'function get_monoton ic _boottime (*ptr)' which is used in the function 'mark_page_accessed'. Initially the frequency 'freq' parameter inside the page structure is incremented as it is assigned to compound and then it is left shifted by six bits in order to increase the frequency parameter. Finally the compound parameter will have the boot time along with the shifted frequency parameter summed up. The 'compound' parameter is used for the page replacement of pages which are least frequently used and not recently accessed inside the system. Selection of victim page is done based on the compound function by taking the assumption that pages which are referred minimum number of times as well as which are not referred for a long time period, are not referred further by the system.

3.5.5 MRU-LFU

The MRU-LFU page replacement algorithm also the same 'compound' variable. The operation that was performed in the 'mark_page _accessed' function is changed to meet the new needs. The 'compound' parameter is used for the page replacement which are least frequently and recently accessed inside the system. Selection of victim page is done based on the 'compound' variable by taking the assumption that pages which are referred minimum number of times as well as which

are referred in the recent time period is not referred further by the system

3.5.6 LRU-MFU

As in the case of LRU_LFU, here also 'compound' parameter such as time stamp and frequency are used. The operation that was performed in the 'mark_page_accessed' function is changed to meet the new needs. The time parameter of the system is obtained using the function 'get_monotonic_ boottime (*ptr)', which is used in the function 'mark_page_accessed'. The 'compound', parameter is initialized with the 'freq' parameter which is incremented and left shifted six times and summed to the boot time of the system. The selection of the victim page is done by sorting the pages based on the 'compound' variable. The sorting used to identify the page that are referred maximum number of times as well as not referred for a long time period.

3.5.7 MRU-MFU

In MRU-MFU the pages are sorted inside the new function named freq_sort which sorts the 'compound' variable values associated with each of the pages. The boot time of the system is obtained using the function 'get_monotonic_boottime (*ptr)', which is used inside the function, 'mark_page_accessed'. Initially the 'freq' parameter inside the page structure is incremented as is assigned to 'compound' and then it is left shifted by six bits. Finally the 'compound' parameter will have the summation of boot time and shifted frequency variable. Selection of victim page is done based on the 'compound' variable by taking the assumption that pages which are referred maximum number of times and which are referred in a recent time period are not being referred further by the system.

3.6 Performance Analysis of Various Page Replacement Methods

The analysis is done by comparing various algorithms. This is done with the help of finding the page faults for different programs of different sizes. For the analysis, various inbuilt programs as well as user created programs are considered. The number of page fault along with user time, system time, elapsed time and execution time are considered. The page fault can be in terms of major fault or minor fault. The time of execution of the program is measured in terms of user time and system time.Table 3.1 will give specification of the testing environment.

Table 3.1: Specification of the testing environment

Specification
Processor:Intel i5 4 GB RAM 700 GB IBM-DTLA- 307030 SATA IDE hard drive. Linux kernel 3.14

Both the forward and backward loops should be considered whenever the discussion of replacement is considered. Different programs with same file size are considered in order to analyse the effect of backward and forward loop references and alsoto understand how the changes are reflected in the fault rate and the execution time. For final conclusion one programme from each category (small, medium and large) are selected and are shown in Table 3.2, Table 3.3, Table 3.4, Table 3.5 and Table 3.6. In all the cases each program is executed many number (minimum 15) of times and the average of each one is taken and recorded in the Table. When the program is executed many number of times it is noted that there is a small change in the value due to the presence of the requested page in the respective memory area.

Table 3.2: Details of major page fault in various page replacement methods

Major page fault(average numbers)									
FILE SIZE(KB)	LRU	MRU	LFU	MFU	LRU-LFU	LRU-MFU	MRU-LFU	MRU-MFU	File selected
2.7	10.6	10.5	10.4	10.6	10.7	10.5	10.7	10.7	selected
7.1	0	0	0	0	0	0	0	0	
7.2	0	0	0	0	0	0	0	0	
7.2	0	0	0	0	0	0	0	0	
7.3	0	0	0	0	0	0	0	0	
7.3	0	0	0	0	0	0	0	0	
7.3	0	0	0	0	0	0	0	0	
26.8	0.4	0.3	0.4	0.5	0.6	0.5	0.3	0.5	
36.4	0.4	0.5	0.5	0.3	0.2	0.4	0.5	0.3	
59	8.7	23.4	9.9	3.4	3.6	2.9	2.6	3.1	selected
59	4.1	0.3	4.4	0.5	4.4	4.6	4	0.4	
59	0.5	4.2	0.4	4.3	0.5	0.4	0.8	4.5	
59	5.7	1.6	1.9	8.5	8.1	8.8	9.1	8.5	
686.7	4.6	4.6	5.5	4.6	5.5	5.5	5.5	5.5	selected

LRU page replacement algorithm which is currently implemented has been modified with MRU, LFU, MFU, LRU-LFU, MRU-LFU, LRU-MFU and MRU-MFU. Table 3.2 and Table 3.3 shows the details of the page fault. From these Tables it is clear that there is some difference in the value of the number of page faults in different methods for both major and minor fault generated by the system. Even though this difference is small, it makes a huge improvement in the overall system performance, because the page replacement technique adapted during the occurrence of a major fault or minor fault inside the system triggering a lot of complicated procedures. Experimental results of the above page replacement policies and their respective diagrams are shown in Figure 3.1 and Figure 3.2. From the Table 3.2 and Table 3.3. it is clear that there are some noise in the simulation result. Hence the confidence level and confidence interval is calculated for these samples. Graphical representation of these are given in Figure 3.3 and Figure 3.4 respectively. It is observed that except the file size 59KB all are coming under the confidence interval. Actually 59KB file is taken as special test case which is added with maximum number of forward and backward branching. Hence it will give different fault rate.In both the cases the confidence level is set as 95% and the confidence interval for major fault is calculated as 4.18564 and 0.61436 and for minor fault the confidence interval is calculated as 21364.4719 and 4234.1567.

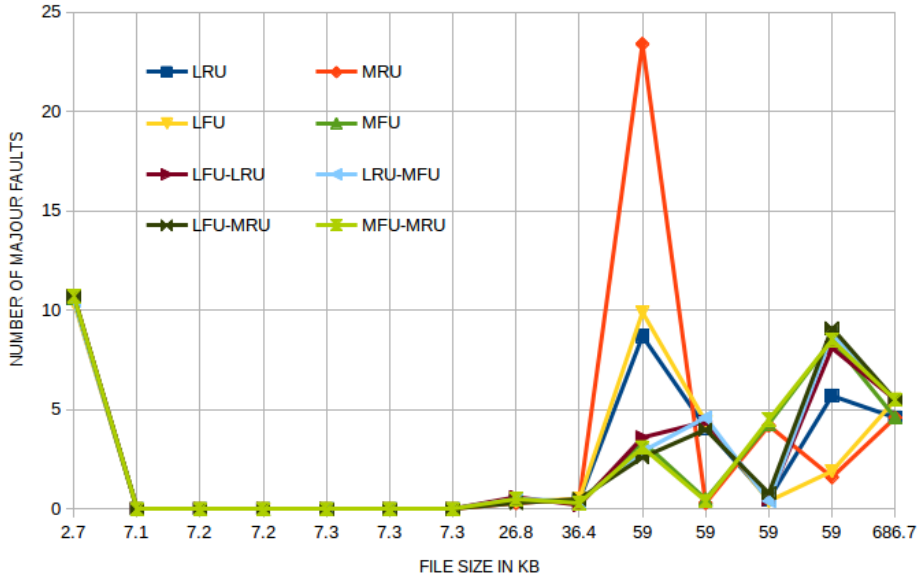

Figure 3.1: Comparison of major fault in various page replacement methods

In Table 3.2 it can be seen that for large program size, especially by using system program, the page fault number will be almost same in case of MRU, MFU and LRU. But it is observed that for the user program (medium size) the major page fault rate shoots up in the case of MRU. In the case of LFU there is a slight increase in major fault rate and due to this the performance is degraded when compared with the LRU. In the case of combinations of the various methods the results of LRU-MFU and MRU-LFU give almost the same result. LRU-LFU performance slightly less than the above two methods in the user program. In the case of MRU-MFU, it is observed that the number of major page fault is reduced in the case of medium size program when compared with the LFU-LRU and is much reduced with LRU algorithm. And it is also observed that for medium size programs the combination will always give a better performance. Even though we are not able to make any far-reaching change in the number of major fault, the number of processor cycles required for each major fault is in the range of thousand of cycles [Computer systems a programmers

Table 3.3: Details of minor page fault in various page replacement methods

Minor page fault (average numbers)									
FILE SIZE(KB)	LRU	MRU	LFU	MFU	LRU-LFU	LRU-MFU	MRU-LFU	MRU-MFU	File selected
2.7	24600.3	24583.4	22533.2	24611	23486.7	24165	24599.6	23516.9	selected
7.1	147.3	148.1	148	147.5	147.9	147.6	148.8	147.7	
7.2	146.4	147.8	147.8	147.8	147.2	147.5	147.8	147.6	
7.2	146.1	146.8	147.1	147.1	147.1	146.8	147.4	147.9	
7.3	146.9	147.2	147.1	147.1	147.6	147.1	147.4	147.7	
7.3	147	147.7	147.8	147.8	148.3	148.3	148.6	148.6	
7.3	145.7	146.6	146.5	146.8	146.8	146.5	146.5	146.4	
26.8	687.8	689.9	691.7	692.4	690	689.7	689.6	691.9	
36.4	937.2	937	936.6	936.9	937.3	936.9	937	936.5	
59	39216.6	39377.5	40813.5	36729.1	38217.3	37961	35205.5	39705.2	selected
59	32902.2	36491.6	31687.8	33033.2	33096.3	31724.1	37528.6	34899.8	
59	37033.5	34356.5	38700.9	37462.2	37336	38838.1	32686.6	35858.3	
59	38578.4	38847.3	35997	40167.1	38724.9	38573.2	41507.7	37269.6	
686.7	5717.7	5786.6	5837.4	6077	5817	5844.9	5694.7	5774.2	selected

perspective, Randal E Bryant, David O Hallaron, Pearson Education]. Hence a small degradation in the major fault will improve the overall system performance.

In Table 3.3 the analysis of the minor fault describes the fact that for small user programs, even LRU, MRU and MRU-LFU shows the same result, but the number of minor faults gets reduced in the case of LFU, LRU-LFU and MRU-MFU. When the size of the program is increased, the existing LRU gives better performance. In the case of LRU-LFU and LRU-MFU the performance of the system is degraded for large size programs. MRU-LFU is able to reduce the minor fault for the small, medium and large size programs when compared to the existing LRU method. The diagramatic representation of Table 3.3 is given in Figure 3.2.

Table 3.4, Table 3.5 and Table 3.6 show the data regarding the various time parameters, which are extracted from the system using the system built-in command over a set of programs that are used for the measurement of page faults. The analysis of this result will give a clear idea about the relationship of the execution time and the page fault number.

In Table 3.4 the analysis of the user time is given. It shows that the existing LRU gives the better user time considering small, medium and

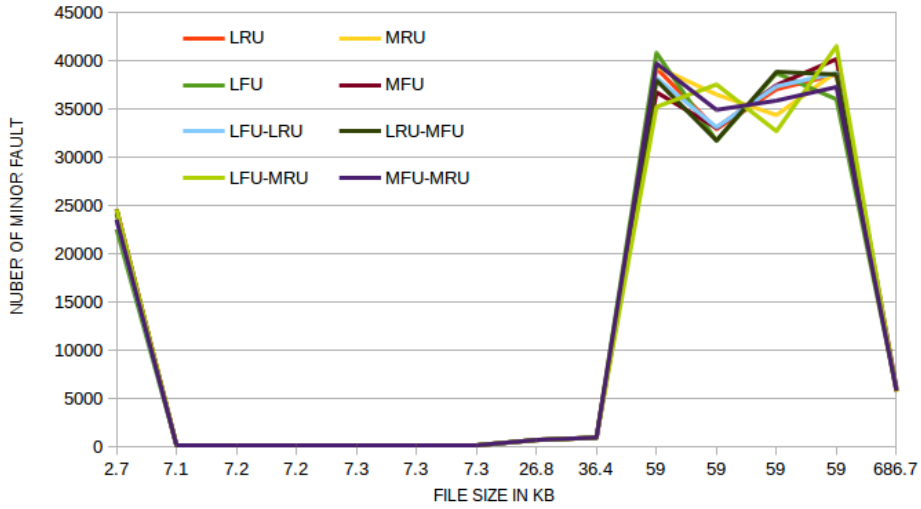

Figure 3.2: Comparison of minor fault in various page replacement methods

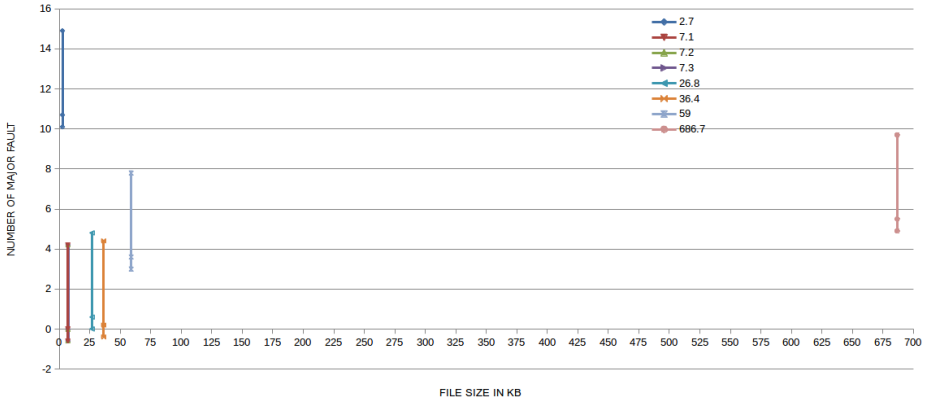

Figure 3.3: Confidence level plot for major fault

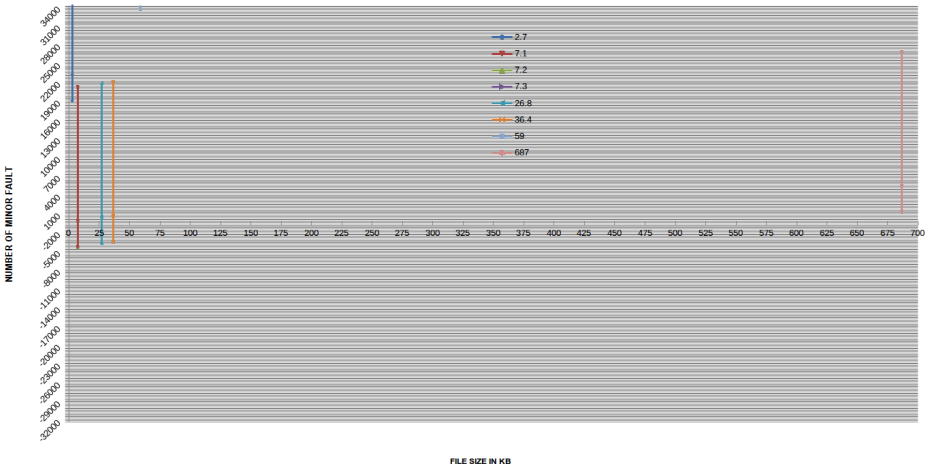

Figure 3.4: Confidence level plot for minor fault

Table 3.4: Details of user time in various page replacement methods

Average User Time(Sec)									
FILE SIZE(KB)	LRU	MRU	LFU	MFU	LRU-LFU	LRU-MFU	MRU-LFU	MRU-MFU	File selected
2.7	2.211	2.017	2.001	1.906	2.147	2.785	2.512	2.35	selected
7.1	0	0	0	0	0	0	0	0	
7.2	0	0	0	0	0	0	0	0	
7.2	0	0	0	0	0	0	0	0	
7.3	0	0	0	0	0	0	0	0	
7.3	0	0	0	0	0	0	0	0	
7.3	0	0	0	0	0	0	0	0	
26.8	0.018	0.014	0.013	0.015	0.027	0.031	0.03	0.039	
36.4	0.032	0.032	0.045	0.03	0.033	0.031	0.039	0.039	
59	1.326	1.269	2.13	1.313	1.605	1.589	1.36	1.986	selected
59	1.36	1.779	1.639	1.439	1.759	1.539	2.378	1.746	
59	1.456	1.109	1.928	1.627	1.782	1.947	1.124	1.577	
59	1.338	1.347	1.214	1.595	1.543	1.708	1.951	1.575	
686.7	1.693	2	0.648	5.714	2.776	3.082	2.004	2.989	selected

large size programs. MFU is the one which consumes the maximum user time when the size of the program increases. The time taken by the LRU-LFU and MRU-MFU are almost the same and is closer to the user time of LRU. The diagramatic representation of the above Table is shown below in Figure 3.3.

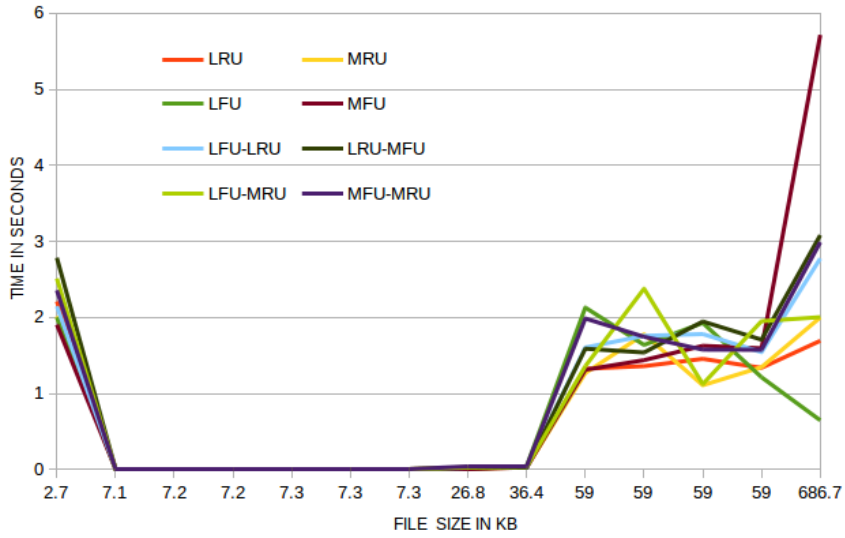

Figure 3.5: Comparison of user time in various page replacement methods

The analysis of system time is shown in the Table 3.4. All methods except LRU consume less amount of system time for programs with average and large size. It is noted that when the system commands are executed inside the system, system time is almost zero in the case when replacement is done with any method other than LRU. LRU-LFU gives better performance while considering the small, medium and large program compared to other methods. The diagramatic representation of the above Table 3.4 is given in Figure 3.4. In elapsed time calculation the system pre-emption condition is also considered because the pre-emption condition may occur due to the unavailability of the data. Hence this is also considered along with user time and system time measurements for the evaluation. The actual execution

Table 3.5: Details of system time in various page replacement methods

Average system time (Sec)									
FILE SIZE(KB)	LRU	MRU	LFU	MFU	LRU-LFU	LRU-MFU	MRU-LFU	MRU-MFU	File selected
2.7	2.718	2.388	2.112	2.433	2.059	2.617	2.313	2.26	selected
7.1	0.004	0.197	0.182	0.171	0.162	0.152	0.175	0.169	
7.2	0	0.236	0.192	0.132	0.162	0.155	0.19	0.181	
7.2	0.002	2.317	2.341	1.605	1.962	1.998	2.027	2.167	
7.3	0	1.981	1.747	1.688	1.584	1.444	2.083	1.83	
7.3	0.004	1.398	2.139	1.687	1.892	1.83	1.502	1.736	
7.3	0	1.964	1.93	1.873	2.07	2.107	2.432	1.919	
26.8	0.138	1.331	1.152	2.239	1.042	0.991	1.045	1.134	
36.4	0.201	0.006	0.004	0.007	0.004	0.005	0	0	
59	1.966	0	0	0	0.001	0	0	0	selected
59	1.473	0.001	0.003	0.002	0.002	0.001	0	0	
59	1.628	0.001	0.002	0	0	0.001	0	0	
59	1.764	0.003	0	0.006	0.003	0	0	0	
686.7	0.968	0.001	0	0.002	0.003	0	0	0	selected

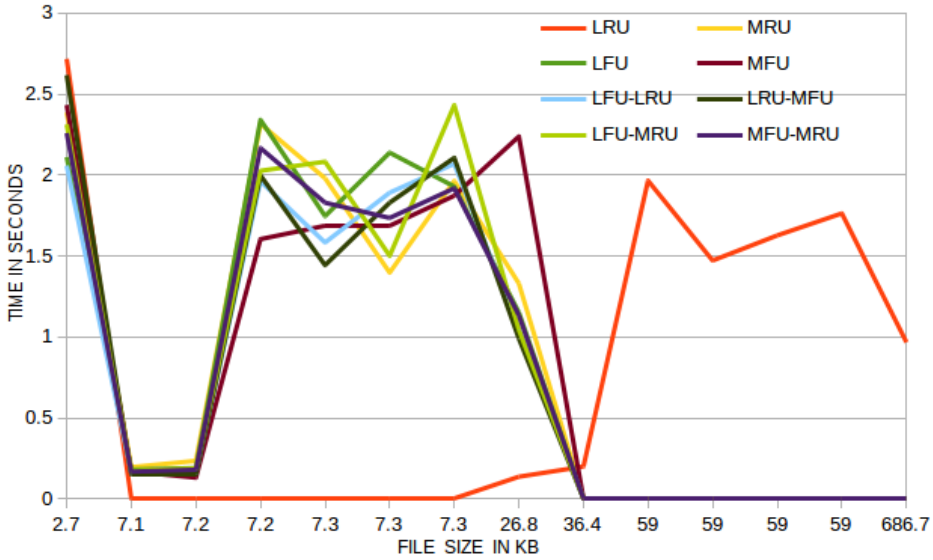

Figure 3.6: Comparison of system time in various page replacement methods

time can be taken by adding the user and system time, which can also be considered for the analysis process. The real time calculation of the elapsed time parameter should also get included. Hence the execution time and waiting time are generated from Table 3.4, Table 3.5 and Table 3.6.

Table 3.7 gives the details of execution time, which is generated from the Table 3.4 and Table 3.5. The execution time is measured as the time taken by the program in both user mode and system mode.

Table 3.6: Details of average elapsed time in various page replacement methods

Average elapsed time (Sec)									
FILE SIZE(KB)	LRU	MRU	LFU	MFU	LRU-LFU	LRU-MFU	MRU-LFU	MRU-MFU	File selected
2.7	2.718	2.388	2.112	2.433	2.059	2.617	2.313	2.26	selected
7.1	0.004	0.197	0.182	0.171	0.162	0.152	0.175	0.169	
7.2	0	0.236	0.192	0.132	0.162	0.155	0.19	0.181	
7.2	0.002	2.317	2.341	1.605	1.962	1.998	2.027	2.167	
7.3	0	1.981	1.747	1.688	1.584	1.444	2.083	1.83	
7.3	0.004	1.398	2.139	1.687	1.892	1.83	1.502	1.736	
7.3	0	1.964	1.93	1.873	2.07	2.107	2.432	1.919	
26.8	0.138	1.331	1.152	2.239	1.042	0.991	1.045	1.134	
36.4	0.201	0.006	0.004	0.007	0.004	0.005	0	0	
59	1.966	0	0	0	0.001	0	0	0	selected
59	1.473	0.001	0.003	0.002	0.002	0.001	0	0	
59	1.628	0.001	0.002	0	0	0.001	0	0	
59	1.764	0.003	0	0.006	0.003	0	0	0	
686.7	0.968	0.001	0	0.002	0.003	0	0	0	selected

Table 3.7: Details of execution time in various page replacement methods

Execution time (MB/Sec)									
FILE SIZE(KB)	LRU	MRU	LFU	MFU	LRU-LFU	LRU-MFU	MRU-LFU	MRU-MFU	File selected
2.7	4.929	4.405	4.113	4.339	4.206	5.402	4.825	4.617	selected
7.1	0.004	0.197	0.182	0.171	0.162	0.152	0.175	0.169	
7.2	0	0.236	0.192	0.132	0.162	0.155	0.19	0.181	
7.2	0.002	2.317	2.341	1.605	1.962	1.998	2.027	2.167	
7.3	0	1.981	1.747	1.688	1.584	1.444	2.083	1.83	
7.3	0.004	1.398	2.139	1.687	1.892	1.83	1.502	1.736	
7.3	0	1.964	1.93	1.873	2.07	2.107	2.432	1.919	
26.8	0.156	1.345	1.165	2.254	1.069	1.022	1.075	1.173	
36.4	0.233	0.038	0.049	0.037	0.037	0.036	0.039	0.039	
59	3.292	1.269	2.13	1.313	1.606	15.89	1.36	1.986	selected
59	2.833	1.78	1.642	1.441	1.761	1.54	2.378	1.746	
59	3.084	1.11	1.93	1.627	1.782	1.948	1.124	1.577	
59	3102	1.35	1.214	1.601	1.546	1.708	1.951	1.575	
686.7	2.661	2.001	0.648	5.716	2.779	3.082	2.004	2.989	selected

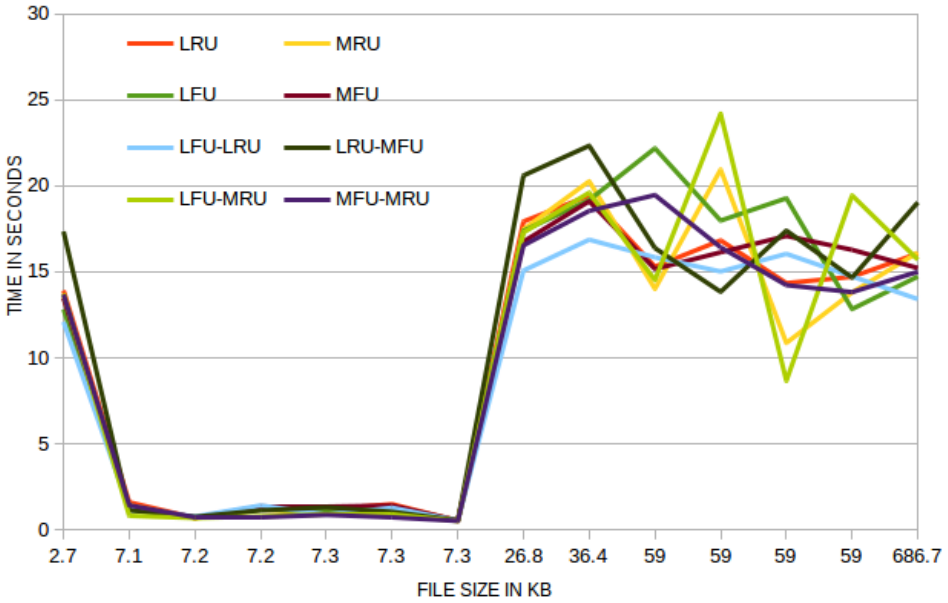

Figure 3.7: Comparison of elapsed time in various page replacement methods

The diagramatic representation of above Table 3.6 is given in Figure 3.5 and that gives a clear picture of the comparison of elapsed time.

The diagramatic representation of Table 3.7 is given in Figure 3.6 and that gives a clear picture of the comparison of actual execution time. The execution time shown in Table 3.7 is generated directly from Table 3.4 and Table 3.5.

To conclude the result analysis, the parameters major fault, minor fault and elapsed time are considered . From the above Tables it is observed that LRU-LFU gives a better value when compared to other methods.

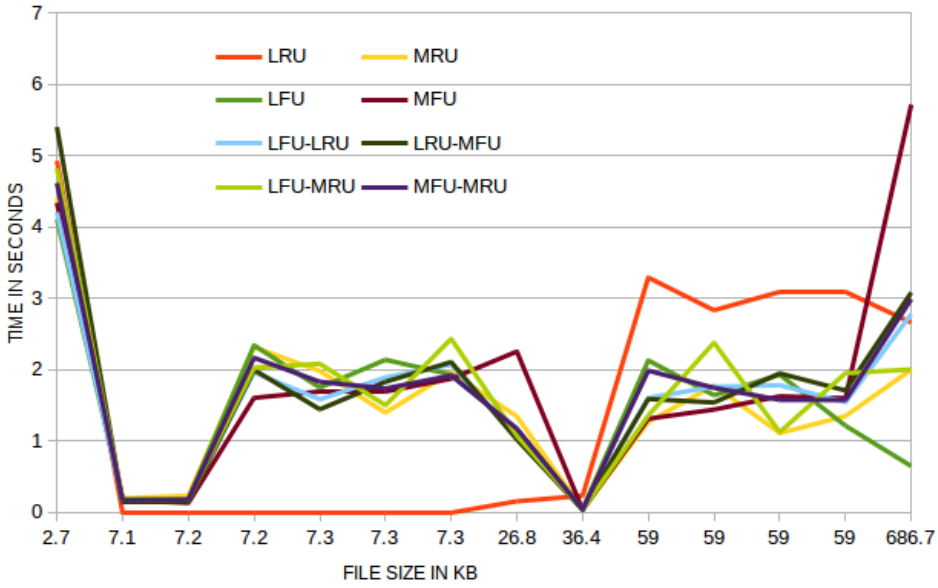

Figure 3.8: Comparison of execution time in various page replacement methods

3.7 Conclusion

It is observed that during the replacement of data from various memory levels, using page replacement algorithm, if the frequency parameter is considered along with time stamp then the performance of the system can be enhanced significantly. The execution time of a file depends on the number of page faults which the system under goes. Hence with a reduction in the page fault number inside, the execution time can be reduced and the system performance can be increased. Various existing LRU, MRU, LFU, MFU algorithms and combinations of these methods are considered and found that LRU-LFU is the one which reduces the major fault and minor fault compared to existing LRU. Due to the reduction in major fault, minor fault and elapsed time the performance of the system gets increased.

3. HANDLING OF VARIOUS PAGE REPLACEMENT TECHNIQUES

4

CaMMEBH for Page Fault Reduction

4.1 Abstract

This chapter deals with Cache Memory Management with Efficient Branch Handling (CaMMEBH) and also discuss the reconfiguration of Linux kernel to improve the system throughput. It aims to keep most probable pages in the cache along with the recently accessed pages through prefetching. Ensuring the availability of required pages in cache is one of the major tasks of kernel in memory management. Any failure of kernel in this functionality may lead to cache miss instead of cache hit. CaMMEBH focuses mainly on branch handling statements to reduce the fault rate in memory. Kernel of an operating system is organized as different modules based on the functions. It can be modified with required changes to reflect the need. Proper kernel modules can be identified and updated so that hit ratio in the cache will be enhanced. The property of locality of reference allows the system to prefetch the pages from one memory level to the next one. CaMMEBH designed to load all the probable destination pages into cache while loading from

main memory, so that maximum hit ratio is found when searching in the cache itself. Normally when a branch statement is encountered in the program a prefetching action is activated only when the branch condition is true and the branch addressed data containing block will be moved to the cache area. While performing this, branch handler is not activated if the branch condition is false. Here an attempt is made to find the solution to this problem. CaMMEBH tries to maintain all required pages in cache memory in both the conditions.

4.2 Study Of Branch Handling

The processor performance is highly dependent on the supply of correct instruction at the proper time. If a data is missed out in the process, the searching of the missed data will be done based on principle of inclusion. Principle of inclusion states that [Computer System Architecture, 3rd edition, M Morris Mano, Person] when ever a data miss occurs it will search the data as per the memory hierarchy. Each memory level i will be always a subset of its higher level i+1. Data will be transferred between various memory levels in a parallel manner. If the data miss occurs in the cache memory the procesinsor has to spend many cycles for fetching the data from the lower memory levels. One of the methods used to reduce cache miss while searching for the instruction is the instruction prefetching, which in turn will increase the number of instructions to be supplied to the processor. Even modern processors uses branch target buffer to predict the target address of branches so that they can be fetched ahead of an instruction stream and that will increase the concurrency and performance of the processor [A.J. Smith, 1982], [N.Fukumoto, et al., 2008], [J.Gummaraju and M.Franklin, 2000]. All the developments in these fields indicate that in future the gap between processing speeds of processor and data transfer speed of memory is likely to be increased. Branch predictor plays a critical role in achieving effective performance in many

modern pipelined microprocessor architecture [Advanced computer organization Processor Architecture, Mattan Erez, 2000], [Marius Evers, 2000]. Commonly used methods for branch predictions are software prefetching and hardware prefetching [Tien_Fu Chen, Jean_Loup Baer, 1994], [Bumyong Choi, Leo Porter, Dean M Tullsen, 2008]. In software prefetching the compiler will insert a prefetch code in the program [A.H.Badawy, et.al. 2004], [C.K Luk, T.C Mowry, 1996]. In this case as the actual memory capacity is not known to the compiler, it may lead to some harmful prefetches. In hardware prefetching, instead of inserting prefetch code hardware monitors memory accesses by looking for common patterns[D.Callahan, K.Kennedy, A.Porterfield, 1991], [S Pintar and A Y Tango, 1996]. The guessed addresses are placed into prefetch queue. Prefetchers trade bandwidth for latency, extra bandwidth is used only when guessing incorrectly and latency reduced only when guessing correctly. In addition to all this during the execution of a program if the processor is waiting for the availability of the next instruction then the performance of the system gets degraded due to idle condition. In a pipelined system the time that is wasted in case of branch misprediction is equal to the number of stages in the pipeline, starting from fetch stage to execution stage [Advanced computer Architecture, parallelism, scalability, programmability, 2nd edition, K Hwang and N Jotwari, TMH]. All the prefetching methods give importance only in fetching the instruction for execution, not to the overall performance of the processor [D Joseph and D Grunwald, 1999]. The section 4.2.1 and section 4.2.2 describe about some branch prefetching methods.

4.2.1 Software Prefetching.

4.2.1.1 Long Cache Lines

Among the various prefetching techniques, Long cache line is the simplest form [A.J. Smith, 1982]. In this method the execution of instruction is considered only in sequential order. In this cache miss

delays can be reduced or eliminated by replacing the current instructions with new instructions that are going to be executed are moved into the cache in advance. When ever the cache lines are moved in to the cache, tag storage should be updated with an entry. In order to reduce the space requirement of the tag storage, longer cache lines are used. When the cache lines are longer the time taken to move it to the cache is more thereby it increases the memory traffic, and they contribute to cache pollution due to the larger replacement granularity [A.J. Smith, 1982]. Structure of a standard program [Advanced Computer Architecture, Lecture 5, Addison-Wesley, 2edition] shows that among the instructions used around 56% of instructions are conditional branching, 10% of instructions are unconditional and 8% of the instructions come under the call return pattern. Due to branching and call return pattern of execution the prefetching needs to undergo non-sequential execution of the program in most cases.

4.2.1.2 Lazy Prefetching

There are certain communication patterns where prefetching is inapplicable or insufficient, especially for producer-consumer patterns. To handle producer-consumer patterns we can make use of Lazy prefetc hing. Data transfer is initiated by the producer, which is a natural style of communication. Concept of remote writes is made use by the producer and it initiates the primitives which move data close to the consumer as soon as the data become available and there by minimizing the latency at the consumers read [Aleksardar Milenkovic, Veljko Milutinovic, 1998]. As in data prefetching concept, in lazy prefetching also expected data is moved closer to the processor before it is actually needed. Lazy prefetching makes use of the good properties of both prefetching and remote write to solve the problem of misprediction of the future. Since it is a software prefetching method the prefetched instructions are inserted into the application code, either explicitly by the programmer or automatically by the compiler. In lazy prefetching consumer (processor) anticipates its future needs of instruction using

ask instruction. The instruction code checks the data cache and if the data is not present in the cache then a cache miss occurs and a request to memory is initiated. A memory agent which is present in the memory accepts the request and checks the memory. If the data requested by 'ask' instruction is a valid one in the memory then the agent sends the requested block to the processor. If the data is not a valid copy the producer have to produce the data from the next memory level and supplied to the consumer, so that it initiates memory update operations using write back instruction. The write back cycle forces the memory agent to deliver the updated block to consumers according to the information from special table [I.K. Chen, C.C. Lee and T.N. Mudge, 1997].

4.2.1.3 Adaptive Prefetching

Timing and scheduling of prefetch instructions is a critical issue in software data prefetching. If a prefetch is issued too early, there is a chance that the prefetched data will be replaced from the cache before its use or it may also lead to replacement of other useful data from the higher levels of the memory hierarchy. If the prefetch is issued too late, the requested data may not arrive before the actual memory reference, thereby introducing processor stall cycles. Adaptive prefetching, is incompatible to conventional prefetching, it prefetches pages by observing the previous paging activity. It is a system in which the subset of pages prefetched is not fixed. The pages are determined by running algorithms which decide the subset of pages eligible for prefetching by observing historical paging activity for a particular system and for a particular user [Alaa R Alameldeen, David A wood, 2004]. Obviously the pages prefetched in one system will be different from another system, though both run the same operating system. It is because the usage pattern of a user will be different from another and hence its paging activity also differs. Most of the operating systems make use of this adaptive prefetching concept and different vendors call it by different names. In Linux operating system it is known by the names

preload or prefetch or read ahead etc. The difference lies in the implementation details like the data structure used for holding prefetch data, number of pages to be prefetched, criteria for preloading random access memory etc.

Even multiprocessor systems make use of adaptive prefetching. Scheduling of prefetch instructions is the key for the success of any software prefetching algorithm. In multiprocessor system the scheduling is more critical task because the prefetched data is shared among the various cores present in the processor. If there is a negative interaction among different processor cores it will generate large problem inside the system. The latest versions of many architectures have chip multiprocessors (CMPs) with a shared L2/L3 cache [D Kongetira, K Aingraran, K olukotun Niagra, 2005], [Alaa R Alameldeen, David A wood, 2007]. Like any other shared resources the cores present in CMPs will compete for the cache also. In the context of CMPs with shared on chip caches, existing compiler algorithms for scheduling software prefetch instructions and static techniques to compute prefetch distances may not be effective. The L2 cache itself is the last line of defence in off-chip memory accesses, therefore achieving a high accuracy of prefetches are of critical importance. Apart from useless prefetches, the impact of harmful prefetches is also high in case of chip multiprocessors with shared L2 cache. A prefetch is harmful if the prefetched data replaces the data which are already in use by other cores. The harmful prefetches can occur among the accesses made by a single core, or by the access from multicores. The contribution of harmful prefetches also increases with the increased number of cores. Hence there is a direct correlation between the degradation in the effectiveness of prefetching and the increase in the fraction of harmful prefetches. The harmful prefetch patterns change dramatically across the different phases of program execution, and in the execution phases of a given application, the total number of processor cores that are involved in harmful prefetches is relatively small. Based on these observations, dynamic and adaptive complementary techniques to mitigate the impact of harmful prefetches are proposed. In these techniques, if the perfetch is found to be harmful it is suppressed and if it is found

to occur frequently the rharmful prefetch data is blocked in the L2 cache itself [Sparsh Mittal, 2016], [Mahmut Kandemir, ISBN-978-3-9810801]. These two techniques are very effective in mitigating the impact of harmful prefetches.

4.2.2 Hardware Prefetching

4.2.2.1 Next-Line Prefetching

Next-line prefetching is another instruction prefetching technique which comes under the hardware prefetching scheme. Next line prefetching tries to prefetch sequential cache lines before they are needed by the CPUs fetch unit [I.K. Chen, C.C. Lee, T.N. Mudge, 1997]. Current cache line is the one from which the processor is currently accessing the instruction or data. The cache line which is followed sequentially after t4.1he current cache line is refered as the next line. If the next line is not resident in the cache, it will be perfected from some distance into the current cache line which is accessed. This specified distance is measured from the end of the current cache line and is called the fetch ahead distance. Next-line prefetching predicts that execution will continue along the sequential path for conditional branches in the current line.

In order to calculate the address of the next cache line a little additional hardware is required. Unfortunately, next-line prefetching is unlikely to reduce misses when execution proceeds with non-sequential execution paths caused by conditional branches, jumps and subroutine calls. In these cases, the next line guess will be incorrect except in the case of short branches and the correct execution path will not be prefetched. Performance of the scheme dependents upon the choice of fetch ahead distance. If the fetch ahead distance is large, the prefetch has to be initiated early and the next line is likely to be received from memory in time for the CPU to access it. If the current instruction is branch instruction, it may increase the fetch ahead distance. If

the prefetch is useless, it will increase both memory traffic and cache pollution. For the calculation of the wastage of processor cycle, fetch ahead distance can be used. Run ahead Execution [O. Mutlu, J. Stark, C. Wilkerson, Y. N. Patt, 2003] prefetches by speculatively executing the application, but ignoring dependences on long latency misses. In spite of these shortcomings, next-line prefetching has been shown to be an effective strategy, sometimes reducing cache misses by 20-50% [J. Pierce, 1995].

4.2.2.2 Target-Line Prefetching

The main draw back of next-line prefetching is the inability to correctly prefetch non-sequential cache lines, target-line prefetching overcomes this problem. When instructions in the current line are being executed, the next cache line accessed might be the next sequential cache line or it might be a line containing the target of a control instruction found in the current line. Since unconditional jump and subroutine call instructions have a fixed target and conditional branch instructions are handled based on history of pervious execution for fetching the target page [V.Srinivasan, E.S.Davidson, G.S.Tyson, 2004]. For this a good heuristic function is used to prefetch on the previous behaviour of the current line. Heuristic function will analyse the previous history and prefetch the next line. Target-line prefetching uses a target prefetch table maintained in hardware to supply the address of the next line to prefetch when the current line is called for execution. The table contains current line and successor line pairs. When an instruction execution transfers from one cache line to another line, two things happen in the prefetch table. The successor entry of the previous line is updated to the address of the new current line. Also, a lookup is done in the table to find the successor line of the new line. If a successor line entry exists in the table and if that line does not currently reside in the cache, the line has to be prefetched from memory. By using this scheme, instruction cache misses will be avoided or at least their miss

penalty will be reduced if the execution flow follows the path of the previous execution.

4.2.2.3 Wrong-Path Instruction Prefetching

In this method, during the execution of instruction mispredicted paths are actually reduced along with the number of cache misses occurring during correct path execution [G. Reinman, T. Austin and B. Calde, 1999], [J. Pierce and T.N. Mudge, 1996]. This suggests that prefetching instruction cache lines down mispredicted paths might have a positive result. It is a hybrid scheme which combines both target and Next-line prefetching. The next line is prefetched whenever instructions are accessed inside the fetch ahead distance as described earlier. The major difference with reference to target prefetching is that no target line addresses are saved and no attempt is made to prefetch only the correct execution path. Instead, the line containing the target of a conditional branch is prefetched immediately after the branch instruction is recognized in the decode stage. Thus, both paths of conditional branches are always prefetched: the fall-through direction with next-line prefetching and the target path with target prefetching. Unfortunately, as the target is computed at a later stage, prefetching the target line when the branch is taken is unproductive. In this case, if the target address is not in the cache, a fetch miss and a prefetch request of the same line will be generated simultaneously. Similarly, prefetching unconditional jump or subroutine call targets is useless, since the target data has been always taken early and the prefetch address would be produced too late. To reiterate, the target prefetching part of the algorithm can only perform a potentially useful prefetch for a branch, which is not taken. This is why the algorithm is called wrong-path prefetching. However, if execution returns to the branch in the near future, and the branch is then taken, the target line will probably reside in the cache because of the prefetch. The hardware requirements for wrong-path prefetching are roughly equivalent to what is required for next-line prefetching since the target

prefetch addresses are generated by the existing decoder and no target addresses are saved. The obvious advantage of wrong-path prefetching over the hybrid algorithm is that there is a lower hardware cost. The performance of wrong path prefetching might also compare favourably with other schemes. Wrong-path prefetching can prefetch target paths, which are to be executed unlike the table-based schemes that require a first execution pass to create the cache line links. In addition, wrong-path prefetching performs better than correct-path only when there exist a large disparity between the CPU cycle time and the memory speed. Wrong-path prefetching, prefetches lines down a path which is not immediately taken thus it potentially has more time to prefetch the line from a slow memory before the path is executed. The potential advantages of wrong-path prefetching would not come without cost. Prefetching down will add some lines into the cache that are never accessed. This will increase both memory traffic and cache pollution. For the algorithm to be successful, the benefits of prefetching must overcome the added pollution misses. The extra traffic is not reduced, but memory bandwidth which is viewed as a hardware resource is utilized more effectively to reduce the performance degradation caused by instruction cache misses.

4.2.2.4 Fetch Directed Instruction Prefetching

Modern high-performance processor [T. Alexander, G Kedem, 1996] is composed of two processing engines: the front-end processor and the execution core. The front-end processor is responsible for fetching and preparing (e.g., de-coding, re-naming, etc.) instructions for execution. The execution core arranges the execution of instructions and the retirement of their register and memory results to non-speculative storage. Typically, these processing engines are connected by a buffering stage in some form, e.g., instruction fetch queues or reservation stations. The front-end acts as a producer, which fills the connecting buffers with instructions that are utilized by the execution core. This producer/consumer relationship between the front-end and execution

core creates a fundamental bottleneck in computing, i.e., execution performance is strictly limited by fetch performance. Efficient instruction cache performance is critical in order to keep the execution core satisfied. Instruction cache prefetching is an effective technique for improving instruction fetch performance [Mahmut Kandemir, Yuanrui zhang, Ozcan Ozturk, 2009].

A scalable front-end architecture is a mechanism to relieve the bottleneck of the fetch end caused due to earlier fetch cycle and later processing [Lawrence Spracklen, Yuan chou, Santhosh G Abraham, 2005]. One aspect of that architecture was to decouple the branch predictor from the instruction cache. The branch predictor produces fetch blocks into a Fetch Target Queue (FTQ), where they are eventually consumed by the instruction cache. This decoupling allows the branch predictor to run ahead of the instruction fetch. This can be beneficial when the instruction cache has a miss, or when the execution core backs up. This second case can occur because of data cache misses or long latency instructions in the pipeline. Fetch block addresses in the FTQ are used to provide Fetch Directed Prefetching (FDP) for the instruction cache, and to guide instruction prefetching. Future branch prediction architectures may be able to hold more state than the instruction cache, especially multi-level branch predictors and those that are able to predict large fetch blocks or traces. When a cache block is evicted from the instruction cache, its corresponding entry in the FTB is marked to make it invalid , which is similar to a branch target buffer, but can predict larger fetch blocks. If a branch predicted FTB entry is marked as being evicted, then prefetch the predicted fetch block using the fetch directed prefetching architecture. Prefetching blocks that are already contained in the instruction cache results in wasted bus bandwidth. When the instruction cache has an idle port, the port can be used to check whether or not a potential prefetch address is already present in the cache. This technique is called Cache Probe Filtering (CPF). If the address is found in the cache, the prefetch request can be cancelled, thereby saving band width. If the address is not found in the cache, then in the next cycle the block can be prefetched if the cache bus is free. Cache probe filtering is only needed to access the

instruction caches tag array. It is found to be beneficial to add an extra port for CPF, which would only affect the timing of the tag access. An instruction cache (port) may be idle and will be unused because of an instruction cache miss due to insufficient predicted fetch blocks. If an instruction cache miss occurs, then the fetch engine will stall until the miss is resolved. When the instruction window becomes full because of a slow instruction in the execution core, the instruction cache has to stall since the fetch buffer is full. To use the idle cache ports to perform cache probe filtering during a cache miss, the cache needs to be lock up free.

4.2.2.5 Branch Target Instruction Prefetching

Branch target instruction prefetching is a hardware prefetching technique. This method combines nextline prefetching along with the prefetching of all control instruction targets regardless of the predicted direction of conditional branches [Y. Zhang, S. Haga, R. Barua, 2002]. Unlike wrong-path instructions prefetching, the line containing the target of a conditional branch, jump and functional call are prefetched immediately after the branch instruction is recognized by the prefetch unit instead of decode stage, and stored in a separate buffer called branch target buffer (BTB). Branch target buffers (BTB) have been effectively used as a mechanism for branch and instruction fetch prediction to increase the instruction fetch rate [D Kongetira, K Aingraran, K olukotun Niagra, 2005]. Thus, both paths of conditional branches are always prefetched. Unlike target line prefetching, there is no need to maintain a target prefetch table. The main advantage of this method is the reduction in cache pollution when wrong prefetch occurs with a separate memory. Target instructions along with its address tag will be stored in BTB. Prefetch unit will fetch cache line size instructions into BTB. If the processor accesses these BTB instructions, next line prefetching unit prefetches next sequential instructions and stores them in instruction cache (I-cache) if required. Whenever processor executes branch instructions, for target addresses, first it will check

BTB, if it is not present then it will check in the I-cache. Either I-cache or BTB will contain branch target address instructions. The execution unit need not wait because of the unavailability of target address of control instructions and the main memory. Therefore, a better performance is expected. The prefetch unit indicates that when a line is to be prefetched, it computes the lines address, and performs a cache tag lookup to see if the line is resident in the cache. If the line is not resident, prefetch is initiated and a memory request is then sent to the non-blocking cache handler. The cache tag structure can be accessed simultaneously by both the fetch and prefetch units. For the next-line prefetching, the default fetch-ahead distance is 3/4 of the line size [D Kongetira, K Aingraran, K olukotun Niagra, 2005]. For wrong-path prefetching, a target prefetch is suggested during the cycle in which a conditional branch is decoded [D Kongetira, K Aingraran, K olukotun Niagra, 2005]. A single path execution stream can be speculatively exploited by branch prediction mechanism. Most efforts are focused on improving the prediction accuracy or reducing the branch execution penalty. The problem with branch prediction is the deeper the speculation the less likely speculated path will be executed. Since there is no execution at all on the wrong path, the machine may suffer a large penalty on misprediction even with a high accurate prediction.

4.3 Suggestion for Prefetching Techniques

The survey report here clearly defines the applicability of instruction cache prefetching schemes in the current and next generation microprocessor designs, where memory latencies are likely to be longer. However, a new prefetching algorithm is examined that was inspired by previous studies about the effect of speculation down mispredicted paths. Data prefetching has been used in the past as one of the mechanism to hide memory access latencies [Jeffery A Brown, Leo Porter, Dean M Tullsen, 2011], [A.Stoutchini M, et.al, 2001], [G.H.Loh and D S Henry, MIP-9702281]. But prefetching requires accurate timing, to

be effective in practice. Inaccurate timing may adversely affect processor performance in the case of multiple processors; hence it is handled with the help of adaptive prefetching method. The design of an effective prefetching algorithm will minimize the prefetching overhead. Since it is a big challenge, it needs more thought and effort on it. Even if a number of prefetching methods are available none of them is proved to be miss free. Even more, it is observed that in both hardware and software implementation of the prefetching techniques, branch prediction and handling is taking place during runtime. Instead of applying these at runtime, compilation procedure can be used for handling this. During the initial step of compilation that is in lexical analysis phase along with the symbol table creation, branching table can be generated by observing the tokens which can be used for handling the prefetching. The look ahead distance for branching is also considered for better utilisation of the available memory capacity as in the case of memory management section.

To classify memory access behaviours in hardware [J. D. Collins, et al., 2001] the branch access stream is matched against the behaviour specific table in parallel. Study shows that timing and scheduling of prefetch instructions [R. S. Chappell, et.al, SSMT] is a critical issue in software data prefetching and prefetch instructions must be issued in a timely manner for them to be useful. If a prefetch is issued too early, there is a chance that the prefetched data will be replaced from the cache before its use or it may also lead to replacement of other useful data from the higher levels of the memory hierarchy. If the prefetch is issued too late, the requested data may not arrive before the actual memory reference is made, thereby introducing processor stall cycles. Making use of this concept will pre-empt the current running sequence from the processor.

4.4 Need of Branch Handling

During the occurrence of a branch condition the system takes steps to prefetch the blocks corresponding to both true and false conditions to the respective memory area with out adding any extra hardware.

Today, all the processors come in to the market with the basic feature of pipelining [A.J. Smith, 1982]. The time required for moving an instruction one step down the pipeline is a processor cycle. The length of the processor cycle is determined by the time required for the slowest pipe stage. If the stages are perfectly balanced, then the time per instruction on the pipelined processor is calculated as given in equation 4.1.

$$\frac{Timeperinstructiononunpipelinedmachine}{Numberofpipelinestages} \qquad 4.1$$

Pipelining yields a reduction in the average execution time per instruction. It is an implementation technique that exploits parallelism among the instructions in the sequential instruction stream. Pipeline overhead arises from the combination of pipeline register delay and clock skew. The pipeline registers add setup time plus propagation delay to the clock cycle. The major hurdle of pipelining is structural, data and control hazards. Here more stress is given to control hazards, arising from the pipelining of branches and other instructions that changes the program counter (PC). Control hazards can cause a greater performance loss for pipeline than data hazards do. When branch (conditional/unconditional) is executed, it may or may not change the PC to some value other than its current value. The simplest scheme to handle branches is to freeze or flush the pipeline, holding or detecting any instructions until the branch destination is known.

Branching will destroy the entire parallelism achieved through pipe lining in the system. If branching can be predicted early to some extent it can overcome the delay. That is, the required instruction

can be brought to the cache region early. Prefetching can be done either with software or hardware. The most significant source of lost performance is when the process is waiting for the availability of the next instruction [Weifeng Zhang, Dean M Tullsen, Brad Calder, 2007], [J Lu, H Chen, et al., 2003]. In both the cases it has to flush some instruction out from the pipe. Whenever branching is encountered, the system will go for a context switching and the pipe will be filled with instruction for the next process. Here the problems which are discussed were considered and comes up with a solution.

4.5 What is CaMMEBH?

CaMMEBH is a proposed software prefetcing technique which makes use of the basic concept of both adaptive prefetching (software prefetching) and branch target prefetching (hardware prefetching). CaMMEBH implemented and tested on Linux environment.

Programs interact with the hardware through kernel and kernel maintains the operations of hardware. Linux is a type of monolithic kernel in which all functional components have access to all internal data structures and routines. The primary memory is partitioned into kernel space and user space in this context. Kernel space is loaded with code of kernel during boot time and the user space is loaded with user programs. These user programs cannot access the kernel space directly since it is purely dedicated to kernel. All operating system services are provided through predefined functions called system calls. Memory management is one of the main functionality of kernel. It allows processes to safely access the system memory as they require it. This memory includes cache, primary memory and secondary memory. Cache memory is used to hold frequently accessed data. Cache management is achieved through system calls that provide access to kernel functions. CaMMEBH will make necessary changes to the kernel source code to reflect the increased system performance and efficiency

by reducing the page fault rate. It includes identification and modification of existing kernel module to achieve the goal. CaMMEBH focuses on prefetching of branch handling instructions.

CaMMEBH mainly focused on the conditional statements and the looping constructs in which the execution flow changes from the sequential manner. When the execution flow changes from a sequential manner, chance of increasing page fault in cache memory is more which reduces the throughput of the system. CaMMEBH reduces page fault in memory through prefetching, because it make use of the concept of both adaptive and branch target prefetching. Its objective is, in addition to the targeted instruction it also loads the next instruction that follows a branch handling jump instruction onto memory while executing the jump.

4.6 Implementation of CaMMEBH.

Implementation of CaMMEBH is done on Linux operating system. Inside the system the branching and prefetching operations are handled by the file 'jump_label.h'. All the branch instructions present inside the program will be converted into CALL function. Kernel source code contains so many data structures and functions related to the execution of jump statements. A function call is also implemented through jump statement [http://www.spinics.net/list/linux_assembly/]. Assembly language specifies the invocation of a function or procedure through the statement CALL FUNCTION_OFFSET. But it is actually treated as JMP ADDR by the system. ADDR is the memory address where the function is stored in the memory [https://www.autoitscript.com], [https://github.com], [https: //en.wikipedia.org].

All the conditional and unconditional branch statements are specified as jmp instruction. There is a structure in the memory called 'jump_table' whose entries correspond to each jump statement in the program. It includes the fields like code, target and a key value. Code

represents the address of the jump instruction. Target specifies the target address of the jump, is the location to where the jump has to be taken in order to continue execution. Key is the index to jump_table. There are various data structures used to hold the branching information like source address, destination address, etc. These data structures are used in various kernel functions to manage branch handling. One of the major data structures related to branch handling is jump entry. It includes the fields for code, target and key as in jump_table. So this data structure is used to make an entry in jump_table as shown in Figure 4.1.

```
typedef u32 jump_label_t ;
structjump_entry
{
 jump_label_t code;
 jump_label_t target;
 jump_label_t key;
}
```

Figure 4.1: Data structure of jump_table

u32 is a data type used in C which is similar to the data type 'int' but u32 is used for explicitly sized data. CaMMEBH aims to load the next executable instruction into the memory along with the current instruction. The goal can be attained by modifying the structure jump_table and also the data structure jump_entry. The structure has to include an additional field to hold the address of the next instruction as target2. The modified data structure is shown in Figure 4.2

Idea of prefetching is based on the concept of locality of reference; the block which should be executed next is prefetched in to main memory from secondary memory. In Linux this is done with the help of a function 'mem_cpy()'. This function is invoked in a routine called

```
typedef u32 jump_label_t ;
structjump_entry
{
  jump_label_t code;
  jump_label_t target1;
  jump_label_t target2;
  jump_label_t key;
  }
```

Figure 4.2: Modified data structure of jump_table

'jump_label_transform ()' which resides in the file arch/x86/kernel/jump _label.c. In Linux 3.13 normal program statements as follows: memcpy(&code, ideal_nops [NOP_ATOMIC5], JUMP_LABEL_NOP _ SIZE)

Two new statements as given in Figure 4.3 are added to function is added to function 'prefetch_range()' to reduce the page fault by loading the next executable block also into the cache .

```
entry->target2=entry->code+JUMP_LABEL_NOP_SIZE;
prefetch_range(entry->target2,code.offset);
```

Figure 4.3: New statements

'Prefetch_range()' is the function which loads the instructions in to the cache memory. First line of code will point to the starting of the memory area which is to be loaded to the higher level memory. The structure element code.offset is set as the number of instructions to be moved to the cache. Table 4.1 will give specification of the testing environment.

Table 4.1: Specification details of testing environment

Specification
Processor:Intel i5 4 GB RAM 700 GB IBM-DTLA- 307030 SATA IDE hard drive. Linux kernel 3.14

4.7 Analysis of CaMMEBH with Current System.

To analyse the performance of CaMMEBH a new module is implemented for Linux kernel version 3.14 and checked for suitability and efficiency. The experiments were aimed at measuring the page fault occurrence in cache memory while running various programs with different sizes. The Linux version 3.14 is modified to reflect the changes like reduction in major fault, minor fault and various time parameters of each program. The results shows that there is a substantial decrease in the page fault rate after compiling the modified source code in accordance with the branch handling policy.

The results obtained from CaMMEBH experiments are analysed by making comparative study with the current system. For the analysis, programs with varying sizes from small to large are executed using both original source code and modified code and results were recorded. Table 4.2 will give the details about the number of major faults. Table 4.3 contains the details about minor fault. Each of the programs is executed minimum 15 times and the average is taken, the result is rounded and recorded in the Tables.

In Table 4.2 it can be found that there is a notable change in the number of major fault in system when modified version is implemented. This change in major fault shows an improvement of the system. As mentioned in chapter 3 whenever a major fault is occurring inside the

Table 4.2: Number of major faults

File Size	Major Fault (Numbers)	
	Exsisting Method	CaMMEBH
300KB	7	5
350KB	6	4
912KB	3	1
1.43MB	6	5
2MB	9	8
15MB	5	4
20MB	23	20
25MB	3	2
65MB	24	21
70MB	115	108
75MB	54	49
78MB	56	52

system, the processor have to spend 1000 of cycles in order to handle it [Computer systems a programmers perspective, Randal E Bryant, D OHallaron, Pearson]. By reducing the major fault the processor can be free from such operations and it can perform some useful work in a multiprocessing environment. Experimental results shown in Table 4.2 is diagramatically given in Figure 4.4.

Table 4.3 gives the details of minor fault occurring inside the system when the modified version of the kernel is implemented. Results show that CaMMEBH is able to reduce the number of minor fault occurring inside the system. As mentioned earlier even though major fault is more expensive than minor fault, minor fault will also play its on roll in the performance of the processor. Thus change in minor fault will also help to increase the processor performance. The diagramatic representation of Table 4.3 is given in Figure 4.5.

Table 4.4 and Table 4.5 will give the details about the user time, system time and elapsed time with respect to the various programs. The results show that there is no markable change noted in the case

Table 4.3: Number of minor faults

File Size	Minor Fault (Numbers)	
	Exsisting Method	CaMMEBH
300KB	6998	6960
350KB	2161	2042
912KB	5139	5116
1.43MB	5601	5510
2MB	240	238
15MB	240	238
20MB	239	238
25MB	239	237
65MB	27481	26457
70MB	36586	33724
75MB	35873	34456
78MB	36487	35343

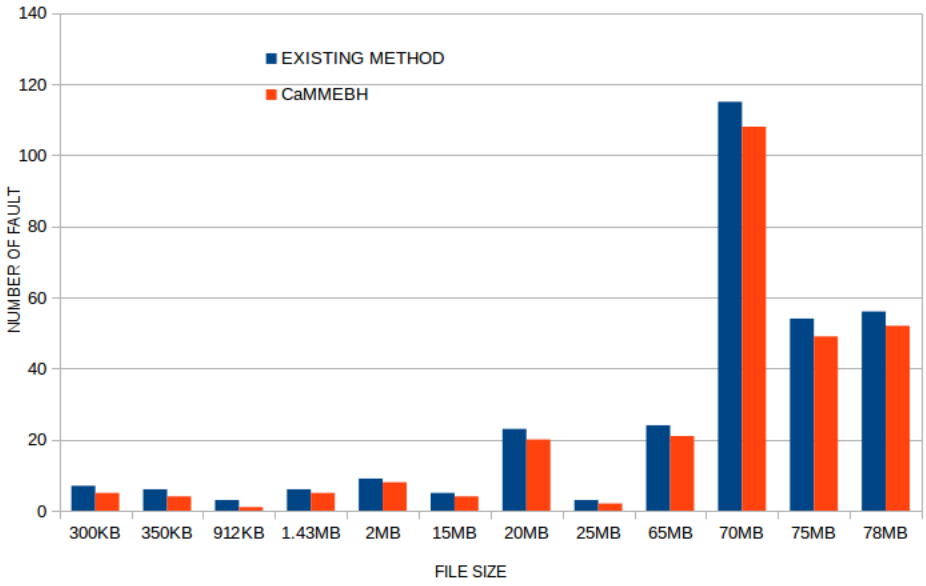

Figure 4.4: Comparison of major faults

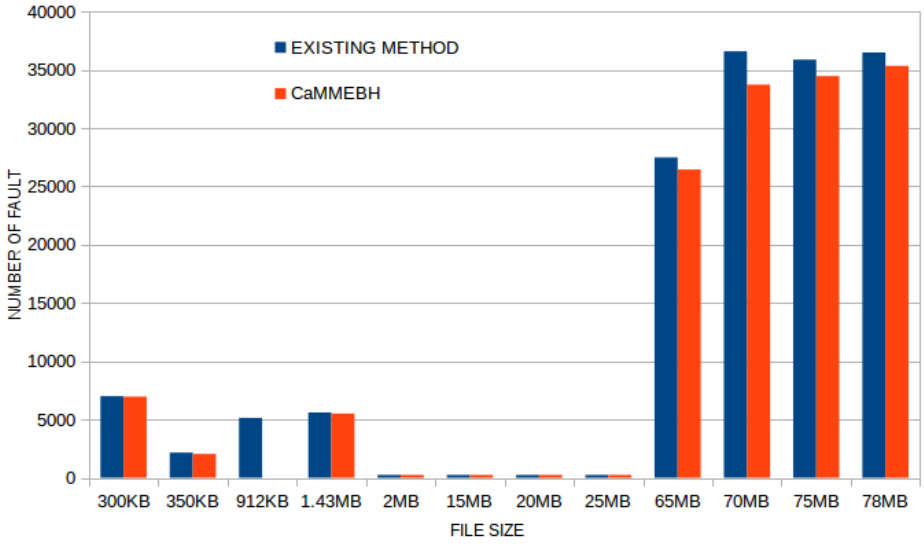

Figure 4.5: Comparison of minor fault

Table 4.4: Details about the various time parameters in CaMMEBH method

File Size	CaMMEBH		
	USER TIME(Sec.)	SYSTEM TIME(Sec.)	ELAPSED TIME(Sec.)
300KB	0.22	0.041	0.461
350KB	0.57	0.042	1.172
912KB	0.48	0.042	0.992
1.43MB	0.2	0.041	0.441
2MB	0.57	0.043	1.173
15MB	0.19	0.042	0.422
20MB	0.44	0.042	0.912
25MB	0.36	0.041	0.761
65MB	0.57	0.043	1.183
70MB	1.02	0.041	2.091
75MB	0.48	0.043	1.003
78MB	0.56	0.041	1.161

Table 4.5: Details about the various time parameters in existing method

File Size	Exsisting Method		
	USER TIME(Sec.)	SYSTEM TIME(Sec.)	ELAPSED TIME(Sec.)
300KB	0.22	0.041	0.461
350KB	0.57	0.042	1.172
912KB	0.48	0.042	0.992
1.43MB	0.2	0.041	0.441
2MB	0.57	0.043	1.173
15MB	0.19	0.042	0.422
20MB	0.44	0.042	0.912
25MB	0.36	0.041	0.761
65MB	0.57	0.043	1.183
70MB	1.02	0.041	2.091
75MB	0.48	0.043	1.003
78MB	0.56	0.041	1.161

of user time and a slight change is noted in the system time. Because here the work is based on the branch handling which is not able make any change in the user time. But it has some influence on the system time. Here elapsed time is mainly influenced by fault rate.

From the above Tables the time taken to execute a program can be calculated by adding the user time and system time. Elapsed time can be used to find how long the process is under the waiting condition. Here waiting can be calculated by subtracting the execution time from the elapsed time. Table 4.6 gives the details about the execution time and waiting time for the various processes. One of the reasons for increase in waiting time is due to the pre-emption condition of the process. The pre-emption condition can occur due to the presence of pagefault. Hence by reducing the page fault the waiting time can be reduced. By implementing CaMMEBH, pagefault rate is reduced significantly as show in Figure 4.4 and Figure 4.5.

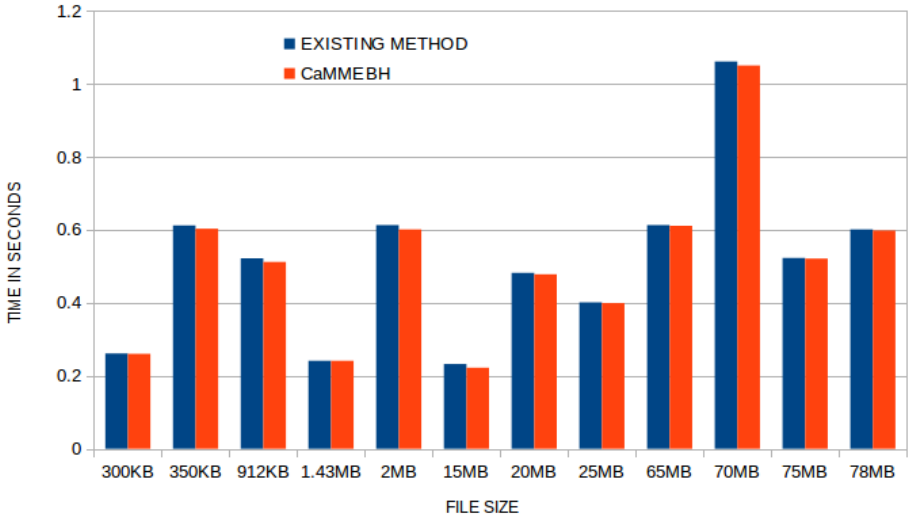

Figure 4.6: Comparison of execution time

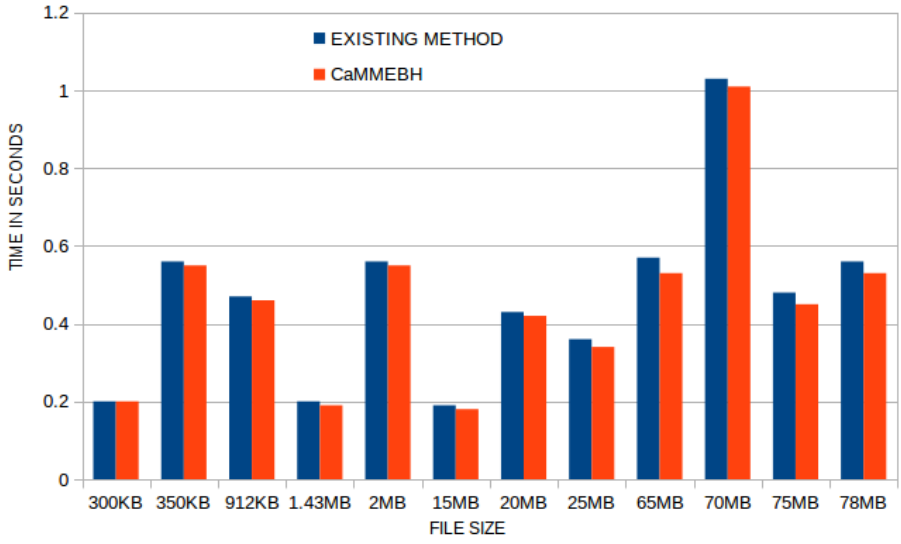

Figure 4.7: Comparison of waiting time

Table 4.6: Details about the execution time and waiting time for existing and CaMMEBH

File Size	Exsisting Method		CaMMEBH	
	EXECUTION TIME(Sec.)	WAITING TIME(Sec.)	EXECUTIONTIME(Sec.)	WAITING TIME(Sec.)
300KB	0.261	0.2	0.26	0.2
350KB	0.612	0.56	0.603	0.55
912KB	0.522	0.47	0.512	0.46
1.43MB	0.241	0.2	0.241	0.19
2MB	0.613	0.56	0.601	0.55
15MB	0.232	0.19	0.222	0.18
20MB	0.482	0.43	0.478	0.42
25MB	0.401	0.36	0.399	0.34
65MB	0.613	0.57	0.611	0.53
70MB	1.061	1.03	1.05	1.01
75MB	0.523	0.48	0.521	0.45
78MB	0.601	0.56	0.598	0.53

The diagramatic representation of Table 4.6 is given in Figure 4.6 and Figure 4.7. From the figures it is clear that by the introduction of the CaMMEBH the total waiting time gets reduced for the various processes.

4.8 Conclusion

By proper handling of the branches in the system the number of page fault can be reduced. CaMMEBH is concentrated only in the branching concept. The complete removal of the page fault is not possible with CaMMEBH because it deals only with the branching statements. There are a number of prefetching methods available and none of the methods are completely fault free. Main reason behind this is the size limit of the memory along with the demand paging concept. In CaMMEBH the data structures and the kernel functions that deal with the branch handling are edited to increase the hit ratio in the cache memory and thereby reduces the page fault. The CaMMEBH can be extended to prefetch the instructions other than the branch handling instructions to enhance the efficiency of the system to a large extend. Performance enhancement is done by relieving the processor from the fault handling operations.

5

Processor Performance Enhancement using MBFQV2, LRU-LFU and CaMMEBH

5.1 Abstract

Processor performance can be measured in terms of how much time the processor spends on performing useful work. This chapter gives a consolidated work of discussions in previous chapters. In chapter 2 it is discussed about how MBFQV2 improved the system performance compared to CFQ. The current system makes use of the LRU page replacement method which works based on the time stamp. Chapter 3 focuses about a new page replacement method LRU-LFU which makes use of time and frequency parameter for scheduling process. It also clearly shows how the system performance is improved using LRU-LFU. A new method CaMMEBH is introduced in chapter 4 for handling the

branch instructions. Whenever branch statements come across both true and false conditions, are handled effectively in CaMMEBH . Due to this the page fault generated by branch statements are avoided. The increase in processor performance is obtained by reducing the page fault and there by reduces the elapsed time

5.2 Introduction

Disk scheduler is the one responsible for transfer of data from secondary to primary memory. It works based on the principle of round robin scheduling where a time slice is given to each of the requests which come to the scheduler list. The accessing is done based on the queue data structure. Pre-emption process is activated inside the system in order to maintain the CFQ concept so that each request gets a fair allotment. The problem present in CFQ is that the arrival of a synchronous request may be arbitrarily delayed by a scheduler by just delaying the dispatching of the preceding request [Paolo Valente, Fabio Checconi, 2010]. Only because of the delayed invocation the request may get higher timestamp. Only because of higher time stamp the request has to wait in the request queue for a longer time period. Instead of setting the time slot a new concept of bandwidth is introduced in the BFQ. The size of the file is also considered by the scheduler. But here also the allotment of slot is kept static and once the budget is fixed it remains the same for the entire system operation. The B_max is also fixed as the maximum budget value. Due to this the request with less budget value has to wait for a longer time period. These two things are taken in consideration and modification is done on the BFQ method and formed MBFQV2. The result in chapter 2 shows that MBFQV2 gives a better result compared to BFQ. Hence in the kernel the disk scheduler is replaced with new MBFQV2 scheduler.

Once the data is made available in the primary memory, the next level of data movement is from primary to cache memory. Whenever the data has to make a move to the next level before moving the data

the availability of the space or room should be checked. Once the space is available, only then the movement takes place. If there is no space available then space should be created by replacing one of the existing pages called victim page. This is done by the page replacement scheduler. The working of the scheduler can be done based on various algorithms. LRU is the most commonly used replacement algorithm in almost all the operating systems. All the page replacement algorithms either take the arrival time of the page to the memory or how many number of times the page is referred by the system inside the memory. New replacement algorithm is suggested by modifying current LRU algorithm where the time stamp is mixed with the concept of frequency parameter. By making this modification, both major and minor page-fault will get reduced and the performance of the system will increase. In chapter 3 the results show how LRU_LFU gives a better performance than the LRU. Hence in the kernel the page replacement scheduler is replaced with LRU_LFU.

Another reason behind the page fault is the problem caused by the branch handler. In the existing system the branch handler will give preference to the true condition only. If the current method is modified to a new concept CaMMEBH, then equal weightage is given to both true and false branch conditions. It means an attempt is made to combine the concept of adaptive prefetching and branch target buffer prefetching. For this the existing data structure used by the handler is modified by adding a new field to it and which handles the false condition also. By this the handler will ensure that the pages which are required by both true and false conditions should be present in the cache memory.

5.3 Processor Performance Enhancement with Reduced Page Fault

This chapter discusses about the improvement in overall system performance by reducing the pagefault at different levels. The technique used for transferring data discussed under Chapter 2 is not only applicable for secondary memory to primary memory, but it can be used whenever the data has to be moved from one memory level to another memory level. Similarly, the page replaement method discussed under Chapter 3 is applicable for replacement of pages in primary memory and the replacement of cachelines in order to give space for the incoming new pages or cache blocks respectively. Once the modifiction is done on any of these methods, it not only reflects in one memory level but also affects the other levels of memory management module of the operating system. Hence an enhancement is obtained not only in one memory level but also in other memory levels also.

As per Amdahl's Law [Computer Architecture A Quantitative Approch, 4th edition, John L Hennessy, David A Patterson, Elsevier] the speedup can be gained by using equation 5.1.

$$Speedup = \frac{performancleade\ for\ \ entire\ task\ using\ the\ enhancement\ when\ possible}{performance\ for\ entire\ task\ without\ using\ the\ enhancement} \quad 5.1$$

-

Alternatively as in equation 5.2

$$Speedup = \frac{Execution\ time\ of\ entire\ taskwithout\ using\ the\ enhancement}{Execution\ time\ of\ entire\ task\ using\ the\ enhancement\ when\ possible} \quad 5.2$$

Speedup tells how fast a task will run using the computer with the enhancement as opposed to the original computer. In this work the enhanement is done on various memory levels to reduce the pagefault rate. The performance of the processor is measured in terms of CPU time as shown in equation 5.3 and equation 5.4.

$$CPUtime = CPUclockcyclesforaprogramXclockcycletime \quad 5.3$$

or

$$CPUtime = \frac{CPU\ clock\ cycles\ for\ a\ program}{Clock\ rate} \quad 5.4$$

From this it is clear that the processr performance can be increased

by reducing the CPU clock cycles for a program. While handling the page fault condition the processor has to spend a lot of clock cycles. In this work a reduction is made in the pagefault which will in turn reduce the clock cycle used by the processor. As per the equation 5.3, when the number of clock cycles gets reduced the CPU time also gets reduced.

5.4 Implementation of Various Phases

Implementation of MBFQV2, LRU-LFU and CaMMEBH are already discussed in Chapter 2, Chapter 3 and Chapter 4 in detail. The proposed system can be implemented through step by step procedures of the above three methods. It includes the installation and compilation of Linux 3.13 with modified program code.

5.4.1 LINUX Kernal

Linux kernel is implemented as a collection of modules. Each module is dedicated to perform some specific function. The entire kernel source code is partitioned into a number of subsections which incorporates millions of code lines [http://www.linux.org], [http://www.linux howtos.org/].

Linux kernel supports a number of architectures and the architecture specific code of kernel is included in the arch module of source code. Some of the architectures incorporated are alpha, ia64 etc. Minimum 32 bit processor with or without MMU and gcc should be there for implementation of these architectures [http://www.linuxjournal.com /1052], [http://www. phoronix.com], [http://www.linuxjournal.com /7105] . There are architectures which supports both 32 and 64 bit processors. Various file systems are handled here by dividing the entire 'fs' module into subdirectories, one for each file system. The developer

code includes some files in common which controls the whole file system. The file system module deals with the storage and management of files in storage devices. Ext4 is one among the file systems handled by Linux operating systems [UNIX concept and applications, 4th edition, S.Das, TMH], [Understanding Linux kernel source code deeply, C.Lijun, Posts and Telecom Press, 2002]. It has scalability and reliability supporting large 64bit file systems. Kernel code area is where all memory management operations are performed in 'mm' module. Both kernel space and user space are taken into considerations while managing memory. The page replacement policies, allocation and deallocation of cache memory regions etc are handled in 'mm' module [The complete reference Linux, 2nd edition, Peterson, TMH], [Advanced Linux Programming, Y Zong-de, et.al. Post and telecommunications press, 2008].

5.4.2 Module Creation

Development of Linux kernel is achieved through creation of kernel modules and loading it on demand. If any module functionality is not applicable kernel will unload that module. Major advantage of kernel capability improvement is by inserting new modules. Hence rebuilding and recompiling of entire kernel is not needed, the newly included module is compiled independently and is made as a part of existing kernel [http://www.tldp.org], [www.kroah.com], [https://www.inso.tuwienac .at/upload/], [www.makelinux .net/books/lkd2].

Care should be taken while writing code since whatever is included in the kernel module will be executed only in available kernel space. Compilation of the new module is done using a special file called 'makefile'. Once the creation of makefile is completed use the make command to compile the new module [https://www.kernel.org], [https:/en.wikipedia.org], [www. make linux.net].

5.4.3 Kernal Compilation

Kernel building involves creation, loading and installing of modules and these steps are explained with the use of commands make, make modules and make modules install respectively. Make command is the one which does building or compiling of program. Loading of module is achieved by the next command make modules. The installation is performed by the command make modules install.Table 5.1 shows the specification of the testing environment.

Table 5.1: Specification of the testing environment

Specification
Processor:Intel i5 4 GB RAM 700 GB IBM-DTLA- 307030 SATA IDE hard drive. Linux kernel 3.14

5.5 Performance Analysis

Table 5.2and Table 5.3 show the details of change in major and minor fault after appling the various methods. The work is mainly concentrated in the improvement of system performance. For that keep the processor free from fault handling by reducing the major and minor fault rate.

Table 5.2 shows the details about the major fault in various methods. Here the various scenarios are taken for observation. In the first case only the CaMMEBH is considered. Results show that the number of major faults get decreased compared to existing system. The reason behind this is that there is no major fault generated inside the program due to the branching statements. CaMMEBH is not able to

Table 5.2: Number of major faults occurred in various methods

File Size	MAJOR FAULT(Number)				
	EXISTING	CaMMEBH	CaMMEBH+LRU-LFU	CaMMEBH +MBFQV2	CaMMEBH+LRU-LFU+MBFQV2P
300KB	7	5	5	4	3
350KB	6	4	3	3	2
912KB	3	1	1	1	1
1.43MB	6	5	4	4	4
2MB	9	8	6	5	3
15MB	5	4	4	4	3
20MB	23	20	20	19	16
25MB	3	2	2	2	1
65MB	24	21	21	20	18
70MB	115	108	100	98	95
75MB	54	49	50	50	47
78MB	56	52	51	50	48

reduce any major fault generation other than branching. To handle the fault generation due to other condition, other methods should be included along with CaMMEBH. Hence CaMMEBH is combined with LRU_LFU and the results are shown in Table 5.2. Number of major fault is reduced in all the cases compared to existing system.

CaMMEBH is merged with MBFQV2 and the results are shown in Table 5.2. in this case also the number of major fault is reducing in all the cases. The reason behind this may be how fast the pages are made available in the respective memory levels. If the corresponding page is not available in the memory it should be taken from the lower memory level. This depends on how fast the data can be moved from one level to the other level. Even if the chance of occurrence of the fault is detected and the page can't be brought to the main memory then it leads to the major fault. This condition is handled in this scenario.

Finally CaMMEBH is merged with both LRU_LFU and MBFQV2, the results are shown in Table 5.2. Here also in all the cases the major fault gets reduced. Fault due to branching, improper page replacement method and delay due to the waiting in the request queue for transferring the page from secondary to primary are considered. Results show that there is a drastic change in the major fault number in certain cases. On an average around 40% of reduction in major fault is observed when compared with the existing system. But the major fault can't be completely removed from the system. The diagramatic representation of Table 5.2 is given in Figure 5.1

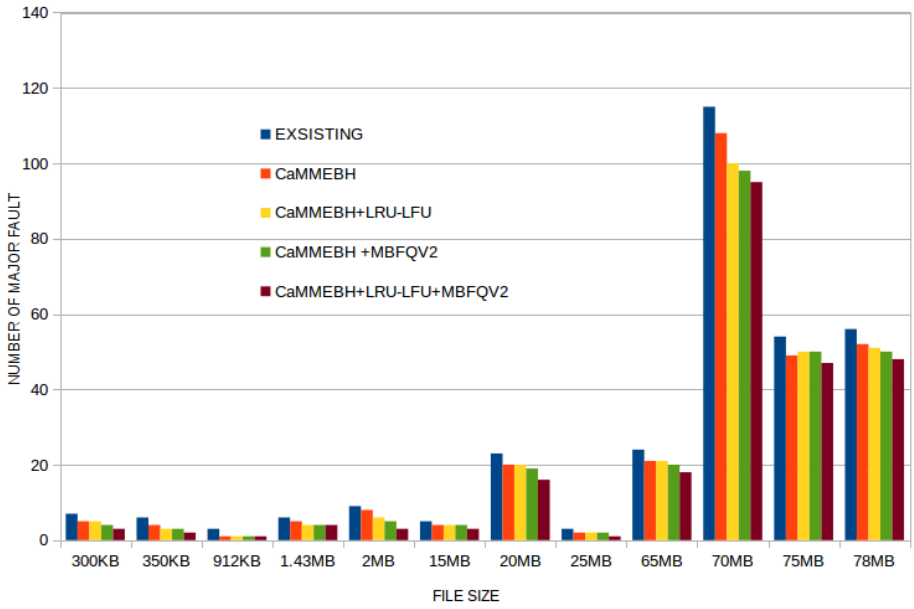

Figure 5.1: Comparison of major fault occured in various methods

Table 5.3: Number of minor faults occurred in various methods

File Size	MINOR FAULT(number)				
	EXISTING	CaMMEBH	CaMMEBH+LRU-LFU	CaMMEBH +MBFQV2	CaMMEBH+LRU-LFU+MBFQV2P
300KB	6998	6960	6583	6815	6557
350KB	2161	2042	1624	1635	1478
912KB	5139	5116	5110	5205	5104
1.43MB	5601	5510	5837	5680	5456
2MB	240	238	239	237	237
15MB	240	238	239	239	237
20MB	239	238	238	238	236
25MB	239	237	237	237	238
65MB	27481	26457	25783	25233	24346
70MB	36586	33724	33653	33862	33242
75MB	35873	34456	33686	34327	33456
78MB	36487	35343	34586	33726	34543

Table 5.3 gives the details about the minor fault. In this case also the various methods are considered as in the case of major fault.

Here initially CaMMEBH alone considered. Results show that the number of minor fault get decreased compared to existing system. CaMMEBH is not able to reduce any minor fault generation other than branching. To handle the page fault generation due to other conditions, other methods should be included along with CaMMEBH. CaMMEBH is combined with LRU-LFU and the results are shown in the Table 5.3. Number of minor fault is reduced in all the cases compared to existing system.

CaMMEBH is merged with MBFQV2, the results are shown in Table 5.3 compared to the existing system. In this case also the number of minor fault gets reduced in all the cases. The reason behind this may be how fast the pages are made available in the respective memory levels. If the corresponding page is not available in the memory, it should be taken from the lower memory level. This depends on how fast the data can be moved from one memory level to the other memory level. Even if the chance of a fault occurrence is detected and the page can not be brought to the cache memory it leads to the minor fault. Such condition is handled in this scenario.

Finally CaMMEBH is merged with both LRU_LFU and MBFQV2, the results are shown in Table 5.3. Here also in all the cases the minor fault is reduced. Faults due to branching, improper page replacement

method and delay due to the waiting in the request queue for transferring the page from primary to cache are considered. Results show that there is a large variation in the minor fault number in certain cases. On an average around 6% of reduction in minor fault is observed when compared with the existing system. But the minor fault can not be completely removed from the system. The diagramatic representation of Table 5.3 is given in Figure 5.2.

In Table 5.4, Table 5.5, Table 5.6, Table 5.7 and Table 5.8 the details about various time parameters with different file sizes are considered. The results show that even if the changes is made on the data transfer method, page replacement method or the branch handler it does not affect the user time or system time parameters significantly. By the above methods the fault rate can be reduced along with a reduction in wasting time.

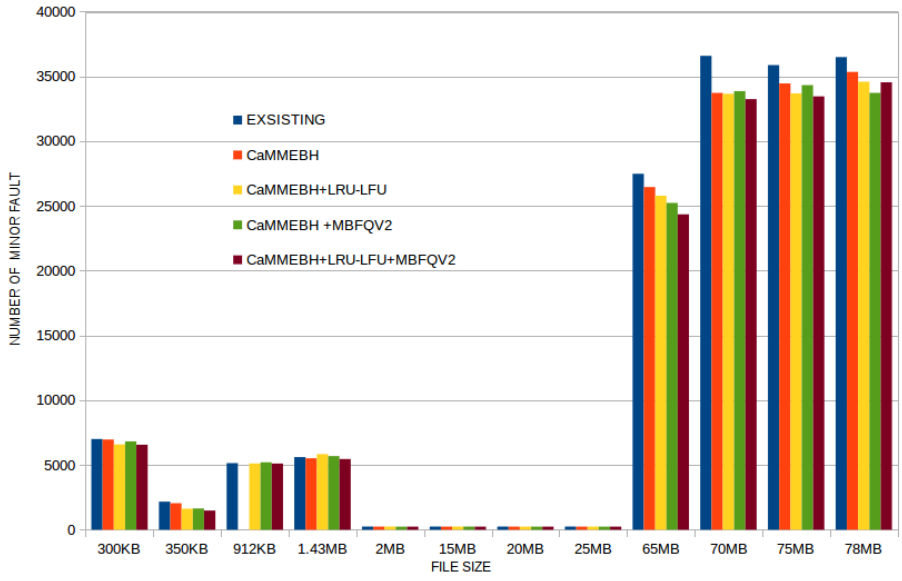

Figure 5.2: Comparison of minor fault occured in various methods with existing one

In Table 5.4 a slight reduction is observed while combining all

Table 5.4: Comparison of user time in various methods with existing method

File Size	USER TIME(Sec)				
	EXISTING	CaMMEBH	CaMMEBH+LRU-LFU	CaMMEBH +MBFQV2	CaMMEBH+LRU-LFU+MBFQV2P
300KB	60.22	0.22	0.22	0.22	0.21
350KB	0.57	0.56	0.56	0.55	0.55
912KB	0.48	0.47	0.47	0.47	0.46
1.43MB	0.2	0.2	0.21	0.21	0.2
2MB	0.57	0.56	0.56	0.55	0.55
15MB	E0.19	0.18	0.17	0.17	0.17
20MB	0.44	0.44	0.43	0.43	0.43
25MB	0.36	0.36	0.37	0.37	0.35
65MB	0.57	0.57	0.56	0.55	0.55
70MB	1.02	1.02	1.15	1.02	1.02
75MB	0.48	0.48	0.48	0.48	0.46
78MB	0.56	0.56	0.55	0.54	0.54

the methods. The diagramatic representation of Table 5.4 is given in Figure 5.3.

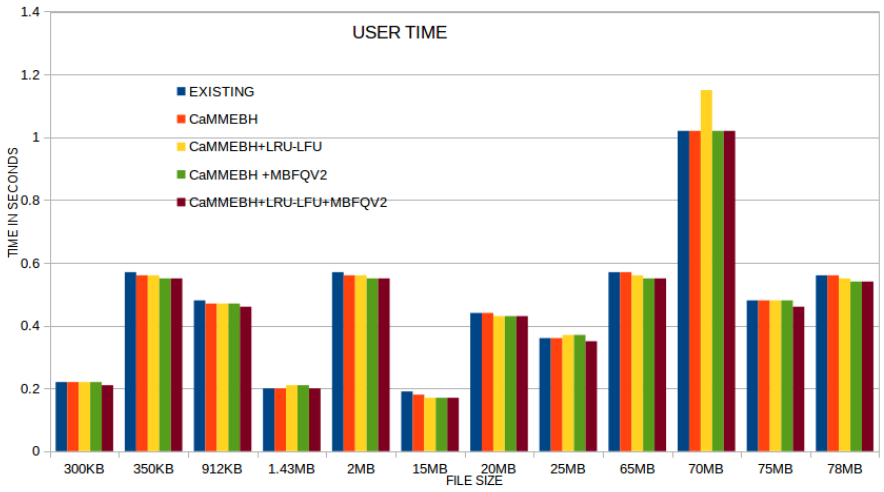

Figure 5.3: Comparison of user time in various methods with existing one

Table 5.5 gives the details about the system time. A slight reduction is visible in the case of combining all the methods. The diagramatic representation of the Table 5.5 is given in Figure 5.4.

Table 5.6 gives the details about the elapsed time. The table shows that elapsed time is reduced in all the cases. From the above Tables

Table 5.5: Comparison of system time in various methods with existing method

File Size	SYSTEM TIME(Sec)				
	EXISTING	CaMMEBH	CaMMEBH+LRU-LFU	CaMMEBH +MBFQV2	CaMMEBH+LRU-LFU+MBFQV2P
300KB	0.041	0.04	0.042	0.041	0.04
350KB	0.042	0.043	0.042	0.04	0.038
912KB	0.042	0.042	0.041	0.042	0.04
1.43MB	0.041	0.041	0.043	0.041	0.037
2MB	0.043	0.041	0.043	0.041	0.04
15MB	0.042	0.042	0.041	0.042	0.041
20MB	0.042	0.038	0.042	0.04	0.039
25MB	0.041	0.039	0.041	0.043	0.038
65MB	0.043	0.041	0.042	0.043	0.041
70MB	0.041	0.03	0.041	0.04	0.036
75MB	0.043	0.041	0.042	0.041	0.04
78MB	0.041	0.038	0.042	0.039	0.039

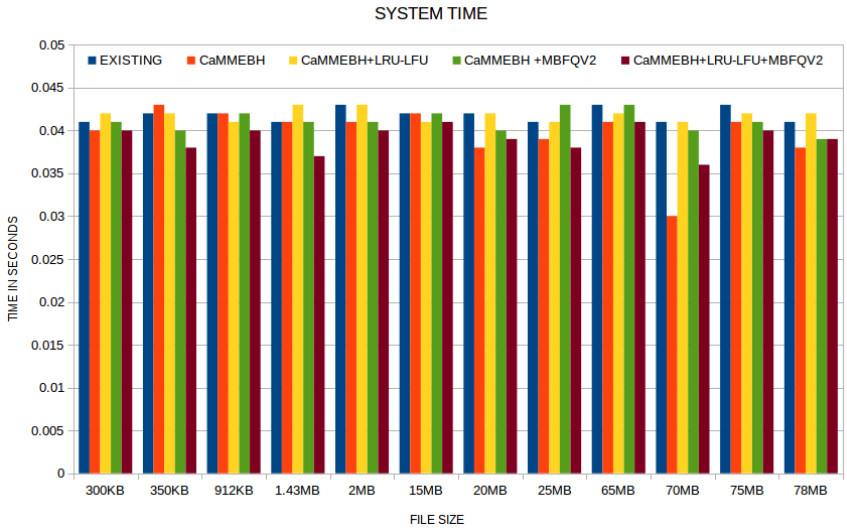

Figure 5.4: Comparison of system time in various methods with existing one

Table 5.6: Comparison of elapsed time in various methods with existing method

File Size	ELAPSED TIME(Sec)				
	EXISTING	CaMMEBH	CaMMEBH+LRU-LFU	CaMMEBH +MBFQV2	CaMMEBH+LRU-LFU+MBFQV2P
300KB	0.461	0.462	0.482	0.45	0.43
350KB	1.172	1.153	1.192	1.141	0.92
912KB	0.992	0.972	0.991	0.972	0.942
1.43MB	0.441	0.431	0.453	0.423	0.41
2MB	1.173	1.151	1.173	1.121	0.966
15MB	0.422	0.402	0.411	0.401	0.401
20MB	0.912	0.898	0.902	0.898	0.891
25MB	0.761	0.739	0.771	0.737	0.722
65MB	1.183	1.141	1.182	1.139	1.028
70MB	2.091	2.06	2.221	2.06	1.93
75MB	1.003	0.971	1.002	0.969	0.903
78MB	1.161	1.128	1.152	1.126	0.992

Table 5.7: Comparison of execution time in various methods with existing method

File Size	EXECUTION TIME(Sec)				
	EXISTING	CaMMEBH	CaMMEBH+LRU-LFU	CaMMEBH +MBFQV2	CaMMEBH+LRU-LFU+MBFQV2P
300KB	0.261	0.26	0.262	0.261	0.25
350KB	0.612	0.603	0.602	0.59	0.588
912KB	0.522	0.512	0.511	0.512	0.5
1.43MB	0.241	0.241	0.253	0.251	0.237
2MB	0.613	0.601	0.603	0.591	0.59
15MB	0.232	0.222	0.211	0.212	0.211
20MB	0.482	0.478	0.472	0.47	0.469
25MB	0.401	0.399	0.411	0.413	0.388
65MB	0.613	0.611	0.602	0.593	0.591
70MB	1.061	1.05	1.191	1.06	1.056
75MB	0.523	0.521	0.522	0.521	0.5
78MB	0.601	0.598	0.592	0.579	0.579

it is already seen that user time and system time are not affected significantly by these methods. But the change in elapsed time shows that the waiting time is decreased inside the system for the various processes.

From Table 5.4 and Table 5.5 the execution time of a program can be obtained by adding the user time and system time. The details about execution time are given in Table 5.7. It shows a slight reduction in execution time using various methods.

Table 5.8 is obtained from the Table 5.6 and Table 5.7. Here waiting time is calculated by subtracting the execution time from the elapsed time.

As shown in Table 5.8, in all the cases waiting time is reduced and

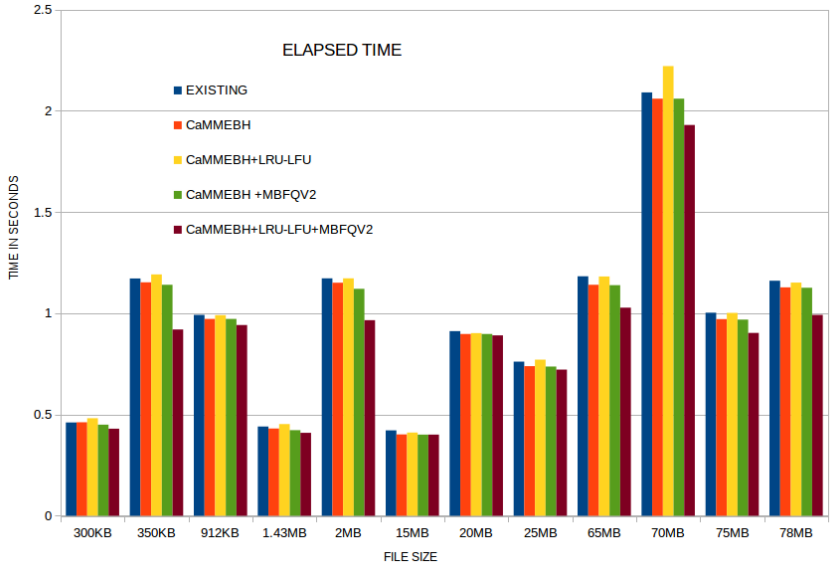

Figure 5.5: Comparison of elapsed time in various methods with existing method

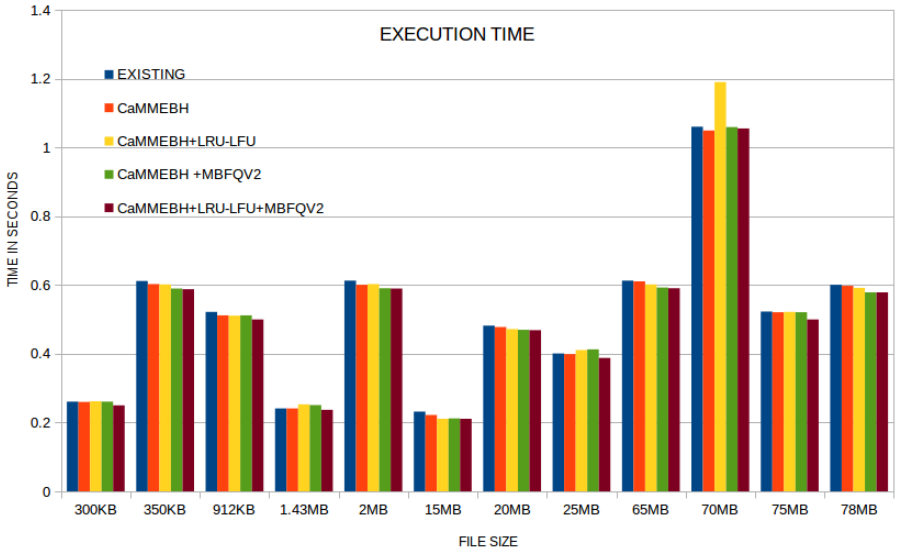

Figure 5.6: Comparison of excecution time in various methods with existing method

Table 5.8: Comparison of waiting time in various methods with existing method

File Size	WAITING TIME (Sec)				
	EXISTING	CaMMEBH	CaMMEBH+LRU-LFU	CaMMEBH +MBFQV2	CaMMEBH+LRU-LFU+MBFQV2P
300KB	0.2	0.2	0.22	0.189	0.18
350KB	0.56	0.55	0.59	0.551	0.332
912KB	0.47	0.46	0.48	0.46	0.442
1.43MB	0.2	0.19	0.2	0.172	0.173
2MB	0.56	0.55	0.57	0.53	0.376
15MB	0.19	0.18	0.2	0.189	0.19
20MB	0.43	0.42	0.43	0.428	0.422
25MB	0.36	0.34	0.36	0.324	0.334
65MB	0.57	0.53	0.58	0.546	0.437
70MB	1.03	1.01	1.03	1	0.874
75MB	0.48	0.45	0.48	0.448	0.403
78MB	0.56	0.53	0.56	0.547	0.413

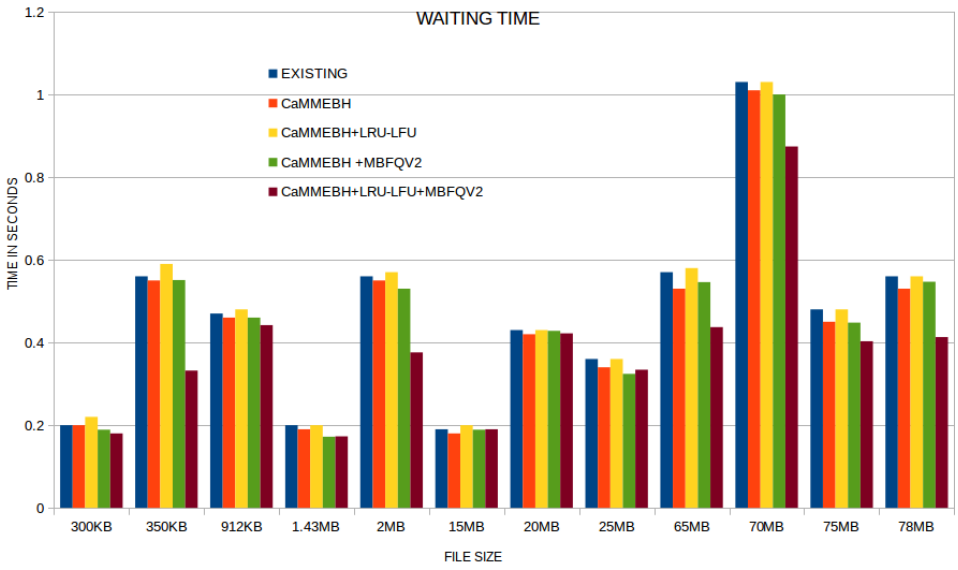

Figure 5.7: Comparison of waiting time in various methods with existing method

there by the system performance is increased. The reduction in waiting time is on an average around 16% of the existing system. Diagramatic representation of Table 5.8 is shown in Figure 5.7.

Even if all the operating systems are designed with multi-threading capability the compilers are not able to utilise this feature approappritaly. The execution will be done in sequential order only. This is the reason why there is no change in the user time or in the system time parameters. To increase the execution speed the facility of multithread should be utilized in a proper way. Chapter 6 deals with this feature.

5.6 Conclusion

This chapter actually deals with the merging of the various methods discussed in Chapter 2, Chapter 3 and Chapter 4. The results show that by incorporating these various methods in the different Linux versions, number of major and minor faults can be reduced. If the fault rate is reduced the performance of the system gets increased by freeing the processor for fault handing conditions. It also shows that CaM-MEBH along with LRU_LFU and MBFQV2 methods a reduction in page fault rate and waiting time is noticed. Hence an improvement in over all system performance is obtained.

5. PROCESSOR PERFORMANCE ENHANCEMENT USING MBFQV2, LRU-LFU AND CAMMEBH

6

Parallel Execution of Multiple Threads

6.1 Abstract

Nowaday the processors are multicores such as dual-core, quad-core and octa-core processors. To get maximum utility of these cores, parallel programs should be used. Currently existing applications are sequential in nature and when run on multiple cores, utilizes only one core. To increase the performance of application program, parallelization is an important technique to make efficient use of all the cores. Manual parallelization requires huge effort, in terms of time and money and hence there is a need for automatic code parallelization. This chapter introduces Automated Code Parallelizer using Open Multi-Processing(OpenMP), which automates the insertion of compiler directives to facilitate parallel processing on shared memory machines with multiple cores. It converts an input sequential program into a multi-threaded program for multi-core shared memory architectures. This work focuses on loops and speculatively parallelizes the different iterations of a loop while taking care of data dependency between the

different iterations. While executing the various iterations the window size is varied and found the optimal window size.

6.2 Introduction

The availability of multi-core architectures [V.Vidya, Priti Ranadive, SudhakarSah, 2010] allow users not only to run several applications at the same time, but also to run parallel code. However, the manual development of parallel versions of existent sequential applications is an extremely difficult task because it needs (a) an in-depth knowledge of the problem to be solved (b) understanding of the underlying architecture and (c) knowledge of the parallel programming model to be used [Gao L, et al., 2013]. Many parallel languages and parallel extensions to sequential languages have been proposed to exploit the capabilities of modern multi-core systems. The most successful proposal in the domain of shared memory system is OpenMP, a directive-based parallel extension to sequential languages such as FORTRAN, C or C++ that allows the parallelization of user-defined code regions. OpenMP does not ensure correct execution of the code according to sequential semantics, making the programmer responsible for such tasks. Possible dependence violations that may occur between iterations during execution need to be addressed by the programmer [M.Gonzalez, J.Oliver, et al., 2001].

On the other hand, automatic parallelization offered by compilers only extract parallelism from loops when the compiler can assure that there is no risk of a dependence violation at runtime [Jose Rodr'igez-Rosa, Jose Oliver, et al., 2001], [P.P.Athavale, et.al. 2012]. Only a small fraction of loops falls into this category, leaving many potentially parallel loops unexploited [V.G.Vaidya and S Sah, 2012], [V.G.Vaidya and S Sah, 2014],[M.Chandi, I Foster and K.Kenney, 1994].

Thread-Level Speculation (TLS) [Shengyue Wang, Xiaoru Dai, et al. 2005] technique allow the extraction of parallelism from fragments

of code that cannot be analyzed at compile time, namely, the compiler cannot ensure that the loop can be safely run in parallel. To ensure that the code can be run in parallel, the programmer should be able to classify all variables present in the code into two groups as private variables and read-only shared variables. Private variables are always written outside the loop and read-only shared variables are only readable and can not be written anything in to the variable during the iteration. Hence if all variables in a loop are either private or read-only shared, then the loop can be safely parallelized. If a single variable is found that does not fit in these two categories, then it is identified that the loop is not parallelizable at compile time itself. Under these situations, the loop can be executed in parallel by software-based speculative parallelization. In speculative execution, the loops are executed as if the iterations are independent even if there are dependencies. TLS can deal with these situations in which dependence violations may occur, leading the parallel loop to correctly analyse its execution [http://software.Intel.com/en-us/articles/automatic-parallelization-with-intel-compilers/]. The main problem with this technique is that the code needs to be manually augmented in order to handle speculative execution and monitor the possible dependences. Programmers have to specify the variables that may lead to a dependence violation, as speculative variables.

OpenMP allows the user to mark variables as speculative, that enables the automatic transformation of the code to support its execution by TLS runtime library during compile time. The transformations proposed in OpenMP are transparent to programmers, who do not need to know anything about the TLS parallel model. Programmers only have to classify variables depending on their accesses, letting the system perform all the changes needed in the source code. To do so, a new OpenMP clause 'speculative' is used to handle dependency violation.

A new clause for speculative is added in to OpenMP framework implementation. This pass transforms the loop with the corresponding OpenMP parallel directive, inserting the runtime TLS calls needed to

(a) distribute blocks of iterations among processors (b) perform speculative loads and stores of speculative variables and (c) perform partial commits of the correct results generated so far. The TLS runtime library used is based on the same design principles as the speculative parallelization library [Cintra M, Llanos D.R, 2003].

6.3 Current Speculative Technique

Multi-core technologies have increased the performance of computing systems during the last decade [V.Vadiya, S. Sah, P.Ranadive, 2010]. However, unlike previous advances in computer architecture, existent code cannot immediately take advantage of these architectural improvements. To fully exploit multi-core capabilities programmers should have an in-depth knowledge of both the application and the underlying computer architecture [P.Marcuello, 2003], [C.Zilles and G Sohi, 2001].

Due to the huge number of sequential programs already written for many decades a new technique that automatically parallelizes them is quite desirable. However, automatic parallelization techniques currently implemented in many commercial compilers are not able to parallelize most of the loops because of data dependencies.

OpenMP is a shared-memory Application Programming Interface (API) whose features are based on prior efforts to facilitate shared-memory parallel programming [www.openmp.org]. The appropriate insertion of OpenMP features into a sequential program will allow different applications to benefit from shared-memory parallel architecture. In practice, many applications have considerable parallelism that can be exploited. However, OpenMP fails to parallelize execution when there are data dependencies within the loops. Such loops require speculative execution.

Existing speculative techniques require manual intervention of expert programmers. These programmers firstly need to extract certain information about the source code that they want to parallelize. Without automatic tools programmers have to manually extract the information, such as variable usages within each loop or I/O functions that complicate or even preclude the parallelization. They should also determine whether it is worth parallelizing a loop or if the thread-management overheads would be larger than the benefit of parallelizing. This information extraction is the first step to speculatively parallelize a source code. The second step is to add all the functions and structures needed to handle the speculative execution. However, manual parallelization task is usually tedious and mistakes are easily committed [J Subholle, et al., PPOPP'93].

6.4 Thread-Level Speculation

Thread-Level Speculation aims to automatically extract loop and task-level parallelism when a compile-time dependency analysis cannot guarantee that a given sequential code is safely parallelizable. TLS optimistically assumes that the code can be executed in parallel, relying on a runtime monitor to ensure that no dependence violations are produced. A dependency violation appears when a given thread generates a data that has already been consumed by a successor in the original sequential order. In this case, the results generated so far by the successor (called the offending thread) are not valid and should be discarded. The original code is augmented with function calls that distribute iterations among processors and monitor the use of all the variables that may lead to a dependency violation. The commits to store the results obtained by successful iterations should be in order. If a dependency violation appears at runtime, the library functions stop the offending threads and restart them in order to use the updated values and thus preserve the sequential semantics.

Unfortunately, most loops have variables whose values might be written in a particular iteration and later be read in a subsequent iteration. Sequential semantics impose a total order for both operations, and if these two operations are done out-of-order by different threads a dependency violation occurs. The results generated by such threads that consume the outdated value of such speculative variables should be discarded with all the results generated by its successors [Aldea S, Llanos D.R, Gonzlez-Escribano, 2012]. This is called a squash operation.

At compile time TLS requires that the original code be augmented to perform speculative loads, speculative stores and in-order commits. In addition, it also requires that the loop structure be rearranged in order to follow the re-execution of squashed operations. The plug in automatically performs all these changes required by the TLS runtime library. The speculative clause triggers significant changes into the code. Read and write operations to speculative variables are replaced at compile time with function calls of 'loading' and 'storing' that handle these operations. Read operations are changed for function calls that obtain the most up-to-date value of the element being accessed. Write operations are changed for function calls that write the data in the version copy of the current processor. It also ensures that no thread executing a subsequent iteration has already consumed an outdated value for this structure element called dependency violation. If such a violation is detected, the offending thread and its successors are stopped and restarted. The loop annotated with the speculative clause is transformed into a loop with as many iterations as available threads. At the beginning of the loop body, a scheduling method assigns the block of iterations to be executed to the current thread. Once a thread has finished the execution of the assigned chunk of iterations, a function is called in order to verify the correct execution of iterations carried out. If the execution was successful the version copy of the data is committed to the main copy otherwise version data is discarded. Programmers just need to use the proposed OpenMP speculative clause to point out which variables may lead to a dependency violation.

6.5 OpenMP Speculative Clause

There are two different ways to handle the OpenMP for speculative parallelization. First way is by the addition of a new directive, for example pragma omp speculative. However, this option is more difficult to implement because there are many OpenMP related components that should be modified. It is preferable to use the second approach, which is a proposal of a new clause for the OpenMP, pragma omp speculative parallel loop, construct. This new clause would enable the programmer to enumerate which variables should be updated speculatively. The definition of 'pragma omp speculative' clause point out conflictive variables that may lead to a dependency violation is rather useful. Because this clause is the first step to transform the code automatically and handle the dependencies easily.

Taking this into consideration, a new clause is proposed for Open MP to provide support for speculative parallelization of 'for' loops. The target of TLS systems considered are 'for' loops. Other loops can be considered as well, but as long as their number of iterations cannot be so easily predicted, the applicability of TLS solutions is limited in these cases.

The new OpenMP clause is called speculative, and it needs to be used as the clause of a parallel directive. This new clause is shown in Figure 6.1, where 'list' contains variables that may lead to any dependency violation. The parser identifies OpenMP directives and

```
#pragma omp parallel for speculative(list)
  for-loop
```

Figure 6.1: OpenMP speculative clause for 'for-loop'

clauses, and emits the corresponding unified tree form, called generic representation. Initially, the generic representation of the new clause is created like other standard clauses. Then, the compiler is prepared to recognize and parse the clause as part of the parallel loop construct.

When the new clause has been parsed the plug_in detects the clause and starts all the transformations needed on the code.

Programmers write OpenMP [Milovanovi M, Ferrer R, Unsal, et al., 2008] programs as usual, but now with the capability of annotating as speculative those variables that could lead to a dependency violation. With this method, programmers need not bother about handling these violations. Now the speculative runtime system is responsible for such task [Dang F.H, Yu H., Rauchwerger, 2002], [Xekalakis P, Ioannou N, Cintra, 2009]. Once a programmer annotates each variable to its type, a compiler plugs in augments the code to integrate the TLS runtime library.

From the point of view of a programmer, the structure of a loop being speculatively parallelized due to the proposed clause is not so different from a loop parallelized with regular OpenMP directives. Current OpenMP parallel constructs force the programmer to explicitly declare the variables used in the parallel region according to their role. This can be an extremely hard and error-prone task if the loop has more than a few dozen lines. It is possible that, if the programmer is unsure about the use of a certain variable or structure, then simply label them as speculative.

```
# pragma omp parallel for default(none)
        private (i,x, temp)
          speculative (a)
        for(i=0;i<max;i++)
  {
           x=i%(max+1);
           temp=a[x-1];
            x=(2*temp)%(max+1);
           a[x-1]=temp;
  }
```

Figure 6.2: Example of for loop with speculative clause

Figure 6.2 shows an example of the use of the proposed clause. Variable 'i' is private, since it is the variable that controls the iterations of the 'for' loop. Variables 'x' and 'temp' are private because they are always written before being read in the context of iteration. And finally, variable 'a' is speculative because accesses to this variable can lead to dependency violations. During parallel execution a particular iteration may read from a non-updated value leading to incorrect execution. A speculative management of 'a' allows the parallel execution of this loop properly.

However, the use of the new clause forces the compiler plug_in to perform several changes into the source code. The clause points out those variables which may lead to a dependency violation [Gao L, Li, 2013]. The compiler has to rewrite part of the loop in order to handle possible violations by ensuring the correct parallelization of the loop. Under speculative execution each thread maintains a version copy of the data structure that is accessed speculatively. At compile time, the original code is augmented to perform speculative stores, speculative loads and in-order commits. In addition, the loop structure is rearranged in order to allow the re-execution of squashed iterations.

6.5.1 Speculative Stores

At compile time, all write operations to the data structure being speculatively accessed should be replaced with calls to a speculative store function. This function writes the datum in the version copy of the current thread, and ensures that no thread executing a subsequent iteration has already consumed an outdated value for this structure element. If such a violation is detected, the offending thread and its successors are stopped and restarted.

6.5.2 Speculative Loads

At compile time, all reads to the speculative data structure should be replaced with calls to a function that performs a speculative load. This function obtains the most up-to-date value of the element being accessed. This operation is called forwarding. If a predecessor has already defined or used that element then that value is forwarded. If not, the function obtains the value from the reference copy of the data structure.

6.5.3 Commit Operation

If no dependency violation arises during the execution of a given thread, its changes to the speculative data structure should be committed to the reference copy of the data structure. Note that commits should be done in order to ensure that the most up-to-date values are stored. After performing the commit operation, a thread can receive a new iteration or block of iterations to continue the parallel work.

6.5.4 Scheduling Iterations under TLS

Finally, the original loop to be speculatively parallelized should be augmented with a scheduling method that assigns to each free thread the following chunk of iterations to be executed. If a thread has successfully finished a chunk then it will receive a brand new chunk not yet executed. Otherwise, the scheduling method may assign the same chunk whose execution had failed to the thread to improve locality and cache reutilization.

6.6 Speculative Engine

Software speculative schemes should allocate some additional memory in order to hold the information related to speculative executions. The use of this data is mandatory to enable recovery operations that could arise in an optimistic execution. In this context, memory needed could be allocated dynamically or statically and the use of an approach instead of the other is a critical decision that directly influences the overall memory used in a program.

6.6.1 Data Structures

The data structures needed by the speculative library are shown in Figure 6.3 [Aldea, Sergio, Alvaro Estebanez, Diego R. Llanos, Arturo Gonzalez-Escribano, 2014]. A matrix W having four window slots as shown in Figure 6.3 implements a sliding window that manages the runtime of the library. Each slot has the responsibility of management of speculative execution of a particular set of iterations. The slots assigned to the non-speculative and the most-speculative threads are indicated by two variables, non-spec and most-spec. Each slot is composed of two fields, STATE with the state of the execution being carried out and a pointer to maintain the position of the speculative variables used during the execution.

An example of the execution of a loop is also shown in Figure 6.3. The loop has been divided into three chunks of iterations, and it will be executed in parallel using three threads. It is very important to understand that there is no fixed association between threads and slots. A thread is assigned with a new chunk of iterations and a slot performs the operation [Estebanez A, Llanos D.R, Gonzalez-Escribano, 2014]. An order should be maintained among the threads and chunks being executed.

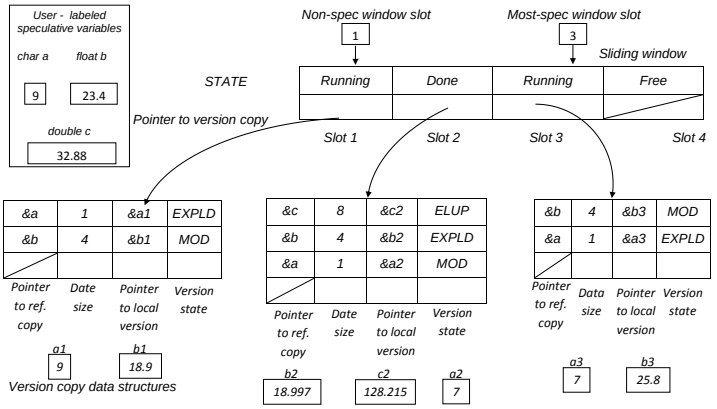

Figure 6.3: Data structures of speculative library

Thread working under slot 1 is executing the non-speculative chunk of iterations as indicated by its RUNNING state in Figure 6.3. The next chunk has been already executed and its data has been left these to commit after the non-spec chunk finishes as indicated by DONE state. While the last one, the most-speculative chunk launched is also in RUNNING state. In other words, the thread in charge of the second chunk has already finished, while the non-spec and most-spec threads are working. If more chunks were pending then the free thread would be assigned with the following chunk by starting its execution in slot 4. Slot 2 cannot be re-used yet, because commit operation of speculative variables after the execution of chunk 2 are yet to be completed. When the non-speculative thread working in slot1 finishes, it will commit its results and then the results stored in all subsequent DONE slots, can also be committed in order.

In addition to its STATE, each slot points to a data structure that holds the version copies of the data being speculatively accessed. Figure 6.3 represents a loop with three speculative variables. At a

given moment, the thread executing the non-speculative chunk has speculatively accessed variables ‘a’ and ‘b’. Each row of the version copy data structure keeps the information needed to manage access to each speculative variable. The first column indicates the address of the original variable, known as the reference copy. The second one indicates the data size. The third one indicates the address of the local copy of this variable associated to the window slot. Finally, the fourth column indicates the state associated to this local copy. Once accessed by a thread, the version copies of the speculative data can be in three different states: Exposed Loaded, indicated as ’EXPLD’ shows that the thread has forwarded its value from a predecessor or from the main copy. The state Modified indicated as ’MOD’ shows that the thread has written new values in to the variable. The state Exposed Loaded and Updated indicated as ’ELUP’ states that the thread has first forwarded the value to the variable and has later modified it.

Figure 6.3 represents a situation where the thread working in slot1 has performed a speculative load from variable ‘a’ and a speculative store to variable ‘b’. Regarding ‘a’, the figure shows that the thread working in slot 2 modified and forwarded its value to slot 3. With respect to variable ‘b’, the information in the Figure 6.3 shows that ‘b’ has been overwritten by threads working in both slots 1 and slot 3.

6.7 Partial Commit Operation

The partial commit operation is exclusively carried out by the non-speculative thread. Every time a thread should check if its data have to be committed or discarded. It first checks whether the data has to be squashed and if it is a non-speculative thread [Aldea, Sergio, Alvaro Estebanez, et al. 2014]. If the thread is speculative, the state of corresponding slot is left committed by the non-spec window.

In Figure 6.3 if the non-spec thread working in slot 1 is completed then the state of the variable in the data structure should be scanned to

verify whether it is ELUP or MOD state. In our example, 'b' has been modified in slot 1 version copy as 'b1' and hence 'b' in shared memory should be updated with 'b1'. After committing the version copy data structure associated with slot 1, change the slot 1 state to 'Free' and advances the non-spec window to slot 2. As long as slot 2 is marked as 'Done', its data should be committed as well. In our example, data stored in 'c2' and 'a2' should be committed to the user-defined variables 'a' and 'c'. After this, the state of slot 2 is also changed to 'Free' and the non-spec pointer is advanced. Thread working in slot 3 is still running and when it finishes data 'i', 'b3' is copied to user defined variable 'b'. All these commit operations are carried out with the help of auxiliary data structures that store list of elements in 'ELUP' or 'MOD' states only. Because of this unnecessarily traversing through local copy can be avoided.

It is interesting to note that each thread writes data in its local data structure version copy only and hence no critical sections are required to protect them. It is important to see that thread in the preceding slot should be completed before advancing to the next slot.

6.8 Loop Transformation for Speculative Execution

Figure 6.4 briefly shows the transformation of a parallel for-loop using speculative engine. This transformation is automatically carried out by the compiler plug-in. The changes are briefly described below:

Lines 2-3: Define the additional internal variables

Line 4: omp_get_num_threads() function is called to obtain the number of available threads.

Line 5: A specbegin() function is called to initialize the execution of the following parallel loop. If it is the first loop being parallelized,

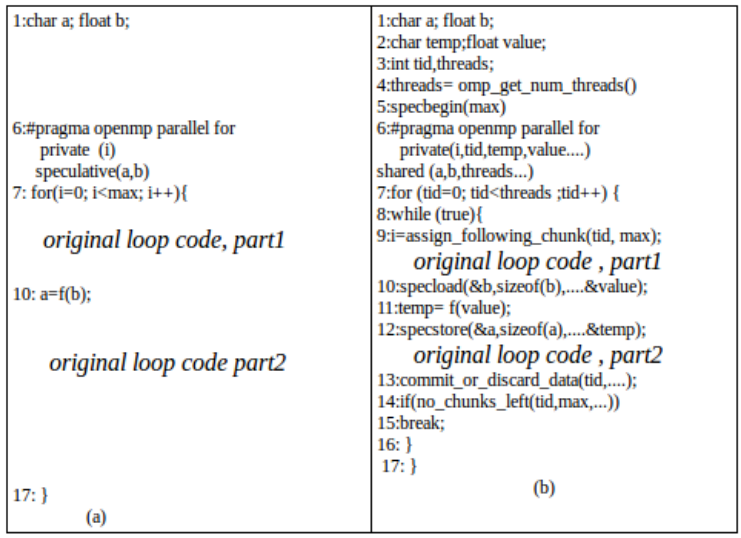

Figure 6.4: Loop Transformation (a) Original code (b) Transformed code using speculative engine

this function also initializes the runtime speculative library.

Line 6: All variables labelled as speculative are automatically reclassified as shared. Besides this change Partial Commit Operation, all reads and stores inside the loop body on that speculative variables are replaced with calls to specload() and specstore() functions, in order to keep sequential consistency as described.

Line 7: The original loop structure is replaced with a parallel 'for' loop with just 'threads' iterations. This launches the number of desired threads.

Line 8: A while (true) loop ensures that each thread repeatedly requires a chunk of iterations from the original loop to be processed. If no chunks are left, a break statement is used to exit from then loop and end of thread is reached (line 14).

Line 9: Inside the loop, each thread receives the index of the first iteration of its assigned chunk and proceeds with the original loop

body.

Lines 10-12: The read of 'b' variable is replaced with a call to the specload() function, that recovers the mo3st up-to-date value for this variable. The value is stored in a private 'temp' location. Line 10 of Figure 6.4(a) performs a write on 'a'. This write is replaced with a call to specstore() , that first stores the value in a local version copy and then checks whether a successor has already consumed an outdated value of 'a'. If so, the offending thread and all of its successors are squashed.

It is important to highlight that only the lines of the original loop body that involve speculative variables are changed in this way and the remaining code is left unchanged.

Line 13: Once the original loop body is completed, a call to commit_or _discard_data() checks whether the thread has been squashed or not. If a squash operation is issued by a predecessor, local copies of speculative data will be discarded. If the thread has not been squashed and if it is a non-spec one then a partial commit will occur.

Line 14: After completing their tasks related to the current chunk, all the threads check whether there are any pending chunks to be executed. If there is no pending chunk then threads leave the while loop.

When all threads have exited the while (true) loop, the end of the parallel section has been reached. Hence all chunks of iterations are successfully executed and their results have been committed to the speculative variables.

6.9 Implementation and Analysis of Result

The experiments were done to study the performance improvement gained by the system on using speculative parallelization. Test programs were executed on various environments and the execution times were analyzed.Table 6.1 shows the specification of the testing environment.

Table 6.1: Specification of the testing environment

Specification
Processor:4xIntel(R) core (TM)i5-2450M CPU@2.50GHz Memory: 4GB of DDR3 700 GB IBM-DTLA- 307030 SATA IDE hard drive. Operating system: OS Luna GCC Version:4.6.2

All the flags are completely removed during compile time to get error free code optimization so that the correct analysis of performance can be obtained [GNU Project: GCC internals (2013)], [http://gcc.gnu.org/ onlinedocs/ gccint/], [IBM: Thread-level speculative execution for C/ C++. IBM XL C/C++ for Blue Gene, 2012]. For that use the following commands.
$COMPILER -fplugin=$PLUGIN $PARAM $FLAGS -o $EXEC $ SOURCES $PLUGINDIR/speccode.o $PLUGINDIR/auxiliar _ functions.o
In addition to this make some changes in the system specific flags as follows:
COMPILERDIR=/usr/gcc-4.6.2/bin
COMPILER=$COMPILERDIR/gcc-4.6.2
specprag/Makefile
CFLAGS2 = -fopenmp $(THREADS_FLAG) $(ITER_FLAG) $(POINTER _FLAG) $(BLOCK_FLAG)

Table 6.2: Execution time of program 1 with two different window size, different number of cores and threads

Number of Cores	1		2		4	
Threads	window size n+1	window size 2n	window size n +1	window size 2n	window size n+1	window size 2n
1	5.22	5.21	5.22	5.22	5.24	5.35
2	5.79	8.56	5.76	5.86	5.71	5.84
4	10.54	9.31	10.23	7.44	3.11	3.06
8	13.82	12	13.6	7.1	5.97	4.95

Table 6.3: Execution time of program 2 with two different window size , different number of cores and threads

Number of Cores	1		2		4	
Threads	window size n+1	window size 2n	window size n +1	window size 2n	window size n+1	window size 2n
1	6.5	6.51	6.5	6.52	6.5	6.56
2	5.95	9.77	6	6.4	5.86	5.89
4	10.87	10.69	10.47	7.8	3.18	3.14
8	13.81	12.51	13.76	7.29	5.94	5.19

Modified GCC variable to a more generic value to facilitate execution: GCC=/usr/gcc-4.6.2/bin/gcc-4.6.2 1specprag/user_parameters.h

As specified earlier the window size will determine the number of slots available for the threads. When the window size is changed, it is observed that the optimal result is achieved while doubling the size of threads. The test results are shown with two conditions. In first condition the window is equal to thread plus1 [Aldea, Sergio, et al., 2014] and in second condition the window size is double the number of threads.

The experiments were conducted on single core, dual core and quad core environments with one, two, four and eight threads per execution. Table 6.2 to Table 6.6 show the execution time for different programs under various environments.

Table 6.4: Execution time of program 3 with two different window size, different number of cores and threads

Number of Cores	1		2		4	
Threads	window size n+1	window size 2n	window size n +1	window size 2n	window size n+1	window size 2n
1	5.87	5.24	5.07	5.1	5.08	5.13
2	6	8.17	5.94	6	6	5.95
4	10.66	9.28	10.74	7.59	3.62	3.14
8	13.78	11.62	13.65	7.26	6.12	5.16

Table 6.5: Execution time of program 4 with two different window size, different number of cores and threads

Number of Cores	1		2		4	
Threads	window size n+1	window size 2n	window size n +1	window size 2n	window size n+1	window size 2n
1	10.54	10.85	10.54	10.59	10.53	10.63
2	6.6	13.71	6.39	6.26	6.3	6.12
4	10.93	13	11.04	7.87	4.16	3.88
8	14.56	14.63	14.3	7.32	7.26	5.45

Table 6.6: Execution time of program 5 with two different window size, different number of cores and threads

Number of Cores	1		2		4	
Threads	window size n+1	window size 2n	window size n +1	window size 2n	window size n+1	window size 2n
1	15.18	15.2	15.17	15.25	15.14	15.31
2	14.11	19.6	14.03	13.96	13.83	13.78
4	22.25	18.75	22.11	13.96	7.69	7.67
8	23	18.78	22.91	13.79	12.95	8.05

Analysis of the Tables show that if the number of threads is equal to one and even if the execution is done on a single core, dual core or quad core processor the variation in execution time is negligible. Hence if a single thread is executing it means that the execution is done in sequential order only and hence no significance for multiple cores. All the programs get minimum execution time when the number of cores and number of threads are increased. It is also observed that the maximum performance is obtained when the number of cores is equal to the number of threads. From the above Tables it is seen that maximum performance were obtained when the number of threads is equal to 4 and the number of cores is equal to 4 which gives the maximum performance. Even if the number of threads is changed to 8 the execution time is not decreased because the maximum number of cores is 4 only. It shows that increasing the number of cores alone does not improve the system performance. A better utilization of the cores is required, that is parallel usage of the cores.

In the first condition the window size is limited to one more than the number of threads. It was found that a higher performance is achieved if the window size is doubled as shown in Figure 6.5, Figure 6.6, Figure 6.7, Figure 6.8 and Figure 6.9. Modern multiprocessing systems have the mechanism to handle higher levels of speculation due to the availability of multiple cores. The number of window size

was fixed as double the threads through trial and error and extensive experimentation.

It is clear from the figures that speculative multi-threaded execution significantly improves the performance of the system. It can also be seen that the performance of multi-core systems are at their best when the number of threads is equal to the number of processing cores. In dual core and quad core systems, two and four threaded executions respectively take the least time for completion.

The improvement is highly evident for four threads. There is always a speculative overhead associated with parallel speculative execution. At times, the overhead overrules the improvement gained by parallelization. This is the reason why two-threaded executions are sometimes slower than single threaded execution in dual core systems, as seen in Figure 6.5, Figure 6.6, Figure 6.7, Figure 6.8 and Figure 6.9. It can also be seen from the same diagram that as the number of threads increases, the performance improvement due to speculative parallelization dominates over the delays caused by speculative overhead and thus completes the execution faster.

The performance improvement with the increase in the number of threads is not linear. It can be seen from Table 6.2, Table 6.3, Table 6.4, Table 6.5 and Table 6.6 that increase in the number of threads doesnt reflect too much on the performance. Performance improvement is higher when the tasks are CPU bounded. For other tasks like I/O bounded the performance improvement will be minimal.

It is clear from Figure 6.5, Figure 6.6, Figure 6.7, Figure 6.8 and Figure 6.9 that the execution speed increases in a non linear fashion as the program size increases. This is due to the increase in the number of loops with the increase in the file size. It is also observed that the execution speed is the best when the number of cores is equal to the number of threads.

The performance improves as the number of threads increases, until the number of threads equals the number of cores. If the number

of threads is further increased, the performance gradually decreases. This is because when the number of threads is made higher than the number of processing cores, then multiple threads are sharing the same core. As a result, there occurs some context switching between threads which, cost time.

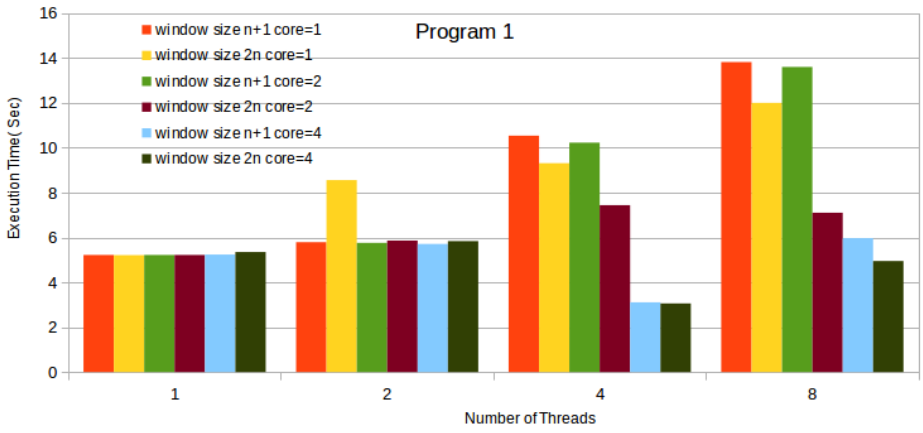

Figure 6.5: Execution time of program 1 having window size n+1 and 2n with different number of threads and cores.

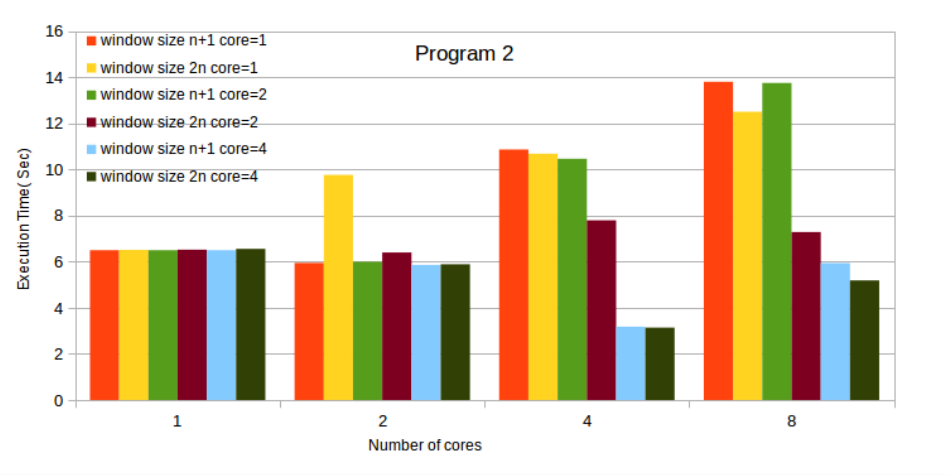

Figure 6.6: Execution time of program 2 having window size n+1 and 2n with different number of threads and cores.

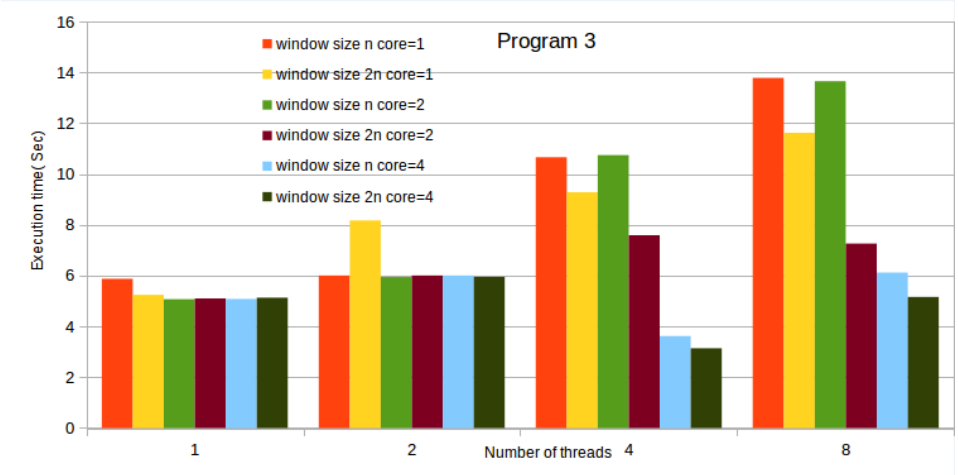

Figure 6.7: Execution time of program 3 having window size n+1 and 2n with different number of threads and cores.

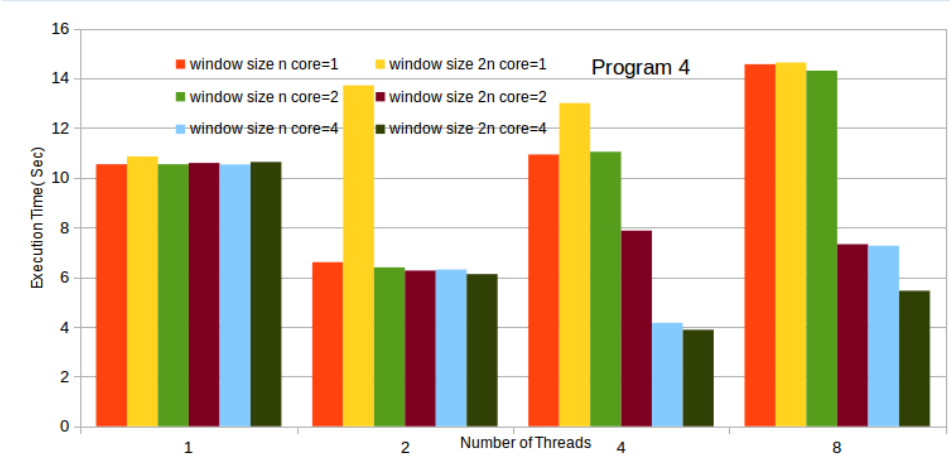

Figure 6.8: Execution time of program 4 having window size n+1 and 2n with different number of threads and cores.

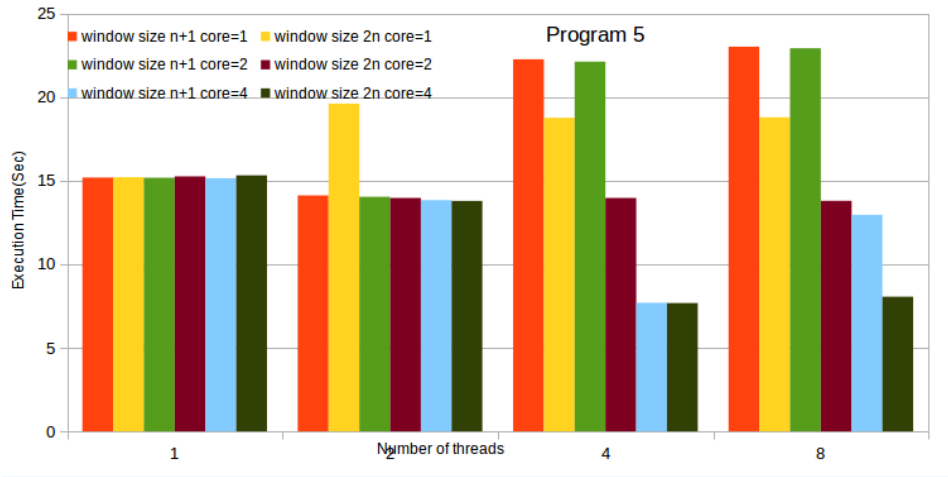

Figure 6.9: Execution time of program 5 having window size n+1 and 2n with different number of threads and cores.

6.10 Conclusion

GCC Compiler for Thread-Level Speculation Using OpenMP is a compile time system that automatically adds the code needed to handle speculatively parallel execution of a loop. It uses a new OpenMP clause to find those variables that may lead to a dependency violation. The plug_in mechanism provided by GCC is used to support the new OpenMP clause. By this method, programmers can point out the speculative variables and need not know anything about the speculative parallelization model. For the generation of the parallel code the programmer has to add only one line, instead of the significant number of lines required by the manual parallelization. The system performance will also depend on the relationship between the number of threads and window size. The experiment results show that maximum performance is obtained when the window size is set as double the number of threads and the number of threads is equal to the number of cores.

7

Summary of Results, Conclusions and Future Works

7.1 Abstract

The summary of the results on the study of different disk scheduling algorithms, various page replacement methods, branch handling techniques and the conversion of sequential program to parallel program are presented in this chapter. Also the performance evaluations of novel algorithms BFQ, MBFQV1 and MBFQV2 for achieving better data transferring speed between various memory levels are summarized. Evaluation of various page replacement methods along with the new LRU-LFU, MRU-LFU, LRU-MFU, MRU-MFU are done to reduce page fault rate. New CaMMEBH algorithm was suggested for branch handling during the perfetching process. All these algorithms are compared with the current algorithms and alternate usages of the new algorithms are suggested. Usage of multicore processor in sequen-

tial execution environment, and inefficient usage of recourses are also discussed. For better utilization of the recourses, the method to convert a sequential program to a parallel program is also suggested.

7.2 MBFQV2

When the disk receives multiple synchronous requests for the accessing of data, these requests are handled in such a way that it gives maximum throughput. To achieve this instead of CFQ, BFQ and modified versions of BFQ are used. Implementation of BFQ gives a better result while comparing with the current technique CFQ. Some of the draw backs of BFQ are identified which leads to the modified versions MBFQV1 and MBFQV2. It is found that MBFQV1 gives a better performance when compared with the BFQ. MBFQV2 the new disk scheduler combined with proper back-shifting of request timestamps may allow a timestamp based disk scheduler to preserve both guarantees and a high throughput. In MBFQV2 it is observed that the throughput and speed of data transfer were better compared to the other schedulers for normal size applications.

7.3 LRU-LFU

It is observed that during the replacement of the data from various memory levels page replacement algorithms make use of either frequency parameter or time parameter. In LRU-LFU both frequency and time parameters are used. The performance of the system depends on the number of page faults which the system undergoes. Hence by reducing the page fault number during the page replacement process the system performance can increase. Here MRU, LFU and MFU were implemented and a comparative study was done with the existing LRU. Along with these, the combination of all methods such as LRU-LFU,

MRU-LFU, LRU-MFU and MFU-MRU are done and a comparative study is made with existing LRU. It is observed that, in the case of LRU-LFU the major fault and minor fault were less compared to existing LRU. Due to the reduction in major fault and minor fault the elapsed time is also reduced for LRU-LFU.

7.4 CaMMEBH

CaMMEBH is about reducing page fault by managing the branch handling functions. While handling the conditional branch instruction the prefetch function will give importance to only one condition in existing systems. But here branch handler will consider both the conditions. For this the data structures and the kernel functions that deal with branch handling are edited to increase the hit ratio in the cache memory and thereby reducing the page fault. Implementation of CaMMEBH is done and a comparison is done with the existing algorithm for the parameters major fault, minor fault and elapsed time. It is observed that CaMMEBH is able to give a better result when compared with the existing method.

7.5 Performance Enhancement

Here all the techniques disused so far are consolidated such as MBFQV2 along with some modified page replacement policies LRU-LFU and CaMMEBH which reduces page fault, elapsed time and improves the system performance significantly.

7.6 TLS

Thread-Level Speculation Using OpenMP is a compile-time system that automatically adds the code needed to handle the speculative parallel execution of a loop and uses a new OpenMP clause to find those variables that may lead to a dependence violation. The plug-in mechanism provided by GCC is used to support the new OpenMP clause. By this method, programmers can point out the speculative variables and need not know anything about the speculative parallelization model. For the generation of the parallel code the programmer has to add only one line to identify the variable with dependency instead of significant amount of lines required by the manual parallelization. The system performance will also depends on the relationship between the number of cores, number of threads and window size. The experiments show that when the window size is doubled the overall performance of the system gets increased. Implementaion of TLS is done with varying window size, the numbers of cores and threads. The result obtained shows that when the window size is double maximum performance is obtained. The performance evaluation of the program is considered using execution time .

7.7 Research Conclusions

The work includes an elaborated study on the memory management modules of the Linux Operating Systems. Elaborated studies on different branch prefetching methods are also done for reducing the page fault. Different algorithms are implemented on each of the memory levels and improvements have been attained on these areas resulting in proposal of new algorithms. Performance studies were conducted to assess the merits of the new algorithms. The data transfer method CFQ is modified with a new service oriented scheme BFQ and the draw back of BFQ is identified and modification is suggested as new algorithms MBFQV1 and MBFQV2. Better throughput is achieved using

MBFQV2, where each time new budget value is calculated based on the processes present in the request queue. The page replacement methods are then studied. The various methods are implemented by considering time and frequency parameters. The results show that LRU-LFU will give a better performance when compared with the other methods. Branch handling conditions are addressed next. A new algorithm CaMMEBH is developed to handle true and false conditions for branch statements. In CaMMEBH the data structures and the kernel functions that deal with the branch handling are edited to increase the hit ratio in the cache memory and thereby reducing the page fault. All the above methods are merged and results were observed. Result shows that around 40% reduction in major fault, 6% reduction in minor fault and 16% reduction in waiting time is obtained by these methods. Parallel execution of the program is taken in to consideration as the last phase. Experiments are done on OpenMP by varying the number of cores, threads and window size and the results are observed. System performance can be sgnificantily improved by using all the cores present in the processor with double window size.

Altogether the work aims to the enhancement of processor performance and which contributes an efficient system. There is no need to change the existing hardware which helps to reduce E-waste. Better performance is obtained compared to existing system.

7.8 Future Work

Following are few suggestions for future work:

1.The CaMMEBH can be extended to prefetch the instructions other than branch handling instructions to enhance the efficiency to a greater extent.

2. Currently the Thread level speculation concept is implemented only for the ‘for loop’. This speculation concept can be extended to other

types of loops by which a complete level of parallelism can be achieved inside a multi-core system.

Published Work of the Author

[1] Abraham, J.P. & Mathew, S. (2012). Study on branch handling. *National Conference on Emerging Trends in computing Technology*, 56 – 60.

[2] Abraham, J.P. & Mathew, S. (2013). An attempt to improve the processor performance by proper memorymanagement for branch handling. *International journal of computer science, engineering and application*, **3**, 81 – 88.

[3] Abraham, J.P. & Mathew, S. (2013). Cacheline override in branch target buffer used in branch handling. *International Journal of Emerging trends and Technology in Computer Science*, **2**, 62 – 65.

[4] Abraham, J.P. & Mathew, S. (2013). The effective way of processor performance enhancement by proper branch handling. *Third international Conference on computer Science and information Technology(CCSIT-2013) AIRCC*, 451 – 457.

[5] Abraham, J.P. & Mathew, S. (2015). A novel approach to improve the processor performance with page replacement method. *International Conference on information and communication Technology Published*, **46**, 1371 – 1376.

PUBLISHED WORK OF THE AUTHOR

[6] Abraham, J.P. & Mathew, S. (2016). High Throughput disk scheduling with equivalent bandwidth sharing. *IEEE - Inernational conference on Innovations in Information, Embedded and Communication Systems*, 653 – 657.

[7] Abraham, J.P. & Mathew, S. (2016). A novel approach to improve the system performance by proper scheduling in memory management. *Lecture Notes in Electrical Engineering*, **394**, 79 – 92.

[8] Abraham, J.P. & Mathew, S. (2016). Modified gcc compiler pass for TLS by modifying the window size using Openmp. *CSA-2016 . 5Th international conference in computer science and application*, 205 – 212.

[9] Abraham, J.P. & Mathew, S. (2016). Software optimization technique for the reduction page fault to increase the processor performance. *International Journal of Engineering and Technology*, **9**, 1180 – 1186.

[10] Abraham, J.P. & Mathew, S. (2017). High Throughput disk scheduling with equivalent bandwidth sharing. *Indian Journal of Science And Tecnology(Journal extention of the conference),(Accepted).*

References

[1] "Cache Memories," *ALAN JAY SMITH Computing Surveys*, vol. 14, no. 3, pp. 473 – 530, 1982.

[2] A. R. Alameldeen and Wood, "Interaction Between Compression and Prefetching in Chip Multiprocessors," *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 228 – 239, 2007.

[3] A. R. Alameldeen and D. A. Wood, " Adaptive cache Compression for High-performance Processors." *In Proc. Of the 31st Annual International Symposium on Computer Architecture,*, pp. 212 – 223, 2004.

[4] S. Albers, *Competitive online Algorithms*. Max-planck-institute for informatik, Im stadlwald 66123, Germany.

[5] S. Aldea, D. R. Llanos, and A. González-Escribano, "Support for thread-level speculation into OpenMP." LNCS,Springer, Heidelberg, 2012, vol. 7312, pp. 275 – 278.

[6] S. Aldea and A. Gonzalez-Escribano, "A New GCC Plugin-Based Compiler Pass to Add Support for Thread-Level Speculation into OpenMP." Springer International Publishing,, 2014, pp. 234 – 245.

[7] T. Alexander and G. Kedem, "Distributed prefetch-buffer/cache design for high performance memory systems," *Proceedings of*

the Second International Symposium on High-Performance Computer Architecture,San Jose, CA, USA,, 1996.

[8] J. N. Amaral, A. Douillet, Stoutchinin, G. R. Gao, J. Dehnert, and S. Jain, "Speculative Prefetching of Induction Pointers," *in CC-10*, 2001.

[9] P. P. Athavale, P. Ranadive, M. N. Babu, S. Sah, V. Vaidya, and C. Rajguru, "Utomatic Sequential to Parallel Code Conversion: The S 2 P Tool and Performance Analysis," *GSTF Journal on Computing(JoC)*, 2012.

[10] A.-H. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tsen, "The Efficacy of Software Prefetching and Locality Optimizations on Future Memory Systems," *The Journal of Instruction-Level Parallelism*, vol. 6, no. ISSN - 1942 - 9525, 2004.

[11] J. A. Brown, L. Porter, and D. M. Tullsen, "Fast Thread Migration via Working Set Prediction," *In proceedings of the 17th International Synposium on High Performance Computing (HPCA 2011)*, 2011.

[12] R. E. Bryant and D. O. Hallaron, *Computer systems a programmers perspective.* Pearson Education.

[13] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," *4th Conference on Architectural Support of Programming Languages & Operating Systems. New York, NY, USA: ACM.*, no. ISBN - 0 - 89791 - 380 - 9, pp. 40 – 52, 1991.

[14] M. Chandi, I. Foster, and K. Kenney, "Integrated Support for task & Data parallelism," *The International Journal of High Performance Computing application*, 1994.

[15] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (SSMT)," *Proceedings of the 26th International Symposium on Computer Architecture*, no. ISSN - 1063 - 6897, pp. 186 – 195, 1999.

[16] M. Charikar, "Advanced Algorithm Design: Marking Algorithm,Lectured by," *Transcribed by Borislav Hristov*, 2013.

[17] T.-F. Chen and J. L. Baer, "A performance study of Software and Hardware Data Prefetching Schemes," *In Proc. Of the 21st Annual International Symposium on Computer Architecture*, 1994.

[18] B. Choi, L. Porter, and D. M. Tullsen, "Accurate Branch Prediction for Short Threads," *ASPLOS08 March 1-5, Seattle, Washington, USA.*

[19] M. Cintra and D. R. Llanos, "Toward efficient and robust software speculative parallelization on multiprocessors," *In: Proceedings of PPoPP*, pp. 13 – 24, 2003.

[20] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," *28th Annual International Symposium on Computer Architecture , 2001,*, no. ISSN - 1063 - 6897, pp. 14 – 25, 2001.

[21] N. Dafre and D. Kapgate, "Novel Cache Replacement Algorithm," *International Journal Of Engineering And Computer Science*, vol. 3, no. 6, pp. 6260 – 6266, June - 2014.

[22] F. H. Dang, H. Yu, and L. Rauchwerger, "The R-LRPD test: Speculative parallelization of partially parallel loops," *In: Proceedings of 16th IPDPS*, vol. 29, pp. 20 – 29, 2002.

[23] S. Das, *UNIX concept and applications*, 4th ed. Tata McGraw-Hill.

[24] M. Dhamdhere, "Operating System A concept-based approch."

[25] A. Douillet, Stoutchinin, J. N. Amaral, and G. R. Gao, "Speculative Prefetching of Induction Pointers," *in CC-10*, 2001.

[26] M. Erez and M. Madhav, "Branch Prediction," *IEE482:AdvancedComputer Organization, Stanford University, Lecture-3 Processor Architecture Stanford University*, 2000.

[27] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, "New Data Structures to Handle Speculative Parallelization at Runtime," *In: Proceedings of HLPP 2014*, vol. 2014, 2014.

[28] M. Evers, "Improving Branch Prediction by Understanding Branch Behavior," *PhD thesis, University of Michigan,*, 2000.

[29] N. Fukumoto, T. Mihrara, K. Inoue, and K. Murakami, "Analyzing the Impact of Data Prefetching on Chip MultiProcessors," *[published in: Computer Systems Architecture Conference, 2008. ACSAC 2008. 13th Asia-Pacific, IEEE Xplore: 16 September 2008*, no. INSPEC - Accession - Number - 10220671, 2008.

[30] L. Gao and L. Li, "SEED: A statically greedy and dynamically adaptive approach for speculative loop execution," *EEE Trans. Comput*, vol. 62, no. 5, pp. 1004 – 1016, 2013.

[31] M. Gonzalez, J. Oliver, X. Martorell, E. Ayguade, J. Labarta, and N. Navarro, "OpenMP extensions for thread groups and their run-time support," *InProceedings of the Workshop on Languages and Compilers for Parallel Computing, volume 2017 ofLNCS*, vol. 2017, pp. 324 – 338, 2001.

[32] J. Gummaraju and M. Franklin, "Branch prediction in multithreaded processors," *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques*, no. ISSN - 1089 - 795X, pp. 179 – 188, 2000.

[33] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*, 4th ed. Elsevier.

[34] K. Hwang and F. A. Briggs, *Computer architecture and parallel processing*. Tata McGraw-Hill.

[35] K. Hwang and N. Jotwari, *Advanced computer Architecture, Parallelism,Scalability,Programmability*, 2nd ed. McGraw-Hill.

[36] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 121 – 133, 1999.

[37] B. Juurlink, "Approximating the Optimal Replacement Algorithm," *CF04 April 1416, 2004, Ischia, Italy.Copyright 2004 ACM 1-58113-741-9/04/0004*, 2004.

[38] M. Kandemir, "Adaptive prefetching for sharedcache based chip multiprocessors," *Proceedings of the Conference on Design, Automation and Test in Europe*, no. ISBN - 978 - 3 - 9810801 - 5 - 5, pp. 773 – 778.

[39] S. Khajoueinejad, M. Sabeghi, and AzamSadeghzadeh, "A Fuzzy Cache Replacement Policy and its Experimental Performance Assessment," *IEEE.12 February 2007*, no. INSPEC - Accession - Number - 9309205, 2007.

[40] C. Lijun, *Understanding Linux kernel source code deeply.* 2002 :Posts & Telecom Press, 2002.

[41] G. H. Loh and D. S. Henry, "Predicting Conditional Branches With Fusion-Based Hybrid Predictors," *NSF Grant*, no. MIP - 9702281.

[42] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen, "The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System," *IEEE.*, 2003.

[43] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," *7th Conference on Architectural Support of Programming Languages & Operating Systems. New York, NY, USA: ACM*, no. ISBN 0-89791-767-7, pp. 222 – 233, 1996.

[44] M. M. Mano, *Computer System Architecture*, 3rd ed. PEARSON.

[45] P. Marcuello, "Speculative Multithreaded Processors," *Ph.D Thesis, Universital Politecnical de Catalunya*, 2003.

[46] A. Meyerson, "Online algorithms for network design," *In: Proceedings of the Annual ACM Symposium on Parallelism in Algorithms and Architectures, ACM16th Press*, pp. 275 – 280, 2004.

[47] M. Milovanović, R. Ferrer, O. S. Unsal, A. Cristal, X. Martorell, E. Ayguadé, J. Labarta, and M. Valero, "Transactional memory and OpenMP," *In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS*, vol. 4935, pp. 37 – 53, 2008.

[48] S. Mittal, "Survey of recent prefetching techniques for processor caches," *ACM COMPUTING SURVEY*, vol. 49, no. issue2, 2016.

[49] C. E. Oancea and A. Mycroft, "Software thread-level speculation: An optimistic library implementation," *Proceedings of 1st international workshop on multicore software engineering-2008*, no. ISBN - 978 - 1 - 60558 - 031 - 9, pp. 23 – 32, 2008.

[50] E. J. O'Neill, P. E. O'Neill, and G. Weikum, "The LRU-K page replacement Algoritham For Database Disk Buffering," *SIGMOD Washington,DC,USA 1993 ACM*, 1993.

[51] M. A. P. Valente, "Improving Application Responsiveness with the BFQ Disk I/O Scheduler," *Proceed- ings of the 5th Annual International Systems and Storage Conference (SYSTOR 12). ACM, New York, NY, USA*, 2014.

[52] Peterson, *The Complete reference Linux*, 2nd ed. Tata McGraw-Hill.

[53] J. Pierce and T. N. Mudge, "Wrong-path instruction prefetching," *In international Symposium on Microarchitecture*, pp. 165 – 175, 1996.

[54] S. Pintar and A. Y. Tango, "A hardware-based data prefetching technique for superscalar processors," *In Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 214 – 225, 1996.

[55] A. Sane, P. Ranadive, and S. Sah, "Data dependency analysis using data write detection techniques," *ICSTE 2010*, vol. Vol - 1, pp. VI – 9 – VI – 12, 2010.

[56] A. Silberschatz and Bear, *Operating systems concepts*, 5th ed. New York :John Wiley Sons, Inc, 1999.

[57] A. Silberschatz and P. Bear, *Operating systems concepts*, 4th ed. New York :John Wiley Sons, Inc, 1999.

[58] N. Sivasubramaniam and P. Senniappan, "Enhanced core stateless Fair Queuing with Multiple Queue Priority Schedular," *[The International Arab Journal of Information Technology- march 2*, vol. 11, no. 2, pp. 159 – 167, 2014.

[59] S.layer and Druschel, "Anticipatory scheduling:A Disk Scheduling Framework to overcome Deceptive Idleness in synchronousI/O," *Proc.18th ACM Symp. Operating systems Principles*, 2001.

[60] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Communication of the ACM*, vol. 28, no. ACM - 0001 - 0782 - 85 - 0200 - 0202, pp. 202 – 209, 1985.

[61] M. G. Sobell, *A Practical Guide to Solaris*.

[62] X. Z. SongJianga, "Token-ordered LRU:An effective pagereplacement policy and its implementation in Linux systems," *Elsevier*, vol. 60, no. 1 - 4, pp. 5 – 29, 2004.

[63] W. Stallings, *Operating Systems*, 4th ed. Person education.

[64] D. Stephens, J. Bennett, and H. Zhang, "Implementing Scheduling Algorithms in High speed Networks," *IEEE J.Selected Areas Comm.*, vol. 17, no. 6, pp. 1145 – 1158, 1999.

[65] J. Subholle, J. M. Slichnoth, d. R. O'Hallaron, and TomasGrose, "Exploiting task and data parallelism," *ProceedingPPOPP '93 Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 13 – 22.

[66] A. S. Tanenbaum and A. S. Woodhull, *Operating system Design and Implementation*. PHI.

[67] V, Srinivasan, E. S. Davidson, and G. S. Tyson, "Branch History Guided Instruction Prefetching," *In proceedings of the 7 thInternational Conference High-Performance Computer Architecture, 2001. HPCA*, no. INSPEC - Accession - Number - 6846686, 2001.

[68] V. Vaidya, S. Sah, and P. Ranadive, "Optimal Task Scheduler For Multicore Processor," *ICSTE2010*, vol. 1, pp. pp – VI – 1 – VI – 4.

[69] V. G. Vaidya and S. Sah, "A Review of Parallelization Tools and Introduction to Easypar," *International Journal of Computer Applications*, vol. 56, no. Issue - 12, pp. 30 – 34, 2012.

[70] V. G. Vaidya and S. Shah, "Dependency aware ahead of time static scheduler for multicore," *IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)*, no. DOI - 10 - 1109 - ICIS - 2014 - 6912156Conference, 2014.

[71] P. Valente and F. Checconi, "High Throughput Disk Scheduling With Fair Bandwidth Distribution," *IEEEtransactions on computers*, vol. 59, no. INSPEC - Accession - Number - 11446908, pp. 1172 – 1186, 2010.

[72] S. Wang, X. Dai, K. S, and Y. A. Zhai, "Loop Selection for Thread-Level Speculation," *International Workshop on Languages and Compilers for Parallel Computing LCPC 2005,LCNS*, vol. 4339, pp. 289 – 303, 2005.

[73] P. Xekalakis, N. Ioannou, and M. Cintra, "Combining thread level speculation helper threads and runahead execution," *In: Proceedings of ICS*, vol. 2009, pp. 410 – 420, 2009.

[74] P. Yiapanis, G. Brown, and M. Lujan, "Compiler-driven Software Speculation for Tread-level Parallelism," *ACM Transaction on Programming Languages and systems-2016*, no. ISSN - 0164 - 0925 - EISSN - 1558 - 4593, 2016.

[75] H.-j. B. Zhan-sheng Li, Da-wei Liu, "CRFP: A Novel Adaptive Replacement Policy Combined the LRU and LFU Policies," *Computer and Information Technology Workshops, 2008. CIT Workshops 2008. IEEE 8th International Conference on*, 2008.

[76] W. Zhang, D. M. Tullsen, and B. Calder, "Accelerating and Adapting Precomputation Threads for Efficient Prefetching," *n Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA 2007) IEEE*, pp. 85 – 95, 2007.

[77] Y. Zhang, S. Haga, and R. Barua, "Execution history guided prefetching," *In Proceedings of the 16thinternational conference on Supercomputing,New York,USA*, pp. 199 – 208, 2002.

[78] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," *SIGARCH Computer Architecture News*, 2001.

[79] Y. Zong-de and D. Y. chun andZHENG Qing-hua, *Advanced Linux Programming.* The People's Posts and Telecommunications Press, 2008.

[80] [Online]. Available: http://algo2.iti.kit.edu/vanstee/courses/caching.pdf

[81] http://blog.scoutapp.com/articles/2015/04/10/understandingpage-faults-and-memory-swap-in-outs-when.-should-you-worry

[82] http://mirror.linux.org.au/pub/linux.conf.au/2007/video/talks/123.pdf,2010

[83] http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers/,accessedOct2011

[84] http://www.cs.utexas.edu/users/mckinley/352/lectures/23.pdf

[85] http://www.linux.org/threads/the-linux-kernel-the-source-code.4204/

[86] http://www.linuxhowtos.org/

[87] http://www.linuxjournal.com/article/1052

[88] http://www.lohninger.com/helpcsuite/how_to_use_fifos.htm

[89] http://www.phoronix.com/scan.php?page=news_item&px=more-x86-asm-to-c-linux

[90] http://www.spinics.net/lists/linux-assembly/msg00293.html

[91] http://www.tldp.org/LDP/khg/HyperNews/get/syscall/syscall86.html

[92] http://www.tutorialspoint.com/assembly_programming/assembly_arithmetic_instructions.htm

[93] https://en.wikipedia.org/wiki/Linux_kernel

[94] https://github.com/torvalds/linux

[95] https://www.autoitscript.com/autoit3/docs/functions/Call.htm

[96] https://www.inso.tuwien.ac.at/uploads/media/OSKP_MonoMicroExo.pdf

[97] https://www.kernel.org

[98] IBM:Thread-levelspeculativeexecutionforC/C++.IBMXLC/C++forBlueGene,Tech.report(2012)

[99] www.kroah.com/lkn/

[100] www.linuxjournal.com/article/7105

[101] www.makelinux.net/books/lkd2/

[102] www.openmp.org

[103] www.akhilnarang/kernal_bullheado

[104] http://web.njit.edu/rlopes/Mod5.3.pdf

[105] http://istc-bigdata.org/index.php/
memory-wall-what-memory-wall

[106] *Programming Manual,Logix5000 Controllers Major, Minor, and I/O Faults.*

[107] "GNU Project: GCC internals (2013," 2013 http://gcc.gnu.org/
onlinedocs/gccint/