Energy Efficient Techniques for Multi-tasking Embedded Systems – Cache Design and Task Scheduling Algorithms

THESIS

Submitted in partial fulfillment
of the requirements for the degree of
DOCTOR OF PHILOSOPHY

by

BIJU K R

Under the Supervision of
Prof. S Gurunarayanan

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN) INDIA
March 2009

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

PILANI (RAJASTHAN) INDIA

CERTIFICATE

This is to certify that the thesis entitled "**Energy Efficient Techniques for Multi-tasking Embedded Systems – Cache Design and Task Scheduling Algorithms**" and submitted by **Mr. Biju K R** ID No. **2003PHXF418P** for award of Ph.D. degree of the Institute embodies original work done by him under my supervision.

_____

Signature of the Supervisor

Date: Name: Dr. S. GURUNARAYANAN

Professor (Electronics & Instrumentation)

Dedicated

To God

for gifting me with the best

Parents, Teachers & Friends

one can have…

ACKNOWLEDGEMENTS

This thesis arose in part out of years of research that has been done. During this period, I have met several people who have made a significant contribution in assorted ways to the research and the making of this thesis and who deserve special mention. I am glad to take the opportunity to convey my gratitude to them in my humble acknowledgement.

In the first place, I deem it a great pleasure to express my gratitude whole-heartedly to Prof. S Gurunarayanan, Dean, FD II and Admissions for his supervision, valuable advice, suggestions and guidance from the very early stages of this research as well as giving me extraordinary experiences throughout the work. Besides providing me unflinching encouragement and support in various ways, he has also allowed me the freedom to experiment with my innovation which has in a major proportion enhanced and nourished my intellectual growth.

I am greatly indebted to Prof. J. P. Misra, Unit Chief, IPC Unit, Prof. Sundar Balasubramaniam, Assistant Unit Chief, IPC Unit and, Dr. T S B Sudarshan, Group Leader, CS-IS Group for their crucial contribution, constructive comments, motivation and collaboration. I thank them for their willingness to share their knowledge with me, which was very fruitful in shaping my ideas and research. I acknowledge their promptness in providing me with all the equipment to facilitate the research. Collective and individual acknowledgements are due to all my colleagues who have directly or indirectly helped me in my work.

Thanks are due to the Microsoft Research, India (MSRI) for providing the financial support for the research. A special word of appreciation is owed to Prof. Rahul Banerjee, Unit Chief, SDETU for providing the necessary aid, support and facilities on several occasions. Thanks are due to Prof. L K Maheshwari, Vice-chancellor and Prof. G. Raghurama, Deputy Director (Academic) for the constant support and concern. I would like to gratefully acknowledge Prof. Ravi Prakash, Dean, RCD and many others for their indispensable help and for creating a pleasant working atmosphere.

Words fail me to express my gratitude to my parents and sister who deserve a special mention for their inseparable support and prayers. Without their care and love, this thesis would not have been possible. Many thanks are due to Dr. Shikha Tripathi, Mrs. Ashwathi Sidharth, and all my other friends who were always ready to lend a hand. I thank everybody who was important to the successful realization of this thesis, as well as express my apology that I could not mention personally one by one. Finally, I would like to thank God for always guiding me.

ABSTRACT

In most Real-Time Embedded Systems, the limited battery life is a major cause of interest and concern. In a bid to optimize the energy consumption, this issue is addressed at various levels – Architecture level (memory subsystem, Dynamic Voltage Scaling / Dynamic Frequency Scaling), Systems level (process management, memory management and compiler techniques), and Application level (efficient data structures and algorithm design). Of the various components, the memory subsystem (architecture level) and the operating system-related activities (systems level) share a considerable proportion of the energy consumption by Embedded Systems.

This thesis addresses the issue of optimizing energy consumption in Embedded System at the Architecture and the Systems levels. The primary source of energy consumption at the architecture level is the memory subsystem, especially the cache memory architecture. This work presents various techniques to reduce this cache-related energy consumption, majority of which is attributed to the data movement across the memory hierarchy demanded and initiated by cache misses. One way to reduce this energy consumption is to improve the cache performance which entails an enhanced cache hit rate. This work proposes a new replacement policy called Late Least Recently Used (LLRU) replacement policy which while deciding on replacement, particularly considers cache lines that are shared among processes. Different hardware designs and implementations of the LRU and LLRU replacement policy have been put forth. Here, a way – predictive placement scheme, a modification of the way – predictive cache, for reducing the cache access time and power consumption has also been proposed and evaluated. One other means of achieving reduced energy consumption in Embedded Systems' cache is to power down all the unused data and tag ways. Motivated by this reasoning, this work proposes two process aware cache architectures, the Process Aware Selective Placement (PASP) and the Shared Memory Process Aware Selective Placement (SMPASP) which are designed to facilitate the powering on of only one selected way and the shutting down of all the other ways. This significantly reduces the dynamic power consumption of the tag and data arrays.

At the Operating Systems level, there have been three basic approaches to resolve the power consumption problem: process scheduling techniques, efficient paging systems,

and performance tuning. In this thesis, to improve the energy efficiency at the Operating Systems level, new scheduling algorithms have been devised. The scheduling algorithms can achieve energy efficiency in both platform-dependent (clock, device characteristics, or memory technology) and platform independent (preemption reduction) scenarios.

This work aims at increasing the platform independent energy efficiency by optimizing the number of preemptions caused in a schedule. The direct cost among other overheads associated with preemptions in a schedule is the time and energy spent for loading and saving the context of relevant processes, which is effectively nonproductive and hence, this reasoning advocates the design of the proposed scheduling algorithms. IntFragment – a static real-time scheduling algorithm establishes this motive. This thesis also proposes two dynamic real-time priority scheduling algorithms – Earliest Deadline First with Reduced Context Switches (EDFRCS) and Rate Monotonic with Reduced Context Switches (RMRCS), which are modifications of EDF and RM respectively. These two schemes also cut down on the number of preemptions caused in a schedule by allowing the execution of the currently executing task, whenever possible. The heuristic used in these schemes is very aggressive and satisfies the scheduling optimality (schedulability) condition of the respective original algorithms.

An indirect but more significant cost brought about by preemptions is the cache flushes which may, eventually even increase the cache miss rate. This thesis proposes a dynamic priority – based Reduced Cache Impact (RCI) algorithm, a modification of the existing Earliest Deadline First (EDF), which reduces the number of cache impact points in a schedule.

All the cache architectures were simulated, evaluated and tested for performance through simulation studies using SPEC95 and Simplescalar benchmarks. The various scheduling algorithms and cache conscious scheduling algorithms proposed were simulated and tested using synthetic benchmark suites.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AFC	Application / File level Characterization
ARJ	Absolute Response time Jitter
BTB	Branch Target Buffer
Conv	Conventional
CRPD	Cache Related Preemption Delay
DFS	Dynamic Frequency Scaling
DVS	Dynamic Voltage Scaling
ECAT	Effective Cache Access Time
ED	Energy Delay
EDF	Earliest Deadline First
EDFRCS	Earliest Deadline First with Reduced Context Switches
EELRU	Early Eviction Least Recently Used
EM	Evict Me
FCFS	First Come First Serve
FIFO	First In First Out
GDR	Inter-reference Gap Distribution Replacement
GDS	Greedy Dual Size
HBTC	History Based Tag Comparison
ICC	Indirect Index Cache
IRG	Inter Reference Gap
LFU	Least Frequently Used
LFUDA	Least Frequently Used Dynamic Aging
LLF	Least Laxity First

LLRU	Late Least Recently Used
LPS	Limited Preemptive Scheduling
LRU	Least Recently Used
MLLF	Modified Least Laxity First
MMUF	Modified Maximum Urgency First
MQ	Multi Queue
MRU	Most Recently Used
MUF	Maximum Urgency First
PASP	Process Aware Selective Placement
PLRU	Pseudo Least Recently Used
PSA	Predictive Sequential Associative
RAC	Reactive Associative Cache
RCI	Reduced Cache Impact
RCS	Reduced Context Switch
RM	Rate Monotonic
RMRCS	Rate Monotonic with Reduced Context Switches
RRJ	Relative Response time Jitter
SA	Set-Associative
SMPASP	Shared Memory Process Aware Selective Placement
SOC	Spatial Oriented Cache
TOC	Temporal Oriented Cache
UBM	Unified Buffer Management
WCRT	Worst Case Response Time
WP	Way Predictive

CHAPTER 1

INTRODUCTION

The explosive growth of Embedded System products has forced researchers to address various issues associated with them like energy consumption and performance. Modern Embedded Systems are complex integrated systems where real-time tasks execute in multi-tasking environments and compete for shared resources like processor, memory, etc. Multitasking is the ability to execute multiple separate tasks in a fashion that is seemingly simultaneous. The basic requirements of multi-tasking real-time embedded system are context switching, inter task communication, managing priorities and establishing timing control for managing multi-tasking in a real-time environment.

There are various factors that influence the performance of Embedded Systems. Embedded System designers have to strike upon beneficial compromise among the factors, i.e., size, energy consumption, cost and performance [Chandrakasan 1995] [Mudge 2000]. The energy consumption in an Embedded System is determined by functionalities of the system and speed of the processor. Most Embedded Systems are battery-driven and due to their limited battery life, energy consumption emerges as an important limiting factor. The energy consumption in an Embedded System can be addressed at various levels of the design hierarchy such as at the technology level, circuit level, architecture level, operating system level, and at the compiler level.

The reduction in dynamic energy consumption at the technology and circuit levels can be accomplished by reducing the average number of circuit switching per clock cycle and reducing the load capacitance. There exist various techniques to minimize this energy consumption. The average number of circuit switching is achieved using techniques like minimizing the Hamming distance between operations/instructions [Lee 2000] or minimizing the number of operations [Hong 1997]. The load capacitance can be minimized by using place and route optimizations. The system level energy reduction can be attained by the Dynamic Voltage Scaling / Dynamic Frequency Scaling (DVS/DFS). This is achieved by running the real-time tasks at a reduced applied voltage for a longer time (reduced frequency) while ensuring that no task misses its deadlines. With the advancement in technology, the most important component of technology level energy

consumption is the leakage energy which is due to the leakage current between the supply voltage and the ground. One of the main components of the leakage energy is the sub-threshold leakage energy and it is dominated by temperature and threshold voltage. The static (leakage) energy consumption occurs as long as the CMOS device is powered on. Instruction scheduling by an energy-aware compiler can reduce the average number of circuit switching resulting in reduced energy consumption. The Compiler can also be used to identify shared, spatial and temporal accesses to improve the memory access performance.

In an Embedded processor, the major source of energy consumption is at the architecture and operating system levels. At the architecture level, the major source of energy consumption is the memory subsystem. A significant fraction or part of the memory subsystem energy consumption is caused by the cache memory activities. This connotes that saving a considerable portion of energy consumed by the cache memory will have a considerable impact on the overall energy consumption. Cache memory energy consumption can be reduced by reducing the cache miss rate, internal activities, associated control and replacement circuitry and shutting down the unused cache blocks. In a multi-tasking environment, majority of the cache misses are the aftereffects of task preemptions (context switches). The time and energy spent in transferring the context of the tasks and serving cache misses are unproductive. This energy consumption can be addressed at the operating system level with the help of an efficient task scheduling algorithm [Xu 2005][Jianli 2005].

This thesis addresses the energy consumption issue of multi-tasking real-time embedded systems at the architecture and the operating system levels, as the activities at these levels constitute a majority of the Embedded System energy consumption. In this thesis, various techniques are proposed to reduce the cache and scheduling-related energy consumption.

1.1. ENERGY EFFICIENT CACHE ARCHITECTURE

The cache memory subsystem consumes a significant amount of energy in Embedded Systems. One way to reduce cache memory related energy consumption is to reduce the cache miss rate, which depends on the mapping scheme used. The cache memory mapping schemes like direct-mapping and set-associative mapping have a substantial

impact on the cache miss rate and on the dynamic energy consumption per cache access. Both these schemes have their own merits and demerits in terms of the dynamic power consumption and performance. A direct-mapped cache accesses only one tag block and one data block per cache access, whereas an N-way set-associative cache accesses N tag blocks and N data blocks per cache access. So the dynamic power consumption of a direct-mapped cache is much lower as compared to that of a set-associative cache of the same size. On the other hand, a set-associative cache offers a better cache hit rate and thus lesser cache miss related energy consumption as compared to that by a direct-mapped cache. A direct-mapped cache may not always result in less overall power consumption. Set-associative caches are used for applications, which require a high cache hit rate and low energy consumption, even though they have an additional overhead of increased dynamic power consumption for tag comparisons. Moreover, the set-associative mapping scheme provides adequate support for energy efficient caching schemes like way shutdown, way concatenation, way prediction and process aware caching.

Experimental studies [Hennessy 2007] prove that an increase in associativity causes an accompanied decrease in miss rate, hence reduced energy consumption. For example, the average miss rate for the SPEC92 benchmark programs is 4.6% for an 8Kbyte direct-mapped cache, 3.8% for an 8Kbyte 2-way set-associative cache and 2.9% for an 8Kbyte 4-way set-associative cache. Even though the decrease in miss rate is small, set-associative cache causes a significant performance improvement due to the large cycle time penalty overhead for a cache miss. Hence, on measuring the performance of a cache in terms of the energy consumption, a set-associative cache gives better performance than the direct-mapped cache. The energy consumption characteristic of a cache memory varies with the cache size as well. Though a small cache size is energy efficient and has less access latency, it suffers from poor hit rate.

In set-associative and fully associative schemes, the hit rate of the cache depends on the placement / replacement circuit in use. These schemes use the replacement circuit to determine the cache lines to be evicted. The replacement algorithm improves the cache hit rate and hence, reduces the dynamic energy consumption of the cache memory.

The replacement schemes are categorized as optimal, random, arrival, recency, frequency and combinations of some of these based on how they choose a victim line for replacement. If the selection of a victim line is based on future references, which is practically impossible for a dynamic-scheduled system, then the replacement strategy to use is optimal replacement. If the victim line is selected randomly from the set, then the random replacement is used. If the choice of a victim line is based on the arrival time of blocks into the cache, then the FIFO replacement algorithm is used. If the selection of a victim line is based on past references then recency, frequency or combination of recency and frequency based replacement schemes are used. Some of the commonly used replacement strategies under recency, frequency and their combinations are Least Recently Used (LRU), Most Recently Used (MRU), Least Frequently Used (LFU), and Least Recently/Frequently Used (LRFU) [Smith 1982] [Lee 2001a].

LRU and its variants are the most widely used replacement algorithms for the cache memory. The performance of these algorithms is good if the workload maintains temporal locality and is close to that of the optimal replacement algorithm when the associativity is less. The LRU replacement algorithm is prone to wrong victim line selection because of bypass block, dead block and live block [Kampe 2004]. LRU can improve its performance if the bypass block is not allowed to enter into the cache, if the dead block is replaced at the earliest after leaving the MRU position and if the live block is held in the cache for a longer period. Some of the other situations where the LRU performance is not good are multi-tasking systems with a common cache for all the tasks, wherein the tasks exhibit changes in the memory access pattern during execution.

There exist various replacement algorithms like Early Eviction LRU (EELRU), Pseudo LRU (PLRU), modified LRU with non-temporal cache hint, Cache System with Replacement Controls (Cache/RC), Reference Locality Replacement (RLR), Software assisted LRU, Evict Me (EM) LRU, Self-correcting LRU (SCLRU) and LRU-SEQ, which address various issues associated with the LRU replacement strategy to improve cache hit rate.

From the analysis of the LRU replacement strategy, it is found that its performance is not good for multi-tasking real-time embedded systems with data sharing among tasks [Wang 2004]. The LRU performance degrades further with increase in preemptions between

tasks. None of the above mentioned replacement schemes address the issue of increasing the life span of shared data for improving cache hit performance in multi-tasking real-time embedded systems with data sharing.

To address this issue, this thesis proposes a new variant of the LRU replacement strategy called the Late LRU (LLRU) replacement strategy which helps in increasing the life span of shared cache lines. In LLRU, the shared cache line(s) gets a higher priority over the non-shared cache lines to guarantee lesser cache misses after preemption. This is because of the basic understanding that the new process may access the shared cache line. This scheme expects the compiler to issue the shared information through special instructions to enable efficient handling of shared pages during replacement. LLRU adds an extra shared bit per cache line to store this sharing information. The replacement circuit finds the cache line to be evicted based on the shared bit and the LRU value. A cache simulator was implemented which uses the Simplescalar benchmark address traces with varying data sharing for evaluating the cache hit rate of both LRU and LLRU replacement strategies. The hardware implementations of LRU and LLRU based on the square matrix as well as the counter were carried out to measure parameters like area, clock frequency, critical path delay and number of transistors using Modelsim, Leonardo spectrum and IC station. Though LLRU requires more area and increased number of transistors, this scheme improves cache hit rate and operating frequency and reduces critical path delay and effective cache energy consumption over the basic LRU scheme.

Another way of reducing the dynamic energy consumption in the cache memory is to reduce the internal activity of the cache during a cache access. The internal activities of a cache are defined as reading and comparing tags in tag arrays, and reading/writing data in data arrays. The minimum cache internal energy consumption can be achieved if the cache subsystem encounters minimum conflict misses. It can be further minimized if each cache hit results in reading and comparing only one tag entry, enabling and accessing one data entry and if each cache miss results in reading and comparing only one tag entry, and accessing no data entries. This can be achieved using the techniques of hardware prefetching, vertical partitioning, horizontal partitioning, reconfiguring cache, optimizing the control circuitry, making use of the compiler and operating system information to improve the performance and various combinations of some of these.

Horizontal cache partitioning schemes like phased lookup cache, difference bit cache, partial tag matching cache and way-prediction cache save dynamic energy at the cost of performance. The way – prediction cache [Inoue 1999] [Inoue 2001] saves a significant amount of energy by speculatively selecting one way for tag comparison. If the data is not available in the selected way (prediction miss), all the N-1 ways are enabled for tag comparisons in the next cycle.

The way – prediction scheme suffers performance degradation because of the cycle time penalty incurred for handling mispredictions. The way prediction cache uses the most recently used (MRU) way information of the set to decide the way to be selected. It requires $\log_2$ (N) bits per set to maintain this MRU way information for a N way set-associative cache. The MRU information of the way prediction cache is stored as a table where each row ($\log_2$ (N) bits) of the table corresponds to a set. This scheme requires a table lookup to obtain the MRU information of the selected set, which is possible only after extracting the set number from the physical address. This adds an extra time delay to the critical path. The performance of this scheme degrades slightly due to the elongated access time as a result of the table lookup. The way prediction scheme requires k * $\log_2$ (N) bits where k indicates the number of sets, to maintain the MRU information and these bits need to be updated during a misprediction or a cache miss. This increases the hardware complexity, effective cache access time and consequently, the energy consumption.

To address these issues, this thesis proposes a new caching scheme called the way predictive placement scheme which avoids the table lookup and also improves the prediction hit rate. This is achieved by replacing the table in the way prediction cache with a global $\log_2$ (N)-bit register. To improve the prediction hit rate and the cache hit rate in a way predictive placement scheme, a modified placement / replacement strategy called the Aligned LRU (ALRU) replacement strategy is proposed. This strategy aligns cache lines into the same way, whenever possible. For experimentation, Simplescalar 2.0 [Burger 1997] cache simulator was employed with different cache configurations. SPEC95 benchmark programs were used to obtain the prediction hit rate, cache hit rate, number of tag comparisons and the energy saving for various cache configurations of

way predictive placement cache. The selection of the SPEC95 benchmark program suite guarantees uniformity in evaluation as most of the existing cache architectures used this benchmark program suite for evaluation. The improvement in prediction hit rate, cache hit rate, number of tag comparisons and effective cache access time results in energy saving in a way predictive placement cache in comparison with a way prediction cache. Way predictive placement cache reduces the hardware complexity as well.

Multi-tasking real-time embedded systems prefer partitioning schemes as they reduce the dynamic energy consumption and make the cache predictable. The cache memory energy consumption can further be reduced with the help of cache – operating system – compiler – application program interaction. The energy reduction can be achieved by accurately predicting the cache line where the required data is available. This interaction can help the real-time energy aware scheduler to perform better in a multi-tasking environment. The interaction can also overcome the replacement mistakes because of bypass block, dead block and live block. Various software-controlled cache architectures transfer locality related information and information about whether the cache block is a bypass block, dead block or live block to the hardware through modified instructions. Some of the existing microprocessors have instructions for flushing the entire cache, cleaning a cache line and locking a cache line to reduce cache pollution and replacement mistakes.

Though some of the existing cache architectures can control cache pollution and replacement mistakes to an extent through cache – operating system – compiler interaction, they are not process aware. These schemes do not use process related information to improve the cache hit rate, shutdown unused ways and reduce energy consumption. The cache hit and energy consumption performance of these schemes are not consistent with varying context switching time. A process aware cache with dynamic allocation of cache ways can reduce the energy consumption further by shutting down all the unused ways.

In this thesis, two software controlled process aware energy efficient cache architectures - Process Aware Selective Placement (PASP) scheme and Shared Memory Process Aware Selective Placement (SMPASP) scheme are proposed for multi-tasking applications. These schemes make use of the cache – operating system – compiler interaction. The PASP scheme consists of an N-way set-associative cache and a small victim set of 3 – 5

cache lines. The SMPASP scheme consists of an N-way set-associative cache, a small victim set of 3 – 5 cache lines and a small shared set. In these schemes, to obtain consistent performance with varying context switch durations, one way / bank is allocated dynamically to a process. This converts an N-way set-associative cache into a direct-mapped cache, thus reducing the circuit complexity (because of replacement circuit) and dynamic energy consumption. The victim set is used for collecting spill out data from the dedicated way, thus improving the cache hit performance. The SMPASP scheme enhances the PASP scheme to further improve the cache performance by exploiting the large amount of data sharing exhibited among tasks in a multi-tasking real-time embedded system. In SMPASP scheme, the shared set is used for improving the cache hit performance by storing all the shared data of various tasks in the multi-tasking system. The PASP and SMPASP schemes transfer process-related information to the cache controller through a special instruction, which is added as a part of the context switching routine. The SMPASP scheme transfers shared information - obtained by using the compiler - to the cache through modified instructions. This information helps the cache controller to predict and enable the required cache way / cache line and power down all the unused cache ways / cache lines which results in reducing the cache miss rate and energy consumption. These schemes reduce cache misses, first cycle misses, number of tag comparisons, effective cache access time and dynamic energy consumption as compared to the conventional and way-prediction cache for both shared and non-shared data sets. A cache simulator CACHEMEM 1.0 was implemented, which uses SPEC 95 benchmark address traces with and without data sharing for evaluating the cache hit rate, first cycle hit rate, number of tag comparisons, and effective cache access time of the conventional, way prediction, PASP and SMPASP caches. The CACHEMEM 1.0 can accommodate different cache configurations including cache size, cache line size and context switching duration. The dynamic and leakage power consumption for the various caching schemes are obtained using the eCACTI cycle-based power estimation model.

1.2. ENERGY EFFICIENT TASK SCHEDULING

Task scheduling in multi-tasking real-time embedded systems constitutes a significant proportion of the energy consumption at the operating system level. Real-time task scheduling algorithms primarily focus on generating a feasible schedule without causing any deadline misses. In multi-tasking real-time embedded systems, constraints like availability of power, size of the memory, complexity of the algorithm and speed of the processor may affect the scheduling policies and algorithms. The energy efficient scheduling techniques designed to minimize the energy consumption in multi-tasking real-time system can be platform-dependent, which is architecture-specific such as multiple clock frequencies and multiple voltage levels. The Dynamic Voltage Scaling (DVS) and Dynamic Frequency Scaling (DFS) are such techniques that are used for reducing the dynamic energy of a system by executing the jobs at reduced operating frequencies and voltage levels. There exist platform independent factors such as the idle time, number of preemptions and cache impact which also affect the time of execution and energy consumption of a schedule. The preemptions in a schedule result in increased execution time due to the additional time required for context switch. This additional time spent during preemptions may affect the schedulability of the task set. The work done during preemptions is unproductive and hence, the energy consumed by the preemption routine is waste. A more significant but indirect impact of preemptions is reflected as cache misses.

There exist various real-time scheduling algorithms in literature which are designed to reduce the preemption count or its impact on a schedule. The motive of the Modified LLF (MLLF) and Optimized Minimum Laxity First (OMLF) scheduling algorithms is to fix the frequent preemption problem of Least Laxity First (LLF). Wang and Saksena [Wang 1999] proposed a fixed priority scheduling algorithm with a regular priority and preemption threshold to reduce preemptions in the schedule. In this scheme, a task having a lower preemption threshold than the executing task cannot preempt the executing task, even if it has a higher regular priority value. Vahid et al. [Vahid 2005] proposed the Modified Maximum Urgency First (MMUF) algorithm which replaces non-strict LLF in Maximum Urgency First (MUF) algorithm with MLLF for reducing preemptions and

improving the schedulability. The technique proposed by Jianli and Chaitali [Jianli 2005] demonstrates a preemption control technique for scheduling in processors with Dynamic Voltage Scaling. This technique is effective in reducing the number of unnecessary context switches caused by dynamic voltage scaling, particularly under low and medium processor utilization levels.

Though the above mentioned scheduling algorithms reduce preemptions when compared with their base versions, they are not aggressive enough to reduce it to the maximum possible extent. For instance, the number of preemptions in schedules produced by MLLF and OMLF is still higher than that of the Earliest Deadline First (EDF). Real-time scheduling algorithms which can reduce the preemptions aggressively, thus improving the schedulability and reducing the energy consumption are required.

To address this issue, a platform independent static scheduling algorithm called IntFragment and two platform independent dynamic scheduling algorithms called Earliest Deadline First with Reduced Context Switch (EDFRCS) and Rate Monotonic with Reduced Context Switch (RMRCS) are proposed. These real-time scheduling algorithms aggressively reduce the preemptions in a schedule without requiring extensive computations. The IntFragment scheduling algorithm reduces the number of context switches by generating a schedule with maximum fragments in between the execution of two instances (jobs) of the same task. The adjacent instances of the same task execute as close as possible or as distant as possible, which results in the creation of the maximum fragment. The grouping of similar jobs, as described above, results in reducing the cache impact. The EDFRCS and RMRCS scheduling algorithms try to extend the execution of currently running task for the maximum possible duration without affecting the schedulability of other tasks in the system. The heuristic used in these algorithms is not as simple as the MLLF heuristic, but it is much more efficient than the exhaustive search proposed by Wang and Saksena [Wang 1999] or the data flow technique proposed by Lee et al. [Lee 1999]. A simulator which simulates the EDF, RM, LLF, MLLF, IntFragment, EDFRCS and RMRCS schedules was implemented and the response time, response time jitter, latency and preemption count was measured using test suites. The test suites are randomly generated under certain conditions: each test suite is characterized by either a fixed number of tasks with utilization varying from low (50%) to high (100%) or by a

fixed utilization with the number of tasks varying from 2 to 20. Each test suite includes 100 different task sets of varying hyperperiods – from 100 to 32000. The energy consumption is calculated based on the preemptions and scheduling complexity. EDF, RM, EDFRCS and RMRCS algorithms were implemented in RTLinux and their performances were verified.

1.3. CACHE CONSCIOUS SCHEDULING

The preemption of jobs in a multitasking real-time system introduces additional cache misses in a schedule. This leads to an increase in the task execution-time which leads to deadline misses by the low priority tasks. As a cache miss consumes more energy than preemption, the minimum preemption schedule may not be the optimal energy efficient schedule. Hence, an energy efficient schedule should try to optimize the number of preemptions and the amount of data transferred across the memory hierarchies. The cache impact of a program is not constant throughout the program execution because the amount of data usage differs. Thus establishing the best possible preemption point where the cache impact is minimal is not a straightforward task.

In multi-tasking real-time systems, deadline misses caused due to cache misses can be avoided by calculating the tighter upper bound of the cache related preemption delay (CRPD) of each task accurately and adding it to the worst case execution time of that task [Lee 1998] [Negi 2003] [Staschulat 2005a]. Lee et al. proposed the Limited Preemptive Scheduling (LPS) which uses trace-based offline data flow analysis technique to determine the preemption points where the cache impact is minimal [Lee 1999]. Deadline misses because of cache misses can be avoided by minimizing the number of preemptions as cache impact at preemption points is usually high. Various preemption reduction schedulers like MLLF and OMLF produce a valid schedule with lesser cache misses than the schedule produced by its base scheduling algorithm (LLF).

None of the above mentioned scheduling policies exploit the properties of a periodic task set in a multi-tasking real-time system. The cache misses in a schedule can be reduced by combining jobs with maximum sharing. In a periodic task set, this can be achieved by combining jobs of the same tasks together as the code section and the global data section is shared among them.

To achieve this, a new cache-conscious scheduling algorithm called Reduced Cache Impact (RCI) algorithm is proposed, which combines similar jobs of the highest frequency task together without affecting the schedulability of the task set. RCI produces a schedule with significant reduction in the number of cache misses and the cache related preemption delay (CRPD), which results in energy saving. A simulator to calculate preemption count and cache impact of EDF, RM, LLF and RCI schedule, which uses the same test suite designed for preemption reduction algorithms was implemented. The implementation complexity and the schedulability of RCI algorithm were also analyzed.

1.4. THESIS ORGANIZATION

This thesis comprises eight chapters. Chapter 2 reviews various energy efficient cache architectures, energy aware real-time task scheduling algorithms and cache conscious real-time scheduling techniques. In Chapter 3, the proposed cache replacement policy LLRU is explained, experimentally evaluated and analyzed. Chapter 4 focuses on the issues related to way-prediction scheme and elaborates the proposed way predictive placement scheme, a modification to overcome some of the drawbacks of the way-prediction cache. The PASP and SMPASP which are process aware cache architectures for energy efficient embedded systems are proposed, explained, experimentally evaluated and analyzed in Chapter 5. In Chapter 6, the preemption reduction heuristic for the real-time static scheduling algorithm – IntFragment – is proposed. This chapter also proposes the preemption reduction variants of EDF and RM – EDFRCS and RMRCS respectively – which aggressively reduce the number of preemptions in a real-time schedule without causing any deadline misses. All these algorithms and the conventional real-time scheduling algorithms like EDF, RM, and LLF and MLLF, a reduced preemption version of LLF are studied in detail against various performance metrics in this chapter. Further, Chapter 7 proposes a cache conscious real-time scheduling algorithm – RCI – for energy efficient scheduling. The RCI algorithm is experimentally evaluated and its performance is analyzed against the EDF, RM and LLF. Chapter 8 concludes the thesis and briefly explains the future work.

CHAPTER 2

LITERATURE SURVEY

2.1. INTRODUCTION

The performance, cost, size and power consumption are some of the major concerns in Embedded Systems design. Reduction in power consumption is one of the most important tasks in embedded systems as most of the systems are battery operated. The power consumption issue of embedded systems has been addressed at various levels – technology level, architecture level, operating systems level, compiler level and system and application program level. The main sources of power consumption in embedded systems are at application and system programs level, operating system level and embedded architecture level.

The power consumed by application and system programs can be reduced by selecting the right data structure and algorithms for implementation, customizing the programs for the specific hardware, fine tuning / optimizing the application with the help of a compiler and reducing the complexity (time and space) of the program, etc.

The power consumed by architecture level can be reduced by redesigning the instruction set, redesigning / optimizing the memory subsystem, using a configurable architecture, and managing I/O devices efficiently.

At the operating systems level, the power consumption reduction issue can be addressed by improving the process scheduling techniques, inter process communication techniques, paging systems, and performance tuning for the specific hardware.

This work focuses on optimizing power consumption of embedded real-time systems at the architecture and operating system levels. This is because of the understanding that the majority of embedded system power consumption is due to memory subsystems and operating system functionalities. This chapter discusses various cache architectures proposed to minimize power consumption. This chapter also discusses power efficient scheduling algorithms and optimal scheduling and caching strategies for power efficiency.

2.2 CACHE ARCHITECTURES

Benini and Micheli [Benini 2000] conducted an exhaustive survey on various techniques and tools used in system level power optimization. According to Benini and Micheli [Benini 2000], one of the major components of architecture level power consumption is the memory subsystem. Benini et al. [Benini 2003] analyzed in detail about various architectures and optimization techniques used in embedded memories. Panda et al. [Panda 2001] surveyed various techniques used in data and memory related optimizations in embedded systems. From the analysis [Benini 2003] [Panda 2001], it is clear that majority of the memory subsystem power consumption is due to cache memory activities. Cache memories remain one of the hot topics in computer architecture research, as the ever-increasing speed gap between processor and memory emphasizes the need for more efficient memory hierarchy. Studies show that 42% and 23% of the total processor power in StrongARM 110 [Montanaro 1997] and Power PC [Bechade 1994] respectively is consumed by the cache. These figures show that saving cache energy will have a considerable impact on the overall energy consumption.

Several hardware (architecture level) and software techniques have been proposed to reduce the power consumption and improve the performance of memory subsystem. Each of these techniques has its own merits and demerits. The hardware techniques may lead to complex circuit implementations while incorporating a variety of applications. The software techniques tuned for a particular application can hardly be reused for other applications. These issues are very crucial for embedded system design as increase in the cost of hardware pushes the system towards non-application specific designs.

Reduction in cache power consumption can be achieved by reducing the number of cache misses, latency (delay) per access, power consumption per access, and the cache miss penalty, shutting down a part of the cache, reconfiguring the cache for specific applications and various combinations of some of these. Various architecture level techniques described in literature to attain these, include hardware prefetching, vertical cache partitioning, horizontal cache partitioning, reconfiguring cache architecture, optimizing cache control circuitry, modifying the replacement circuitry to improve hit

rate, making use of the compiler and operating system information (software controlled cache) to improve performance and various combinations of some of these.

The widely used mapping schemes available are direct-mapped, set-associative and fully associative. The embedded system requires low power consumption, low access time and high cache hit rate. Hence, a suitable mapping scheme should be selected for the purpose. A direct-mapped cache maps each memory block to a unique cache block, whether or not the cache block is empty. This is the simplest mapping scheme available with minimum access delay and circuit complexity. Cache hit rate of this scheme is low, as compared to other schemes. The fully associative scheme allows a memory block to be mapped to any of the empty cache blocks, if one exists. If there is no empty cache block, a replacement policy is used to select one of the cache blocks for replacement. In this case, number of tag comparisons required for finding the requested cache line is equal to the number of cache lines. The circuit complexity, access delay and dynamic power consumption of this method is very high when compared to that of the direct-mapped cache. This scheme provides the minimum cache misses (number of cache misses is based on the replacement algorithm in use), when compared to all other schemes. The set-associative mapping scheme is a compromise between the direct-mapped and fully associative mapping schemes. A set-associative cache divides the cache into sets and allows a memory block to be mapped to any of the N empty cache blocks within a set. If all the blocks in the set are occupied, then a block is selected for replacement based on the replacement policy in use.

While designing a cache, one has to choose between the direct-map and set-associative mapping schemes as they are the most energy efficient mapping schemes available. Both these schemes have their own merits and demerits in terms of cache access time, dynamic power consumption and cache hit rate. A direct-mapped cache accesses only one tag block and one data block per cache access, where as a N-way set-associative cache accesses N tag blocks and N data blocks per cache access. Literature shows that a direct-map cache consumes much lesser dynamic power per cache access than a set-associative cache. For instance, Hennessy and Patterson [Hennessy 2007] reported 55% more dynamic power consumption per access for a 4-way set-associative cache as compared to that of a direct-map cache. For cache sizes of 8K and 16K, a direct-mapped cache

consumes 74.7% and 68.8% less power respectively than a same sized 8-way set-associative cache. Also, a direct-mapped cache is 29.5% and 19.3% faster than a same sized 8-way cache of size of 8kB and 16kB, respectively. A direct-mapped cache is also simple to design, easy to implement, and accounts for lesser area. But the cache hit rate of a direct-mapped cache is very poor when compared to that of a set-associative cache with the same line size and cache size [Hennessy 2007].

Hennessy and Patterson [Hennessy 2007] shown experimentally that an increase in associativity results in a decrease in the miss rate and hence, reduces power consumption. This shows that a set-associative cache is favorable for applications that require a high cache hit rate and low energy consumption, even though it has an additional overhead of power consumption due to increase in tag comparisons. For example, the average miss rate for the SPEC92 benchmarks is 4.6% for a direct-mapped 8KB cache, 3.8% for two-way 8KB set-associative cache and 2.9% for a 4-way 8KB set-associative cache. Though the miss rate reduction is small, it results in a significant performance improvement which depends heavily on the hit rate and access time, as the large cycle penalty of a cache miss is now avoided. So, if we measure the performance of a cache in terms of the power consumption, the set-associative cache may give better performance than the direct-mapping scheme because energy overhead due to miss penalty is much higher than the per access power. Thus, applications which require a higher cache hit rate prefer a set-associative cache to a direct-mapped cache. The cache power consumption characteristic varies with the total cache size as well. Small cache size is energy efficient and has less access latency but suffers because of poor hit rate. Set-associative mapping scheme also provides support for energy efficient caching schemes like way shutdown, way concatenation, way prediction and process aware caching efficiently. Thus, set-associative mapping scheme is chosen for this work.

2.2.1 REPLACEMENT SCHEMES IN CACHE

The three types of misses incurred in the cache are the compulsory, capacity and conflict misses. A compulsory miss is caused by the first access to a block that has never been in the cache before. A capacity miss happens when the cache is not big enough to accommodate all the blocks needed for program execution. A conflict miss occurs when

multiple blocks map to the same set. This occurs in the direct- mapped and set-associative cache, but not in the fully associative cache. Conflict misses are the major cause of cache misses during program execution. The three important factors affecting cache performance are cache size, cache line (block) size, and cache associativity. Increase in the cache size results in a reduced number of capacity misses but increased data access time because of load capacitance. Increase in the cache line size provides a reduced number of compulsory misses but increased cache miss penalty due to the increase in data transfer. Increase in associativity results in reduction in the number of conflict misses but increases the per cache access time.

The ideal cache is one with the minimum number of capacity, compulsory and conflict misses, minimum per cache access time, minimum cache miss service time, and minimum load capacitance. The conflict miss rate reduction with minimum cache miss service time is very important for a high performance cache with low power consumption. A set-associative mapping scheme offers a good balance of cache hit rate, cache access time, dynamic power consumption and hardware implementation cost. For a set-associative cache, improvement in the cache hit rate and hence, reduction in the dynamic power consumption can be achieved with the help of an efficient replacement algorithm. A replacement policy determines the effectiveness of this set-associative scheme with a proper memory block mapping technique. A replacement strategy is needed when all the cache lines in a set become full and a new block of memory needs to be placed in the cache memory. The cache controller, with the help of a replacement algorithm identifies a cache memory line. Then it replaces the line with a new data from the main memory. The replacement algorithm helps in reducing the number of conflict misses and hence, the power consumption. The performance of a cache replacement mechanism mainly depends on how accurately cache can predict the future reference pattern based on the past references. The future reference pattern may depend on the past reference pattern and input data. It is relatively easy to find the reference pattern in a static scheduled system than in a dynamic-scheduled system. The choice of a replacement policy is one of the most critical cache design issues. Selection of a suitable line/block replacement algorithm, in the case of fully associative and set-associative caches, can have a significant impact on the overall system performance.

The current processors employ various replacement policies such as Random, Round Robin (or FIFO – First-In-First-Out), LFU (Least Frequently Used), LRU (Least Recently Used), PLRU (Pseudo LRU), MRU (Most Recently Used) and variants of these [Smith 1982]. The performance of all these policies are compared and analyzed with reference to the optimal replacement policy (OPT). The OPT decides the cache line to be evicted based on future references. This strategy cannot be implemented in the case of dynamic scheduling systems, as the future cache references are not available [Jeong 1999]. Even if the future references are known, it is impractical to implement this scheme because of the computational complexity involved in finding the cache line to be evicted. However, it is very useful in determining the lower limit for the number of cache misses.

As the optimal cache miss performance can be achieved only by knowing the future references and the future references are unavailable in the case of dynamic scheduling, one has to go for heuristics near the optimal solution. In general, most of the policies anticipate the future memory references by looking at the past behavior of programs (program's memory access patterns). The purpose of a replacement algorithm is to identify a cache line which should be purged in order to make room for the newly referenced cache request that previously experienced a miss in the cache. Relative performance of these algorithms depends mainly on the length of the history consulted.

The heuristic used for finding a cache line to be evicted can be a random pick from the available cache lines, a cache block that arrived first in the cache, the least frequently used cache line, the least recently used cache line, the most recently used cache line or variants of some of these heuristics.

The random replacement (RAND) heuristic chooses a cache line to be evicted randomly with all the available cache lines having an equal probability of being evicted. So in RAND, the cache arbitrarily replaces a block. Though RAND has the minimum implementation complexity among all the cache replacement algorithms, it may increase the cache miss rate. An additional hardware cost required for its implementation is only the random number generator. This scheme doesn't require any storage facility, as the cache line to be evicted is not chosen based on past references.

In the First-In-First-Out (FIFO) replacement scheme, when a replacement is necessary, a cache block that first entered the cache memory is chosen as the block to be evicted i.e., the cache selects a cache block that has been residing in the cache for the longest time. This scheme is one of the simplest replacement policies to implement. The hardware implementation of this scheme requires a register per cache line to store the entry time which is to be compared during a cache miss. One other implementation can be, maintaining a FIFO queue based on the arrival time of the blocks into the cache. When a replacement is necessary, the block at the head of the queue is removed.

The Least Frequently Used (LFU) replacement scheme selects the cache line to be evicted based on the frequency of access of the cache lines. LFU requires maintaining a frequency count register per cache line and is incremented by one, each time a reference is made to the cache line. So a register is updated for every cache access. LFU finds the cache line with the lowest frequency count as the one to be evicted. The LFU cannot differentiate between references that occurred way in the past and the more recent ones. Whenever a new block is copied from main memory, the frequency count of that block is reset to 0. The motivation for using LFU and other frequency-based algorithms is that the frequency count can be used as an estimate of the probability of a block being referenced. Updating a register after every cache access increases the cache access time and thus degrades performance. The hardware implementation complexity of this policy is more as compared to that of the RAND and FIFO.

The LFU policy can suffer from cache pollution if a previously popular cache line becomes unpopular. This cache line then remains in the cache for a long time, preventing other newly or slightly less popular blocks from replacing it. This mainly happens because of temporal locality, especially after the completion of the execution of a loop for large iterations. To reduce cache pollution, the replacement scheme should address not only the access frequency, but also age of the cache line in the cache. One hardware implementation solution is to introduce a reference count for aging. The reference count is incremented dynamically for every cache access. Whenever the reference count exceeds some predetermined maximum value specified by the algorithm, the frequency count of all the cache lines is reduced. This variant of LFU is called Least Frequently Used –Dynamic Aging (LFUDA). In LFUDA, the dynamic aging is accomplished by

shifting the value in each of the frequency count registers by one position to its right (divided by 2), when the reference count reaches its maximum value. This requires an additional reference count register which should be updated for every cache access and which adds an additional overhead to the cache access time and power consumption. LFUDA increases the cache age during the eviction of a block/object by setting it to the evicted object's key value. Thus, the cache age is always less than or equal to the minimum key value in the cache.

The Least Recently Used (LRU) replacement strategy selects the element that has not been accessed for the longest period, for eviction. This results in a higher cache hit rate, at the cost of additional hardware for manipulating / maintaining LRU state information and decision-making. The basic idea is that the blocks that have been referenced in the recent past are likely to be referenced again in the near future, because of the temporal locality of the workload. This policy uses a program's memory access patterns to guess that the block that is least likely to be accessed in the near future is the one that has been accessed least recently. LRU and its variants are the most widely used replacement strategy in the cache because of their high performance [Smith 1982]. There exist various ways to implement LRU in hardware, which include Counter, Square matrix, Skewed matrix, Link list, Phase, and Systolic array method [Sudarshan 2004].

Literature reveals that the LRU strategy performs close to the optimal replacement strategy, when associativity is less. As the associativity grows, the performance considerably degrades [Wong 2000]. In fact, in [Al-Zoubi 2004], it is reported that the optimal performance of a data cache of a certain size is roughly equal to the LRU performance of a cache twice as big, with the same number of ways.

In the case of FIFO and Random, the replacement circuit complexity and the extra hardware requirements are relatively less when compared with that of the LRU and LFU [Deville 1992] [Sukumar 1993]. The implementation complexities of FIFO and Random schemes are relatively low, irrespective of the associativity. In LRU and LFU strategies, information update has to happen for each reference to a cache line. FIFO strategy does it only once when a new page comes into the cache line, whereas in random, modification / update is not required. So the LRU and LFU take more time per access, when compared with that of the FIFO and Random replacements. The main drawback of random and

FIFO replacement strategies is their high cache miss rate in comparison to that of the LRU and LFU. LRU usually follows stack implementation whereas, FIFO follows a queue implementation. The most commonly used hardware implementations for the LRU replacement strategy, are the LRU counter and LRU square matrix. The hardware components used are storage elements for storing the last reference information and associated logic circuitry for decision-making. For a square matrix implementation, the storage elements (D – flip flops) used per set are N^2 whereas for a counter implementation, only $N * \log_2 N$ storage elements are used, where N is the associativity of the cache. The logic circuit required for selecting the cache line to be evicted is very small for the square matrix implementation whereas, it is more complex for the counter implementation. The square matrix implementation is very good for low associativity caches, but as the associativity increases, the storage element requirement increases quadratically which makes it unsuitable for high associativity caches. The counter implementation logic circuit complexity also increases with the associativity, but it provides a lesser overall complexity as the storage element requirement increases only linearly, which makes it suitable for high associative caches when compared to the square matrix implementation [Sudarshan 2004].

According to [Kampe 2004], the LRU is prone to two major types of selection failures. The first selection failure is to keep a cache block that will not be accessed again until it leaves the cache. This happens when many blocks are accessed in the Most Recently Used (MRU) position and are not accessed again there after until the cache line leaves the cache. Most of these cache lines in this category will have only one cache access and thus, should be bypassed or replaced immediately than keeping them in the cache for a long time (called Bypass block). Another occurrence of this type of failure is when cache lines are accessed repeatedly for several times in the MRU position and then, have no access thereafter (called Dead block). These blocks should be replaced as soon as they leave the MRU position in the cache. The second selection failure is to replace a cache block (Live block) that will be referenced in the immediate future. This happens when a just replaced cache from the set is accessed right after the next miss. LRU can improve its performance if the bypass block is not allowed to enter into the cache, the dead block is

replaced at the earliest after leaving the MRU position and the live block is held in the cache for a longer period.

There exist various other situations where the LRU does not perform well. One such situation is in time-shared systems where multiple processes use the same cache and when there is data streaming in applications. The LRU policy often performs poorly for applications in which the cache memory requirements and memory access patterns change during execution. One other drawback for the LRU is that it considers only the time of the most recent reference to each block for eviction and it cannot differentiate between frequently and infrequently referenced blocks.

A variant of LRU replacement policy is Early Eviction LRU (EELRU)[Smaragdakis 1999]. The EELRU dynamically chooses to evict the LRU page or the e^{th} most recently used page. This policy performs LRU replacement by default, but chooses to evict cache lines early when it observes that too many cache lines are being touched in a roughly cyclic pattern which is larger than the main memory. The LRU reference history determines e, the early eviction point, but it is too expensive to store and use in caches. This approach eliminates capacity page misses in a fully associative memory.

A pseudo LRU (PLRU) method finds the cache line to be evicted from a set that was assumed to be the least recently accessed and is overwritten. The results show that the PLRU techniques can approximate and even outperform LRU with much lower complexity, for a wide range of cache organizations.

Some of the modern processors like Itanium have cache hit instruction(s) to improve cache performance [Veidenbaum 1999]. The memory accessing instructions of Itanium can be accompanied by a nt (non-temporal) cache hint. The Itanium-2 implemented a modified LRU replacement algorithm honoring the nt cache hint [Veidenbaum 1999]. In case of execution of a memory instruction with nt cache hint, the replacement algorithm does not change the rank of the touched cache line (in normal case the rank is set as the highest). In this modified LRU replacement mechanism, the data accessed by instructions with nt hint is more likely to be evicted on a subsequent cache miss. By not changing the value of rank in the set, the algorithm retains the old priority of the replaced block and the data without temporal reuse (Bypass block) does not get preference unnecessarily. This is

one way of avoiding the Bypass block to become the MRU and in turn, stay in the cache for a long time. This replacement scheme relies on the compiler to give the nt hint to the instructions accessing data without temporal reuse. This architecture effectively prevents cache pollution and thus has the potential to achieve better cache locality.

Maki et al. [Maki 1999] try to improve the LRU replacement decision with the help of an additional bit (lock/release) per cache line and lock and release operations. This process-aware scheme reported 60.9% reduction in cache miss ratio and faster execution than the LRU replacement strategy.

Wang et al. [Wang 2004] proposed a replacement algorithm which improves the cache hit performance or in the worst case performs similar to LRU for set-associative caches. This work used compiler-generated auxiliary information to improve the cache replacement decisions for scientific programs. The compiler generates special purpose instructions to set the Evict Me (EM) bit and thus explicitly controls cache replacement. One tag bit (MSB) is used to represent the EM bit. The extra hardware used in Evict-Me cache does not increase the cycle time as it is not a part of the critical path. This is effective only in the case of set-associative caches. Results show that the EM bit can reduce miss rates in set-associative caches up to 45% over the LRU. The EM bit is very similar to Alpha's evict instruction [Kessler 1999] except that Alpha's evict instruction evicts a cache line immediately and thus requires high precision, whereas the Evict Me scheme evicts the cache line only when a cache miss happens in the same set. The evict instruction in Alpha processor is designed mainly to help maintain cache coherency, while Evict Me is for enhancing locality.

Wong and Baer [Wong 2000] proposed an enhancement for the LRU replacement policy with a temporal bit per cache line. This temporal bit acts opposite to the EM bit in [Wang 2004], i.e., it specifies the cache lines to be retained in the cache rather than cache lines to be evicted. The temporal bit settings are determined by offline profiling or an online hardware history table. This bit is set when there is a cache hit in that line and is reset when a non-LRU line is evicted from the set.

Rivers et al [Rivers 1998] used a hardware detection unit to dynamically determine the access as temporal/non-temporal and cacheable/non-cacheable. This information can help

the LRU to perform better, as a non-cacheable access bypasses the cache to avoid pollution. Intel IA-64 [Dulong 1998] provides instructions to control caching so that non-temporal accesses will bypass the cache to avoid cache pollution.

Prabhat Jain et al. [Jain 2001] proposed a software-assisted cache replacement scheme to kill/keep the cache line. They proposed a methodology which ensures that cache pollution does not degrade the overall performance when software or hardware prefetching methods are used. This scheme requires additional software-controlled state information that affects the cache replacement decision. It uses software instructions to kill/keep a particular cache line. Kill and keep require one bit of additional state per cache line. They provided conditions under which kill instructions can be inserted into a program code, such that the resulting performance is guaranteed to be as good as or better than the original program execution using standard LRU policy. They combined prefetching and cache replacement to achieve different associated performance guarantees.

Martin Kampe et al. [Kampe 2004] proposed Self-correcting LRU, which is based on LRU augmented with a feedback loop to constantly monitor and correct the mistakes done during replacement. By adopting mechanisms to detect mistakes in each set of the data cache, the proposed scheme could reduce the miss rate by up to 24% for a 4-way set-associative cache.

Praveen Kalla et al. [Kalla 2003] designed a technique (LRU-SEQ) for reducing the transition energy in instruction cache sub-banks by redirecting the sequential cache fills to the last bank accessed. By regrouping sequential accesses, the policy reduces inter-bank transitions and increases the chances that a bank can be shut down for a longer period (thus, reduces leakage energy). This scheme reduces the total energy by 23% on an average.

O'Neil et al. proposed the LRU-2 method [O'Neil 1999] that evicts the memory block with a minimum timestamp of the second to last reference. Wong and Baer [Wong 2000] enhance the LRU with a temporal bit for each cache line. Lai et al. [Lai 2001] use a hardware history table to predict when a cache block is dead and which block to prefetch

and replace the dead one. The drawback of this algorithm is that the size of the history table limits the length of the history consulted.

Hussein Al-Zoubi et al. [Al-Zoubi 2004] evaluated the LRU, PLRU, OPT, and FIFO with the SPEC2000 benchmark suite. The PLRU schemes employ approximations of the LRU mechanism to speed up operations and reduce the complexity of implementations [So 1988]. For the first-level instruction and data caches, the PLRU heuristics are very efficient in approximating the LRU policy. The PLRU techniques are consistently close to the LRU during whole program execution. For the second-level unified cache, the PLRU techniques outperform the LRU for even more cache organizations than those for the first-level caches.

There exist replacement policies which combine the LRU and LFU replacement strategies. Frequency Based Replacement (FBR) [Robinson 1990] is one such hybrid replacement policy. It combines the benefits of both LRU and LFU without the associated drawbacks. In this scheme, the cache lines in a set are ordered based on the LRU value, but the replacement is primarily based on the frequency count. So FBR records the reference count of each cache line to achieve better replacement. The scheme uses a stack for implementation. The FBR is achieved by dividing the stack (cache) into three partitions: a new partition, a middle partition and an old partition based on the reference recency. The old partition contains LRU cache lines; the new partition contains MRU cache lines and the middle partition contains all the cache lines which are neither old nor new. If the reference is to a cache line in the new partition, then the reference count of that cache line is not incremented. This is to ensure that multiple references to the cache lines in the new partition in a short period of time do not promote them more than required. If the reference is to a cache line in the old or the middle partitions, then the reference count of that cache line is incremented by one. The FBR replacement policy identifies a cache line with the lowest reference count from the old partition for eviction.

In [Lee 2001a], Lee et al. shows that there exists a spectrum of block replacement policies that subsumes both the LRU and LFU policies. The spectrum is formed according to how much more weight is given to the recent history over the older history and is referred to as the Least Recently/Frequently Used (LRFU) policy. LRFU increases the weight of a memory block by one when it is referenced, and decays the weights of all

memory blocks according to their backward distance. LRFU policy uses the complete reference history by using only a few words for each block. It associates each block with a Combined Recency and Frequency (CRF) value. During a cache miss, the block with the smallest CRF is selected for eviction. In this scheme, when the k value becomes 0, then it performs similar to the LFU and when the k value is 1, it matches the LRU. Though LRFU replacement scheme performs better in many cases in comparison with the LRU and LFU, it suffers because of its complex hardware implementation, large time complexity (the time complexity of LRFU varies from O(1) to ($\log_2 n$) where as LRU time complexity is O(1)), difficulty in tuning accurately for each system and workload for maximum performance and difficulty in predetermining the exact optimal scale.

Yannis Smaragdakis [Smaragdakis 2004] shows how to combine any two existing replacement policies so that the resulting policy can provably, never perform worse than either of the original policies by more than a small factor. This policy performs very well with real program data, often outperforming LRU (as well as all the other policies it adopts) by more than 40%. Jaafar Alghazo et al. [Alghazo 2004] proposed SF-LRU (Second Chance-Frequency - Least Recently Used) that combines LRU and LFU (Least Frequently Used) using the second chance concept. Experimental results show that the SF-LRU significantly reduces the number of cache misses when compared with the LRU (upto 6.3%) and LFU (upto 9.3%). S-LRU [Kampe 2004] has been proposed to try to partially take account of the frequencies while making the LRU decisions and to keep the overhead low.

Inter-reference Gap Distribution Replacement (IGDR) is based on a reference model and adapts to reference patterns that prefer LRU or LFU. It attaches a weight to each memory block, and selects the smallest weight block for replacement. The difference in time between successive references of a memory block is called Inter-Reference Gap (IRG). This scheme claims achieving cache miss reduction upto 46.1% (19.8% average) and upto 48.9% (12.9% average) speed up over the LRU scheme when working with SPEC2000 benchmarks.

Most Recently Used (MRU) policy selects the most recently used cache line from a set for eviction. This algorithm is not widely used in the cache memory system because of its bad temporal locality. The Priority Cache (PC) [Aguilar 2004] selects a cache line from a

set, based on the runtime and compile-time information, for eviction. PC associates a data priority bit with each cache block. The compiler, through two additional bits associated with each memory access instruction, assigns priorities. These two bits indicate whether the data priority bit should be set as good as the priority of the block, i.e. low or high. The cache block with the lowest priority is the one to be replaced.

In addition to all these replacement policies, there exist various replacement strategies which are very specific to architectures like victim cache [Jouppi 1990], skewed-associative cache [Seznec 1993], elbow cache [Spjuth 2004] etc. Comparison of the widely used replacement algorithms is in Annexure A.

2.2.2 ENERGY EFFICIENT CACHE ARCHITECTURES

Cache memory power analysis reveals that the data lines and data sense amplifiers are the main sources of power consumption [Wilton 1996]. Wilton and Jouppi [Wilton 1996] reported power consumption by the data lines and data sense amplifiers as 55%, 65% and 75% of the total cache sub system power consumption for the direct-mapped, 2-way set-associative and 4-way set-associative mapping schemes respectively. One way to minimize the dynamic power consumption is to minimize the internal activity of the cache during a cache access. Minimum cache power consumption can be achieved if the cache incurs minimum conflict misses. Also, if each cache hit results in reading and comparing only one tag entry, enabling and accessing only that one data entry and if each cache miss results in only reading and comparing one tag entry.

Hardware prefetching [Chen 1995] is a popular technique for enhancing the cache performance in conventional systems. The prefetching techniques try to reduce the cache miss rate by prefetching instructions into the internal cache. This may result in replacing useful data in the cache [Gupta 1990]. Unfortunately, most of the existing prefetch techniques are not very effective in embedded systems because of real time processing constraints and react to stochastic execution flow.

The real-time / embedded systems employ various partitioning schemes to make cache - energy efficient and deterministic. This ensures the smooth execution of higher priority time-critical tasks. A cache partitioning can be either static (fixed) or dynamic. A fixed partitioning scheme partitions the entire cache into N equal/ unequal sizes and assigns

them to the tasks. In case of dynamic partitioning scheme, the cache is partitioned based on various parameters such as the size of the task, priority of the task, number of cache blocks in use etc.

Another way of partitioning the cache is vertical and horizontal partitioning. The vertical cache partitioning scheme [Su 1995] aims at optimizing the capacitance of each cache access by introducing a small cache between the CPU and the main cache. Accessing a smaller cache needs less power because of low load capacitance. Buffering cache [Bunda 1994], filter cache [Kin 1997] and loop cache [Bajwa 1997] are some of the ways of realizing the vertical cache partitioning scheme. These schemes result in reducing the power consumption, but the amount of power saved depends on the spatial locality of the applications and the cache line sizes.

The buffer cache [Bunda 1994] is closer to the processor than the conventional L1 cache. In this scheme, the processor checks the availability of data in the block buffer. If it is a hit, the data is directly read from the block buffer and the cache is not operated. The cache is operated only if there is a block miss. The effectiveness of block buffering strongly depends on the spatial locality of applications and the block sizes. The higher the spatial locality of the access patterns (e.g. an instruction sequence), the larger the amount of energy which can be saved by block buffering. The block size is also very important in block buffering.

Bellas et al. [Bellas 1999] proposed a L0 cache to store the most frequently executed portions of a code. A L0 cache resides between a L1 cache and the processor and is directly accessed by the processor. The selection of the most frequently executed code to accommodate in a L0 cache is dynamic. Bajwa et al. proposed a loop cache [Bajwa 1997]. [Kin 1997] used a loop cache to store the most frequently executing instructions and uses it as a L0 cache. Vahid and Cotterell [Collerell 2002a] [Collerell 2002b] described the use of a loop cache in embedded system. The content of the loop cache is loaded dynamically during program execution and used as a static memory. Given their very small sizes (128–256 bytes), loop caches negatively affect the miss rate, but decrease the overall energy. When compared with pre-decoded instruction buffers, loop caches are less energy efficient for programs with very high locality, but they are more

flexible. The loop cache has a greater area in comparison to a same-sized buffer cache, because of the tag memory and tag match circuit.

Kin et al. [Kin 1997] proposed the filter cache, a small L1 cache which reduces power consumption drastically with small performance degradation. In this scheme a small L2 cache which is similar to the conventional L1 cache is placed behind the filter cache to reduce the cache misses and this explains the performance loss. A typical 256B direct-mapped filter cache achieves 58% power reduction at the cost of 21% performance degradation.

A HotSpot cache is used to reduce dynamic power consumption for both instruction and data caches [Yang 2004]. It adds a small cache between the CPU and L1 instruction cache. It identifies the frequently accessed instructions dynamically and stores them in a L0 cache. Yang and Lee [Yang 2004] designed a mechanism that can successfully identify frequently accessed basic blocks in each program phase at runtime. Only basic blocks declared as hot blocks are stored in the L0 cache. The L1 cache is augmented with a block buffer for exploiting spatial locality within a cache line for further energy savings.

Victim caches [Jouppi 1990] are small, fully-associative buffers that provide limited additional associativity for heavily utilized entries of a direct-mapped cache. The victim cache can be accessed in parallel with the main cache (L1) or in the next cycle if and only if the cache access in the main cache is a miss. The parallel access does not increase the effective access time much, due to its small size. Accessing the victim cache after detecting a cache miss in the main cache results in, wasting one cycle in case of a victim cache hit or cache miss. In this scheme, the victim cache normally employs FIFO replacement policy. If a match is found in the victim, the cache line in the victim is swapped with the main cache's cache line in the specific location. The victim cache improves the cache performance, but for the average cache access time. Stiliadis and Varma [Stilliadis 1997] proposed and evaluated an improvement of this scheme, called the selective victim caching for improving the cache hit rate without affecting the access time. In this scheme, incoming blocks into the first level cache are placed selectively in the main cache or a small victim cache by the use of a prediction scheme which is based on past history. If the probability of a new block to be placed in the cache is less than that

of the existing block in the cache then the new block is stored in the victim cache. Otherwise, the new block is stored in the main cache and the existing block in that location is moved to the victim cache. In addition, block exchange between the main cache and the victim cache are also performed selectively.

The horizontal partitioning scheme [Su 1995] partitions the cache memory into various segments. For instance, a 4-way set-associative cache can be partitioned into 4 segments (also called sub-banks) where each cache way is in one segment. Each segment can be accessed, shutdown, or activated (power up) individually. So if one can predict the segment in which the accessed data is available, a majority of the power consumption can be saved by powering down the remaining segments and thus, eliminating unnecessary accesses. Power saving in the horizontal partitioning scheme depends on the number of segments one can turn off while accessing the data. There exist various ways to accomplish the horizontal partitioning scheme. Cache sub-banking proposed in [Su 1995], is one of the horizontal cache partition techniques which partition the data array of a cache into several banks called sub-banks. Each sub-bank can be accessed and managed individually. All the (N-1) sub-banks can be in the power down state while accessing data from a sub-bank. The amount of power saving and the cache hit performance depends on the number of sub-banks. Some of the other horizontal partitioning schemes are way shutdown, way prediction, phase lookup, and various combinations of some of these. One advantage of the horizontal partitioning scheme over the vertical partitioning scheme is the effective cache hit time of a horizontal partitioning cache. The effective cache hit time of a horizontal partitioning cache can be as fast as the conventional cache as the logic circuit that decides which cache bank to select is simple.

The key to energy reduction is to pinpoint the matching way without probing all of the ways. One option to avoid high-energy dissipation at the cost of slower access is by using sequential access, employed in Alpha 21164's L2 cache [Bannon 1995]. In sequential access, the cache waits until the tag array determines the matching way, and *then* accesses only the matching way of the data array, dissipating about 75% less energy than a parallel access cache. Sequential access, however, serializes the tag and data arrays, adding as much as 60% to the cache access time [Powell 2001]. One can implement the horizontal partitioning scheme (sequential access) in a set-associative cache by accessing

it like a direct-mapped cache. This is achieved by a phased lookup set-associative cache [Calder 1996] [Hasegawa 1995] [Lyon 2002]. This cache accesses the tag arrays in the first phase, and in the second phase, it accesses only the data array corresponding to the matching tag, if there exists any. This can eliminate most of the unnecessary activities in data array as the complete data array is in the power down state during the first cycle and in the second cycle, the segment (cache way) which holds the valid data alone is powered on, in case of a cache hit. This scheme accesses at most one data array at the cost of performance overhead due to one extra phase and hence results in a longer cache access time, though it saves energy.

Juan et al. [Juan 1996] proposed the difference-bit cache which is a 2-way set-associative cache. This design is to achieve access time close to that of a direct-mapped cache of the same capacity and line size. This is achieved by a single tag bit and works based on the fact that two tags of a set have to differ by at least one bit. This is achieved by separating the selection of a proper way from the detection of a hit, and selecting the way using the least-significant bit in which both tags of a set differ. The decision of a cache hit in the difference-bit cache is of one cycle. Zhou and Petrov [Zhou 2006] used the application knowledge regarding the nature of memory references to eliminate the tag address translations for most of the cache accesses in a virtual memory system. Application knowledge regarding the nature of the data memory references is used to distinguish references as private data and, consequently, handle them in a more energy-efficient way. In this system, both virtual and physical tags co-exist and for private data, address translation is avoided by directly using the virtual tag.

A skewed associative cache was first introduced by Seznec et al. [Seznec 1993]. A skewed cache is conceptually divided into multiple sub-banks, each indexed by a different hash function. For a skewed 2-way set-associative cache, cache blocks that map to the same location in one of the banks are likely to map to different locations in the other. The skewed-associative cache [Seznec 1993] is an alternative option. A two-way set-associative cache can achieve the equivalent hit performance of a 4-way cache by employing different mapping functions for each way. One of the challenges with skewed caches is the replacement algorithm. Since there are no fixed sets, any combination of victim pairs, one from each bank is possible. This makes it difficult to implement an

exact ordering-based replacement algorithm like the LRU. To solve this issue, Not Recently Used, Enhanced (NRUE) [Seznec 1993] replacement scheme has been proposed which is the best performing replacement algorithm for skewed caches.

[Spjuth 2004] proposed an elbow cache which extends the skewed cache organization. It adopts a relocation strategy for conflicting blocks. An elbow cache relocates the conflicting blocks to their alternate locations. This reduces the conflict problems while consuming less dynamic power. An elbow cache reduces the miss rate at the cost of complexity.

Another way of accessing the set-associative cache as a direct mapped cache is the pseudo set-associative cache [Huang 2001][Yongioon 1999] [Agarwal 1993]. Pseudo Set-Associative Caches are set-associative caches with multiple hit times. It has one tag array and one data array like a direct-mapped cache. On a miss, an index bit is flipped and a second cache entry is checked for a hit—the first and second locations thus form a pseudo-set. Here, dynamic power is reduced at the expense of performance. This scheme requires extra time whenever prediction results in a miss. Panwar and Rennels [Panwar 1995] proposed a method to skip tag comparisons when accessing the last accessed cache line again.

Like the Pseudo-associative cache, a Hash-Rehash cache [Agarwal 1988] was proposed to reduce the miss rate of a direct-mapped cache. It is used to reduce the probability of thrashing in a cache by providing multiple locations to store data in the cache. When a memory reference is presented to the cache, the direct-mapped location is checked. If there is a miss, a hash function is used to index the next cache entry. Like in a pseudo set-associative cache, the most-recently-accessed cache line will be moved to the direct-mapped location. However, exchanging large cache lines consumes large amount of power, cycle time and bus bandwidth. The main drawback of a hash-rehash cache is that every miss at the first cache address results in a second cache lookup at a rehash address. The cache miss (miss in both the first and second cache addresses) in the hash-rehash cache might replace useful data at the hashed location which results in secondary thrashing. The secondary thrashing affects the performance of this scheme which may even degrade it to below that of a direct-mapped cache.

A column-associative cache [Agarwal 1993] improves upon the hash-rehash by inhibiting a rehash access if the location reached by the first time access itself contains data written by a hashed address. It employs two different mapping schemes. The first is used when a cache access is issued, whereas the second is applied only in the case of a miss in the first attempt. This scheme has an additional bit called a rehash bit per cache entry. This bit indicates whether the data in the entry was written using a hashing function or not. In practice, this cache behaves like a 2-way set-associative cache with sequential search and uses the LRU information to guide both the replacement policy and the search order. Even in the column-associative cache, thrashing may still occur if data from 3 memory locations is frequently used and is stored using a common cache address. If the cache is off-chip, then the swapping of data between cache locations is impractical because of the latency associated with the off-chip cache.

Hallnor et al. [Hallnor 2004] proposed Indirect Index Cache (IIC) as a mechanism to achieve full-associativity through software management. The IIC serializes tag comparison and data lookup by storing a forward pointer in the tag store to identify the corresponding data line. A cache access in the IIC is performed using a structure similar to a hash table with chaining. If a matching tag is not found in the set-associative tag-store, a pointer associated with the set is used as a direct-mapped index into a collision table. Each entry in this second table provides a pointer to the next member of the collision chain. The chain is traversed until either a match is found or the maximum chain length is reached. This scheme requires collision chain traversal, resulting in variable hit latency and port contention. This scheme also requires swapping of tag entries to reduce the average hit latency to a reasonable value. IIC also requires software management of the replacement algorithm.

Patel et al. [Patel 2006] proposed an improved indexing scheme for direct-mapped caches, which reduces the number of conflict misses by using application-specific information. This indexing scheme is based on the selection of a subset of the address bits. The selection is based on the knowledge of the specific addresses used to access the cache.

The half-and-half cache [Theobald 1993], reserves half of the cache lines for direct-mapped access and the other half for associative accesses. This is to exploit the advantages of both direct-mapping and associativity.

Dropsho [Dropsho 2002] discussed an accounting cache architecture that is based on the resizable selective ways cache proposed by Albonesi [Albonesi 1999]. The accounting cache first accesses a part of the ways of a set-associative cache, known as a primary access. If there is a miss, then the cache accesses the other ways, known as secondary access. A swap between the primary and secondary accesses is needed when there is a miss in the primary and a hit in the secondary access. Energy is saved on a hit during the primary access but secondary access consumes large amount of power, and cycle time.

Zhang [Zhang 2006] proposed the balanced cache (B-cache) to reduce the miss rate of direct-mapped caches by balancing the accesses to cache sets. This is achieved by increasing the decoder length using programmable decoders, and this reduces the accesses to heavily used sets without dynamically detecting the cache set usage information. They introduced a replacement policy for B-cache which reduces miss rates significantly. The B-Cache consumes more power per cache access but exhibits less total memory access related energy saving due to the miss rate reductions.

Benini et al. [Benini 2000] described an application-driven partitioning of the on-chip SRAM, based on recursive formulation. This design consists of independently accessible banks. Gonzalez et al [Gonzalez 1996] proposed a logical partition for the on-chip cache into the spatial and temporal cache based on the spatial and temporal data. This approach relies on the dynamic prediction mechanism for the spatial and temporal data which uses a prediction buffer. Park et al. [Park 2007] presented a dual data cache system structure, called a *co-operative cache system*. This design consists of two caches with different associativity and line sizes, i.e., a direct-mapped *temporal-oriented cache* (*TOC*) with an 8 byte line size and a 4-way set-associative *spatial oriented cache* (*SOC*) with a 32 byte line size. They used 8KB TOC and SOC in their design. For a cache read, by default, the cache probes a TOC block in the first cycle . In case of miss in the TOC block when the data block is available in the SOC block, access the data from the SOC and the corresponding TOC block is copied in to the TOC as well. Whenever a cache miss occurs, the corresponding SOC block is fetched from the main memory to the SOC and

the corresponding TOC block is copied into the TOC as well. For a cache write, if the content is available in the TOC, update it directly there. If there is a miss in the TOC but a hit in the SOC, then the corresponding TOC block is copied from the SOC and modified. The data in the SOC is not modified because the TOC has a higher priority than the SOC for a cache read. The cache write into a SOC can be delayed until the replacement the TOC block.

The Most Recently Used (MRU) cache design [Chang 1987] maintains the MRU information associated with each set. When searching for data, the block indicated by the MRU bits is probed. However, the MRU bits must be fetched prior to accessing the cache. The PSA (Predictive Sequential Associative) cache design [Calder 1996] moves the prediction procedure to previous stages of the pipeline so that the MRU information is presented to the cache simultaneously along with the memory reference. The cache controller attempts to make a prediction speculatively of the way where the required data may be located. If the prediction is correct, the cache access latency and the power consumption are similar to those of a direct-mapped cache of the same size.

Inoue et al. [Inoue 1999] [Inoue 2001] proposed the Way prediction set-associative cache scheme. Way-prediction cache [Inoue 1999] speculatively chooses one cache way before the cache line access in a set-associative cache. This scheme reduces the number of tag comparisons by first accessing only the tag array and data array of one way (segment) that is predicted in the first cycle. All the (N-1) data and tag segments are in the power down state during this cycle. If a misprediction occurs, then the remaining (N – 1) ways are accessed in the following cycle. Inoue et al. used $\log_2$ (N) bits per set to maintain the MRU way information which is used for predicting the way. MRU bits of each set have the information of the recently accessed cache line's way in that set. If the prediction is correct, the cache consumes the energy required for only one activated way. Otherwise, the cache searches all of the ways and consumes energy required for all of them. This method saves almost (100*(N – 1) / N) % of the energy in an N – way set-associative cache. Prediction accuracy according to Powell et al. [Powell 2001] is 90% for instruction and 80% for data.

Though the way-prediction scheme is very effective in saving power, it suffers from serious drawbacks, which significantly limit its usage. This scheme suffers performance degradation because of the cycle time penalty for handling mispredictions. In the way prediction scheme a table lookup is needed to identify the MRU information of the selected set. This adds extra time delay to the critical path as one cannot prefetch the MRU information until the set number is available. One can overcome this using information available either early in the pipeline, such as the program counter (PC), or later in the pipeline, such as an XOR-based approximation of the load address [Powell 2001]. Unfortunately, both choices have problems. Way prediction based on information from early pipeline stages suffers from poor accuracy, and way prediction based on late pipeline information introduces a way prediction table lookup delay in the cache access critical path [Batson 2001]. For instance, the way prediction scheme used in [Inoue 1999] inserts a table lookup after the address generation to identify the predicted way. Another drawback of the way prediction scheme is that the MRU information does not always work well with data references [Calder 1996][Batson 2001] [Min 2004].

Powell et al. [Powell 2001] combined way prediction and selective direct-mapping to reduce the L1 cache dynamic energy without performance degradation. This scheme predicts the matching way and searches only in that predicted way, thus saving energy.

Albert Ma et al. [Ma 2001] proposed the way memorization cache to reduce fetch energy in instruction caches. The way memorization cache stores way information (link) within the instruction cache. It also maintains a valid bit per link to guarantee that the way link is valid. In the way prediction scheme, reading at least one tag is compulsory for verifying the prediction correctness. Way memorization requires a link invalidation mechanism to maintain the coherence of link information.

Batson and Vijay Kumar [Batson 2001] proposed the Reactive Associative Cache (RAC) which uses both the way prediction and selective direct-mapping schemes. The RAC provides flexible associativity by placing most blocks in the direct-mapped positions and reactively displacing only conflicting blocks to set-associative positions. To achieve direct-mapped hit times, the RAC organizes the data array like a direct-mapped cache, and the tag array like a set-associative cache. The RAC uses way prediction with feedback for high prediction accuracy. The RAC uses a PC scheme for implementing

way prediction. The RAC hit time is within 1% of the direct-mapped cache and is 25% faster than a 2-way set-associative cache.

Zhang et al. [Zhang 2005] proposed a way-halting cache, which reduces the energy without performance degradation. This has been achieved with the help of a fully associative array called halt tag array. The halt tag array predetermines which tags cannot match due to the mismatch of their lower order 4 bits and halts access to the ways with known mismatch tags, thus saving power. In this method, some energy is wasted in the parallel comparison of low-order 4 bit tags in halt tag array. Here, in worst case, the saving achieved is less.

Energy efficiency can also be achieved by reducing the number of tags for comparisons. Efthymiou and Garside [Efthymiou 2002], Juan et al. [Juan 1996], Min et al. [Min 2004] focused on reducing the number of tag bit comparisons (partial tag matching) to save energy and access time. This method is application-specific and it reduces the energy, access time and traffic for a specific set of applications, though not for all. Inoue et al. [Inoue 2002] proposed a history-based tag-comparison scheme (HBTC) for reducing the energy consumption of direct-mapped instruction caches. HBTC eliminates unnecessary tag checks at runtime by efficiently exploiting the program execution footprints recorded in the Branch Target Buffer (BTB) contents. Zhang et al. [Zhang 2003] proposed a way concatenation cache which is a set-associative cache whose ways can be logically concatenated to result in a 4-way, 2-way, or direct mapped cache all of the same total size. Albonesi [Albonesi 1999] proposed the way shutdown cache for reducing the dynamic power consumption. In this scheme, a simple logic circuit is used to shut down the cache ways. Zhang et al. [Zhang 2003] proposed the Way shutdown cache which increases the miss rate but saves static / leakage power. In this scheme all the unused ways are put into the shutdown state by using circuit level technique proposed by Powell et al [Powell 2000]. Zhang et al. [Zhang 2003] also proposed cache line size configurable cache (16, 32, 64 B) by using a small register. Here, a base 16B line size is used and the larger sizes are implemented by concatenating multiple physical lines. Zhang [Zhang 2007] extended the traditional configurable cache and made the whole on-chip cache memory capacity available to both instruction and data caches. The capacity can then be co-allocated between the data and the instruction caches. When compared with the way

shutdown and way concatenation caches, the capacity co-allocation cache provides a better solution than increasing the associativity.

Ishihara and Fallah [Ishihara 2005] proposed a non-uniform cache architecture. This work uses an algorithm for simultaneous cache configuration optimization and code placement. The non-uniform cache architecture allows different associativity values for different cache sets. An algorithm determines the optimum number of cache ways for each cache set and generates a object code suitable for the non-uniform cache memory. The paper also proposes a compiler technique for reducing redundant cache way accesses and cache-tag accesses. Aly et al. [Aly 2003] proposed a variable way set-associative cache to reduce the power consumption without degrading the performance. Static profiling is used to determine the sets' behavior in a set-associative cache. Each cache set in this design has a different associativity. Qureshi et al. [Qureshi 2005] proposed the *V-Way Cache*, which allows the associativity to vary on a per-set basis by increasing the number of tag-store entries relative to the number of data lines. This scheme uses *Reuse Replacement*, a global replacement policy based on frequency information. The proposed replacement policy selects a victim within five cycles for 99.3% of the evictions.

Chang et al. [Chang 2004] proposed a value conscious (VC) cache to reduce average power consumption during a cache access. This is based on the observation that the majority of the cache access bits are ‘0’. In VC cache power dissipation for accessing a 0 is much less than that for accessing a 1. VC cache achieves this by preventing bitlines from being discharged while accessing 0. The VC cache is a circuit-level technique which saves more power, if the data contains more bits with 0 values.

One of the major concerns in conventional memory architecture is that the cache is transparent to the operating system and application programs (software). The transparency of the cache cause unnecessary power consumption. Recent studies suggest the necessity of cache – compiler – operating system – application program interaction to improve the cache performance. The interaction can reduce the cache power consumption by accurately predicting the cache set where the required data is available, thus allowing the other ways to sleep [Yang 2005]. The interaction can also improve cache predictability and performance by helping in the selection of a victim cache line with minimum modification in the replacement circuitry [Jain 2001][Wang 2002][Sartor

2005]. Yang et al. [Yang 2005] proposed a software-controlled cache that allows application programs to control data allocation on the cache. The mapping between data types and cache regions is determined statically based on the programmer's knowledge of the application behavior and offline profiling information gathered using a cache simulator. Jain et al. [Jain 2001] uses application-specific information about future variable accesses from the program analysis (trace analysis) for replacement decisions, i.e. to keep or evict the cache lines. The compiler can provide information about the spatial and temporal locality of the loops [Wang 2002] in application programs which will helps the cache controller to decide on the cache line to be evicted. Wang et al. [Wang 2002] uses the compiler to find the array elements in a loop that will not be reused again soon and use them for eviction. Sartor et al. [Sartor 2005] uses compiler locality hints to keep or evict a cache line. In all the above described software-controlled cache schemes the information is passed on to the hardware through modified instructions. Some of the existing microprocessors have instructions that can manage the cache by either flushing the entire cache or cleaning a given cache line. This gives these processors the ability to limit cache pollution [Gupta 1990]. One such example is the Compaq Alpha 21264 [Kessler 1998] where the load/store instructions minimize cache pollution by invalidating a cache line after it is used. The microprocessor can prefetch a line or zero out a given line [May 1994] [SunMicrosystems 1997] by using these instructions. Few other processors permit cache line locking within the cache based on the frequency of usage of the elements in the cache line; mainly for removing those cache lines as candidates to be replaced [Cyrix 1998] [Cyrix 1999].

2.3 OPERATING SYSTEM LEVEL ENERGY CONSUMPTION

At the operating system level, there have been three primary approaches to address the energy consumption problem: process scheduling techniques [Smith 1982], paging systems [Leback 2000], and performance tuning [Acquaviva 2003].

Task Scheduling in real-time systems is a well understood and widely studied issue in literature. The primary focus of most real-time task scheduling algorithms is to generate a feasible schedule i.e. a schedule which ensures that no job misses its deadline. In some real-time systems, additional constraints other than feasibility may also apply. For

instance, in an embedded system, availability of power, size of the memory and speed of the processor may, among others, affect the scheduling policies and algorithms ([Kandemir 2003][Dudani 2002] [Pillai 2001] [Pouwelse 2000]).

Task scheduling algorithms may be online or offline. In online scheduling, the scheduling algorithm competes for the processor time along with the tasks being scheduled and is dynamic. In offline scheduling, all tasks-related information required for scheduling such as arrival times, periods, worst case execution times and deadlines are available with the scheduling algorithm well in advance [Liu 2000]. The offline scheduling usually takes place external to the executing environment and is static.

One of the most commonly used offline real-time scheduling algorithm is the clock-driven scheduling algorithm [Liu 2000]. In clock-driven scheduling, a schedule which satisfies all the task deadlines by taking the worst case execution time into consideration is found and is fed into the system in the form of a table. The resultant schedule in the form of a table consists of job / task identifiers and their activation times. The system then generates timer interrupts to schedule the job(s) / task(s) according to the schedule. This clock-driven scheduling algorithm is a static algorithm, i.e., if a job finishes its execution before the worst case execution time and although k other jobs are available for execution in the system, the clock-driven scheduler will allot the next job to the CPU only based on the static schedule table entry. Though this scheduling scheme gives the minimum scheduler overhead, the system throughput and waiting time performance degrade because of the idle time in the middle of the schedule (which results in the underutilization of the system). This scheduling algorithm is useful for resource constrained real-time systems whose tasks execute till the worst case execution time for almost all cases. The efficiency of the schedule in this case depends on the schedule provided by the external environment. The scheduler overhead in clock-driven scheduling is only the time taken by a timer interrupt, which is very small (with O(1) complexity) when compared with any dynamic scheduling algorithm. Moreover, this scheduler is forced to modify and reload a new schedule (static schedule table) every time a new change in the execution time of a task occurs, or a new task is added. Thus, this makes the scheduler static.

On the contrary, online scheduling algorithms are capable of taking decisions on-the-fly. Here, the scheduler also needs to compete with other jobs for the CPU time. This dynamic scheduler does not leave the CPU idle if any job is ready for execution and also grabs some CPU time for its execution too. Almost all online scheduling algorithms are priority-based scheduling algorithms.

2.3.1 PRIORITY-BASED DYNAMIC SCHEDULING ALGORITHMS

The priority-based scheduling algorithms are further categorized into task-level fixed priority scheduling algorithms, task-level varying but job-level fixed priority scheduling algorithms and job-level varying priority algorithms. Liu and Layland [Liu 1973] in their seminal work, proposed the Rate Monotonic (RM) scheduling algorithm for periodic tasks. The RM is a task-level fixed priority scheduling algorithm where the priority of a task is inversely proportional to its period. Another most commonly implemented and analyzed algorithm is the Earliest Deadline First (EDF) [Liu 2000] algorithm. The EDF bases the priority of a job on its deadline. As the priority is fixed for a specific job but varies among multiple instances of the same task, EDF is a task-level varying but job-level fixed priority scheduling algorithm. Least Laxity First (LLF) [Liu 2000] algorithm is a job-level varying priority scheduling algorithm, where the priority of a job at time t is inversely proportional to the slack available to that job. The slack of a job is defined as the difference between the total time available until the job's deadline and the remaining execution time of the job in the CPU, i.e., deadline of the job J_i – current time t – remaining execution time of J_i.

More recently, the proliferation of mobile embedded devices running on limited battery power has brought in focus the issue of power-aware computing – in particular, power-aware scheduling. The above mentioned conventional priority-based scheduling algorithms like RM, EDF and LLF are oriented towards generating a feasible schedule, rather than an energy efficient schedule. The next section discusses energy efficient scheduling algorithms.

2.3.2 ENERGY EFFICIENT SCHEDULING ALGORITHMS

The energy consumption of a system with multiple tasks includes the energy consumption by the CPU and memory for executing the tasks, energy consumption by the

scheduling process, memory management unit and other operating system functionalities, and the energy consumption because of the unproductive but unavoidable activities like context switches, frequency / voltage mode transfer etc. An ideal schedule is the one which results in optimal power consumption by the CPU, i.e., running the CPU at an optimal frequency and voltage level for the minimum time duration, optimal number of memory references, and optimal use of the operating system functionalities like scheduling, memory management, zero context switches and frequency / voltage mode transfer.

The factors affecting power consumption can be classified as platform-dependent and platform-independent. The platform-dependent factors are those which require special architecture level support for energy reduction. Some of the most important platform-dependent factors affecting the power consumption caused by a schedule are the clock frequency and the applied voltage of the CPU. These factors are platform-dependent because the CPU should support the multiple operating frequencies and voltage levels for making it work, which is not common in usual scenarios. The platform-independent factors affecting power consumption include context switching, process idle time and caching impact. The following sections explain in detail about the existing scheduling algorithms which were designed with the aim of reducing the platform-dependent and platform-independent power consumptions.

2.3.2.1 Platform-dependent Energy Efficiency

A hard real-time scheduler takes the worst-case execution time into consideration for determining the schedule. This is to ensure that all the hard real-time tasks meet their respective deadlines. The actual execution time of the tasks is usually just a fraction of the worst-case execution time which results in the CPU being idle for a long time. Thus, to reduce the power consumption of the CPU, the literature adopts two different strategies. The simplest one is to keep the CPU in power down state when no task is available for execution. This method is advantageous if the system can allow the CPU to be in power down mode for a long time as the transfer between power down mode and active mode takes some time and so, powering down of the CPU should not affect the execution of the other tasks in the system. The alternate approach to reduce power

consumption is to reduce the CMOS circuitry power dissipation. Power dissipation in CMOS technology $Pd = C_{eff} * V_{dd}^2 * f$, where C_{eff} is the effective switching capacitance, V_{dd} is the supply voltage and f is the frequency of the clock [Aydin 2004]. The above formula suggests that one can achieve reduced power consumption if the supplied voltage, effective switching capacitance and the frequency of the clock can be reduced. This is possible only if the CPU supports multiple supply voltage levels and frequencies, which makes it platform-dependent.

For the processors which support multiple voltage and frequency levels, reduction in CPU power consumption is possible by reducing the supply voltage and clock frequency, while still meeting all the task deadlines. This technique is called Voltage Scaling / Frequency scaling. The voltage / frequency scaling can be static (SVS/SFS) or dynamic (DVS/DFS).

In SVS / SFS, if the current process utilization is less than 100% even for the worst-case, then the applied voltage and frequency is scaled down to make the resultant utilization 100%. This works under the principle that if the scheduling algorithm can produce a valid schedule (if there exists one) when the utilization of the processor is less than or equal to 100%, then the applied voltage and frequency of the CMOS technology can be scaled down without any job deadline misses. In other words, one can scale up the utilization of the processor (CPU) by making it work for a longer time with lesser applied voltage and frequency and thus save dynamic power. If the current utilization of the system is 60%, by reducing the voltage and frequency (thus increasing the execution time of the task), statically, the utilization of the processor can be scaled up to 100% without any of the tasks missing its deadline. In this case the system continues to work with the worst-case execution time with respect to the new selected frequency and supply voltage. This requires processors with multiple voltage level and frequency support.

Analysis reveals that the worst-case execution time is a never / rarely occurring case, as most of the jobs finish their execution well in advance. This leaves the CPU (processor) idle for a long period, thus resulting in power consumption without any productive work being done. One way to tackle this issue is to power down the CPU whenever there is no job readily available for execution. The CPU consumes some time and power to switch

between the power-down and active state which makes this option ineffective, when the power down time is small and distributed.

One other way of saving power consumption is by adjusting the supply voltage and frequency dynamically. This is technique is called Dynamic Voltage Scaling / Dynamic Frequency Scaling (DVS /DFS) [Gruian 2001] [Krishna 2000] [Pering 1998]. DVS/DFS reduces the applied voltage and frequency and consequently, increases the execution time of tasks, while ensuring that none of the task deadlines is missed.

Pillai et al. [Pillai 2001] proposed various DVS algorithms for real-time systems (RT-DVS) which include cycle conserving DVS for the EDF and RM, and Look – Ahead RT-DVS. The cycle conserving EDF and RM at first, scale-up the utilization by applying SVS (reducing the operating frequency and supply voltage). This scheme assumes the worst-case execution time initially and then executes at a high frequency until the completion of some jobs. Like in the EDF, the cycle conserving EDF also selects the next job to run based on the job deadline. Whenever a job is finishing its execution before its worst-case execution time, the unused CPU time is utilized by recalculating (lowering) the operating frequency and supply voltage for the ready-to-run jobs. This calculation uses the actual execution time to find the utilization until a new job of the same task arrives in the system. On the arrival of a new job of the task, the operating frequency and supply voltage are recalculated with the worst-case execution time of the job. This is higher than or equal to the current operating frequency and supply voltage. The cycle conserving RM works in a similar manner except for the criterion for selection of the next job to run. In cycle conserving RM, the next job is selected based on the period, not the deadline. The cycle conserving RM does not perform the schedulability test, as the test takes $O(N^2)$ time, where N is the number of tasks to be scheduled. The Look-Ahead EDF uses a Look-Ahead technique to determine the future computation need and defers task execution. This approach is more aggressive than the cycle conserving EDF and RM. This Look-Ahead scheme looks at the time interval until the next task deadline and tries to push (defer) as much work as possible beyond the deadline. This scheme sets the operating frequency and supply voltage to the minimum possible value, so that it can finish the minimum work to ensure all the future task deadlines. The task of determining the minimum cycle required and minimum operating frequency is carried out by looking

at the tasks in the reverse EDF order. Although this scheme aggressively reduces the processor operating frequency and supply voltage, it ensures that there are sufficient cycles available for each task to meet its deadline. This approach may thus result in running the processor at its maximum frequency in order to complete all the deferred work in time. The authors adopt this strategy keeping in mind the very high chances of tasks finishing their execution much ahead of their worst-case execution times. In this case, the system never needs to execute at peak execution rates and thus, this heuristic allows the system to continue performing (by meeting all the task deadlines) with low operating frequency and supply voltage.

2.3.2.2 Platform independent Energy Efficiency

One of the most important platform-independent factors affecting power consumption caused by a schedule is context switching [Mok 1983]. The context switch time is the time taken to switch between two processes or threads in a schedule. Thus the context switch duration is a hidden, unproductive duration in a schedule. The context switch duration includes the time taken for saving the context of the current process / thread and loading the context of the next process / thread. This implies that when a process finishes or a new process starts, this interval is not counted as a context switch. Typically, the duration of a context switch between threads is less than that between processes, though the former is not insignificant. In this thesis, it is assumed that context switching time refers to context switching between processes. Most of the issues related to context switching between processes are applicable to threads as well.

Various factors specific to the architecture and the operating system affect the context switch duration. For instance, the impact of register sets, floating point units, and caching schemes on context switching times have been reported [Dittman 2004] [Gooch 1998]. Gooch [Gooch 1998] also refers to the impact of the process queue on the context switching time – in particular, the strong correlation between context switching time and the length of the run (process) queue. We conjecture that this may be a consequence of the time taken for inserting a switched-out process into the data structure for the run queue. Hence this time is likely to be logarithmically or linearly proportional to the number of processes in the queue, depending on the data structure used. Though the data

available is not enough to substantiate our conjecture, hopefully this can be verified experimentally.

The direct impact of context switches in a schedule is the time spent in the act of context switching [Acquaviva 2003]. This time is – depending on the specific architecture and the operating system – small, though not insignificant. The number of context switches in a schedule may even add up to a significant delay in the execution of a process and thus, affect its schedulability. The total time spent in context switches also results in wasted power consumption. An indirect but more significant impact of context switches may be the data movement caused across the memory hierarchy, i.e., cache block replacement and page replacement in the RAM. In fact, the additional energy consumption due to this indirect impact has been reported to be significantly higher [Lee 1998] [Lee 1999] [Acquaviva 2003].

Available analyses or evaluations of scheduling algorithms in literature do not account for context switch time. In particular, they use a simplistic model where the context switch duration is assumed to be 0. This affects the evaluation in two ways: (a) actual execution times may not match the scheduled times and in particular, the hard real time tasks may miss deadlines; (b) the context switch is unproductive and the energy consumed for the operation is a waste and in particular, this may critically impact the performance of a low power system. An indirect but more significant impact of context switches may be due to cache flushes. In fact, the additional energy consumption due to this indirect impact is reported to be significantly higher [Acquaviva 2003].

The amount of energy wasted due to context switches in a schedule is proportional to the product of the number of context switches in the schedule and the average impact of a context switch. A power-aware operating system should account for the impact of scheduling on the power consumption. And a power-aware scheduling algorithm should account for the impact of context switches on power consumption. Several scheduling algorithms have been designed to be preemption-aware, i.e. they reduce the number of preemptions or context switches.

Various techniques have been proposed in literature for reducing the number of context switches in a schedule. These techniques vary in complexity from simple and inexpensive

heuristics to exhaustive search. Some techniques attempt to reduce the number of context switches while others address the indirect impact of context switches by reducing data movement across the memory hierarchy.

Oh and Yang [Oh 1998] propose a variant of LLF to reduce preemptions in a schedule, known as Modified LLF (MLLF) by fixing the "frequent preemption problem of LLF" [Dertouzos 1974] [Mok 1983]. The strict LLF scheduling algorithm suffers because of the frequent context switches in the schedules generated by it. Frequent context switches are possible if there are jobs with the same slack. According to the LLF scheduling algorithm, the slack / laxity of a job is defined as the difference in the time available until the deadline of the job and the job's remaining execution time. In strict LLF, the processes with equal slack / laxity force the scheduler to select the other job to run in the CPU (context switch) after every 2 units of execution in the CPU. This results in loss of time and energy which may even cause the missing of deadlines by some of the processes. When there is a tie in the slack / laxity among processes, the MLLF scheduling algorithm executes the process with the least deadline while freezing the priority of all the other processes with the same slack / laxity. This heuristic fixes the frequent context-switching problem of LLF, without affecting its optimality. The approach is simple and effective in addressing the limitation of LLF, but it does not aggressively remove unnecessary context switches. Furthermore, no detailed analysis on the effectiveness of the algorithm (in reducing preemptions) is available.

Zolfaghari [Zolfaghari 2004] proposed the Optimized Minimum Laxity First (OMLF) scheduling algorithm which overcomes the drawback (large number of context switches) of the Minimum Laxity First scheduling algorithm. This scheme nearly follows the technique followed in MLLF, except that the executing process's priority also decreases. In this scheme, the priority of the process with the longest execution time increases more rapidly than the other processes with the same deadlines. The priority function they used is different from LLF and MLLF. Hildehrandt, et al. [Hildebrandt 1999] proposed and evaluated a universal deterministic scheduling coprocessor that implements the scheduling algorithm, Enhanced Least-Laxity-First-algorithm (ELLF), which can hide the runtime overhead of the LLF (MLF) algorithm.

Wang and Saksena [Wang 1999][Wang 2000] describe a fixed priority scheduling algorithm that reduces context switches. In this model, each task has a regular priority and a preemption threshold priority. This scheduling algorithm allows a task to disable preemptions caused by tasks having up to a specified threshold priority, i.e., tasks having a lower priority than the preemption threshold cannot preempt the running task, even if the priority of the other task is greater than that of the running task. Tasks having a higher priority than the preemption threshold are allowed to preempt the running task. Hence, a certain level of non-preemptability is achieved using the preemption threshold. Preemption thresholds are assigned by a branch-and-bound algorithm using *lateness* heuristic. This paper also proposed an algorithm to find the preemption threshold with $O(N^2)$ complexity. This approach is limited to fixed priority scheduling, as threshold assignment may take exponential time. The authors claim 15% - 20% increase in processor utilization as compared to preemptive scheduling.

Stewart and Khosla [Stewart 1991] proposed Maximum Urgency First (MUF) scheduling algorithm. MUF is an improvement over the RM, which can be used to predictably schedule dynamically changing systems. MUF is a mixed priority scheduling algorithm (combination of fixed and dynamic priority algorithms) and combines the advantages of the RM, EDF and LLF algorithms. The urgency of a task is defined as a combination of two fixed priorities and one dynamic priority. One of the fixed priorities, called the *criticality*, has precedence over the dynamic priority. Meanwhile, the dynamic priority has precedence over the other fixed priority, which is called *user priority*. The dynamic priority is inversely proportional to the laxity of a task. The assignment of criticality and user priority is done apriori. To assign criticality, the tasks are ordered based on their period. The first N-tasks (these are the tasks which do not fail, even if there exists transient overload) whose total worst-case utilization is below 100% are defined as critical tasks. If a critical task does not fall in the critical set, then the period transformation is used. A high criticality is assigned for tasks in the critical task group and low criticality for others. A unique user priority is assigned optimally to every task in the system. The dynamic priority is the inverse of the laxity / slack of the job. The next job to run is the one with the highest criticality. If two or more jobs have the same criticality, then the job with the highest dynamic priority is selected. If there are two or

more jobs with the same criticality and dynamic priority, then the job with the highest user priority is selected. If more than one job has the same criticality, dynamic and user priority, then they are served in a First Come First Serve (FCFS) manner. The MUF works as RM, where the criticality of every task is different.

Vahid et al. [Vahid 2005] propose a modification to the Maximum Urgency First (MUF) scheduling algorithm [Stewart 1991] known as Modified Maximum Urgency First (MMUF). The major drawback of MUF scheduling algorithm is in its rescheduling operation. The rescheduling operation is performed whenever a task is arriving in the ready queue [Vahid 2005] [Stewart 1991], which may even cause a critical task to fail in certain situations. The MUF uses non-strict LLF where the scheduling decision points are those of arrival or completion of a job. This may result in MUF causing the failure of critical tasks.

In MMUF, a unique importance parameter is used, instead of using the tasks' request interval to create the critical set. MMUF uses the EDF or MLLF for defining the dynamic priority. This results in reducing unnecessary context switches introduced by LLF. RM, EDF, LLF and MUF are special cases of MMUF, depending on how the algorithm is setting the importance parameter and context switch reduction logic. The MMUF offers better performance (in schedulability) than MUF because of the lesser number of task preemption counts. It also results in the execution of more non-critical tasks in overloaded situations.

Apart from the above techniques, there have been other approaches where context switch reduction has been considered in conjunction with other techniques or goals. For instance, [Jianli 2005] demonstrates a preemption control technique for scheduling in processors with Dynamic Voltage Scaling. This technique is effective in reducing unnecessary context switches caused by dynamic voltage scaling, particularly under low or medium processor utilization levels. As such, this technique may not be effective in reducing context switches while scheduling under in non-DVS processors (nor equivalently under high processor utilization levels when DVS by itself is not useful).

Apart from these efforts on preemption reduction, there have also been several attempts to characterize energy consumption at the operating system level. In particular, power

analysis of real-time operating system in [Dick 2000] identifies time and energy profiles of different operating system functions and the behavioral characterization in [Stewart 1991] includes energy consumption profiles. But such attempts ignore the effect of task scheduling on energy consumption and do not relate energy savings to other real-time system performance metrics. Gopalakrishnan and Parulkar [Gopalakrishnan 1996] characterize the impact of preemptive scheduling on utilization. Here, a similar approach is adopted in that a count of the number of preemptions is taken, but experimental evaluation is used as the instrument for comparing the different algorithms.

From literature [Pillai 2001] [Gopalakrishnan 1996], it is evident that the real-time systems require techniques for power consumption reduction at the operating system level for better performance. In real-time systems, the optimization of platform-independent parameters like context switches not only reduces the power consumption, but also increases the schedulability of the task set. This prompts for the design of scheduling algorithms – both static and dynamic – which aggressively reduce the context switches independent of the platform in use.

2.4 CACHE CONSCIOUS SCHEDULING ALGORITHMS

Energy efficiency and time saving can be achieved by minimizing the overheads in a schedule. One such factor to minimize in a schedule is the number of preemptions as the time spent on saving and loading the context of processes is unproductive and the energy used for that is wasted. The direct impact of preemption is the time and energy spent for saving and loading the context. However, in addition to the direct costs, preemption introduces indirect costs caused by cache memory flushes. The cache memory blocks belonging to the preempted process may not be available in the cache when that process gets the next chance to execute in CPU. This results in a huge amount of data transfer across the memory-hierarchies, thus causing an increase in power consumption and execution time. The increasing execution-time of tasks leads to deadline misses by low priority tasks.

It is not necessary that a schedule with the minimum number of preemptions will cause minimum cache impact. This is because of the fact that a program in execution consumes a varying number of cache pages at different points in time. As the cache impact leads to

a major share of time and power consumption a schedule which generates preemptions at the minimum cache impact points may perform better than a schedule with the minimum number of preemptions. The cache related preemption delay (CRPD) may affect the schedulability of the task set as well.

The main challenge in addressing the schedulability of a real-time system with a cache memory is its unpredictability because of varying cache related preemption delay. There are two ways to address this issue. The first way is to use cache partitioning, wherein the cache memory is divided into disjoint partitions and one or more partitions are dedicated to each real-time task [Kirk 1989][Liedtke 1997] [Muller 1995] [Wolfe 1994]. In the cache partitioning techniques, each task is allowed to access only its own partition and thus, cache related preemption delay is avoided. However, cache partitioning has a number of drawbacks. One of the main drawbacks is that the existing system (hardware, software or both) may need to undergo serious modifications for the implementation of this scheme. Partitioning of the existing cache has also the drawback of limiting the amount of cache memory that can be used by an individual process at any point of time. The second way is to incorporate the worst case cache related preemption delay as a part of the process execution time and then analyze the schedulability with this new worst case execution time.

Luculli and Natale [Luculli 1997] presented a static scheduling methodology for real-time tasks whose task layout is known at design time and does not change at runtime. This work tries to optimize the extrinsic cache misses.

Basumallick and Nilsen [Basumallick 1994] proposed an improvement over RM with an upper bound on the cache related preemption delay to calculate schedulability. The main drawback of this technique is that it suffers from the pessimistic utilization bound, i.e., many task sets which are failing the schedulability condition can still be executed successfully. To overcome this problem, Mataix et al. [Mataix 1996] proposed a technique based on the response time approach. Even in this scheme the pessimistic assumption that each cache block used by a task replaces a memory block from the cache that is needed by the preempted task holds true. This leads to an overestimation of the cache related preemption delay, as it is possible that the replaced memory block is one

that is no longer needed or one that will be replaced without being referenced even when there were no preemptions.

Tomiyama and Dutt [Tomiyamay 2000] gave an approach to calculate the tight upper bound on the Cache Related Preemption Delay (CRPD) which a task might impose by using integer linear programming. This work determines the program execution path of the task which requires the maximum number of cache blocks. However, they only consider direct-mapped instruction cache. The above overestimation is addressed by Lee et al. in [Lee 1998].

Lee et al. [Lee 1998] proposed a technique to analyze the cache related preemption delays in fixed priority scheduling for the tasks that cause unpredictable variations in the execution time. This technique first performs a per task analysis to estimate the cache related preemption cost for each execution point and then stores this value in a table called the preemption cost table (for each task). This table provides the upper bound on the cache related preemption delay for a given number of preemptions. By using the worst case execution time and the worst case cache related preemption delay, this technique computes the worst case response time of each task by using a linear programming technique. Although this technique is more accurate than the techniques that do not consider the usefulness of cache blocks, it is still subject to a number of overestimation sources. This solution suffers because of two types of overestimation. First, when a task is preempted, not all of its useful cache blocks are replaced from the cache. Second, the worst-case preemption scenario given by the solution may not be feasible during in the actual execution. This leads to an enhancement of this technique by the same authors in [Lee 2001b].

Negi et al. [Negi 2003] refined the approach of Lee et al. in [Lee 1998] by applying path analysis. [Negi 2003] provided a program path analysis technique which will analyzes both the preempted and the preempting tasks to estimate the CRPD. This technique improves the accuracy of the analysis by estimating the possible states of the cache at each possible preemption point than estimating the state of each cache block independently. However, inter-task cache eviction is not considered. Also, WCRT analysis is not mentioned in [Negi 2003].

Lee et al. [Lee 2001b] enhanced the previous technique [Lee 1998] by bounding the cache related preemption delay in the fixed priority scheduling algorithm for instruction caches. While calculating the cache related preemption delay, the enhancement takes into account the relationship between a preempted task and the set of tasks that execute during preemption. The enhancement also considers the phasing of tasks to eliminate many infeasible task interactions. These enhancements are passed as constraints to the linear programming problem to then calculate the guaranteed upper bound on the cache related preemption delay.

Tan and Mooney [Tan 2004b] [Tan 2004c] proposed to analyze the inter-task cache eviction. This approach assumes that all cache lines used by the preempted task and evicted by the preempting task will be reloaded after the preemption. But, Lee et al. [Lee 1998][Lee 2001b] presented that only those cache lines used by “useful" memory blocks of the preempted task need to be reloaded. In [Tan 2004a], Tan and Mooney focused on enhancing the previous approach [Tan 2004b] [Tan 2004c] by incorporating “useful" memory block analysis in the work of Lee et al. So this method first analyzes the maximum set of memory blocks in the preempted task that can possibly cause a cache reload. Then, the method incorporates the inter-task cache eviction behavior by calculating the intersection set of the cache lines used by the preempting task and the preempted task. The new approach results in a more accurate WCRT method than, that by Lee et al and Tan and Mooney for a multi-tasking single-processor system, using set-associative or direct-mapped unified caches.

Staschulat et al [Staschulat 2005a][Staschulat 2005b] proposed the cache related preemption delay analysis for set-associative instruction caches. In this technique, the preemption delay analysis is integrated into a scheduling analysis to determine the response time of tasks accurately. [Staschulat 2005b] used a pseudo-polynomial algorithm, where the designer can decide the tradeoff between the analysis precision and the analysis execution time.

Ju et al. [Ju 2007] presented a way to incorporate the cache related preemption delays (CRPD) in dynamic, preemptive, multitasking real-time schedulers like the EDF. The CRPD is the delay introduced by the higher priority tasks because of cache misses caused via preemptions. The proposed implementation had three steps. In the first step, the

program analysis techniques are used to estimate the maximum CRPD incurred by the task preemptions. The second step bounds the number of preemptions of each task and the third step finds the actual execution time of each task with its total CRPD. The analysis is done by maintaining the possible cache contents at each of the preemption points of the lower priority tasks and of the high priority tasks. If the resultant execution times meet the schedulability, then the tasks are schedulable. These tests though better than the other approaches are not sufficient to find a tight bound for the cache related preemptions.

Ramaprasad and Mueller [Ramaprasad 2006a] [Ramaprasad 2006b] bound the cache interference penalty on real-time tasks by providing accurate predictions of the data cache behavior across preemptions. This is done by deriving the data cache reference pattern for all scalar and non-scalar references. The effects of cache interferences are analyzed by identifying the additional misses due to preemptions. This method calculates the tight upper bound on the number of preemption points for each job of the task and then finds the worst possible impact caused by it. This work proved by experimentation that it is sufficient to consider the N most expensive preemption points, where N is the maximum possible number of preemptions.

Lee et. al [Lee 1998][Lee 1999] proposed a new replacement scheme called Limited Preemptive Scheduling (LPS) that limits the preemptions to execution points with small cache related preemption cost. LPS uses data flow analysis techniques to determine the preemption points with small cache loading costs. LPS reduces the cache related preemptions at the cost of increasing the blocking delay of higher priority tasks. This scheme finds a schedule which maximizes the schedulability of a given task set, while minimizing the cache related preemption delay. The primary limitation of this approach is that it requires extensive data flow analysis and therefore, may not be suitable for dynamic scheduling.

CHAPTER 3

LATE LEAST RECENTLY USED (LLRU) REPLACEMENT STRATEGY

3.1 INTRODUCTION

The performance of cache memory depends heavily on its hit rate and access time. The classical approach to improve the cache performance is to increase the hit rate. One way of improving the hit rate is to reduce the number of conflict misses, which can be achieved by increasing the cache associativity. In [Hennessy 2007], it is proven that conflict miss reduces from 28% to 4% by changing associativity from direct mapping to 8 – way set associative. The set associative and fully associative caching schemes provide an advantage with respect to hit rate over the direct mapping scheme. The replacement policy used in set associative and fully associative caching schemes plays an important role in improving the hit rate, as it determines the next cache line to be replaced.

A replacement strategy is needed when all the cache lines are filled and a new block of memory needs to be placed in the cache. Cache controller identifies a cache line to be replaced. Then it replaces that cache line with new data from the main memory. The replacement algorithm used in cache memory helps in reducing the number of cache misses and thus reduces the power consumption. This reduction in power consumption for set associative cache can thus be achieved with the help of an efficient replacement algorithm. The performance of cache replacement mechanism primarily depends on how accurately the cache can predict the future reference pattern based on the past references. The future reference pattern may depend on past reference pattern and input data.

The current processors employ various replacement policies such as Random, Least Recently Used (LRU), Pseudo LRU (PLRU), Most Recently Used (MRU) and Round robin (or FIFO – First-In-First-Out).

3.2 LEAST RECENTLY USED (LRU) REPLACEMENT STRATEGY

LRU replacement strategy, as the name implies, replaces the element that has not been accessed for the longest period. This results in a higher cache hit rate with the cost of additional time and

hardware for maintaining LRU state information and decision-making. It is the most widely used replacement strategy in conventional cache, because of its high performance [Smith 1982]. The implementation complexity of the LRU scheme increases with increase in associativity [Hennessy 2007][Deville 1992][Sudarshan 2004], thus resulting in consuming more time to detect the line to be replaced. There exist various ways to implement LRU in hardware, which includes counter, square matrix, skewed matrix, Link list, Phase, and Systolic array method[Sudarshan 2004]. There also exist a large number of modifications for LRU strategy, which mainly focus on higher performance and lower implementation complexity [Sudarshan 2004] [Sukumar 1993] [Zhang 1997]. Literature reveals that LRU strategy performs close to optimal replacement, when associativity is less. As the associativity increases, the performance degrades considerably [Wong 2000].

In this replacement strategy, the LRU information is updated for each reference to a cache line. LRU takes more time per access of data from cache, compared to FIFO and random replacements. The LRU algorithm for cache replacement is given below.

3.2.1 LRU REPLACEMENT STRATEGY: ALGORITHM

Input: LRU data structure.

Output: Line number of the Cache line to be evicted (victim line) if CACHE MISS

Search space: All cache lines (N) in a set S.

ON EVERY REFERENCE IN A CACHE SET

```
begin
    if (SQUARE MATRIX implementation) then
        if (Reference is CACHE HIT in ith Cache line) then
            set ith row of LRU data structure to 1; set ith column of LRU data structure to 0
        Victim line number = Row number of LRU data structure with all zeros
    if (COUNTER implementation) then
        if (Reference is CACHE HIT in ith Cache line) then
            for j = 0 to N-1
                if (LRUcount[j] > LRUcount[i]) LRUcount[j] = LRUcount[j] – 1;
            LRUcount[i] = N-1;
        Victim line number = Cache line number whose LRUcount value is zero
end
```

3.2.2 HARDWARE IMPLEMENTATION OF LRU SCHEME FOR N – WAY SET-ASSOCIATIVE CACHE

Most commonly used hardware implementations in LRU are square matrix and counter. The hardware components required for implementing square matrix LRU for a set is a 2:1 multiplexer, (log_2N):N decoder, N x N storage elements (D – Flip flop), N x log N priority encoder and N - OR gates. The hardware components required for implementing counter LRU for a set is a 2:1 multiplexer, a N:1 multiplexer, two 1:N demultiplexers, N counters (each counter is of log_2N storage elements, i.e., D – Flip flops), N (log_2N)-bit comparators, N x log_2N priority encoder and N - AND gates. With increase in associativity, in case of square matrix LRU implementation, the number of storage elements increases quadratically (N^2) whereas increase is linear (N log N) in the case of counter. The associated circuit complexity also increases heavily with associativity.

3.3 WHY LATE LEAST RECENTLY USED (LLRU) REPLACEMENT SCHEME?

The need of LLRU replacement strategy is explained with an example. Assume a system has two ready–to-run processes (P0 and P1) in RAM with 6 non-shared data lines each (P00, P01, P02, P03, P04 and P05 for Process P0 and P10, P11, P12, P13, P14 and P15 for Process P1) and two shared data lines that is shared between the two processes (S0 and S1). Also, assume the cache system is 8 – way set associative, with all the data lines mapped to the same set of the cache. The access pattern is P00, S0, S1, P01, P02, P03, P04, P05, P10, P11, P12, S0, S1, P13, P14 and P15.

If the replacement scheme is LRU, this pattern will result in 16 cache misses and will repeat for every hyper period. If we have a replacement strategy where a shared cache line (data line in cache containing shared data between processes) replacement is delayed for some more time, which results in a better cache hit rate. According to LLRU, the shared cache lines have shared bit set to one, which delays the replacement of these pages. This result in reducing the number of cache misses to 14, from 16. The performance of this algorithm gets even better when the number of shared cache lines increases. For a system with no shared cache lines, the performance of this algorithm is the same as that of LRU.

None of the existing replacement policies address the issue of shared cache lines among processes, as the cache is transparent to the operating system. This work simulates and synthesizes a replacement scheme called Late LRU (LLRU), which takes shared cache lines into

consideration to improve the cache hit rate, thus resulting in reduction in power consumption and improvement in the cache performance.

To the best of author's knowledge, this work is the first attempt to address issues related to shared cache lines during cache replacement. This work is an extension of LRU scheme known as Late-LRU (LLRU) replacement scheme, which takes care of shared pages while replacement (repetition). An analysis of the performance of LLRU scheme by using both software Simplescalar traces and hardware simulation is carried out. Software simulation results provide cache miss rate measure, whereas, the hardware simulation using Modelsim and Leonardo Spectrum provides circuit complexity and size.

3.4 LATE-LRU REPLACEMENT POLICY (LLRU)

The proposed scheme, termed as Late – LRU replacement strategy (LLRU) is an augmentation of LRU replacement strategy, where an additional state affects the replacement decision. Along with the control bits like valid bit, dirty bit (only for write back policy) and LRU state information, which influences LRU replacement decision, this scheme introduces one more bit per cache line termed as 'shared bit' to identify the shared cache lines. This bit is used for identifying the victim line for replacement. By default, the shared bit of all the cache lines is reset. This is to ensure good performance even when there are no shared pages available in the system. The shared bit corresponding to a cache line is set if its content is shared between two or more processes and some of these processes are in ready state, that is, the shared bit is set for the cache lines that are shared (which are likely to be used by the other ready-to-execute processes). The shared bit is reset for the non-shared cache lines, as well as for the shared cache lines whose other sharing processes are not in ready state.

The LLRU replacement policy uses LRU data structures and shared bit. While finding a victim cache line using LLRU replacement policy, one can assign highest priority to shared cache lines, so that they can stay back for a longer time in the cache, thus resulting in an increased cache hit rate. The shared bit of the cache line is reset immediately after the scope expiry, that is, when all the shared processes complete their execution. In case of a cache miss occurs, the LLRU replacement algorithm finds a cache line to be evicted and will copy the data requested to that cache line.

This approach is very useful and effective for embedded systems, which are static-scheduled. At the time of compilation, one can identify the shared contents among the processes and can also get the sequence of execution of these processes.

3.4.1 LLRU CACHE REPLACEMENT ALGORITHM

Here, LLRU replacement decision is based on LRU data structure and shared bit. If the access is a cache hit, then the hit signal is set by the cache controller and the LLRU data structure is updated in the similar fashion as in case of LRU replacement policy. If the access is a cache miss, then with the help of LRU data structure and shared bit, LLRU decides which cache line to be evicted within the cache set.

In LLRU, to find the cache line to be evicted, find the line with minimum LRU count and shared bit status as 0. If each of the cache lines in a set has its shared bit as 1, then apply LRU replacement policy to find the cache line to be evicted. The LLRU algorithm is given below.

Input: LRU data structure and Shared bits.

Output: Line number of the Cache line to be evicted (victim line) if CACHE MISS

Search space: All cache lines (N) in a set S.

```
ON EVERY REFERENCE IN A CACHE SET
begin
    if (SQUARE MATRIX implementation) then
        if (Reference is CACHE HIT in ith Cache line) then
            set ith row of LRU data structure to 1; set ith column of LRU data structure to 0
        Count number of 1's in each row of LRU data structure (log2N bits)
        Append Shared bit as MSB bit to the count value (log2N + 1 bits)
        Victim line number = Row number with minimum count value
    if (COUNTER implementation) then
        if (Reference is CACHE HIT in ith Cache line) then
            for j = 0 to N-1
                if (LRUcount[j] > LRUcount[i]) LRUcount[j] = LRUcount[j] – 1;
            LRUcount[i] = N-1;
        Append Shared bit as MSB to corresponding LRUcount value
        Victim line number = Cache line number whose LRUcount value is minimum
end
```

3.4.2 LLRU HARDWARE IMPLEMENTATIONS

LLRU works in the same way as LRU, when none or all of the cache lines in the set are shared. If any of the cache lines' shared bit is set, then the decision-making is based on the above mentioned LLRU strategy. The LLRU replacement circuitry finds a non-shared cache line with minimum LRU value as the victim cache line for replacement.

The following section describes in detail about square matrix and counter LLRU implementations.

3.4.2.1 Square Matrix Implementation of LLRU

The LLRU selects least recently used non-shared cache line if there exists at least one non-shared cache line, otherwise the least recently used shared cache line is used for replacement. This implementation uses D flip-flops to construct a square matrix LRU and shared bit data structures. The implementation requires N bits for shared bit data structures per cache set. It requires N x N bits per cache set to implement a square matrix LRU data structure for a N-way set associative cache. The global set contains M replications of these data structures, where, M denotes the number of cache sets available in an N – way set associative cache. In LRU square matrix implementation, each of the N rows of the data structure maps to one of the N cache lines of a set, as [Sudarshan 2004]. At reset all data structures are initialized to zero as shown in Figure 3.1.

	Shared	LRU Matrix			
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0

Fig. 3.1: LLRU data structure for a 4-way set associative cache (Square Matrix implementation)

The cache set identification and working set identification is the same, as in LRU replacement scheme described in [Sudarshan 2004]. The square matrix LRU data structure follows a simple logging scheme wherein, it sets the row of access lines to 1 and then sets the column of the access lines to 0. The number of 1's in each row is an indication of the order of cache line access. The

cache line corresponding to the row, which has the maximum number of 1's, is the most recently used and the cache line corresponding to the row, which has all 0's is the least recently used. In case of a cache miss, the corresponding shared bit is concatenated as MSB with the count of 1s in each row to find the cache line to be evicted. The cache line with minimum resultant count will be the one to be evicted, when transferring content from main memory.

Figure 3.2 shows the square matrix implementation of LLRU replacement scheme. LLRU consists of a 'n'-bit 2-to-1 multiplexer, a n: N decoder, a (N x n)-bit counting circuit (combinational circuit), a ((n+1) x n)-bit minimum finder circuit, N shared bits, and N x N edge-triggered D flip-flops with clear and preset. N represents the associativity, while n is $\log_2 N$.

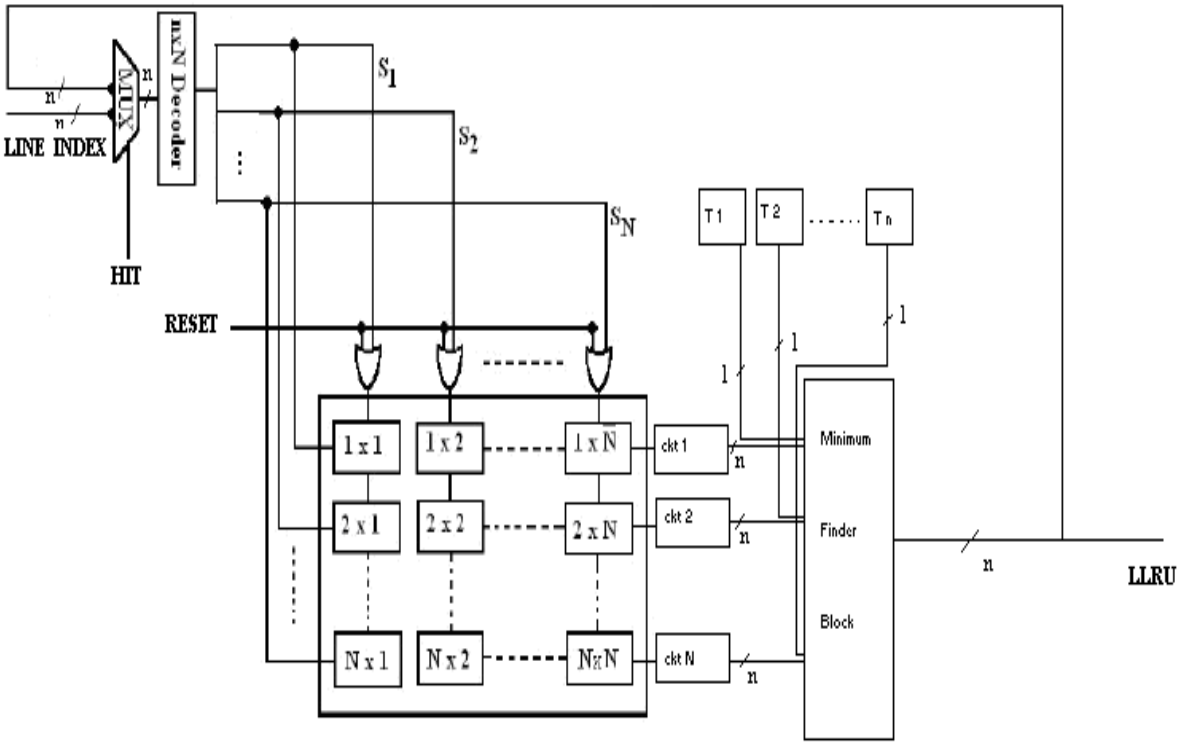

Fig. 3.2: LLRU square matrix implementation

The circuit works as follows.

The 2:1 multiplexer, $\log_2 N$ x N decoder and N x N edge-triggered D flip-flop work in the same way, as in LRU. In the case of a hit, the cache line index is given as the select input of the decoder. The decoder selects the corresponding row and column. The storage element in row is set and column is reset. The ANDed complement output of the D flip-flops in each row is fed into

a counting circuit. The counting circuit counts the number of 1s in the input and gives the resultant count as the output. The output of the counting circuit is of n bits ($\log_2 N$), since each row consists of N elements. The N shared bits are also connected (input) to the minimum finder block. The shared bit is concatenated as MSB to the count provided by the counting circuit and this (($\log_2 N$) + 1) bits per cache line is input to the minimum finder. The minimum finder finds the minimum of all the inputs and its output is the cache line to be evicted, according to LLRU policy. In the case of a miss, this index is given to the multiplexer, which is triggered by the miss signal to enable it and the corresponding row and column are set and reset respectively. RESET signal high initializes the matrix by setting all storage elements to zero.

It can be observed that for a cache hit, the delay involved and time required to service the request is the same as LRU replacement policy. But for a cache miss, as the replacement line has to be obtained from the minimum finder, a delay is added to the replacement implementation. The matrix also needs to be updated with the replacement.

Similar to LRU, square matrix LLRU implementation uses simple data structure and requires minimum associated logic for finding / modifying LRU information. The main drawback of square matrix method is that it does not scale well for large associativity cache. This is because the amount of space required for information increases quadratically with N.

3.4.2.2 Counter Implementation of LLRU

Figure 3.3 shows the data structures and its initial values for a 4-way set associative LLRU counter implementation. In this implementation, a register is used for every individual row to maintain LRU data structure [Sudarshan 2004]. This implementation uses an edge-triggered $\log_2 N$-bit register, which supports operations like reset to zero, decrement by one and load (N-1) externally for a N-way set associative cache. Each cache line in every set is mapped to a register. Thus, for higher values of N, the counter LLRU storage space utilization drops exponentially in comparison with square matrix LLRU implementation.

The values in the register indicate the order in which the cache lines within a set have been accessed. The smallest value (zero) in register corresponds to the least recently accessed cache line and the highest value (N-1) corresponds to the most recently accessed cache line. At reset all the registers and shared bits are initialized to zero.

In case of a cache hit, the LLRU counter data structure is updated in same fashion as LRU counter, i.e., the registers whose value is greater than the active register is decremented by one and the active register is set to the highest value (N-1).

In case of a cache miss, the shared-bit is concatenated as MSB with LRU register to form a $((\log_2 N)+1)$-bit LLRU register. The cache line with minimum resultant register value is the one to be evicted for transferring content from main memory.

	Shared	LRU Count
1	0	0 0
2	0	0 0
3	0	0 0
4	0	0 0

Fig. 3.3: LLRU data structure for a 4-way set associative cache (Counter implementation)

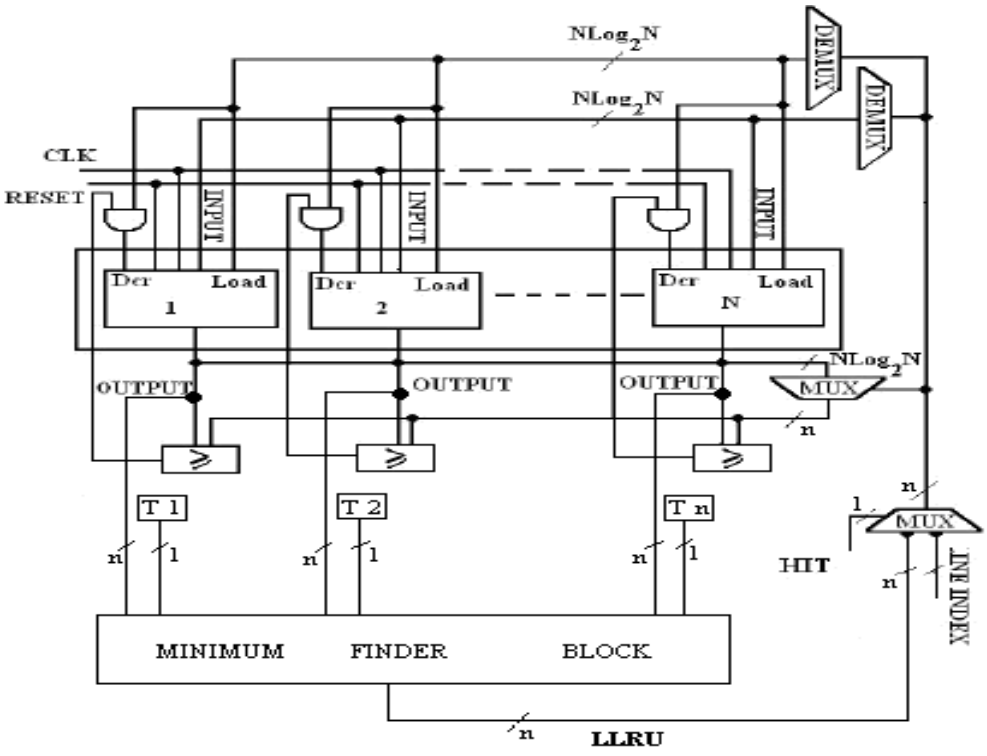

Fig. 3.4: LLRU counter implementation

The LLRU counter implementation has similar structure as the LRU counter implementation [Sudarshan 2004]. Figure 3.4 shows the counter implementation of LLRU replacement scheme. As in LRU, LLRU has a register per cache line to store LRU count information. LLRU has N shared bits, one per cache line indicating whether the cache line is shared or not. In LRU the cache line to be evicted is the one with the minimum count value. In case of LLRU, the cache line to be evicted is the least count non-shared cache line that may not be the least count cache line.

The LLRU replacement circuit is designed in same way as the LRU replacement circuit except the priority encoder is replaced with the minimum finder. The value of registers and shared bits are the inputs to the minimum finder. The minimum finder concatenates the shared bit as the MSB of corresponding register output. This guarantees that the shared cache line's count will be high compared to non-shared cache line, thus rendering shared data cache lines a second chance to reside in the cache. It outputs the index of the cache line, which has the minimum resultant count value. That cache line in the index is the one to be replaced, if the access is a cache miss.

3.5 EXPERIMENTAL RESULTS AND ANALYSIS

3.5.1 SOFTWARE SIMULATOR

A cache simulator is implemented in C with both LRU and LLRU replacement policies for experimentation. The experimentation uses Simplescalar benchmark address traces generated using Simplescalar2.0 simulator [Burger 1997]. In this experiment, 4KB and 8KB, cache sizes are used. Cache hit rate for 2-way and 4-way set associative cache configurations with data sharing (10 test cases with data sharing varied between 0% and 100%) is measured and averaged over the number of test cases. Figure 3.5 and 3.6 show the LRU and LLRU results for 4K, 2-way, 4K, 4-way, 8K, 2-way and 8K, 4-way set associative cache configurations.

In the worst case, i.e., when no shared cache lines are available, LLRU offers the same performance as LRU. Depending on the number of shared cache lines, the performance of LLRU improves. As expected, when the associativity increases, the hit rate of both LRU and LLRU improves and hence, 8K 4-way set associative cache configuration performs better than 8K 2way set associative cache configuration. It is also observed that depending on the cache size, performance varies.

In all these configurations, LLRU performs equally good or better than LRU. It is also observed that depending on the number of shared cache lines, the performance of LLRU improves.

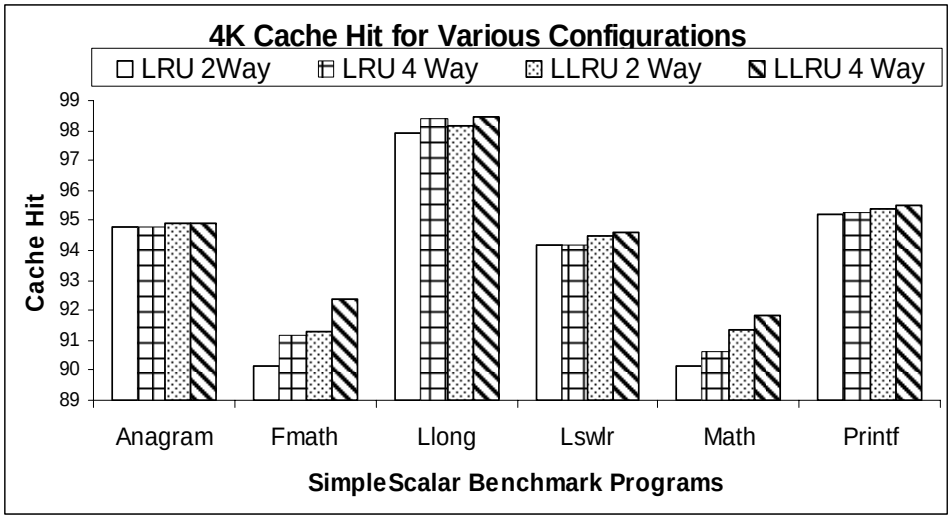

Fig. 3.5: LRU and LLRU performance of a 2-way and 4-way (4K cache) set associative cache.

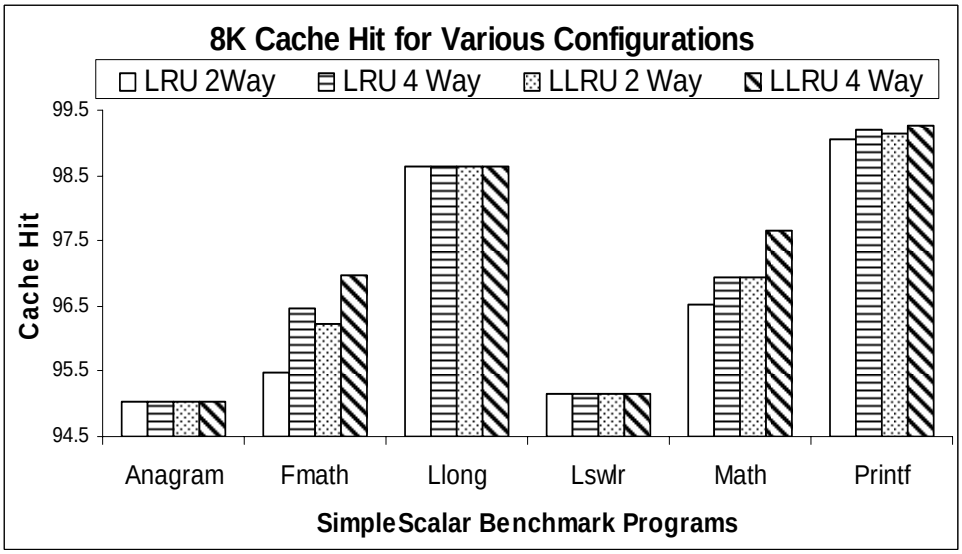

Fig. 3.6: LRU and LLRU performance of a 2-way and 4-way (8K cache) set associative cache.

3.5.2 HARDWARE SIMULATION AND SYNTHESIS

The verilog implementations of square matrix LRU, counter LRU, square matrix LLRU and counter LLRU are carried out. These implementations are simulated for a 4-way set associative cache configuration. The simulation establishes the functional correctness of the various LRU and LLRU hardware implementations. Simulation is carried out using Modelsim. These

implementations are synthesized with Leonardo spectrum using ami05 technology. Area and other layout details are obtained from the IC station.

Table 3.1 shows the maximum clock frequency, critical path delay, number of transistors used and the total area occupied for square matrix and counter implementation of LRU and LLRU replacement strategies. It is evident from the table that the square matrix implementations of the replacement policies (both LRU and LLRU) are faster, when compared to their counter implementations, because of reduced critical path delay. The number of transistors used is more in square matrix implementation compared to counter implementation. So the area occupied by square matrix implementation is more compared to counter implementation.

Table 3.1: Comparison chart of LRU and LLRU

		Clock Freq (MHz)	**Critical path delay (ns)**	**No. of Transistors (CMOS)**	**Area (mm^2)**
Square Matrix Implementation	**LRU**	84.9	11.40	797	0.70
	LLRU	92.2	10.47	1212	0.93
Counter Implementation	**LRU**	66.1	14.75	784	0.679
	LLRU	79.6	12.19	1082	0.899

From Table 3.1 one can observe that the LLRU implementation works at a higher clock frequency than the corresponding LRU implementation (counter and square matrix implementations of LRU).This is mainly due to lower critical path delay for LLRU as compared to LRU implementations. Here, the square matrix LLRU has the least critical path delay (10.47nsec) and counter LRU has the highest critical path delay (14.75nsec). It is also observed that square matrix LRU has lower critical path delay (11.4nsec) as compared to counter LLRU (12.19nsec).

It is also observed that LLRU implementations require more number of transistors and more area than the LRU implementations. LLRU improves the performance of cache and its accessibility, in the case of shared pages. Hence, the trade off for the area is justifiable for higher performance.

3.6. CONCLUSION

This work focused on the implementation and analysis of a new cache replacement strategy, Late LRU (LLRU), which particularly considers shared pages among processes, while deciding on replacement. Hardware implementations of LRU and LLRU based on square matrix as well as counter were carried out. These implementations were simulated in Modelsim and synthesized in Leonardo spectrum. Layouts for these hardware architectures have also been obtained, from the IC station. The cache hit is measured for various cache configurations by using software simulator and Simplescalar traces. The results thus obtained, were analyzed based on the parameters like area, clock frequency, critical path delay, number of transistors and cache-hit rate. From the above results, one can conclude that with minimal extra hardware, LLRU improves the cache performance significantly. This method guarantees that the performance of the modified replacement strategy, LLRU is better and at least as good, in the worst case, as the performance of the original strategy under the LRU replacement policy. It is known that any improvement in cache performance in embedded systems leads to reduction in power consumption and overall improvement of the system performance. Thus LLRU replacement scheme can play a significant role in cache memory design for embedded systems.

CHAPTER 4

WAY-PREDICTIVE PLACEMENT CACHE

4.1 INTRODUCTION

This chapter focuses on the methods adopted for energy efficient cache designs. Way prediction set associative cache is modified to improve cache performance in terms of power, access time and cache hit rate. This chapter discusses way prediction scheme and its limitations. This chapter also explains way predictive placement scheme, a modification of way prediction scheme with global prediction, a new technique to overcome some of the bottlenecks of way prediction scheme. The chapter elaborates on way predictive placement scheme with two alternatives for replacement strategy: LRU and aligned LRU (ALRU). This chapter evaluates conventional cache, way prediction cache and way predictive placement cache using SPEC 95 benchmark suite. The evaluation is based on number of tag comparisons, prediction hit rate, cache hit rate, cache access time and energy saving parameters.

4.2 WHY WAY-PREDICTION SET-ASSOCIATIVE CACHING SCHEME?

Cache memory power consumption can be significantly reduced by managing data lines and sense amplifiers efficiently**,** as these are the most power consuming modules in cache memory sub system. Wilton and Jouppi [Wilton 1996] reported the power consumption of data lines and data sense amplifiers as 55%, 65% and 75% of the total cache subsystem power consumption for direct, 2-way set associative and 4-way set associative mapping schemes respectively. Optimal energy efficiency is possible if each cache hit results in reading and comparing one tag entry, enabling and accessing only one data entry and if each cache miss results in reading and comparing one tag entry. The key to obtaining optimality is to pinpoint the matching way without probing all the ways. There exist various schemes like sequential access, and phased lookup set associative cache. Both these schemes have performance overhead in terms of extra phase as it takes one extra cycle to access the cache. A better approach for attaining energy efficiency is to speculatively choose one way before a cache line access in set associative cache. As functioning of cache is based on temporal locality, Most Recently Used (MRU) heuristic

can be very effective. Inoue et al. [Inoue 1999] proposed this as way prediction set associative cache.

4.3 WAY PREDICTION SET-ASSOCIATIVE CACHE

Figure 4.1 shows the way prediction set associative cache (WP) proposed by Inoue [Inoue 1999]. The algorithm of the Way Prediction caching scheme with LRU replacement strategy is as discussed below.

4.3.1 ALGORITHM FOR WAY-PREDICTION CACHE

```
//Cache operations to be carried out on every reference
begin
    Look up the virtual page number in TLB
    if (TLB HIT) then
        Derive tag (t), index (i) and offset (o) from physical address fetched from TLB.
        WaySelect = Prediction Table[i]
        CYCLE = 1
        while (CYCLE < 4) do
            if (CYCLE = 1) then
                Enable WaySelectth cache way; Disable all the other N-1 ways
                if (Reference is CACHE HIT in ith set, WaySelectth way) then
                    Read / Write data from / to offset location; Update LRU data structure
                    CYCLE = 4
                else CYCLE = 2
            if (CYCLE = 2)
                Disable WaySelectth cache way; Enable all the other N-1 ways
                if (Reference is CACHE HIT in ith set) then
                    Read / Write data from / to offset location; Update LRU data structure
                    Prediction Table[i] = CACHE HIT Way number
                    CYCLE = 4
                else CYCLE = 3
            if (CYCLE = 3)
                Transfer data from Main memory to the victim line (selected by LRU
                replacement strategy) of ith set
                Read / Write data from / to offset location; Update LRU data structure
```

Prediction Table[i] = CACHE HIT Way number

CYCLE = 4

if (TLB MISS) then

Generate trap to handle TLB miss

end

4.3.2 WORKING OF THE WAY – PREDICTION CACHE

WP speculatively chooses one way with the help of way enable circuit before starting the normal cache-access process. This scheme accesses the cache data line and tag line corresponding to index field in the enabled way. If the prediction is correct, i.e. the tag field in address reference is matching with the tag field of the selected cache tag line then the cache access is completed successfully. All the (N-1) ways are in power down state during this cycle. A wrong prediction can be because of the requested data available in another cache way of the same set or because of a cache miss. In WP, prediction-miss results in enabling all the other (N-1) ways and comparing (N-1) tag values with the tag field of the address reference in the next cycle. If the tag field of the address reference is matching with any of the (N-1) tag fields of the enabled cache tag lines, then the cache access has been completed successfully. Otherwise, the access will be cache miss.

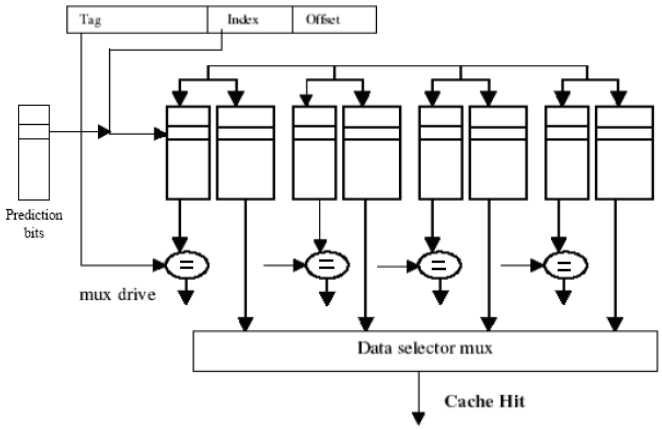

Fig. 4.1: 4-way set-associative cache (Way prediction cache)

Inoue et al. used $\log_2$ (N) bits per set to maintain the MRU way information, which is used for predicting the way. MRU bits of each set have information of recently accessed way of a cache line in that set. If prediction is correct, cache consumes dynamic energy for only one activated way. Otherwise, it consumes dynamic energy same as conventional

cache with an additional cycle time penalty for misprediction. This method saves almost (100*(N – 1) / N) % of the energy in a N – way set associative cache. Prediction accuracy according to Powell et al. [Powell 2001] is 90% for instruction and 80% for data.

On a prediction hit, WP consumes energy only for activating the predicted way. In addition, the cache access can be completed in one cycle. This results in saving 50%, 75%, and 87.5% of the dynamic power for 2-way, 4-way and 8-way set associative cache configurations respectively. On prediction miss (or cache miss), however, the cache-access time of the WP increases due to the extra phase.

The average energy consumption ($E_{WP\text{-}N\text{-}SACache}$) and the average cache-access time ($T_{WP\text{-}N\text{-}SACache}$) for the N-way set-associative WP (WP-N-SACache) are as follows:

$$\mathbf{E_{WP\text{-}N\text{-}SACache} = (E_{Tag} + E_{Data}) + (1 - PHR) * ((N-1)E_{Tag} + (N-1)E_{Data})}$$

$$\mathbf{T_{WP\text{-}N\text{-}SACache} = 1 + (1 - PHR)}$$

Where, PHR is prediction-hit rate.

WP improves the Energy – Delay (ED) product (ED= average cache access time * average energy consumption per cache access) by 60–70% compared to a conventional set-associative cache.

4.3.3 DRAWBACKS OF THE WAY-PREDICTION CACHE

Though way-prediction scheme is very effective in saving power, it suffers with serious drawbacks, which significantly limit its usage. This scheme suffers from performance degradation because of cycle time penalty for handling misprediction. In way prediction scheme, a table lookup is needed to identify the MRU information of the selected set. This adds extra time delay to the critical path as one cannot prefetch the MRU information until the set number is available. As known from literature, the performance of the existing way prediction with MRU information does not always work well. The two choices available for a way prediction are to use information available (1) early in the pipeline, such as the program counter, (2) later in the pipeline, such as a XOR-based approximation of the load address [Powell 2001]. Each of these methods has its own demerits. Way prediction based on information from early pipeline stages suffers from

poor accuracy. To improve the way prediction accuracy, one should go for late pipeline stage information. But the way prediction based on late pipeline information introduces a way prediction table lookup delay in the cache access critical path [Batson 2001]. For instance, the way-prediction scheme used in [Inoue 1999] inserts a table lookup after the address generation to identify the predicted way. The other drawback of way prediction scheme is that MRU information does not always work well with data references [Calder 1996][Batson 2001][Min 2004].

Other than time and performance overhead, WP increases hardware complexity of set associative cache. For a cache containing K sets, K* $\log_2$N bits are required for storing MRU information, where N is the associativity. This information is stored in the form of a table containing K rows where each row specifies MRU information of the corresponding cache set. This table is accessed every time (before a cache access) to determine the last accessed way in that set. This introduces an increase in the cache access time as it lies in the critical path. In addition extra circuitry is required for enabling / disabling the way.

Though WP reduces the energy consumption of a set associative cache significantly, there exist scope of improvements in terms of optimizing / avoiding table lookup time, prediction hit rate and hardware complexity. The following sections explain in detail about way – predictive placement scheme for set associative cache – a modification over way – prediction set associative cache to achieve greater energy saving, high prediction hit, reduced critical path delay, zero table look up and reduced hardware complexity.

4.4 WAY – PREDICTIVE PLACEMENT CACHE

The late pipeline stage information introduces an undesirable way prediction table lookup delay in the cache critical path. We require a scheme to maximize the prediction, minimize the delays and save energy significantly. This way-predictive placement scheme is a modification of way-prediction algorithm, which would help in reducing the number of tag comparisons, increase way-prediction rate and in turn reducing power consumption. The proposed scheme uses only $\log_2$ N global bits in fixed position for the entire cache to store the MRU information independent of number of cache sets. This acts as a unified global MRU for all the sets. This modification helps us to reduce the

hardware complexity and overcome the table lookup, while using late pipeline information.

4.4.1 ALGORITHM FOR WAY-PREDICTIVE PLACEMENT SCHEME

```
//Cache operations to be carried out on every reference
begin
    Look up the virtual page number in TLB
    if (TLB HIT) then
        Derive tag (t), index (i) and offset (o) from physical address fetched from TLB.
        WaySelect = Global MRU bits
        CYCLE = 1
        while (CYCLE < 4) do
            if (CYCLE = 1) then
                Enable WaySelectth cache way; Disable all the other N-1 ways
                if (Reference is CACHE HIT in ith set, WaySelectth way) then
                    Read/Write data from/to offset location; Update ALRU data structure
                    CYCLE = 4
                else CYCLE = 2
            if (CYCLE = 2)
                Disable WaySelectth cache way; Enable all the other N-1 ways
                if (Reference is CACHE HIT in ith set) then
                    Read/Write data from/to offset location; Update ALRU data structure
                    Global MRU bits = CACHE HIT Way number
                    CYCLE = 4
                else CYCLE = 3
            if (CYCLE = 3)
                Transfer data from Main memory to the victim line (selected by ALRU
                replacement strategy) of ith set
                Read/Write data from/to offset location; Update ALRU data structure
                Global MRU bits = CACHE HIT Way number
                CYCLE = 4
    if (TLB MISS) then
        Generate trap to handle TLB miss
end
```

4.4.2 WORKING OF THE WAY – PREDICTIVE PLACEMENT SCHEME

Conventional cache architecture for a 4-way set-associative cache is as shown in Figure 4.2. Consider an 8KB cache with a line size of 64 bytes. The cache line has 32 sets, thus the memory address will have 6-bit offset, 5-bit index and remaining 21-bits of tag fields. This architecture includes 5:32 decoder, word line drivers, four tag arrays, four data arrays, sense amplifiers, comparators, one multiplexer and output drivers. In addition to all these, way – predictive placement scheme has $\log_2$ N bits to store the MRU way information and extra circuitry to enable/disable the tag and data arrays. The way – predictive placement scheme, placement and replacement in the cache are based on this way information. The way – predictive placement scheme is as shown in Figure 4.3 and Figure 4.4.

The way – predictive placement scheme has the following changes to the existing way-prediction schemes. It uses only $\log_2$ N bits for the entire cache, irrespective of the number of cache lines, to store the MRU information. This is a unified MRU for all the sets. For the cache configuration described above, a way-prediction scheme would need 64 bits for storing the MRU information, whereas, the proposed way – predictive placement scheme needs only 2 bits. The proposed modification results in reduced hardware complexity and eliminates the table lookup, while using late pipeline information. For maximizing the prediction hit using MRU information, modification to existing scheme has been proposed. The modifications are as follows. Whenever a decision has to be taken about the page eviction, find if the cache block corresponding to the current MRU bits (way) in the given line is ready for replacement. This replacement strategy helps to explore MRU bits information and avoid misprediction between one index and the other. Secondly, most of the continuously referred pages in different indexes are aligned i.e., map on to the same way. By doing so, any random access to these indexes will not result in misprediction. The modified scheme helps in achieving hardware reduction and energy efficiency.

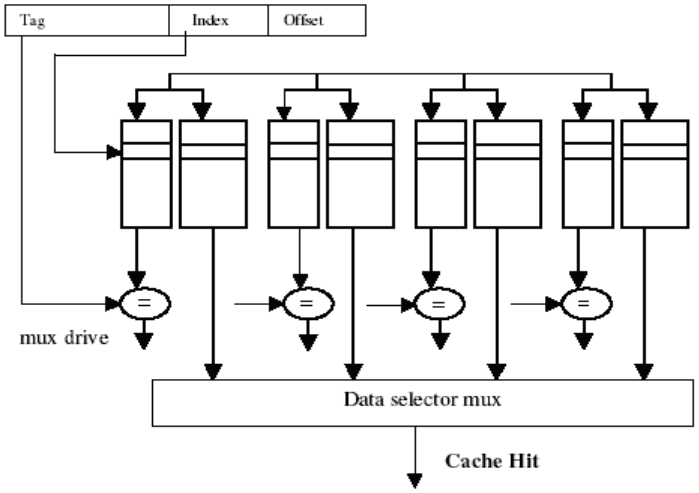

Fig. 4.2: 4-way set-associative cache (Conventional cache)

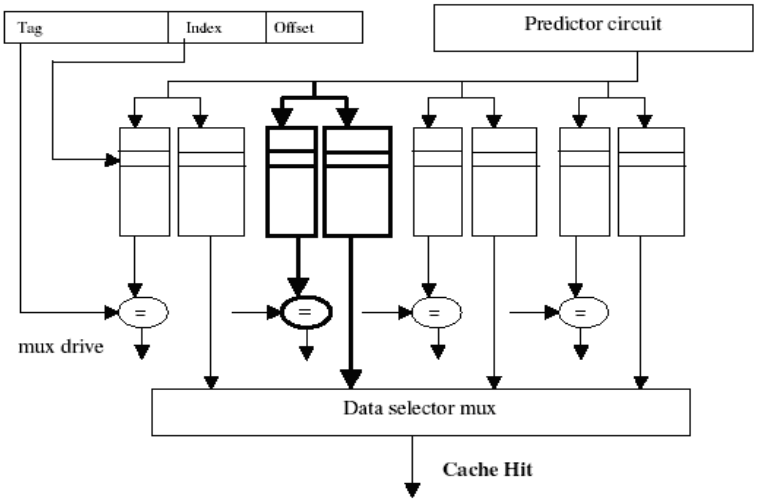

Fig. 4.3: Prediction Hit

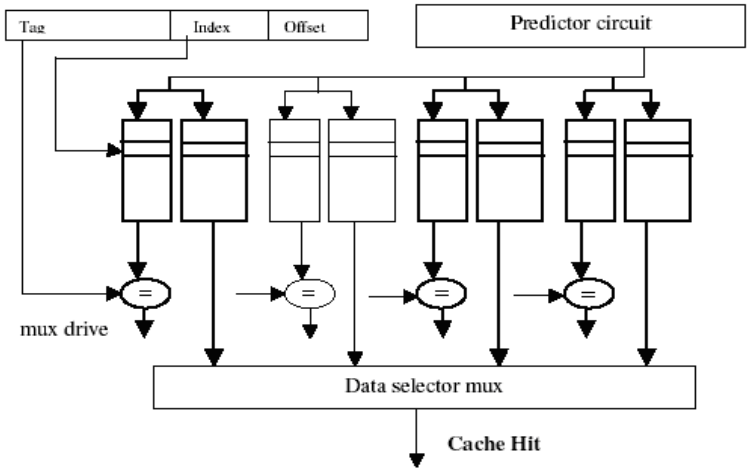

Fig. 4.4: Prediction Miss

On every reference, CPU produces a virtual address. If the corresponding page number is available in the TLB, then fetch the physical address from the corresponding TLB entry. In way – predictive placement scheme, for a cache access, depending on the value of global MRU bits, only one data-way and tag-way are enabled with the help of tag and data way selection circuitry. If the prediction is a hit i.e. if the tag value of the selected cache line (in the enabled cache way) is matching with the address's tag bits, then access the data from the corresponding offset. Update the LRU bits of the corresponding set.

If the prediction is a miss, then in the next cycle, enable all the other (N-1) data and tag ways, where N is the associativity and compare the tags of (N-1) cache lines of the set which are selected based on index bits, to see whether the required data is available in the cache. If the data is available in the cache, access the data from the offset location of the cache line. Update the MRU prediction bits with the new way identifier. Update the LRU replacement circuitry of the corresponding set accordingly. If the required page is not available in cache, then transfer page from main memory into the specific cache line (victim cache line) of the set specified by LRU replacement algorithm. After transferring data from main memory to the selected cache line, access the required data from the offset location of the cache line. Update the MRU prediction bits with the new way identifier. Update the LRU replacement circuitry of the corresponding set accordingly.

The prediction accuracy can be improved if one can modify the replacement circuitry. The modified replacement algorithm should provide better alignment of data lines in the same way. Next section explains in detail about the modified LRU replacement strategy. A cache miss will always lead to a prediction miss. So the performance of way – predictive placement scheme depends on cache miss and prediction miss rates.

4.4.3 REPLACEMENT ALGORITHM - ALIGNED LRU (ALRU): A VARIANT OF LRU

Performance of a way – prediction set-associative cache can be improved by reducing number of mispredictions and cache misses. To achieve this, a new replacement strategy named Aligned LRU (ALRU) is employed, which aligns the data pages in the same cache block wherever possible.

//Algorithm for Cache Placement / Replacement for way – predictive placement scheme

Input: LRU data structure, Global MRU bits.

Output: Line number of the Cache line to be evicted (victim line) if CACHE MISS

Search space: All cache lines (N) in a set S.

ON EVERY REFERENCE IN A CACHE SET

begin

predict = Global MRU bits

if (Reference is CACHE HIT in i^{th} Cache line) then

for j = 0 to N-1

if (LRUcount[j] > LRUcount[i]) LRUcount[j] = LRUcount[j] – 1;

LRUcount[i] = N-1;

if (LRUcount[predict] < N/2) then

Victim line number = predict

else

Victim line number = Cache line number whose LRUcount value is zero

end

The replacement algorithm used in way – predictive placement scheme is a variant of LRU, which replaces one of the LRU pages in the cache set. If the last predicted cache block is one of the candidates for replacement (i.e. the block with LRU count less than (N/2)), then select that cache block as the victim block to load data from primary memory. This guarantees that most of the cache blocks are aligned to the predicted cache way and thus further improve the prediction accuracy.

The prediction correctness can increase with cache page size and can decrease with associativity. Theoretically, with increase in associativity, power consumed approximately reduces by a factor of (N-1) / N for best case, where N is the cache associativity. The power saved in cache increases with increase in block size, as more number of references will fall in the same block. It is not recommended to increase the cache size beyond a limit because of the data transfer delays (miss penalty) and the load capacitance. The worst-case power consumption by any way prediction cache scheme is the same as that of the traditional cache. Similar to the proposed scheme, way – prediction cache also has more average access time than conventional cache. Way –

predictive placement cache has lesser average access time because of zero table lookup time and better prediction accuracy.

4.4.4 WAY – PREDICTIVE PLACEMENT CACHE ENERGY AND ACCESS TIME ANALYSIS

The equations for average energy consumption and average cache access time for N – way set associative way – predictive placement cache are the same as for way – prediction cache except the fact that prediction hit rate is higher and table lookup power is saved. The prediction hit rate is high because of the Aligned LRU replacement strategy implementation with the help of global prediction bits. The average energy consumption ($E_{WPP\text{-}N\text{-}SACache}$) and the average cache-access time ($T_{WPP\text{-}N\text{-}SACache}$) for the N-way set-associative Way – predictive placement cache (WPP-N-SACache) are as follows:

$$\mathbf{E_{WPP\text{-}N\text{-}SACache} = (E_{Tag} + E_{Data}) + (1 - PHR) * ((N\text{-}1)E_{Tag} + (N\text{-}1)E_{Data})}$$

$\mathbf{T_{WPP\text{-}N\text{-}SACache} = 1 + (1 - PHR)}$ Where PHR is prediction-hit rate.

Energy analysis using energy dissipation per cache access model

Energy efficiency of proposed scheme is measured as the difference in energy dissipation by the different caching schemes. In this work the energy dissipation per cache access model proposed by Zhang et al. [Zhang 2005] is considered.

E_{dec} represents the energy dissipation of the address decoder

E_{mux} represents the energy dissipation of the multiplexer and output driver

$E_{tagline}$ represents the energy dissipation of one tag line

$E_{dataline}$ represents the energy dissipation of one data line

$E_{preline}$ represents the energy dissipation of one-line's precharging

$E_{comline}$ represents the energy dissipation of one-line's comparator

E_{saline} represents the energy dissipation of one-line's sense amplifier circuit

E_{tagway} represents the energy dissipation of one tag way

$E_{dataway}$ represents the energy dissipation of one data way

E_{preway} represents the energy dissipation of one way's precharging

E_{comway} represents the energy dissipation of one way's comparator

E_{saway} represents the energy dissipation of one way's sense amplifier circuit

E_{tab} represents the energy dissipation of table lookup

The energy equations of conventional, way prediction and way predictive placement 4 - way set-associative cache can be computed as follows:

$E_{way} = (E_{tagline} + E_{dataline} + E_{preline} + E_{comline} + E_{saline})*$
cachesize/(associativity * cachelinesize) →(4.1)

Conventional cache (E_{con})

$E_{con} = E_{dec} + E_{mux} + (4 * E_{way})$ →(4.2)

Way Prediction cache,

Prediction Hit Energy (E_{Phit})

$E_{Phit} = E_{dec} + E_{mux} + E_{way} + E_{tab}$ →(4.3)

Prediction miss (E_{Pmiss}) and Cache Miss (E_{Cmiss})

$E_{Pmiss} = E_{Cmiss} = 2*(E_{dec} + E_{mux} + E_{tab}) + (4* E_{way})$ →(4.4)

Total Energy

$E_{Total} = Phit * E_{Phit} + (1–Phit – Cmiss) * E_{Pmiss} + Cmiss * E_{Cmiss}$ →(4.5)

Way Predictive placement cache,

Prediction Hit Energy (E_{Phit})

$E_{Phit} = E_{dec} + E_{mux} + E_{way}$ →(4.6)

Prediction miss (E_{Pmiss}) and Cache Miss (E_{Cmiss})

$E_{Pmiss} = E_{Cmiss} = 2*(E_{dec} + E_{mux}) + (4* E_{way})$ →(4.7)

Total Energy

$E_{Total} = Phit * E_{Phit} + (1–Phit – Cmiss) * E_{Pmiss} + Cmiss * E_{Cmiss}$ →(4.8)

4.4.5 EXPERIMENTAL SETUP, RESULTS AND DISCUSSION

4.4.5.1 Experimental setup

For the above calculations, Simplescalar 2.0 [Burger 1997] cache simulator was employed with different cache configurations. SPEC95 benchmark programs [SPEC95] are used to obtain the prediction hit rate and the number of tag comparisons required to

execute each of the selected programs. The selection of SPEC95 benchmark program suite guarantees uniformity in evaluation as most of the existing cache architectures used this benchmark program suite for evaluation.

Input: Cache configuration (shown in Table 4.1) and the trace file from SPEC95 benchmark programs (shown in Table 4.2) to be used to emulate cache behavior.

The different parameters that were varied are as shown in Table 4.1.

Table 4.1: Different cache configurations

Cache Parameters	**Range**
Cache size	8K (in Bytes)
Block size	32, 64, 128 (in Bytes)
Cache associativity	2, 4, 8

Table 4.2: SPEC 95 Benchmark program traces used for experimentation

Exp. No.	**Benchmark Program**	**Exp. No.**	**Benchmark Program**
1	applu	7	mgrid
2	compress95	8	tomcatv
3	fpppp	9	su2cor
4	hydro2d	10	swim
5	ijpeg	11	vortex
6	m88ksim	12	Wave5

Cache Algorithm: Way – predictive placement scheme

Replacement algorithm: a variant of LRU – Aligned LRU (ALRU)

Output:

To estimate the prediction accuracy: Number of hits in first cycle, number of hits in second cycle (mispredictions)

To determine the algorithm efficiency: Total number of cache hits and cache misses

To provide an estimate of the power consumed: Number of tag comparisons required.

4.4.5.2 Results and discussion

Figure 4.5 to 4.7 shows the percentage of tag comparisons in a conventional cache that is required for a 4-way set associative way predictive placement cache with 8KB cache size and 32B, 64B and 128B cache line sizes respectively. This experimentation measures the number of tag comparisons for various SPEC 95 benchmark program suite with 2, 4, and 8 as associativity. From Figure 4.5 to 4.7 one can observe that number of tag comparisons is different for different cache configurations. Increase in associativity results in lesser number of tag comparisons, thus higher savings.

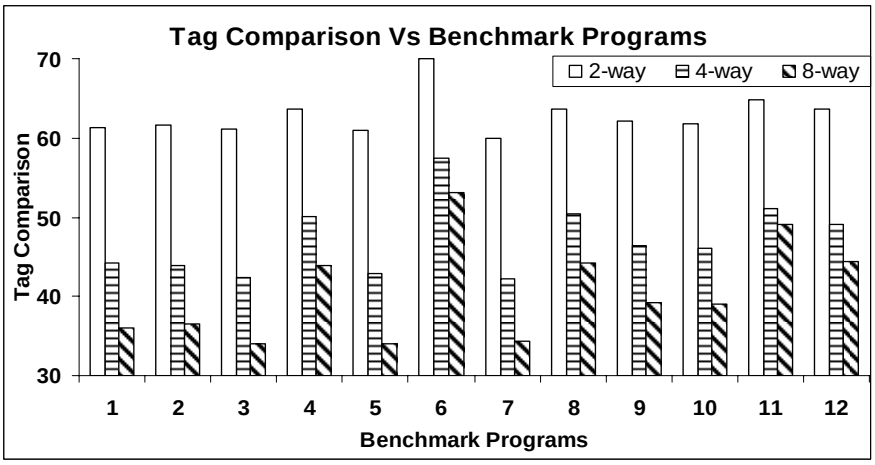

Fig. 4.5: Tag Comparisons for 8KB cache with 32byte cache line size

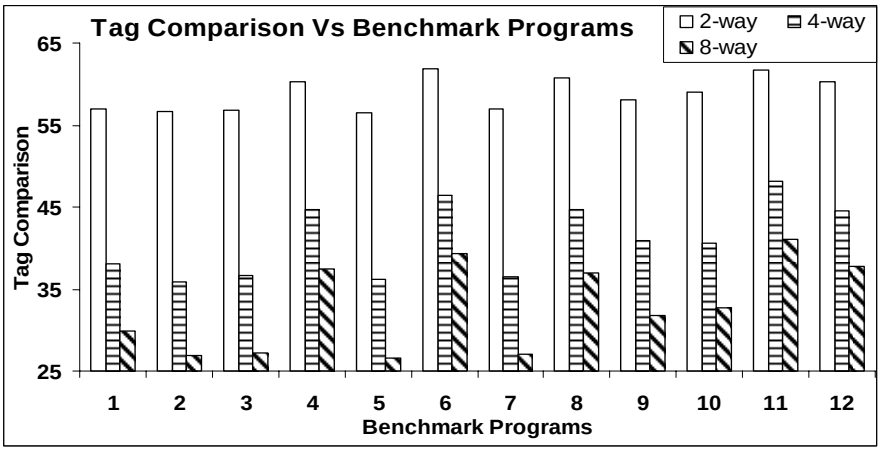

Fig. 4.6: Tag Comparisons for 8KB Cache with 64byte cache line size

Figure 4.8 to 4.10 shows the prediction hit rate of a 4-way set associative way predictive placement cache with 8KB cache size and 32B, 64B and 128B cache line sizes

respectively. This experimentation measures the number of prediction hits over total references for various SPEC 95 benchmark program suite with 2, 4 and 8 as associativity.

From Figure 4.8 to 4.10, one can observe that prediction hit rate reduces with associativity, but still accomplishes the objective of providing an improvement over the way – prediction scheme. Because of low prediction hit rate, higher associativity results in high average access time. In all these cases, power saving of this method is better than conventional cache and way prediction cache. The results are summarized in Table 4.3.

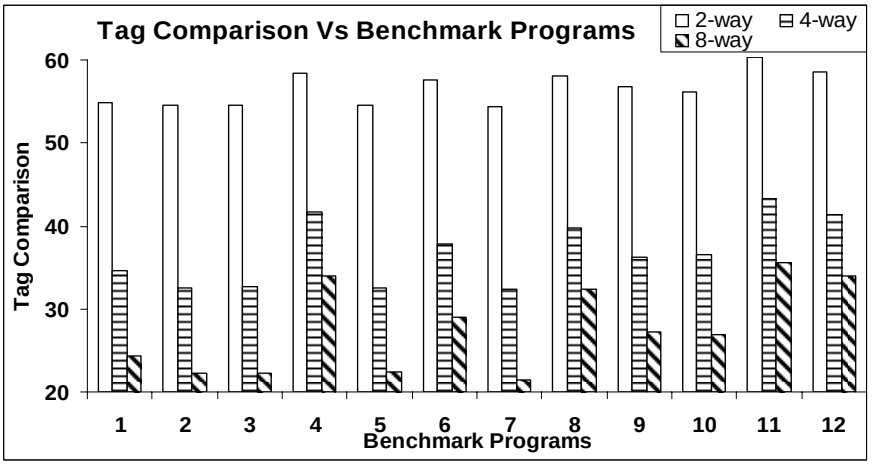

Fig. 4.7: Tag Comparisons for 8KB Cache with 128byte cache line size

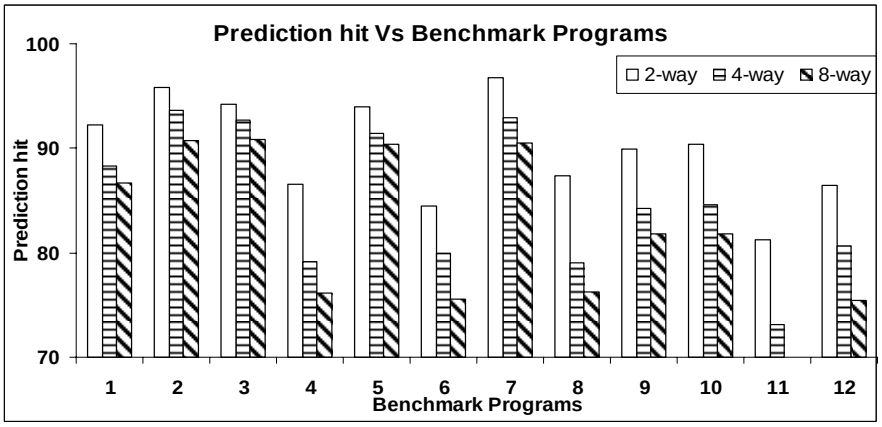

Fig. 4.8: Prediction hit rate for 8KB Cache with 32byte line size

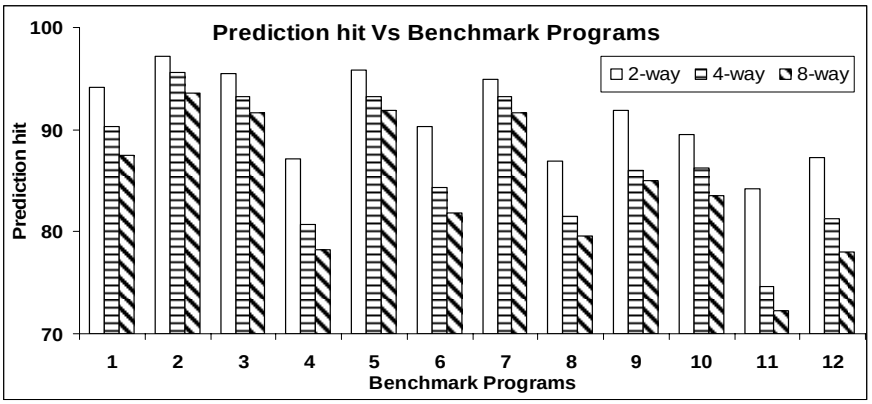

Fig. 4.9: Prediction hit rate for 8KB Cache with 64byte line size

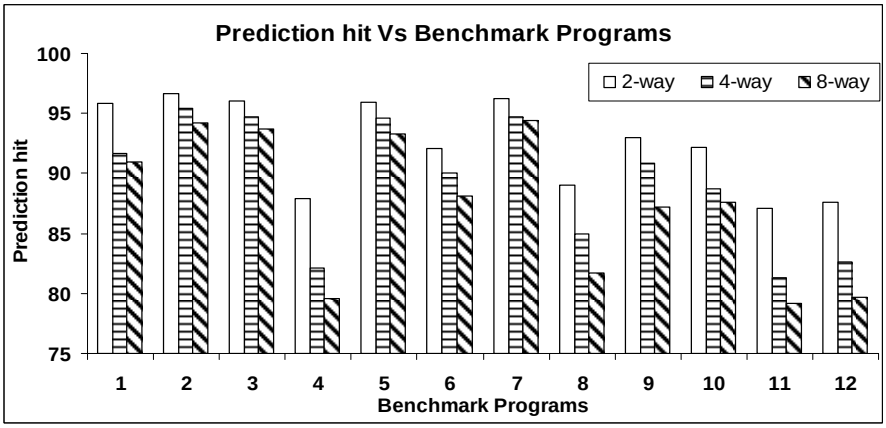

Fig. 4.10: Prediction hit rate for 8KB Cache with 128byte line size

Table 4.3: Comparison of the way predictive-placement scheme with conventional way prediction scheme

Data Cache	Way Predictive Placement Cache	Way Prediction Cache [Inoue 1999]
Average Cache Predict Hit	90.31%	86%
Average Cache Hit	92.92%	NA
Average Tag Comparison	32.26%	NA
Average Increase in Effective Cache Access Time	9.69%	12.975%
Average Energy Saving	67.75%	64.75%

4.4.5.3 Way – predictive placement Cache Vs Way-prediction Cache

For a 4 – way set-associative cache, the average results of way – predictive placement and way – prediction cache as compared with conventional set associative cache are as shown in Table-4.3. The prediction hit rate on an average, using the way-predictive

placement scheme, is 90.31% as compared to 86% for way prediction scheme. The prediction hit rate for conventional cache is always 100%, as all the ways will be available for comparison. The average cache hit for a way predictive placement scheme is 92.92%, which is almost the same as conventional and way prediction scheme. The average tag comparisons of way predictive placement scheme reduces by 67.75% compared to conventional cache, whereas in predictive placement scheme it reduces by 64.75%. The energy saving required for (N-1) tag arrays and data arrays, (N-1) comparators, and (N-1) sense amplifiers amounts to a reduction in 67.75% of energy compared to conventional cache. The energy saving reported for way prediction cache is 64.75%. For the prediction miss an extra cycle is needed to enable the remaining (N-1) ways. The average effective access time thus increases by 9.69% and 12.975% for way predictive placement scheme and way-prediction scheme respectively as compared to conventional set associative cache. The performance improvement of the predictive-placement scheme, with respect to prediction hit rate and energy saving is 5% and 4.63% respectively than the way-prediction cache.

From the simulation result, it is evident that due to high prediction accuracy the proposed scheme saves 67.75% of energy consumption as compared to conventional cache. The improvement in energy saving is achieved due to the replacement policy ALRU and by eliminating the table lookup. The proposed scheme reduces the hardware requirements as it reduces the prediction bits from Number of cache sets * $\log_2$ N to $\log_2$ N. The role of additional hardware is mainly for selectively enabling and disabling the tag and data arrays which is needed for way prediction scheme as well. In case of a conventional cache organization, all the tag and data arrays are enabled for every data access and hence the enabling / disabling circuit is not required.

CHAPTER 5

PROCESS AWARE SELECTIVE PLACEMENT SCHEMES

5.1 INTRODUCTION

This chapter focuses on process aware energy-efficient cache design for Embedded Systems. This chapter presents two new software controlled energy-efficient process-aware caching schemes for an N-way set-associative cache: (i) a process-aware selective placement scheme (PASP) with a victim set and (ii) a shared memory process-aware selective placement (SMPASP) scheme with small shared and victim sets. These two schemes aim at reducing the power consumption and improving the cache hit rate of process aware caches. This chapter elaborates on the working of the proposed process aware cache designs with placement and replacement support. In this chapter, the proposed schemes are evaluated and compared with the conventional set-associative cache and way prediction cache with respect to the cache hit rate, dynamic and leakage power consumption, with the tag comparison count in particular, for various cache sizes, cache line sizes and context switch durations. This performance evaluation is carried out with independent processes and processes which exhibit a considerable amount of data sharing among them. This chapter evaluates the number of tag comparisons and the number of hits obtained for the main cache, victim cache and shared cache separately for various cache configurations (cache size, cache line size and context switch duration) by using a cache simulator CACHEMEM 1.0 in conjunction with the traces of SPEC 95 benchmark suite programs extracted using the SimpleScalar 2.0 simulator. The dynamic and leakage power consumption for the various caching schemes are obtained using eCACTI cycle-based power estimation model.

5.2 WHY PROCESS AWARE CACHE DESIGN?

The conventional N-way set-associative data cache enables all the N tag ways for parallel comparisons while searching the requested data in the cache which results in high dynamic power consumption. One way to minimize the dynamic power consumption is to minimize the internal cache activity during cache access. An ideal situation on a cache hit is to read and compare only one tag entry and access one data entry, whereas on a

cache miss, it is to read and compare only one tag entry. This can be achieved by accessing the set-associative cache as a direct-mapped cache, which requires only one tag comparison to find if the referenced data is available in the cache.

This can be achieved by using a way prediction cache, but in the way prediction cache, the cache lines can be placed in any of the sub banks (usually one sub bank is a cache way). This makes it impossible to put sub banks into deep sleep mode, as the time taken for wake-up from deep sleep mode to active mode is large. This adversely affects the power consumption and performance of the way prediction cache. One solution to this problem is to dedicate one sub bank per process and to switch between the sub banks only when a context switch occurs. This is achieved in the proposed work by assigning one dedicated cache way to one process. This converts the N-way set-associative cache into a direct-mapped cache for the currently running process and all the other cache ways can be in deep sleep mode, until a context switch takes place. The context switching information has to be passed on to the cache controller using special instructions. This results in saving a significant amount of cache power consumption.

Increased power saving is also achieved by reducing the number of conflict misses i.e., by cutting down on the miss penalty overhead. Shutting down N-1 cache ways for a long time results in saving power, but making use of only $1/N^{th}$ of the cache size for a process may degrade the cache hit performance and thus, the increased power consumption. In both Process Aware Selective Placement (PASP) and Shared Memory Process Aware Selective Placement (SMPASP) designs, a process-aware, software-controlled way-selective placement cache mechanism with a victim set is used to improve the hit rate. These schemes thus improve the cache hit rate by providing a small victim set for the spill-out data from the main cache and by modifying the replacement scheme.

Most of the current day Embedded System applications are multithreaded with a large amount of data sharing among the threads, which, if properly exploited may yield a better cache hit rate. A software-controlled cache is used in the proposed scheme to improve the performance, wherein the operating system can partially control the availability of data in the cache by transferring process-related and data sharing-related information to the cache controller for improving the cache hit rate. To facilitate the efficient handling of shared data among processes, the SMPASP scheme is proposed. This scheme improves

the cache performance by providing a small shared set for the shared data among processes and by modifying the replacement scheme.

5.3 CONVENTIONAL AND WAY – PREDICTION CACHE ARCHITECTURES

The proposed schemes (PASP and SMPASP) are compared with the 4-way set-associative conventional (Conv) and way prediction (WP) schemes with respect to the cache hit rate, number of tag comparisons, dynamic, leakage and the total power consumption for various cache sizes, cache line sizes and context switch durations. The conventional cache architecture for a 4-way set-associative cache is shown in Figure 5.1.

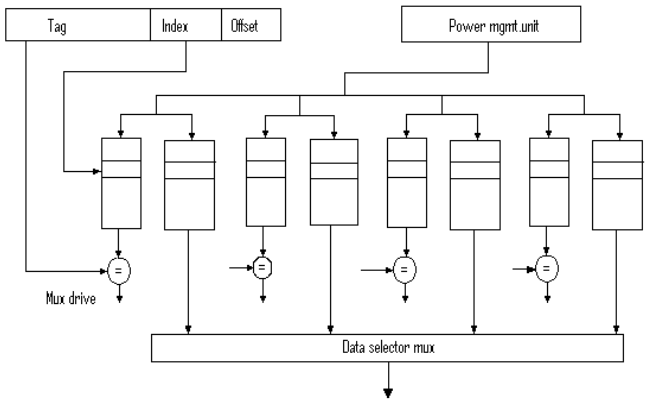

Fig. 5.1: Conventional 4 – way set-associative cache (Conv)

Consider an 8KB, 4 – way set-associative cache with 64 – byte line size. The cache line has 32 sets, thus the memory address will have a 6-bit offset, 5-bit index and remaining 21-bits of tag fields. This architecture includes a 5:32 address decoder for tag and data, word line drivers, four tag arrays, four data arrays, sense amplifiers, comparators, read and write column multiplexers, multiplexer drivers, one output multiplexer and output drivers. The algorithm of the Conventional caching scheme is as discussed below.

5.3.1 ALGORITHM FOR CONVENTIONAL CACHE

//Cache operations to be carried out on every reference

begin

Look up the virtual page number in TLB

if (TLB HIT) then

Derive tag (t), index (i) and offset (o) from physical address fetched from TLB.

if (Reference is CACHE HIT in any one of the cache line in i^{th} set) then

```
            Read / Write data from / to offset location;
            Update Replacement circuit of ith set accordingly
        if (Reference is CACHE MISS in ith set) then
            Transfer data from Main memory to the victim line (selected by the replacement
            strategy in use) of ith set
            Read / Write data from / to offset location;
            Update Replacement circuit of ith set accordingly
    if (TLB MISS) then
            Generate trap to handle TLB miss
end
```

5.3.2 WORKING OF THE CONVENTIONAL CACHE

A data access request to the conventional N-way set-associative cache is handled as follows. On every cache reference, search for the virtual page number in the TLB. If the virtual page number is available in the TLB, fetch the corresponding physical page address from the TLB. The offset bits of the requested address are used to find the exact byte in a cache line after locating the same. The index bits are fed as input to the decoder. Each decoder output line is used to activate one cache set, i.e., N data and tag arrays of the N-way set-associative cache. 'N' data and tag arrays are read out simultaneously through the sense amplifier. N comparators compare the selected address tags in parallel to find which one, if any, out of the N tags is matching with the tag bits of the requested address. The data corresponding to the tag match is selected using the multiplexer and output driver and the replacement circuitry of the corresponding set is updated accordingly. If none of the tags matches then cache line to be evicted is found using the replacement circuit of the corresponding set. Transfer the requested data from the main memory to the selected cache line and update the replacement circuit of the set accordingly. Data is read from the offset location of the selected cache line.

5.3.3 WAY – PREDICTION CACHE

A detailed explanation of the way-prediction cache proposed by Inoue et al. [Inoue 1999] was provided in the Section 4.2. In this scheme, cache line selection is based on the MRU information of the index corresponding to which N bits per cache set are used to store the Most Recently Used (MRU) way information. For the cache configuration

example in section 5.2, 64 bits are needed to store the MRU information. This information is stored as a way-prediction table of 32 entries. The data is selected only if the tag bits of the required address matches with the tag value of the predicted way index bits. If the prediction is a miss, then in the next cycle, the remaining 3 tag ways and data ways are enabled for comparison. This results in spending an extra cycle to find the availability of data in the cache and hence, performance is compromised. Non-availability of data in cache causes cache miss. The replacement algorithm based on the Least Recently Used (LRU) policy decides the victim cache line. This scheme requires 64 bits to save the most recently used (MRU) way information and extra cycles for both prediction miss and cache miss.

5.4 PROCESS AWARE SELECTIVE PLACEMENT CACHE ARCHITECTURE

One of the major concerns in the conventional memory architecture is that the cache is transparent to the operating system and application programs. Recent studies suggest the necessity of cache – compiler – operating system – application program interaction to improve the performance. The interaction can reduce cache power consumption by accurately predicting the cache set where the required data is available, thus deactivating the other ways [Yang 2005]. The interaction can also improve cache predictability and performance by helping the selection of the victim cache line with minimum modification in the replacement circuitry [Jain 2001][Wang 2002][Sartor 2005]. The proposed scheme uses software-controlled cache for reduction in power consumption by shutting down N-1 cache ways and reducing the conflict misses. Here, a process-aware, software-controlled way-selective placement cache mechanism with a victim set is employed. This scheme thus improves the cache hit rate and reduces the power consumption by providing a small victim set for the spill-out data from the main cache and by modifying the replacement scheme.

The proposed design, PASP scheme is as shown in Figures 5.2 to 5.5. Figure 5.2 shows the PASP cache architecture for the 4-way set-associative cache architecture. Figures 5.3 to 5.5 shows the various cache-related operations associated with the PASP cache where the highlighted parts represent the active cache blocks.

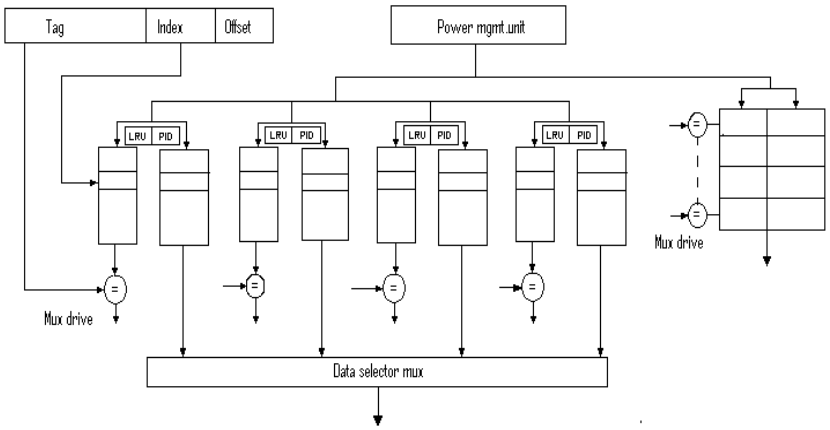

Fig. 5.2: PASP Cache Architecture

This scheme has an N-way set-associative cache (N=4 here), a small fully associative victim cache of 3 to 5 cache lines, one 8 – bit register per cache way to store the process-related information, $\log_2 N$ – bits per cache way to store the LRU way information and a power management unit to activate / inactivate the selected cache ways and the victim set, as shown in Figure 5.3. The fully-associative victim set is available for all the processes and is used to collect the spill-out pages from the main cache. This set improves the cache hit rate, with the help of the FIFO replacement policy.

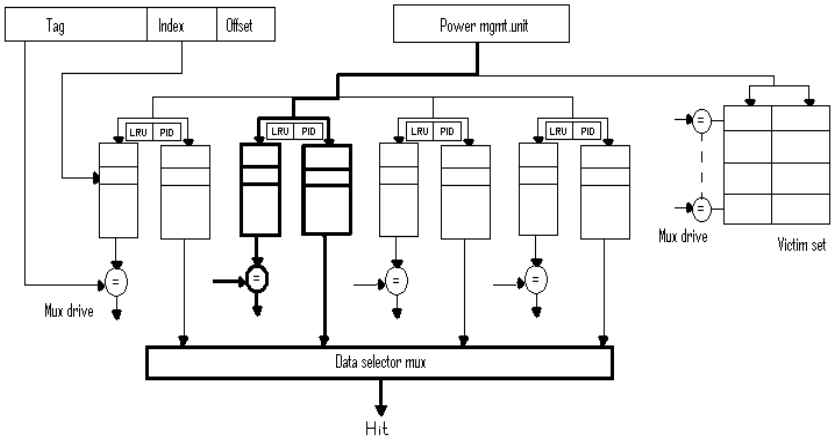

Fig. 5.3: Cache Hit in dedicated cache way

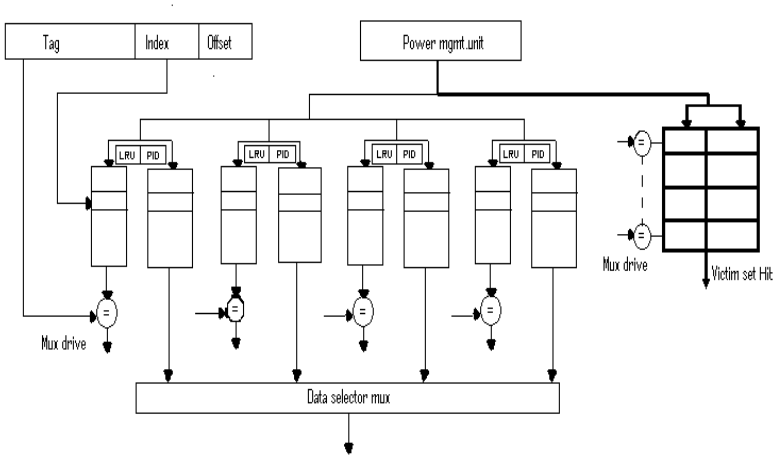

Fig. 5.4: Cache Miss in dedicated cache way, checking in the Victim set for data availability

In the proposed scheme, one way is assigned to one process. The data pages corresponding to the currently executing process are available in either this dedicated way or the victim set. This results in enabling either only the assigned way or the victim set for tag comparisons, as shown in Figure 5.3 and Figure 5.4 respectively. For better power optimization, the entire main cache is disabled while searching the data in the victim set, as highlighted in Figure 5.4. The ways which do not belong to the currently executing process are shut down to save dynamic power consumption. A significant amount of dynamic power saving is possible with this scheme.

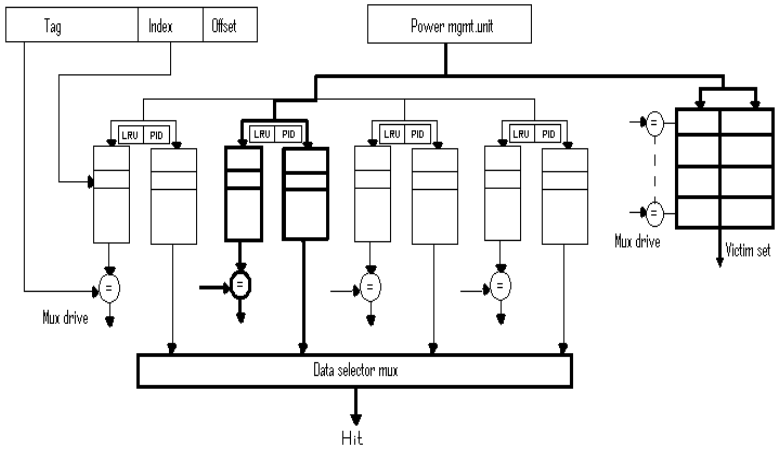

Fig. 5.5: Cache Hit in Victim set (Transferring data to the dedicated cache way – works in the same way as for a Cache Miss)

For a static process scheduling scheme, the process-related information is available at compile time itself, whereas in a dynamic process scheduling scheme, this information is available at runtime only. In this scheme, the operating system transfers the process information to the cache controller with the help of special instructions [Jain 2001][Sartor 2005]. With this information, the cache controller enables different ways for different processes. The context switch routine of the operating system issues special instruction(s) for transferring the process identification information to the cache controller. If the process identifier already exists in one of the ways' register, then the corresponding way is powered up and remaining N-1 ways are powered down. If the process identifier information is not available in any of the registers, then the cache controller finds the way with the minimum LRU value and allocates the way to the process by invalidating all the existing data in that way. Whenever a process terminates, the operating system notifies the cache controller, so that the LRU count of that way is set to zero and all the data in that way is invalidated. By default, the victim set is in the sleep state and is activated only if the requested data is not available in the dedicated way assigned to the process.

5.4.1 PASP ALGORITHM

The algorithm of the Process Aware Selective Placement (PASP) caching scheme is as discussed below.

```
//Cache operations to be carried out on every reference
begin
    Look up the virtual page number in TLB
    if (TLB HIT) then
        Fetch physical address from TLB
        Derive tag (t), index (i) and offset (o) for the Main cache.
        Derive tag (tv) and offset (ov) for the Victim set
        WaySelect = Dedicated Cache way of the currently running process
        WaySelectth way enabled; All other N-1 ways in deep sleep; Victim set in sleep
        CYCLE = 1
        while (CYCLE < 4) do
            if (CYCLE = 1) then
                if (Reference is CACHE HIT in ith set, WaySelectth way) then
                    Read/Write data from/to offset location of ith block in WaySelectth way
```

```
                CYCLE = 4
            else
                CYCLE = 2
                Enable Victim set
        if (CYCLE = 2)
            if (Reference is CACHE HIT in Victim set) then
                Transfer ith block in WaySelectth way to FIFO location in Victim set
                Transfer CACHE HIT Victim set block to ith block in WaySelectth way
                Read/Write data from/to offset location of ith block in WaySelectth way
                Update FIFO replacement circuit of Victim set
                CYCLE = 4
            else CYCLE = 3
        if (CYCLE = 3)
            Transfer ith block in WaySelectth way to FIFO location in Victim set
            Transfer data from Main memory to ith block in WaySelectth way
            Read/Write data from/to offset location of ith block in WaySelectth way
            Update FIFO replacement circuit of Victim set
    if (TLB MISS) then
        Generate trap to handle TLB miss
end
```

The process aware selective placement (PASP) set-associative cache works as follows. The cache obtains the physical address of the requested reference in the same manner as the conventional cache. For every cache access, only the dedicated cache way of that process is enabled with the help of the power management unit and cache controller. In this manner, the set-associative cache is virtually acts as a direct-mapped cache. The corresponding line index of the dedicated cache way is searched for a tag match, as shown in Figure 5.3. On a cache hit, the data from the corresponding offset location is accessed. This results in 75% dynamic power reduction as compared to a conventional 4-way set-associative cache. A tag miss in this scheme can be either because of a miss in the dedicated cache way, but a hit in the victim set (Primary miss but secondary hit) or because of a secondary cache miss, where the requesting data is not available in the dedicated cache way as well as in the victim set. Every prediction miss in the dedicated cache way causes the victim set to be enabled, as shown in Figure 5.4 by the power

management unit and control circuitry in the next cycle for tag comparisons. If any of the tags in the victim set matches, then the corresponding cache line of the victim set is swapped with the indexed location of the dedicated cache way (Figure 5.5). The FIFO bits of the victim set lines are updated accordingly. In the case of a miss in the victim set, the required page is copied from the main memory to the indexed location of the dedicated cache way and any existing valid cache line in corresponding location of the dedicated way is moved into the victim set with the FIFO bits of the victim set lines being updated accordingly.

5.4.2 POWER SAVING CONCERNS

This implementation has the advantage of a direct mapped cache, which simplifies the circuit and mainly helps in reducing the dynamic power consumption caused by additional tag comparisons and the replacement circuitry. The cache access time of the PASP cache during a cache hit is the same as that of a direct-mapped cache. The proposed scheme provides a better performance (cache hit rate and dynamic power consumption) for processes with frequent context switches, as the recently used previous process cache lines are still available in its dedicated cache way. This scheme is power efficient, as it is found from the experimentation that about 85% of the data cache references are directly from the dedicated cache way and very few references are directed to the victim set. The victim set is enabled only when the data is not available in the dedicated cache way and it helps in improving the cache hit rate, thus saving the dynamic power consumption significantly.

The size of the victim set influences the hit rate and the power consumption. If the size of the victim set is more, the overall hit rate of the cache system increases, but the number of parallel comparators and the dynamic power required also increases, which may degrade the performance. So the size of the victim set should be a balanced compromise between the number of comparators and the miss rate. Theoretically, with an increase in the associativity, the dynamic power consumption approximately reduces by a factor of (N-1) / N for the best case, where N is the cache associativity. The power saving will be more for a cache with a larger cache line size because of the improved hit rate, as large number of references fall within the same page.

5.4.3 REPLACEMENT SUPPORT

The replacement support is critical to achieve high performance. The PASP cache uses direct replacement in the dedicated cache way similar to a direct-mapped cache resulting in the dynamic power saving and reduced circuit complexity. Any spill-out cache line from the main cache is moved to the victim set. The victim set follows FIFO replacement policy, which makes the replacement circuitry simple. During a victim set hit, whenever a cache line is transferred from the victim set to the corresponding main cache's dedicated way, that cache line in the victim set is made invalid. If the dedicated cache way has a valid cache line, then that cache line is moved to the victim set and the FIFO bits of the victim set lines are modified appropriately.

5.4.4 LIMITATION OF THE PASP SCHEME

The PASP scheme reduces the dynamic power consumption significantly, but the performance of this scheme is affected when the required data is shared among many processes. In other words, the same cache data block may exist in more than one way at the same time, thus resulting in coherency issues.

For instance, say, two processes P1 and P2 are sharing a cache line S1. Assume the execution sequence is P1 accessing the shared line S1, followed by a context switch to P2 and then P2 accessing the same shared line S1. When P1 accesses the shared line S1 for the first time, a cache miss occurs and S1 is placed in the way assigned to P1, say, way 0. When a context switch occurs and P2 starts its execution with its dedicated cache way as way 1, a request generated for cache line S1 results in a cache miss, as S1 is not available in way 1 and in the victim set. Thus, a copy of S1 from the main memory is placed in way 1. This results in having the same cache line in more than one way at the same time.

In the PASP scheme, to avoid cache coherency problem, the system checks for the existence of the same cache line in other ways and invalidate them, before copying the corresponding shared line to the cache line of the currently executing process's dedicated way. This adds an overhead to the system, as all the ways have to be powered up for tag comparisons resulting into performance degradation for the PASP set-associative cache with a large amount of shared data. To overcome this performance overhead, a new

Shared Memory Process Aware Selective Placement (SMPASP) caching scheme is proposed.

5.5 SHARED MEMORY PROCESS AWARE SELECTIVE PLACEMENT (SMPASP) CACHE ARCHITECTURE

This section explains a Shared Memory Process Aware Selective Placement (SMPASP) scheme for an N-way set-associative cache, designed to efficiently handle shared data among processes, using a small shared set and victim set. This scheme is primarily designed to avoid coherency issues encountered when using the PASP scheme for processes sharing data among them by allocating a separate small shared set to hold the shared data. It aims at reducing the dynamic power consumption of the set-associative cache architecture and enhancing performance with respect to the cache hit rate.

Figure 5.6 represents the Shared Memory Process Aware Selective Placement (SMPASP) scheme for the 4-way set-associative cache architecture. Figures 5.7 to 5.11 shows the various cache-related operations associated with the SMPASP cache (the highlighted parts in the figures represent the active cache blocks). Similar to the PASP cache, the Shared Memory Selective Placement Cache also has a small fully associative victim set for collecting spill-out pages from the main cache. The Shared Memory Selective Placement Cache has an additional small set called the shared set to hold the shared data.

This proposed cache architecture has two variations, based on the mapping scheme – direct-mapped as shown in Figure 5.7 or 2-way set-associative as shown in Figure 5.8 shared set. The Shared Memory Selective Placement Cache architecture contains four registers (8-bit) to store the process identifiers, four 2-bit LRU registers for finding the least recently used way, one decoder for the main cache, one decoder for the 2-way set-associative / direct-mapped shared set (5:32 decoder for the 8KB main cache, 3:8 / 4:16 decoder for the 2-way set-associative / direct-mapped shared set), word line drivers, relevant number of tag arrays, data arrays and comparators, sense amplifiers, one / two multiplexer(s) and output drivers.

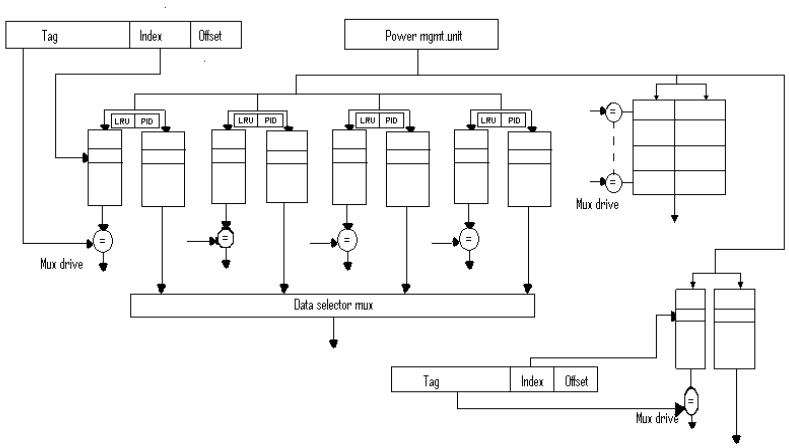

Fig. 5.6: SMPASP Cache Architecture with Direct – mapped Shared set.

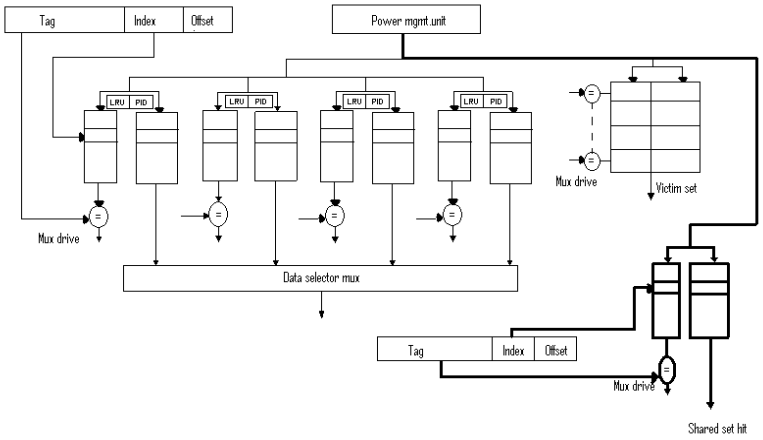

Fig. 5.7: Cache hit in Shared set (Direct-mapped).

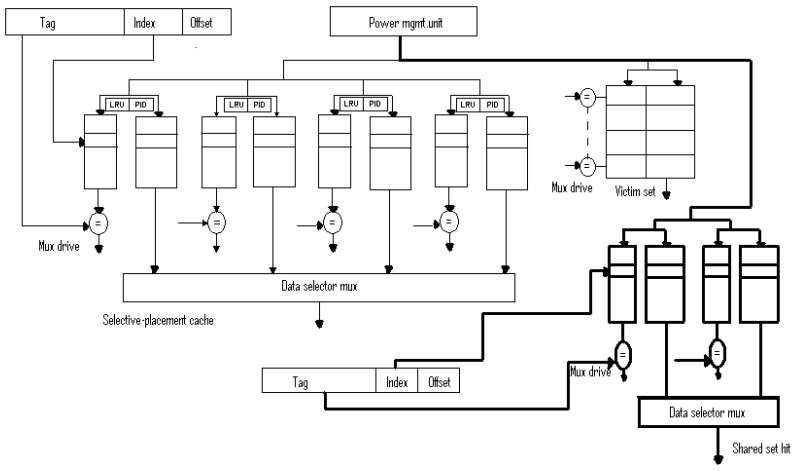

Fig. 5.8: Cache hit in Shared set (2 – Way set-associative mapping).

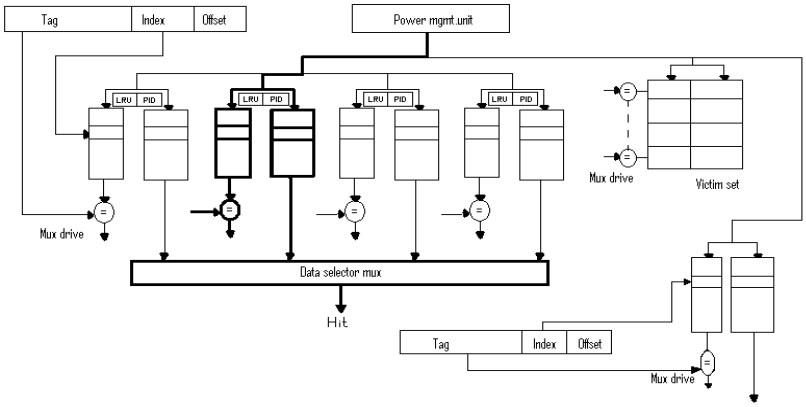

Fig. 5.9: Cache hit in dedicated cache way (non-shared data).

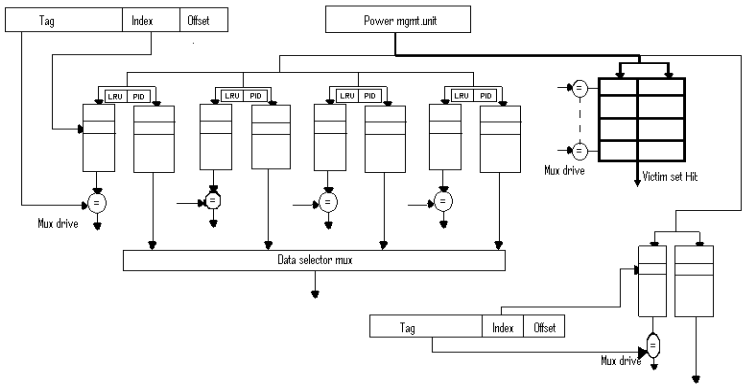

Fig. 5.10: Cache miss in dedicated cache way, checking the victim set for data availability (non – shared data)

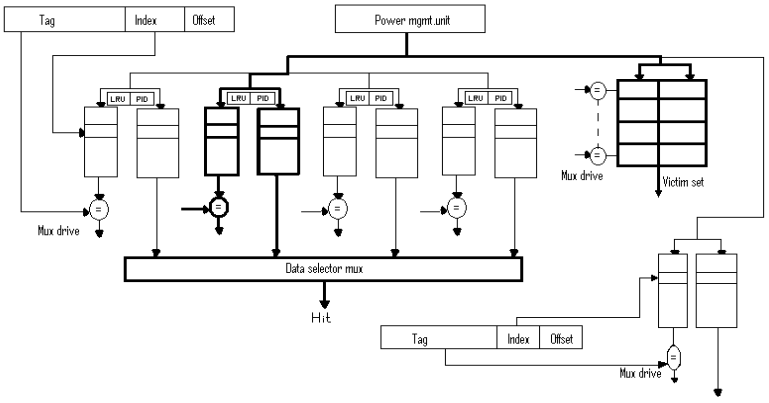

Fig. 5.11: Cache Hit in Victim set (non – shared data; transferring data to the dedicated cache way – works in the same way as for a Cache Miss)

In this scheme, an incoming reference is classified into a shared or a non-shared data reference with the help of compiler information. In case of a cache hit, the non-shared data is available either in the dedicated way of the main cache or in the victim set. The shared data is available only in the shared set. The compiler and the operating system are modified to be intelligent enough to provide the above mentioned sharing information of the incoming data reference and the process-related information to the cache controller using special instructions [Jain 2001][Sartor 2005]. This information is used to enable / disable the cache ways, and to select either the main cache or the shared set for cache line replacement, during a cache miss.

Whenever a context switch occurs or a new process is executed, the operating system passes the process identifier information to the cache controller. The cache controller compares the new identifier value against a 8-bit value stored in each of the 'N' registers. If any of the values matches, then the corresponding way is enabled and all the other N-1 cache ways are disabled. The LRU count of this enabled way is set to the maximum and the LRU count of all the other ways are modified accordingly. If the new process identifier value does not match with any of the existing values in the registers, then the cache way with the minimum LRU count is selected and assigned to this process by storing the process identifier value in the 8-bit register of the selected way, after invalidating all the cache lines of the selected way and modifying the LRU count of all the other ways accordingly.

Whenever a process terminates, the operating system transfers this information to the cache controller. The cache controller sets the LRU count of the way dedicated to that process to the minimum and invalidates all the cache lines of that way.

5.5.1. SMPASP ALGORITHM

The algorithm of the shared memory process aware selective placement (SMPASP) caching scheme is as follows.

```
//Cache operations to be carried out on every reference
begin
    Look up the virtual page number in TLB
    if (TLB HIT) then
        Fetch physical address from TLB
```

Derive tag (t), index (i) and offset (o) for the Main cache.

Derive tag (t_v) and offset (o_v) for the Victim set

Derive tag (t_s), index (i_s) and offset (o_s) for the Shared set.

WaySelect = Dedicated Cache way of the currently running process

if (Reference is Shared) then

WaySelectth way in sleep; All other N-1 ways in deep sleep

Victim set in sleep; Shared set Enabled

if (Reference is CACHE HIT in Shared set) then

if (Shared set is SET ASSOCIATIVE) then

Read/Write data from/to offset location of CACHE HIT block in i_s th set of the Shared set

Update LRU Replacement circuit of i_s th set in Shared set

if (Shared set is DIRECT MAPPED) then

Read/Write data from/to offset location of CACHE HIT block (i_s th block) of the Shared set

if (Reference is CACHE MISS in Shared set) then

if (Shared set is SET ASSOCIATIVE) then

Transfer data from Main memory to LRU block of ith set in Shared set

Read/Write data from/to offset location of LRU block in i_s th set of the Shared set

Update LRU Replacement circuit of i_s th set in Shared set

if (Shared set is DIRECT MAPPED) then

Transfer data from Main memory to ith block in Shared set

Read/Write data from/to offset location of i_s th block of Shared set

if (Reference is NOT Shared)

WaySelectth way Enabled; All other N-1 ways in deep sleep

Victim set in sleep; Shared set in sleep

CYCLE = 1

while (CYCLE < 4) do

if (CYCLE = 1) then

if (Reference is CACHE HIT in ith set, WaySelectth way) then

Read/Write data from/to offset location of ith block in WaySelectth way

CYCLE = 4

else

CYCLE = 2

```
                Enable Victim set
            if (CYCLE = 2)
              if (Reference is CACHE HIT in Victim set) then
                Transfer ith block in WaySelectth way to FIFO location in Victim set
                Transfer CACHE HIT Victim set block to ith block in WaySelectth way
                Read/Write data from/to offset location of ith block in WaySelectth way
                Update FIFO replacement circuit of Victim set
                CYCLE = 4
              else CYCLE = 3
          if (CYCLE = 3)
              Transfer ith block in WaySelectth way to FIFO location in Victim set
              Transfer data from Main memory to ith block in WaySelectth way
              Read/Write data from/to offset location of ith block in WaySelectth way
              Update FIFO replacement circuit of Victim set
    if (TLB MISS) then
        Generate trap to handle TLB miss
end
```

A data access request to the SMPASP cache is handled as follows.

When the CPU issues an address reference, the TLB is first checked to find the corresponding physical address. If the logical reference is a TLB hit, then the physical reference from the TLB is extracted and fed as input to the cache controller. If the logical reference is not available in the TLB, then a trap to handle the TLB miss is generated.

The cache controller has the sharing information about the cache reference beforehand by obtaining it from an earlier pipeline stage. Based on this information, with the help of a power management unit, the cache controller activates either a dedicated cache way of the executing process or the shared set. If the cache reference is a shared one, then the physical reference is for the shared set, which is either direct mapped or 2-way set-associative. If the cache reference is non-shared, then the physical reference is for the main cache or for the victim set.

For a non-shared data reference, in the first cycle, the dedicated cache way of the currently executing process is searched for the referenced data page (Figure 5.9). The tag value for the main cache is compared with the tag stored in the indexed line of that

dedicated way. If a cache hit occurs in the main cache, then the data is accessed from the offset location of the selected cache line in the same cycle. If the non-shared requested data is not available in the dedicated way, then the victim set is activated in the next cycle as shown in Figure 5.10 and searched for the availability of the requested cache line. The victim set follows fully associative mapping scheme. In the case of a cache hit in the victim set, the cache line in the indexed location of the dedicated way is moved to the FIFO location of the victim set and the requested data in the victim set is moved to the indexed location of the dedicated way as shown in Figure 5.11. The requested data is accessed from the offset location and the FIFO replacement circuit of the victim set is updated accordingly. If the referenced data is not available in the victim set, then the cache line in the index location of the dedicated way is moved to the FIFO location of the victim set and the data from the main memory is copied to the indexed location of the dedicated way. The requested data is accessed from the offset location and the FIFO replacement circuit of the victim set is updated accordingly.

For a shared data reference, the requested tag value is compared with the tag value stored in the shared set indexed line(s). The shared set may be direct-mapped as shown in Figure 5.7 or 2-way set-associative as shown in Figure 5.8. If a cache hit occurs in the shared set, then the data is accessed from the offset location of the selected cache line in the same cycle. The LRU bit, in the case of the 2-way set-associative implementation of the shared set, is modified accordingly, i.e., if the cache reference is from way 1 of the shared set, then the LRU bit of the referenced index is set to 0, whereas if the cache reference is from way 2 of the shared set, then the LRU bit is set to 1. This makes the replacement circuit simple and requires only K additional bits for the replacement circuitry, where K is the number of sets in the shared set. If the requesting shared data is not available in the shared set, then the requested shared data is copied from the main memory to the indexed location of the shared set (LRU replacement circuit finds the cache line to be evicted in case of the 2 – way set-associative shared set). The requested data is accessed from the offset location and the LRU bit of the indexed location is updated in case of the 2 – way set-associative shared set.

5.5.2 POWER SAVING CONCERNS

The experimental results presented in Section 5.7 show that the proposed implementation saves 67% power in case of a 4-way set-associative cache, as compared to the conventional cache architecture. The reduction in power consumption is mainly due to the fact that 75% of the main cache is shut down all the time in case of the proposed scheme. The shared set is powered on, only if the cache reference is shared. The victim set is active only if the cache reference is non-shared and the cache line is not available in the dedicated cache way of the main cache. Experimental results show that 90% of the data cache references are either from the dedicated cache way or from the shared set and very few references are directed to the victim set. Based on the sharing information available, the proposed scheme enables only one tag bank, one data bank in the main cache or the relevant tag and data bank(s) in the shared set. Thus, the total number of tag comparisons is reduced in the case of a hit in the shared set, to 1 if the shared set is direct-mapped, or 2 if the shared set is 2-way set-associative, as compared to 'N' for a conventional N-way set-associative cache. A miss in the shared set directly results in a cache miss, thus copying the requested data page from the main memory to the shared set. Even for a non-shared data reference, in case of a hit in the main cache, the number of tag comparisons is reduced to 1. In case of a miss in the dedicated cache way, the victim set is searched in the next cycle. As the hit rate of the main cache and shared set itself is more than 90%, this impact due to the victim set is negligible. Hence, a significant amount of dynamic power saving is possible with this scheme.

Theoretically, with an increase in the associativity, the dynamic power consumption approximately reduces by a factor of (N-1) / N for the best case, where N is the cache associativity. The power saving is more for a cache with a larger page size because of the improved hit rate, as a larger number of references are from the same page. Increasing the size of the victim set, although improves hit rate, has a considerable impact on the power consumed due to increased number of tag comparisons. This demands that the size of the victim set be a beneficial compromise between the number of comparators and the miss rate. The size of the shared set is only controlled by the percentage of shared pages that may exist among the processes.

As the proposed scheme is process-aware, it may also ensure better performance for the processes with frequent context switches, as the recently used pages of the previously executed process are still available in its dedicated cache way. It also ensures better performance than the conventional cache for processes with shared pages among them, as the number of tag comparisons are considerably reduced in both the direct-mapped and 2-way set-associative shared set implementations. This implementation has the advantage of a simple replacement circuitry (the main cache and shared set follow direct mapping) which helps in reducing the dynamic power consumption further. The victim set is enabled only when the non-shared cache reference is not available in the dedicated cache way. The victim set serves the purpose of reducing the miss rate, thus saving dynamic power consumption significantly.

5.5.3 REPLACEMENT SUPPORT

In SMPASP, rudimentary replacement is used in each of the dedicated cache ways, as it is treated like a direct-mapped cache. This gives a great advantage over the conventional and way-prediction schemes. Each spill-out page from the main cache is directed to the victim set. The victim set follows FIFO replacement policy, which makes the replacement circuitry simple. It uses $\log_2 M$ bits per cache line, where M is the number of cache lines in the victim set, for storing the FIFO information. In case of a hit in the victim set, the cache line in the dedicated cache is transferred to the FIFO cache line of the victim set. The selected victim cache line is transferred to the dedicated cache way, the copy in the victim set is made invalid and its FIFO count is set to the least. The FIFO count of all the other cache lines is modified and that of the new cache line (the last spill-out page from the main cache) is set to the maximum, making it the most recent in the victim set. For the shared set, if the shared set is direct-mapped, then the selected cache line is directly replaced, but if the mapping is 2-way set-associative, then the LRU bit is used to find the least recently used cache line for replacement.

5.6 ENERGY ANALYSIS

The energy efficiency of the proposed scheme is measured using the ‘Energy dissipation per cache access’ model proposed by Zhang et al. [Zhang 2005], i.e., the energy efficiency is estimated by measuring the difference in the energy dissipation per cache

access. The energy consumption corresponding to each cache component is defined as follows.

E_{dec} represents the energy dissipation of the address decoder

E_{mux} represents the energy dissipation of the multiplexer and output driver

$E_{tagline}$ represents the energy dissipation of one tag line

$E_{dataline}$ represents the energy dissipation of one data line

$E_{preline}$ represents the energy dissipation of one-line's precharging

$E_{comline}$ represents the energy dissipation of one-line's comparator

E_{saline} represents the energy dissipation of one-line's sense amplifier circuit

E_{tagway} represents the energy dissipation of one tag way

$E_{dataway}$ represents the energy dissipation of one data way

E_{preway} represents the energy dissipation of one way's precharging

E_{comway} represents the energy dissipation of one way's comparator

E_{saway} represents the energy dissipation of one way's sense amplifier circuit

E_{Victim} represents the energy dissipation associated with the fully associative victim set with M cache lines

E_{Shared} represents the energy dissipation associated with the direct-mapped shared set.

cachesize represents the size of the cache in Bytes

cachelinesize represents the size of one cache block and

N represents the associativity of the cache

The energy equations of the conventional set-associative, way-prediction, process aware selective placement and the shared memory process aware selective placement N-way set-associative cache can be computed as follows:

$$E_{way} = (E_{tagline} + E_{dataline} + E_{preline} + E_{comline} + E_{saline}) * cachesize/(N * cachelinesize) \rightarrow (5.1)$$

$$E_{victim} = M*(E_{tagline} + E_{dataline} + E_{preline} + E_{comline} + E_{saline}) \rightarrow (5.2)$$

$$E_{shared} = E_{dec} + E_{tagline} + E_{dataline} + E_{preline} + E_{comline} + E_{saline} \rightarrow (5.3)$$

Conventional cache (Conv):

Energy per cache access (E_{Conv}),

$$E_{Conv} = E_{dec} + E_{mux} + (N * E_{way}) \quad \rightarrow (5.4)$$

Way Prediction cache (WP):

Prediction Hit Energy (E_{Phit}),

$$E_{Phit} = E_{dec} + E_{mux} + E_{way} \quad \rightarrow (5.5)$$

Prediction Miss Energy (E_{Pmiss}) and Cache Miss Energy (E_{Cmiss}),

$$E_{Pmiss} = E_{Cmiss} = 2*(E_{dec} + E_{mux}) + (N* E_{way}) \quad \rightarrow (5.6)$$

Total Energy per cache access (E_{Total}),

$$E_{Total} = Phit * E_{Phit} + (1–Phit – Cmiss) * E_{Pmiss} + Cmiss * E_{Cmiss} \quad \rightarrow (5.7)$$

Where, Phit refers to the percentage of prediction hits (i.e.), Phit = number of prediction hits / total number of cache accesses and Cmiss refers to the percentage of cache misses, which are secondary misses.

Process Aware Selective Placement cache (PASP):

Cache Hit Energy in the dedicated cache way (E_{Ded_hit}),

$$E_{Ded\_hit} = E_{dec} + E_{mux} + E_{way} \quad \rightarrow (5.8)$$

Primary Cache Miss Energy (E_{Victim_hit})

and Secondary Cache Miss Energy (E_{Cmiss}),

$$E_{Victim\_hit} = E_{Cmiss} = E_{dec} + E_{mux} + E_{way} + E_{victim} \quad \rightarrow (5.9)$$

Total Energy per cache access (E_{PASP}),

$$E_{PASP} = Ded\_hit * E_{Ded\_hit} + (1 – Ded\_hit – Cmiss) * E_{Victim\_hit} + Cmiss * E_{Cmiss} \quad \rightarrow (5.10)$$

Where, Ded_hit refers to the percentage of hits in the dedicated way (i.e.), Ded_hit = number of hits in the dedicated cache way / total number of cache accesses.

Shared Memory Process Aware Selective Placement cache (SMPASP):

Cache Hit Energy in the dedicated cache way (E_{Ded_hit}) – non shared references,

$$E_{Ded\_hit} = E_{dec} + E_{mux} + E_{way} \quad \rightarrow (5.11)$$

Cache Hit Energy in the Shared cache set (E_{Shared_hit}) – shared references,

$$E_{Shared\_hit} = E_{shared} \quad \rightarrow (5.12)$$

Primary Cache Miss Energy (E_{Victim_hit}) – non shared references,

$$E_{Victim\_hit} = E_{dec} + E_{mux} + E_{way} + E_{victim} \quad \rightarrow (5.13)$$

Shared Cache Miss Energy (E_{Shared_miss}) – shared references,

$$E_{Shared\_miss} = E_{shared} \quad \rightarrow (5.14)$$

Secondary Cache Miss Energy ($E_{nonShared\_miss}$) – non shared references,

$$E_{nonShared\_miss} = E_{dec} + E_{mux} + E_{way} + E_{victim} \quad \rightarrow (5.15)$$

Total Energy per cache access for the Shared data (E_{shared_Total}),

$$E_{shared\_Total} = (Shared\_hit * E_{Shared\_hit}) + (1 - Shared\_hit) * E_{Shared\_miss} \quad \rightarrow (5.16)$$

Where, Shared_hit refers to the percentage of shared set hits (i.e.),

Shared_hit = number of shared set hits / total number of shared set accesses.

Total Energy per cache access for the non-Shared data ($E_{nonShared\_Total}$),

$$E_{nonShared\_Total} = Ded\_hit * E_{Ded\_hit} + (1 - nonSharedMiss - Ded\_hit) * E_{Victim\_hit} + nonSharedMiss * E_{nonShared\_miss} \quad \rightarrow (5.17)$$

Where, nonSharedMiss refers to the percentage of secondary cache misses for the non-shared data (i.e.), nonSharedMiss = number of secondary cache misses for the non-shared data / total number of cache accesses for the non-shared data.

Total Energy per cache access (E_{SMPASP}),

$$E_{SMPASP} = (Shared * E_{shared\_Total}) + (1 - Shared) * E_{nonShared\_Total} \quad \rightarrow (5.18)$$

Where, Shared refers to the percentage of shared set accesses (i.e.), Shared = number of shared set accesses / total number of accesses for the entire cache subsystem. The energy equations can be modified to simulate a 2-way set-associative shared set, which also causes a considerable power saving.

5.7 EXPERIMENTAL METHODOLOGY, RESULTS AND ANALYSIS

5.7.1 EXPERIMENTAL SETUP

The experimental setup used, measures the cache hit rate, first cycle hit rate, effective cache access time and the power consumption, with the number of tag comparisons in particular. The experimental setup uses a cycle simulator of a cache based on an in-order process model, namely, CACHEMEM 1.0 along with SPEC95 benchmark [SPEC95] suite program traces, extracted by using SimpleScalar 2.0 simulator [Burger 1997], for determining the cache hit rate for the different caching schemes considered for analysis. The effective cache access time is calculated using the cache hit rate and first cycle hit rate with the first cycle cache hit access time assumed to be one cycle. The access time of a first cycle cache miss where the requested data is available in the cache is assumed as two cycles while the cache miss processing is assumed to consume 10 cycles. The energy components are obtained using the power estimation model, eCACTI cycle simulator [Mamidipaka 2004].

CACHEMEM 1.0 is designed to simulate a 4-way set-associative cache with varying cache sizes (1KB, 2KB, 4KB, 8KB, 16KB and 64KB) and line sizes (8, 16, 32, 64, 128, and 256 bytes). The setup simulates a multitasking system, where more than one process is ready to run and is available in the main memory. The system uses a fixed quantum time (100, 200, 500 and 1000 traces) to switch between processes using Round Robin scheduling and is measured as the number of traces executed from a SPEC95 benchmark file. This cache simulator measures the cache hit rate, first cycle hit rate, tag comparison count and effective cache access time of all configurations of cache line size, cache size, context switch duration and SPEC95 benchmark program suite set (with and without data sharing among processes) for different caching schemes like the conventional set-associative cache, way prediction cache, Process Aware Selective Placement (PASP) cache, Shared Memory Process Aware Selective Placement (SMPASP) cache architectures with the direct-mapped shared set and the Shared Memory Process Aware Selective Placement (SMPASP) cache architecture with 2-way set-associative shared set. In CACHEMEM 1.0, one SPEC95 benchmark program is considered as a process and the program set consists of four processes. The power components for the analysis are

obtained by modifying the power analysis model, eCACTI for a specific architecture. The model assumes only cache read operation / no cache write model. In the case of a cache hit, the access time is assumed to be one clock cycle. A cache miss results in writing a tag/data block value into the selected tag/data arrays of the cache. Thus, a cache miss results in one cache read and two cache-write operations, which take 9 additional cycles.

Most of the existing cache architectures in literature use SPEC 95 as benchmark program suite which prompted us to select SPEC 95 over SPEC 2000. The SPEC 2000 benchmark program suite is too big for embedded cache architectures. The number of traces generated by SPEC 95 benchmark program suite is less in comparison with SPEC 2000 benchmark program suite which makes it a better choice for embedded applications. Moreover evaluation with some of the SPEC 2000 benchmark suite traces shows similar performance as that of SPEC 95 benchmark program suite traces. So in this work we use SPEC 95 benchmark program suite for evaluation.

5.7.2 COMPREHENSIVE EVALUATION OF CACHE

Table 5.1: List of SPEC 95 benchmark suite program sets

Exp. No.	SPEC 95 benchmark program set	Data Sharing (in %)
1	compress95, mgrid, fpppp, applu	32.02
2	mgrid, fpppp, applu, swim	34.85
3	fpppp, applu, swim, su2cor	34.90
4	applu, swim, su2cor, tomcatv	32.19
5	swim, su2cor, tomcatv, wave5	31.87
6	su2cor, tomcatv, wave5, ijpeg	26.35
7	tomcatv, wave5, ijpeg, hydro2d	26.53
8	wave5, ijpeg, hydro2d, turb3d	29.05
9	ijpeg, hydro2d, turb3d, vortex	29.24
10	hydro2d, turb3d, vortex, perl	29.49

The cache hit rate is estimated using the cycle-based simulator, CACHEMEM 1.0 for the different caching algorithms. The following table shown in Table 5.1 gives different

SPEC95 benchmark program sets used for the experimentation. The data sharing column gives the percentage of data shared among the processes of a certain program set.

Based on the data sharing among processes, the experimental results are categorized into two cases: (i) Independent processes where the processes do not share any data among them and (ii) Processes where the data section of the processes are shared and the OS and compiler provide the sharing-related information. This work used the above mentioned experimental setup to measure various cache related parameters.

5.7.2.1 Independent processes

In this case, it is ensured that the processes do not share any data among them. It is observed that the PASP and SMPASP (both direct mapped and 2-way set associative shared set) cache architectures have the same cache hit rate, first cycle hit rate, number of tag comparisons and effective cache access time for all the cache configurations, while experimenting with SPEC 95 benchmark programs as independent processes. This is because the shared set in SMPASP architecture is always powered off as the independent processes will not have any shared data, which makes these two selective placement architectures identical. The conventional cache offers lesser effective access time compared to all the other schemes, if the overall cache hit rates are uniform and the first cycle hit rate of the way prediction, PASP and SMPASP cache architectures is lesser than the conventional cache hit rate.

5.7.2.1.1 Cache Hit Rate

The hit rates of various cache architectures are analyzed with respect to the SPEC95 benchmark program sets, cache line size, cache size and context switch duration. The cache hit rate of the conventional and the way prediction cache architectures is always the same, irrespective of the cache configurations and program sets. This is attributed to the fact that these two architectures make use of the complete cache and have the same replacement scheme.

The cache hit rates of an 8K, 4-way set-associative cache for various SPEC95 benchmark program sets running as independent processes is shown in Fig. 5.12. It is observed from the results that the PASP and SMPASP cache architectures offer better cache hit rates for

the majority of the benchmark program sets with different cache size, line size and context switch duration.

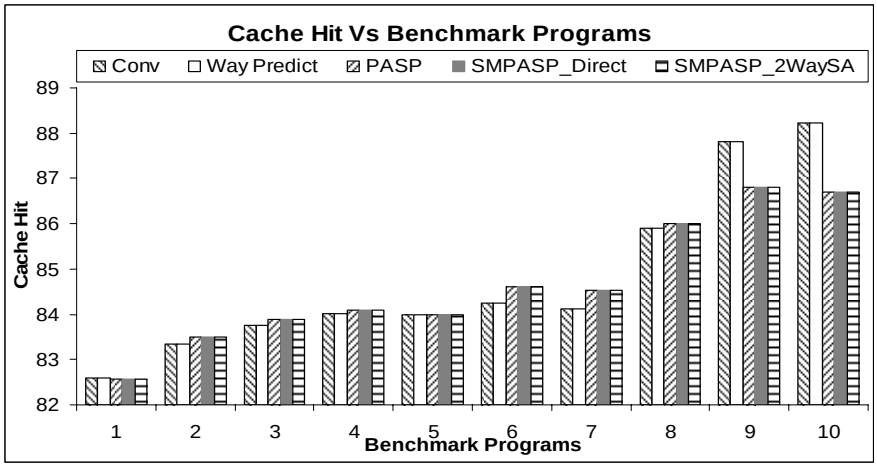

Fig. 5.12: Cache hit rate Vs SPEC95 Program sets for an 8KB, 4-way set-associative cache with 16Byte cache line size and context switch duration of 500 references

Figure 5.13 represents the cache hit rate vs cache line sizes chart for an 8K, 4-way set-associative cache, running SPEC95 benchmark program sets as independent processes. It is observed from these results that irrespective of the architecture the cache hit rate increases with increase in the cache line size, as a greater number of references fall within the same page [Hennessy 2007]. It is also observed that in most of the cases the PASP and SMPASP cache architectures perform better than their conventional and way prediction counterparts. This cache hit performance difference is more significant for smaller cache line size values.

The cache hit rate vs cache size for an 8K, 4-way set-associative cache running SPEC95 benchmark program sets as independent processes is shown in Figure 5.14. From the experimentation, it is clear that for all the architectures, the cache hit rate increases with increase in the cache size. It is also observed that for smaller cache sizes (less than 32KB), the PASP and SMPASP cache architectures offer higher cache hit rate, as compared to the conventional and way prediction architecture.

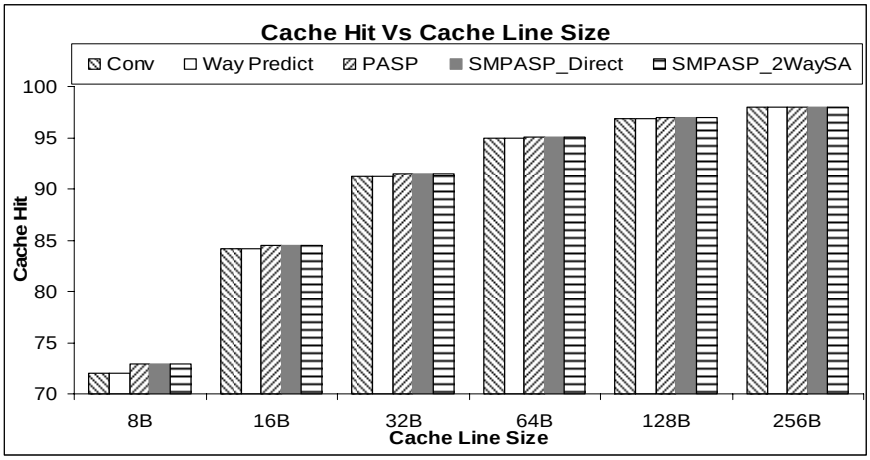

Fig. 5.13: Cache hit rate Vs Cache line size for an 8KB, 4-way set-associative cache with context switch duration of 500 references

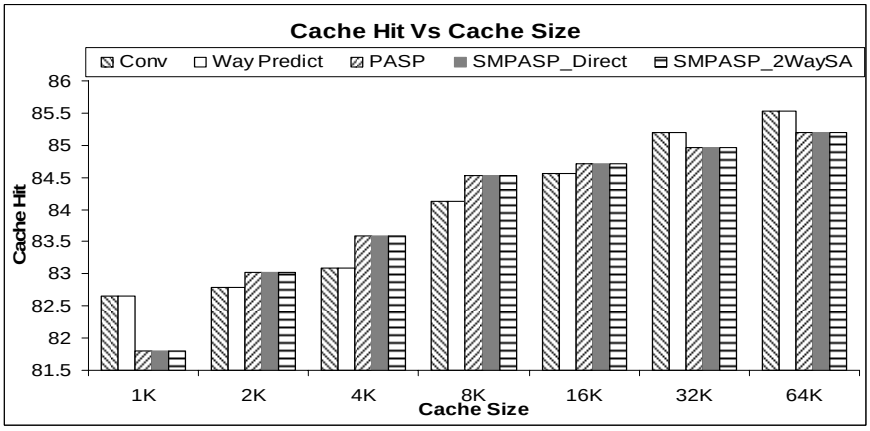

Fig. 5.14: Cache hit rate Vs Cache size for a 4-way set-associative cache with 16 Byte cache line size and context switch duration of 500 references

The cache hit rate vs context switch duration for a 4-way set-associative cache running SPEC95 benchmark program sets as independent processes is shown in Figure 5.15. It is observed from the experimental results that the cache hit performance of the PASP and SMPASP architectures does not degrade with context switch duration. This is because in both these schemes, local replacement equivalent to direct mapping takes place as one way is dedicated to a process. A slight variation of the cache hit rate is possible in selective placement schemes owing to the replacement strategy in victim set which is global. This hardly affects the performance as the victim set is very rarely used because of high first cycle hit rate. Each of the conventional and way prediction cache architectures varies its performance with context switch duration. This is due to the

global replacement strategy, which may replace some cache lines of the next process to execute causing performance variation with the context switching point. One can statically analyze the trace files to get the best context switching points where the cache impact will be least [Lee 1998][Lee 1999]. The brute-force analysis of finding the best context switch point is an NP-hard problem [Lee 1998][Lee 1999]. It is also observed that the PASP and SMPASP cache architectures offer a higher cache hit rate compared to their conventional and way prediction counterparts for all the context switch durations.

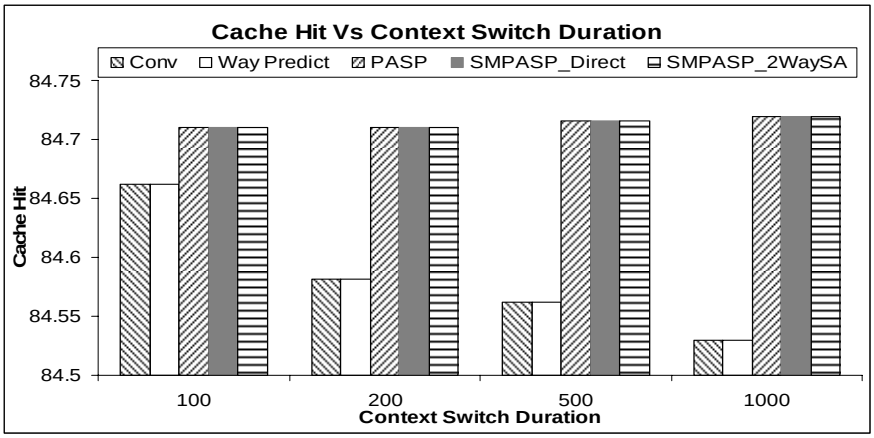

Fig. 5.15: Cache hit rate Vs Context switch duration for a 16K, 4-way set-associative cache with 16 Byte cache line size

5.7.2.1.2 First Cycle Cache Hit Rate

The first cycle hit rate of various cache architectures is analyzed with respect to the SPEC95 benchmark program sets, cache line size, cache size and context switch duration. The first cycle hit rate of the way prediction cache architecture is always less than that of the conventional cache architecture. In case of the conventional cache architecture, all cache hits happen in the first cycle, i.e., the first cycle hit is equal to the cache hit whereas in the way prediction scheme, a first cycle hit occurs only when the prediction is correct, which is usually 80-85% of the total cache hits.

The first cycle hit rates of an 8K, 4-way set-associative cache for various SPEC95 benchmark program sets running as independent processes is shown in Figure 5.16. It is observed from the experimental results that the PASP and SMPASP cache architectures offer greater first cycle hit rates as compared to the way prediction cache for all the benchmark program sets, irrespective of the cache size, line size and context switch

duration. The high first cycle hit rate of the PASP and SMPASP cache can be attributed to the use of local replacement scheme and process aware cache architecture design. It is also seen that in some cases, the cache hit rate of the conventional cache is falling behind that of the PASP and SMPASP schemes.

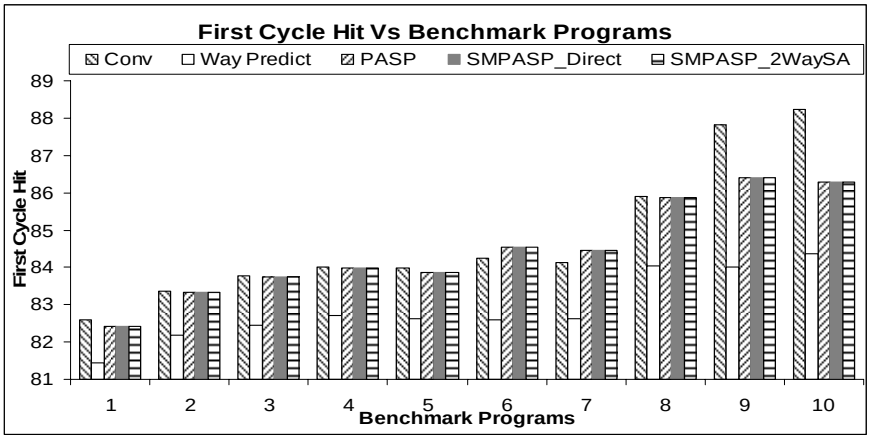

Fig. 5.16: First cycle hit rate Vs SPEC95 Program sets for an 8KB, 4-way set-associative cache with 16 Byte cache line size and context switch duration of 500 references

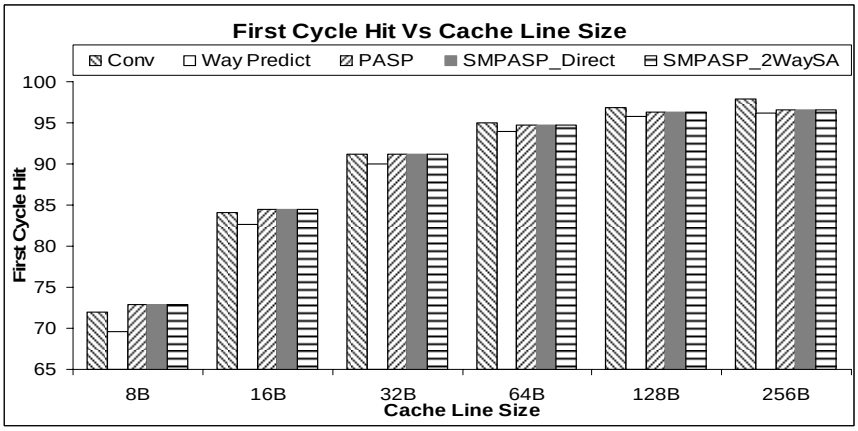

Fig. 5.17: First cycle hit rate Vs Cache line size for an 8KB, 4-way set-associative cache with context switch duration of 500 references

The first cycle hit rate for different cache architectures vs cache line size for an 8K, 4-way set-associative cache running SPEC95 benchmark program sets as independent processes is shown in Figure 5.17. It is observed from the results that for all the architectures, the first cycle hit rate increases with increase in the cache line size. The PASP and SMPASP schemes offer higher first cycle hit rate when compared to the way

prediction scheme, but this difference in the first cycle hit rate of the PASP and SMPASP over the way prediction scheme shrinks with increase in the cache line size.

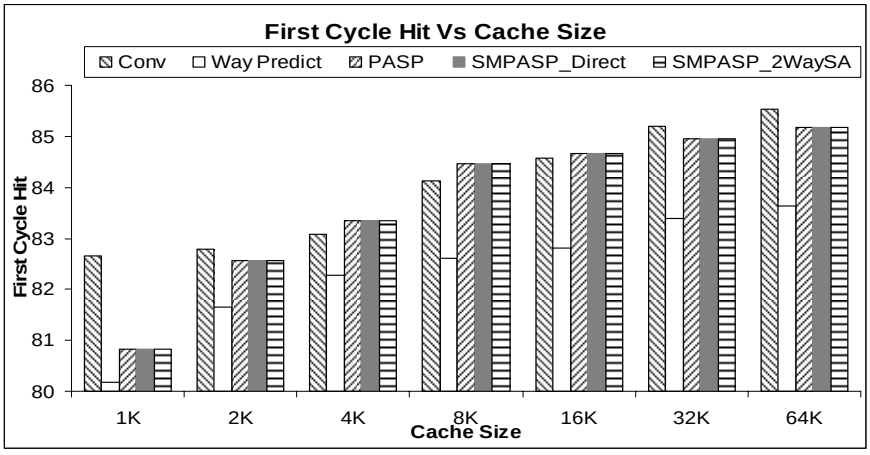

Fig. 5.18: First cycle hit rate Vs Cache size for a 4-way set-associative cache with 16 Byte cache line size and context switch duration of 500 references

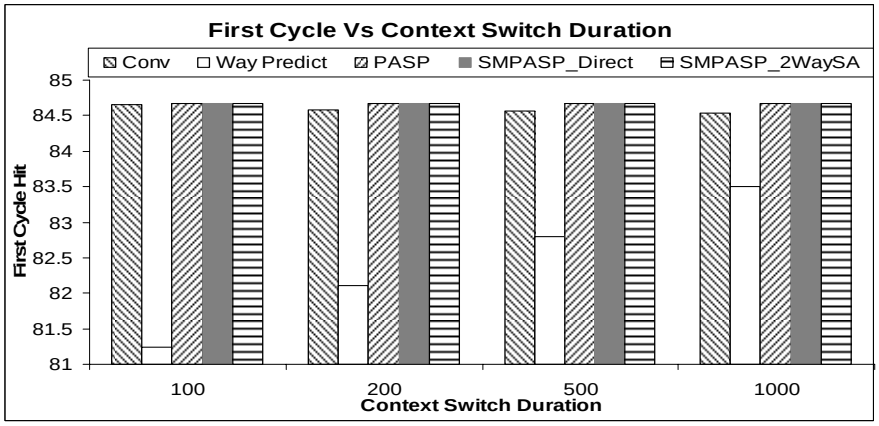

Fig. 5.19: First cycle hit rate Vs Context switch duration for a 16K, 4-way set-associative cache with 16 Byte cache line size

The first cycle hit rate vs cache sizes for an 8K, 4-way set-associative cache running SPEC95 benchmark program sets as independent processes is shown in Figure 5.18. Here, it is observed that for all the architectures the first cycle hit rate increases with increase in the cache size. It is also found that the PASP and SMPASP cache architectures offer higher first cycle hit rate for all cache sizes compared to the way prediction cache architecture. The figure shows that for smaller cache sizes (<32K), the first cycle hit rate of the PASP and SMPASP is better than the cache hit rate of

conventional cache and for larger caches (>32KB), the cache hit rate of conventional cache is better than first cycle hit rate of PASP and SMPASP architectures.

The first cycle hit rate vs context switch duration chart for a 4-way set-associative cache running SPEC95 benchmark program sets as independent processes is shown in Figure 5.19. The simulation results clearly show that the PASP and SMPASP cache architectures always provide better first cycle hit rates than the way prediction cache architecture.

5.7.2.1.3 Tag Comparison Count

The tag comparison count for the different cache architectures is analyzed with respect to the SPEC95 benchmark program sets, cache line size, cache size and context switch duration. The tag comparisons can act as a measure of the number of active data banks and tag banks while running the benchmark program set and can be used as an indirect measure of the energy consumption of the cache memory. The number of tag comparisons is normalized over the number of total references in the benchmark program set under consideration. For a single reference, the normalized number of tag comparisons is N (where N is the cache associativity) for all configurations of the conventional cache, as parallel comparisons are carried out in all the N cache lines. The normalized number of tag comparisons for the conventional cache is always high as compared to the other architectures, irrespective of the cache configuration, program set and data sharing. It is also observed that with increase in the first cycle hit rate, the tag comparison count decreases as the additional cycles are avoided, thus reducing the number of tag comparisons.

The normalized number of tag comparison count of an 8K, 4-way set-associative cache for various SPEC95 benchmark program sets running as independent processes is shown in Figure 5.20. It is observed from the experimental results that the PASP and SMPASP cache architectures yield reduced number of tag comparisons, and thus reduced energy consumption for all the benchmark program sets. As the number of tag comparisons lowers for cache architectures with high first cycle hit rate, the PASP and SMPASP schemes always offer better performance.

The normalized tag comparison count for different cache architectures vs cache line sizes for an 8K, 4-way set-associative cache running SPEC95 benchmark program sets as

independent processes is shown in Figure 5.21. It is observed from the simulation results that for the way prediction, PASP and SMPASP cache architectures the number of tag comparisons reduce with increase in cache line size and thus reduced number of active cache banks. The difference in the tag comparison count and the cache energy consumption for the PASP and SMPASP over that of the way prediction scheme is pronounced for smaller cache line sizes.

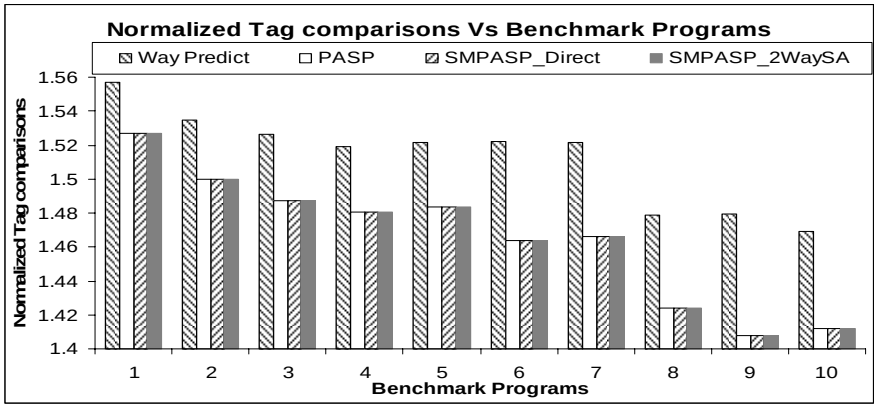

Fig. 5.20: Normalized tag comparison count Vs SPEC95 Program sets for an 8KB, 4-way set-associative cache with 16Byte cache line size and context switch duration of 500 references

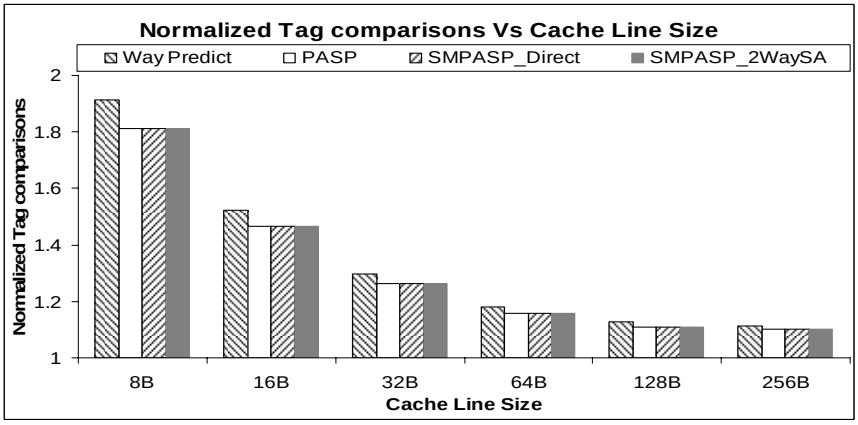

Fig. 5.21: Normalized tag comparison count Vs Cache line size for an 8KB, 4-way set-associative cache with context switch duration of 500 references

The normalized tag comparison count vs cache size chart for an 8K, 4-way set-associative cache running on SPEC95 benchmark program sets as independent processes is shown in Figure 5.22. It is observed from the simulation results that for the case of way prediction, PASP and SMPASP cache architectures the number of tag comparison

reduces with increase in cache size. It is also observed that the difference in the tag comparison count of PASP and SMPASP over the way prediction cache architecture increases with increase in the cache size.

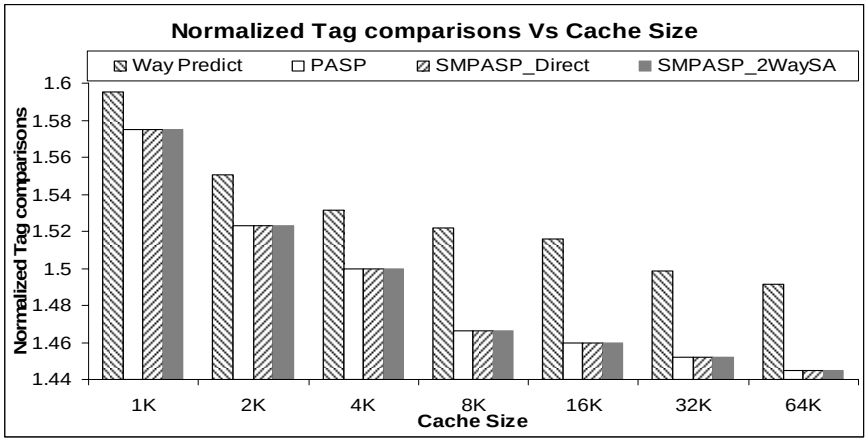

Fig. 5.22: Normalized tag comparison count Vs Cache size for a 4-way set-associative cache with 16 Byte cache line size and context switch duration of 500 references

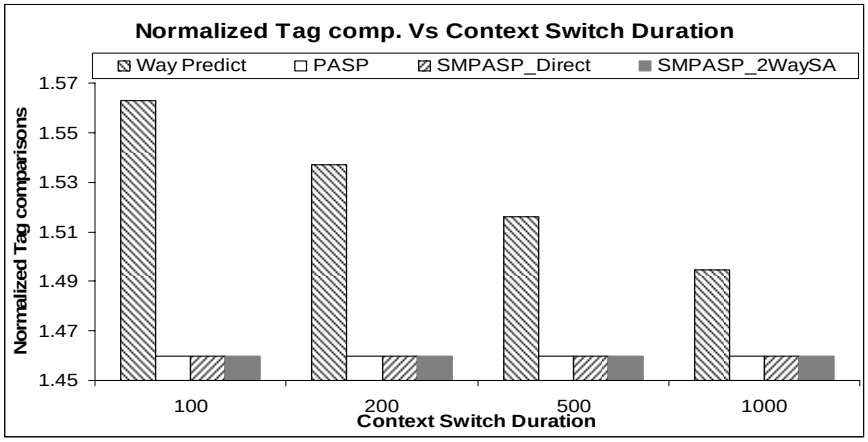

Fig. 5.23: Normalized tag comparison count Vs Context switch duration for an 16K, 4-way set-associative cache with 16 Byte cache line size

The normalized tag comparison count vs context switch duration for a 4-way set-associative cache for SPEC95 benchmark program sets running as independent processes is shown in Figure 5.23. It is observed that the PASP and SMPASP always provide a lesser tag comparison count and consumes less energy as compared to conventional and way prediction scheme. The number of tag comparisons in case of PASP and SMPASP architectures does not vary with context switch duration whereas in case of the way prediction cache the number of tag comparisons decreases with increase in the context

switch duration. This is attributed to the global replacement strategy used in way prediction cache which may replace some of the cache lines of the next process.

5.7.2.1.4 Effective Cache Access Time

The effective cache access time of various cache architectures is analyzed with respect to SPEC95 benchmark program sets, cache line size, cache size and context switch duration. The effective cache access time (ECAT) is calculated using the following formula

ECAT=(First cycle hit * 1) + ((Cache hit – First cycle hit)*2) + (Cache miss * 10) →5.19

Equation 19 assumes that the first cycle hit takes one clock cycle in all architectures. The way prediction, PASP and SMPASP cache architectures take two clock cycles to handle the first cycle miss but cache hit condition while cache miss processing will takes 10 cycles to complete. Similar to the cache hit rate and the first cycle hit rate performance, the PASP and SMPASP cache architectures have the same ECAT for all the cache configurations while running on SPEC95 benchmark programs as independent processes. The ECAT of the conventional cache is always lower than the ECAT of the way prediction cache architectures irrespective of the configurations, program sets and data sharing. From the equation, it is evident that the ECAT for all cache architectures depends on its first cycle hit rate and cache hit rate.

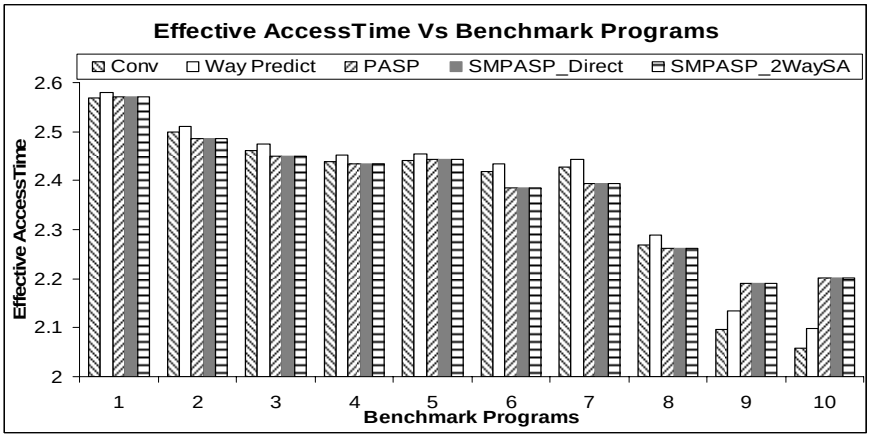

Fig. 5.24: Effective cache access time Vs SPEC95 Program sets for an 8KB, 4-way set-associative cache with 16Byte cache line size and context switch duration of 500 references

The ECAT of an 8K, 4-way set-associative cache for various SPEC95 benchmark program sets running as independent processes is shown in Figure 5.24. It is observed from the experimental results that the PASP and SMPASP cache architectures offer lesser ECAT value for majority of the benchmark program sets. The ECAT performance follows the same pattern observed while evaluating cache hit rate and first cycle hit rate performances. From these experimental results it can be concluded that in case of independent processes, if the cache hit rate is same, the way prediction cache architecture yields a higher ECAT value compared to other cache architectures.

The ECAT of different cache architectures vs cache line sizes for an 8K, 4-way set-associative cache running on SPEC95 benchmark program sets as independent processes is shown in Figure 5.25. It is observed that in case of all the architectures, the ECAT value reduces with increase in the cache line size. Irrespective of the cache line size, the ECAT value of the way prediction cache architecture is high as compared to other cache architectures and this difference in ECAT over other architectures reduces with increase in the cache line size. This is because of the cache hit rate and first cycle hit rate.

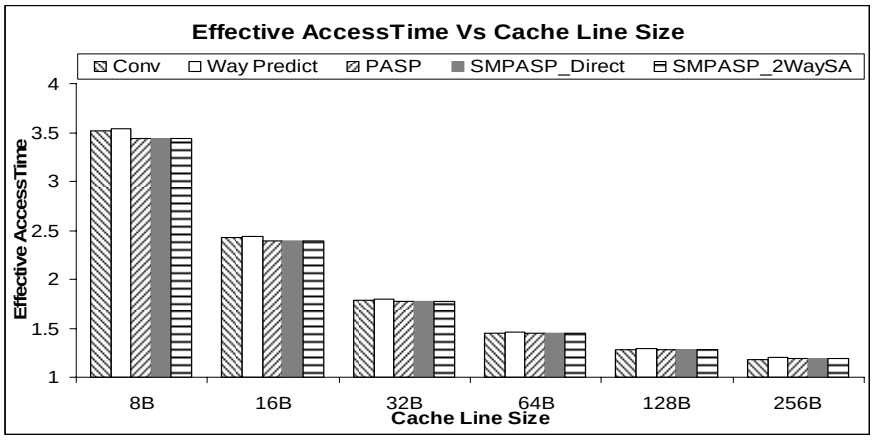

Fig. 5.25: Effective cache access time Vs Cache line size for an 8KB, 4-way set-associative cache with context switch duration of 500 references

The ECAT vs cache size for a 4-way set-associative cache running SPEC95 benchmark program sets as independent processes is shown in Figure 5.26. It is observed from the experimental results that for all the architectures the ECAT value decreases with increase in the cache size. It is also observed that the PASP and SMPASP cache architectures offer lower ECAT value compared to the conventional and way prediction architectures

for cache sizes less than 32KB. The conventional cache architecture performs better for large cache sizes. The way prediction cache architecture performance with respect to ECAT is poor compared to the other cache architectures.

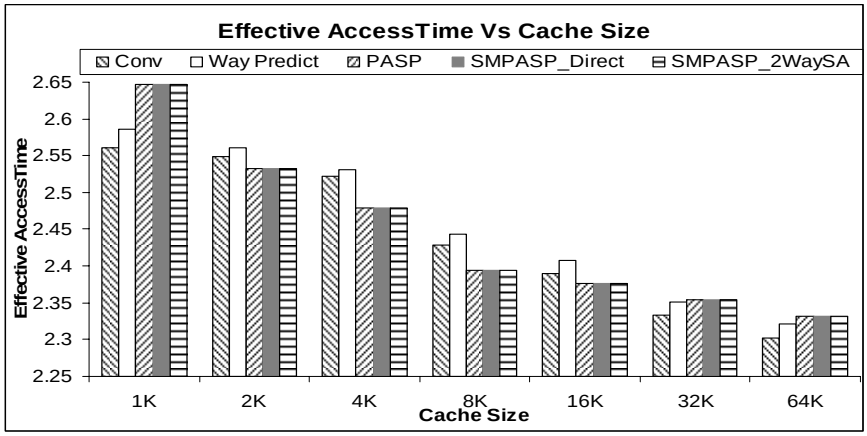

Fig. 5.26: Effective cache access time Vs Cache size for a 4-way set-associative cache with 16 Byte cache line size and context switch duration of 500 references

The relationship between the ECAT of various cache architectures and the context switch duration, for an 8K, 4-way set associative cache running on SPEC95 benchmark program sets as independent processes is shown in Figure 5.27.

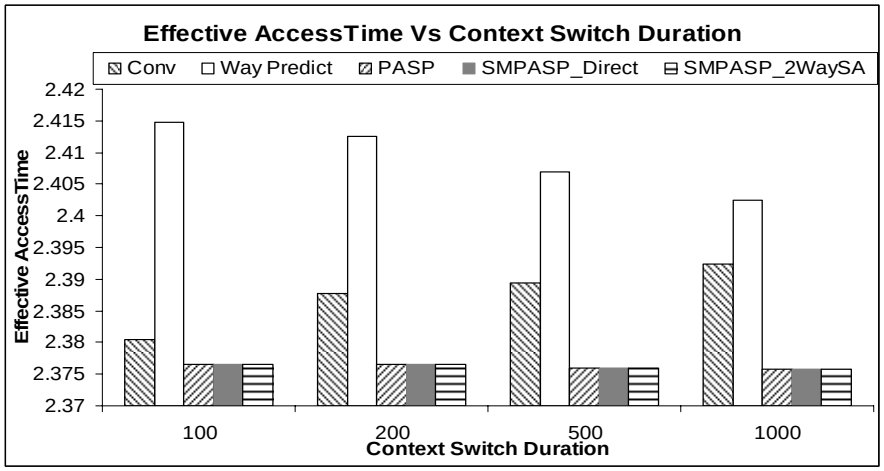

Fig. 5.27: Effective cache access time Vs Context switch duration for an 16K, 4-way set-associative cache with 16 Byte cache line size

It is observed from the experimental results that the ECAT performance of the PASP and SMPASP does not degrade with the context switch duration. This is due to the local replacement carried out in both these schemes with one way dedicated to a process. A

slight variation of the cache hit rate is possible because of the victim set replacement policy which is global. Each of the conventional and way prediction cache architectures varies its ECAT performance with context switch duration. It is also observed that the PASP and SMPASP cache architectures offer lower ECAT value compared to the conventional and way prediction architectures for all the context switch durations.

5.7.2.2 Shared Data among processes

In this section data sharing among the processes is discussed. The data section of the programs is shared and the sharing among the processes varies from 25% to 35% as shown in Table 5.1. This section analyses the cache hit rate, first cycle cache hit rate, tag comparison count and the effective cache access time for various cache configurations of the conventional cache, way prediction cache, PASP cache, SMPASP cache with direct mapped shared set and the SMPASP cache with 2-way set-associative shared set running processes with data sharing among them.

It is observed that the SMPASP (both direct-mapped and 2-way set-associative shared set) always has a better cache hit rate and first cycle hit rate than the PASP cache architecture for all cache configurations while running SPEC95 benchmark programs as processes with data sharing among them, thus justifying the purpose of its design. The better cache hit performance in case of SMPASP is mainly due to the shared cache set which stores all the shared cache lines. Between the two variants of SMPASP, the SMPASP architecture with the 2-way set-associative shared set offers a higher cache hit rate and first cycle hit rate as compared to the SMPASP architecture with direct-mapped shared set. The cache hit rate of the conventional and way prediction cache architectures is always the same irrespective of the configuration program set and data sharing as these two architectures make use of the complete cache and have the same placement and replacement schemes. The cache hit, first cycle hit, tag comparison count and ECAT performance of PASP cache architecture degrades slightly because of the overhead incurred while handling shared data.

5.7.2.2.1 Cache Hit Rate

The hit rate of various cache architectures is analyzed with respect to the SPEC95 benchmark program sets, cache line size, cache size and the context switch duration.

The cache hit rate of an 8K, 4-way set-associative cache for various SPEC95 benchmark program sets running as processes with data sharing among them is shown in Figure 5.28. It is observed from the experimental results that the SMPASP cache architectures performs equally good or better for a majority of the benchmark program sets compared to the conventional, way prediction and PASP cache architectures.

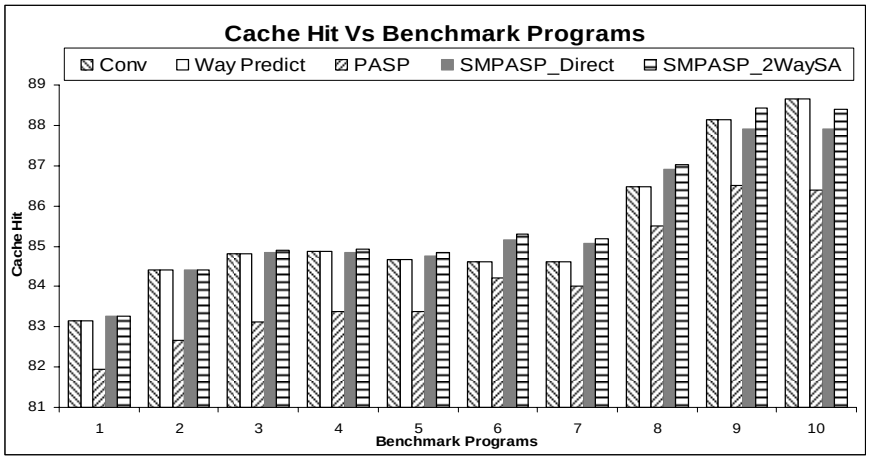

Fig. 5.28: Cache hit rate Vs SPEC95 Program sets for an 8KB, 4-way set-associative cache with 16Byte cache line size and context switch duration of 500 references

The cache hit rate vs cache line size chart for an 8K, 4-way set-associative cache running SPEC95 benchmark program sets as processes with data sharing is shown in Figure 5.29. It is observed that similar to the instance of independent processes, irrespective of the cache architecture, cache hit rate increases with increase in the cache line size. It is also noted that irrespective of the cache line size, the SMPASP architecture performs better than the other architectures. For all the configurations, the SMPASP with 2-way set-associative shared set provides the best cache hit performance because of the shared set.

The cache hit rate vs cache size for an 8K, 4-way set-associative cache running SPEC95 benchmark program sets as processes with data sharing is shown in Figure 5.30. It is found from the experimental results that similar to the independent processes, irrespective of cache architecture, the cache hit rate increases with increase in the cache size. It is also observed that the SMPASP cache architecture offers high cache hit rate in comparison with the conventional and way prediction architectures, for cache sizes less than 32KB. The conventional and way prediction cache architectures perform better for large cache

sizes. The SMPASP with 2-way set-associative cache performs better than the SMPASP with direct mapped cache and PASP cache architectures.

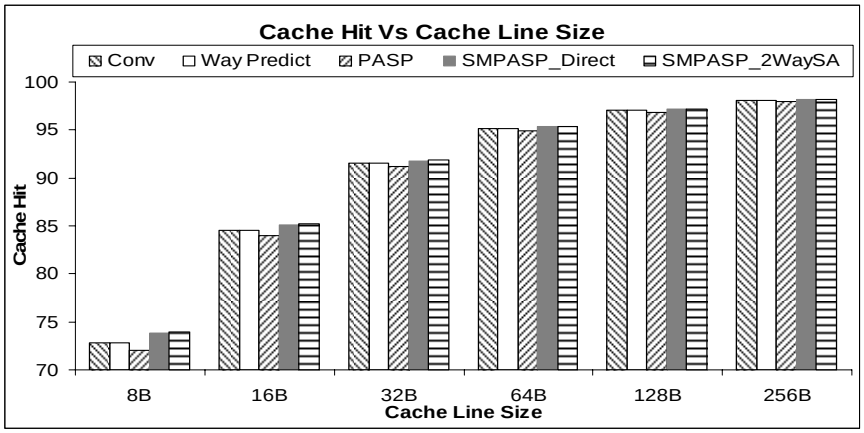

Fig. 5.29: Cache hit rate Vs Cache line size for an 8KB, 4-way set-associative cache with context switch duration of 500 references

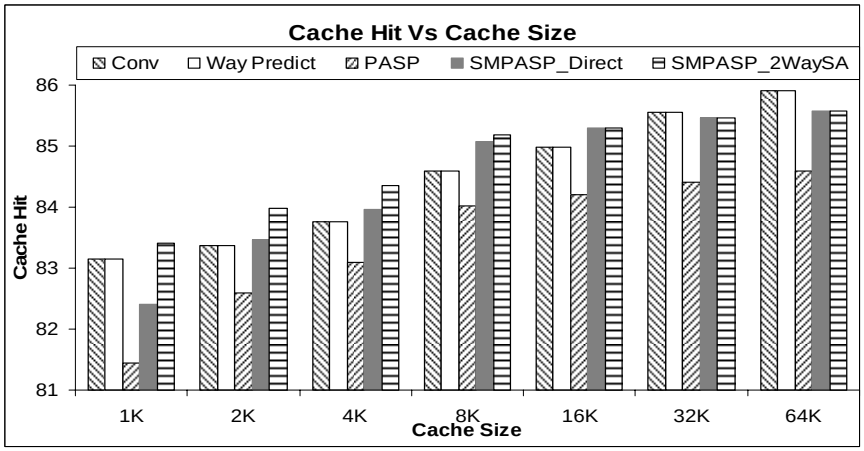

Fig. 5.30: Cache hit rate Vs Cache size for a 4-way set-associative cache with 16 Byte cache line size and context switch duration of 500 references

The cache hit rate vs context switch duration for a 4-way set-associative cache running SPEC95 benchmark program sets as processes with data sharing is shown in Figure 5.31. It is noted that the context switching does not affect the performance of the SMPASP architecture as in this scheme local replacement is carried out by assigning one dedicated way to a process. A slight variation in cache hit rate is possible because of the global victim cache replacement policy but it hardly affects the performance, as the victim set is rarely used owing to very high first cycle hit rate. The PASP cache hit performance increases with lengthening of the context switch duration. This is because of the shared

cache line access by other processes. In case of the PASP architecture, if the context switch duration is high, the system performs as if only one process is executing which results in an improved cache hit performance.

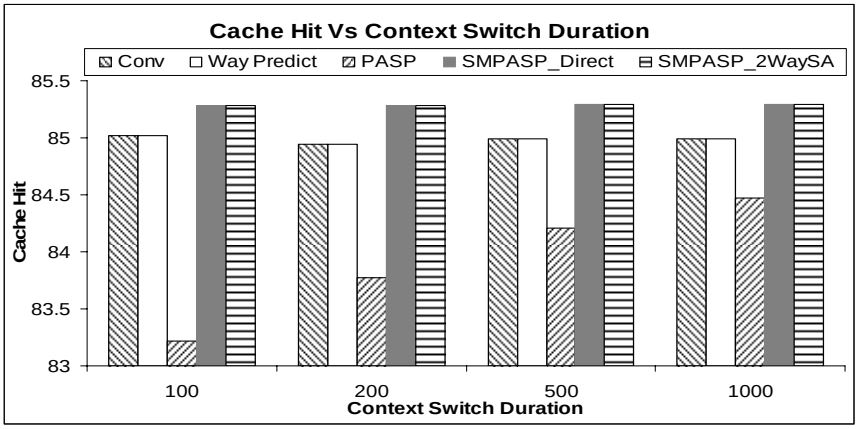

Fig. 5.31: Cache hit rate Vs Context switch duration for a 16K, 4-way set-associative cache with 16 Byte cache line size

The cache hit performance of the conventional and way prediction cache architectures varies with change in context switch duration due to the use of global replacement strategy. It is also seen that the SMPASP cache architecture offers high cache hit rate compared to all other architectures, irrespective of the context switch duration. The SMPASP with 2-way set-associative shared set provides a better cache hit rate compared to SMPASP with direct mapped shared set.

5.7.2.2.2 First Cycle Hit Rate

The first cycle hit rate for various cache architectures is analyzed with respect to the SPEC95 benchmark program sets, cache line size, cache size and the context switch duration. It is observed that the SMPASP (both direct-mapped and 2-way set-associative shared set) always exhibits a better first cycle hit rate than its way prediction and PASP counterparts for all cache configurations while running the SPEC95 benchmark programs as processes with data sharing among them. This is mainly due to the fact that the shared set in the SMPASP architecture stores all the shared cache lines, which results in a better first cycle hit rate. The SMPASP architecture with 2-way set-associative shared set offers higher first cycle hit rate compared to the SMPASP architecture with direct-mapped shared set.

The first cycle hit rates of an 8K, 4-way set-associative cache for various SPEC95 benchmark program sets running as processes with data sharing among them is shown in Figure 5.32. It is observed from the experimental results that the SMPASP cache architecture performs equally good or better for a majority of the benchmark program sets compared to the conventional cache architecture. It is also observed that the SMPASP cache architecture outperforms the way prediction and PASP cache architectures for all the benchmark program sets, thus fulfilling its purpose. Though the first cycle hit performance of the PASP cache architecture degrades slightly because of the inefficiency in managing shared data, it still performs better than the way prediction cache architecture in many cases. The high first cycle hit rate of the PASP and SMPASP compared to the way prediction scheme is attributed to the local replacement strategy used in these two schemes and the process aware cache architecture. It can also be noted that in many cases, the cache hit rate of the conventional cache is falling behind the SMPASP first cycle hit rate.

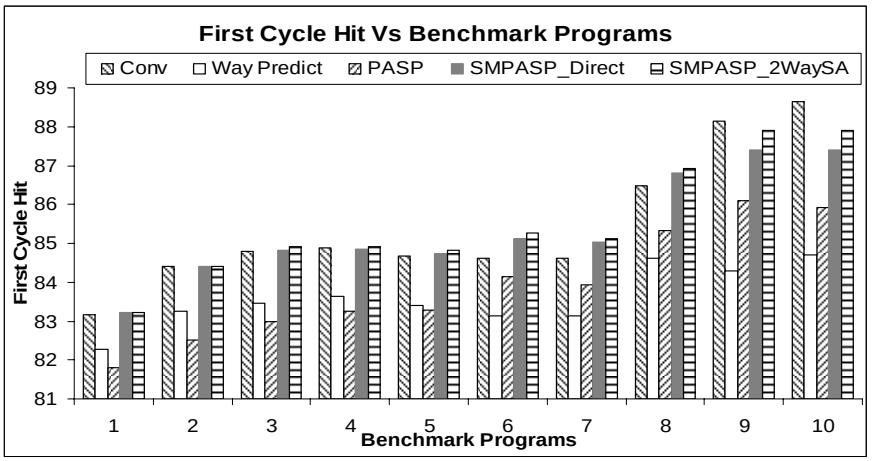

Fig. 5.32: First cycle hit rate Vs SPEC95 Program sets for an 8KB, 4-way set-associative cache with 16 Byte cache line size and context switch duration of 500 references

Figure 5.33 represents the relationship between the first cycle hit rate and the cache line size for an 8K, 4-way set associative cache running SPEC95 benchmark program sets as processes with data sharing. The experimental results reveal that similar to the independent processes, irrespective of the cache architecture, the first cycle hit rate also increases with increase in the cache line size. It is also observed that irrespective of the cache line size, the SMPASP architecture performs better than its way prediction and

PASP counterparts. For all the configurations, the SMPASP with 2-way set-associative shared set provides better first cycle hit performance compared to SMPASP with direct-mapped shared set. It is also observed that irrespective of the cache line size, the first cycle hit rate of the PASP architecture is better than that of the way prediction scheme.

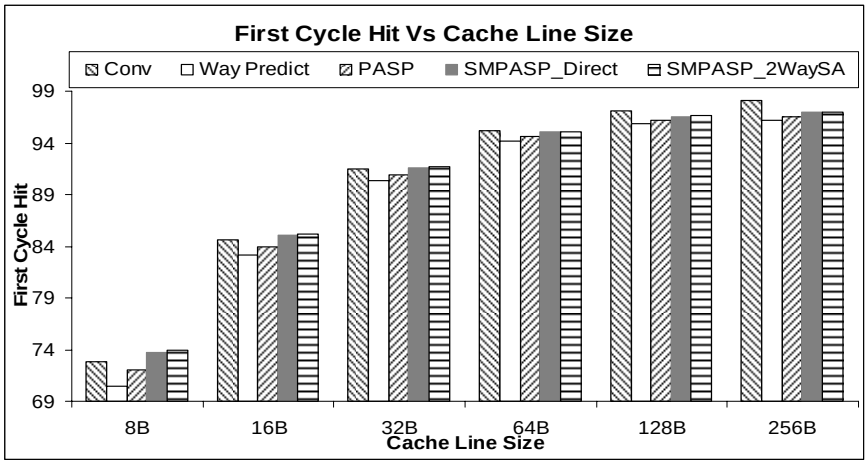

Fig. 5.33: First cycle hit rate Vs Cache line size for an 8KB, 4-way set-associative cache with context switch duration of 500 references

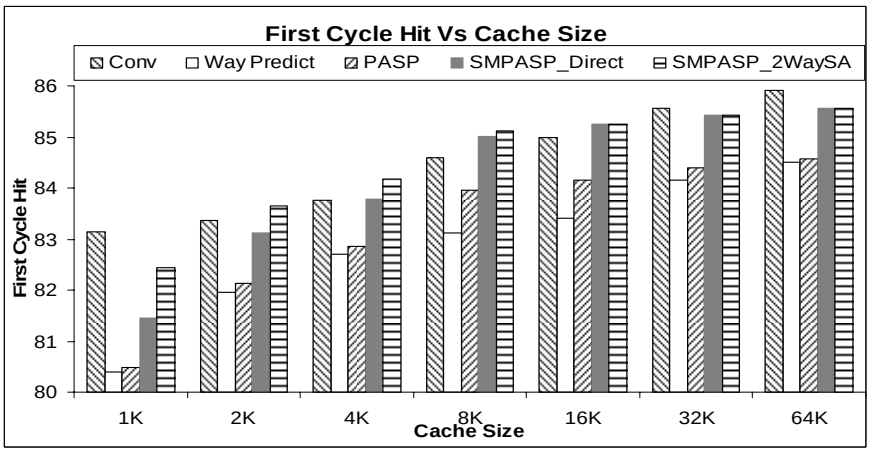

Fig. 5.34: First cycle hit rate Vs Cache size for a 4-way set-associative cache with 16 Byte cache line size and context switch duration of 500 references

The first cycle hit rate vs cache size for an 8K, 4-way set-associative cache running SPEC95 benchmark program sets as processes with data sharing is shown in Figure 5.34. It is found from the experimental results that as in the case of independent processes, irrespective of the cache architecture, the first cycle hit rate increases with increase in the cache size. It is also observed that the SMPASP cache architecture offers better first cycle

hit rate compared to the cache hit rate of the conventional cache architecture for cache sizes less than 32KB. The conventional cache architecture gives a better performance for very large cache sizes. The SMPASP with 2-way set-associative cache performs better than the SMPASP with direct-mapped cache and the PASP cache architectures. It is also observed that for cache sizes less than 32KB, the first cycle hit rate of the PASP architecture is better than way prediction scheme.

The first cycle hit rate vs the context switch duration chart for a 4-way set-associative cache running SPEC95 benchmark program sets as processes with data sharing is shown in Figure 5.35. It is observed from the experimental results that context switching does not affect the performance of SMPASP architecture as in this scheme local replacement is carried out by assigning one dedicated way to a process. In case of PASP architecture, if the context switch duration is high, then the system performs as if only one process is executing which results in an improved first cycle hit performance. The first cycle hit performance of the conventional and way prediction cache architectures varies with change in context switch duration due to the use of global replacement strategy. The SMPASP cache architecture offers a high first cycle hit rate compared to all other architectures, irrespective of the context switch duration. The PASP cache architecture offers a high first cycle hit rate compared to the way prediction cache architecture for all the context switch durations.

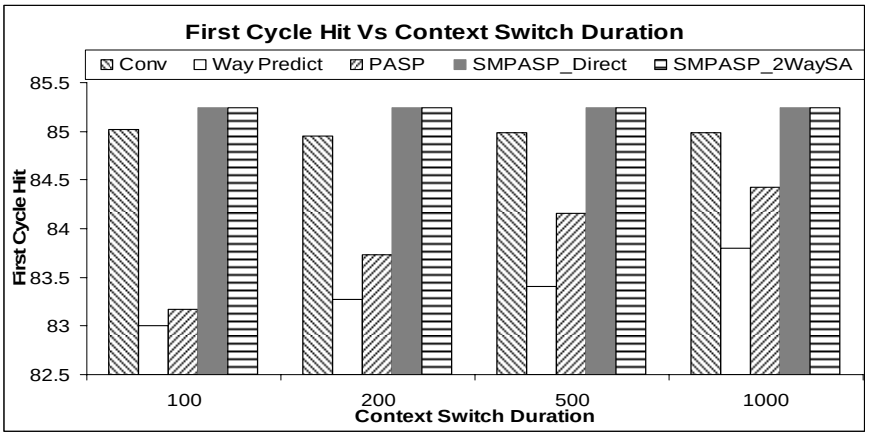

Fig. 5.35: First cycle hit rate Vs Context switch duration for a 16K, 4-way set-associative cache with 16 Byte cache line size

5.7.2.2.3 Tag Comparison Count

The tag comparison count for different cache architectures is analyzed with respect to the SPEC95 benchmark program sets, cache line size, cache size and the context switch duration. The number of tag comparisons is normalized over the total number of references in the benchmark program set. The normalized tag comparison count is N (where N is the cache associativity) for all the conventional cache configurations as it carries out a parallel comparison in all the N cache lines of the selected set. It is observed that the SMPASP with direct-mapped shared set always outperforms all the other architectures with respect to tag comparisons count for all the configurations, context switch durations, program sets and data sharing. The SMPASP with 2-way set-associative shared set gives a higher tag comparison count when compared to the SMPASP with direct-mapped shared set because of the additional tag comparisons in the shared set. The normalized tag comparison count for the conventional cache is always high in comparison to other architectures irrespective of the configurations, context switch durations, program set and data sharing. These results also reveal that with increase in the first cycle hit rate, the tag comparison count decreases. The tag comparison count of SMPASP with 2-way set-associative shared set increases with shared data as additional tag comparisons have to be performed in the shared set. The tag comparisons are also a measure of how many data banks and tag banks are active while running the benchmark program set.

The normalized tag comparison count of an 8K, 4-way set-associative cache for various SPEC95 benchmark program sets running as processes with data sharing among them is shown in Figure 5.36. The SMPASP with direct-mapped shared set cache architecture offers the least tag comparison count for all the benchmark program sets compared to the other architectures. The normalized tag comparison count vs cache line size and normalized tag comparison count vs cache size for an 8K, 4-way set-associative cache running SPEC95 benchmark program sets as processes with data sharing are shown in Figure 5.37 and Figure 5.38 respectively. It is again verified here that similar to the independent processes, the tag comparison count reduces with increase in the cache line size and cache size for the way prediction, PASP and SMPASP cache architectures. This is attributed to the increased first cycle and cache hit rates.

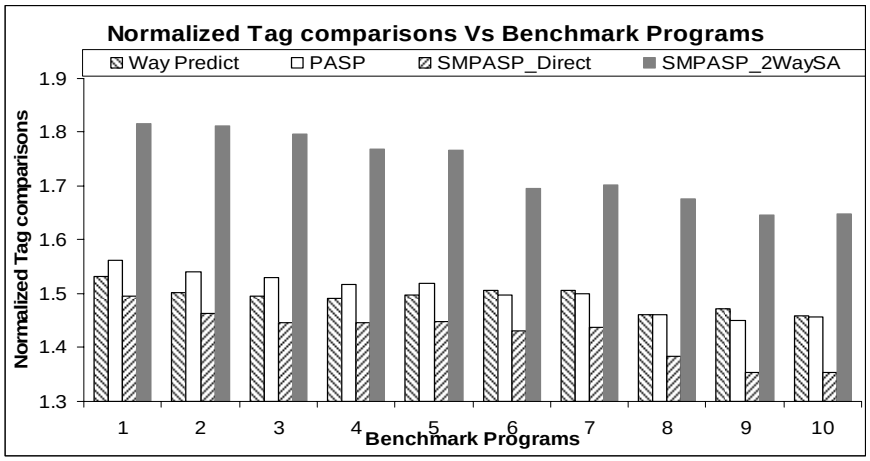

Fig. 5.36: Normalized tag comparison count Vs SPEC95 Program sets for an 8KB, 4-way set-associative cache with 16Byte cache line size and context switch duration of 500 references

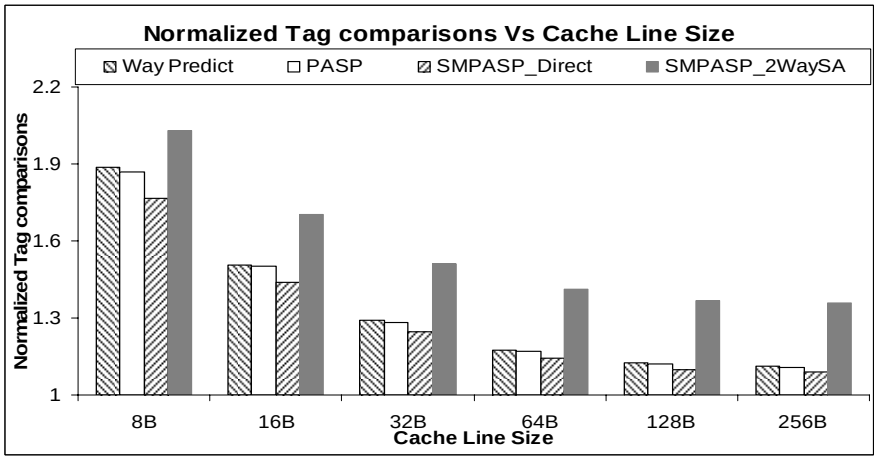

Fig. 5.37: Normalized tag comparison count Vs Cache line size for an 8KB, 4-way set-associative cache with context switch duration of 500 references

Figure 5.39 presents the relationship between the normalized tag comparison count and the context switch duration for a 4-way set-associative cache running SPEC95 benchmark program sets as processes with data sharing. It is observed that the context switching does not affect the performance of the SMPASP architecture. This is attributed to the local replacement carried out in this scheme with one way dedicated to a process. A slight variation of the cache hit rate is possible because of the global victim cache replacement policy but this hardly affects the performance as the victim is rarely used because of high first cycle hit rate. The number of tag comparisons of the PASP reduces

with increase in the context switch duration. If the context switch interval is high, then the PASP cache performs as if only one process is executing (without shared data) which results in an improved cache hit performance for the PASP architecture. The tag comparison count performance of the conventional and way prediction cache architectures varies with change in context switch duration due to the use of global replacement strategy, poor prediction hit and poor cache hit.

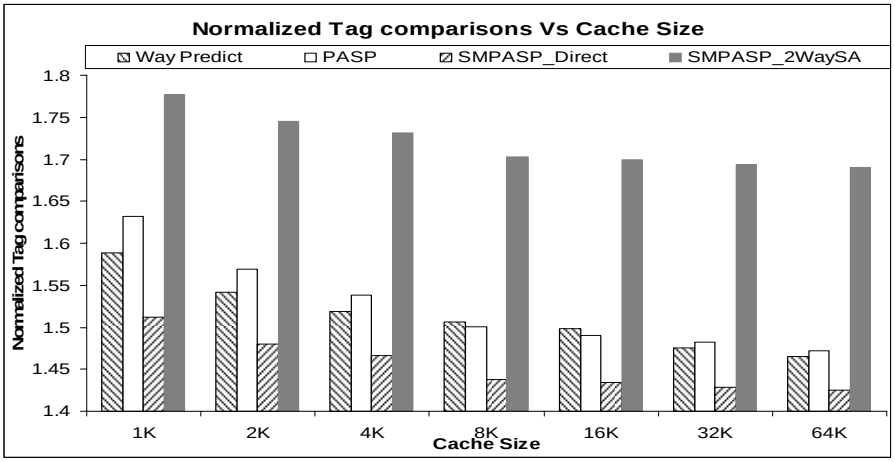

Fig. 5.38: Normalized tag comparison count Vs Cache size for a 4-way set-associative cache with 16 Byte cache line size and context switch duration of 500 references

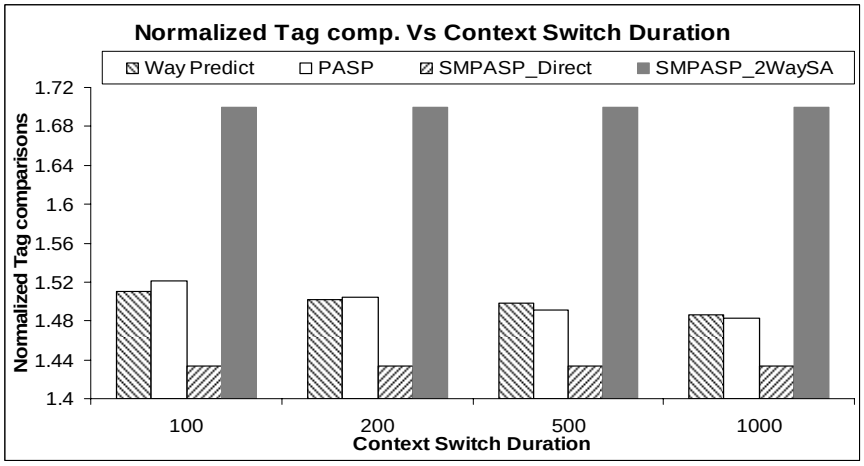

Fig. 5.39: Normalized tag comparison count Vs Context switch duration for an 16K, 4-way set-associative cache with 16 Byte cache line size

5.7.2.2.4 Effective Cache Access Time (ECAT)

As observed in the cache hit rate and first cycle hit rate performances, the ECAT performance of the SMPASP (both direct-mapped and 2-way set-associative shared set)

cache architecture is better than that of all the other cache architectures and context switch duration for all the cache configurations. This is because the shared set of the SMPASP architecture stores all the shared cache lines which results in a better cache hit and first cycle hit performance. The SMPASP with 2-way set-associative shared set offers better performance than the SMPASP with direct-mapped cache owing to the cache hit rate and first cycle hit rate. The ECAT of conventional cache architecture is always lower than that of the way predictive cache architecture, irrespective of the configuration, program set, context switch duration and data sharing which is attributed to the high first cycle hit performance.

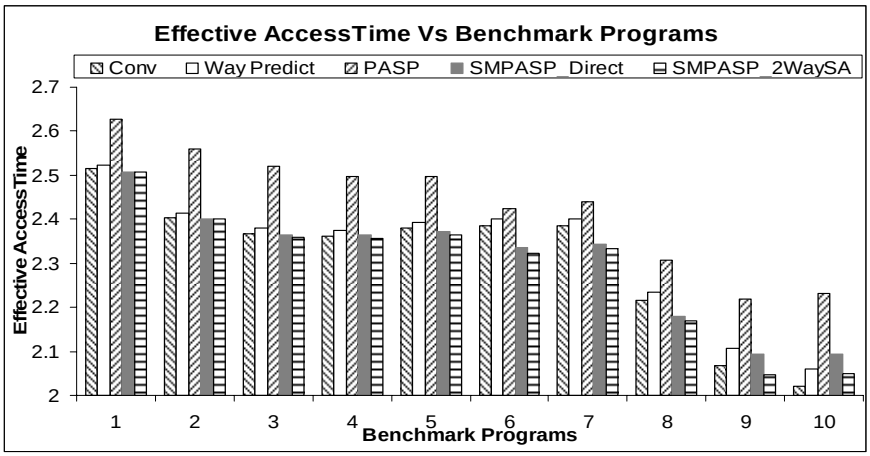

Fig. 5.40: Effective access time Vs SPEC 95 Program sets for an 8KB, 4-way set-associative cache with 16Byte cache line size and context switch duration of 500 references

The ECAT of an 8K, 4-way set-associative cache for various SPEC95 benchmark program sets running as processes with data sharing among them is shown in Figure 5.40. It is observed that the PASP and SMPASP cache architectures offer lesser ECAT for a majority of the benchmark program sets. The ECAT performance follows the same pattern of the cache hit rate and first cycle hit rate performances, as ECAT depends on the first cycle hit rate and the overall cache hit rate. The SMPASP with 2-way shared set offers a lesser ECAT in comparison with the SMPASP with direct-mapped shared set, which is again due to the high first cycle hit rate.

The ECAT vs the cache line size and ECAT vs cache size for an 8K, 4-way set-associative cache for various SPEC95 benchmark program sets running as processes with

data sharing among them is shown in Figure 5.41 and Figure 5.42 respectively. The experimental results reveal that similar to the case of independent processes, here also, irrespective of the cache architecture, the ECAT decreases with increase in the cache line size and cache size. It is also observed that for cache sizes less than 32KB, the SMPASP cache architecture offers low ECAT compared to the other architectures. The conventional and way prediction cache architectures perform better for very large cache sizes.

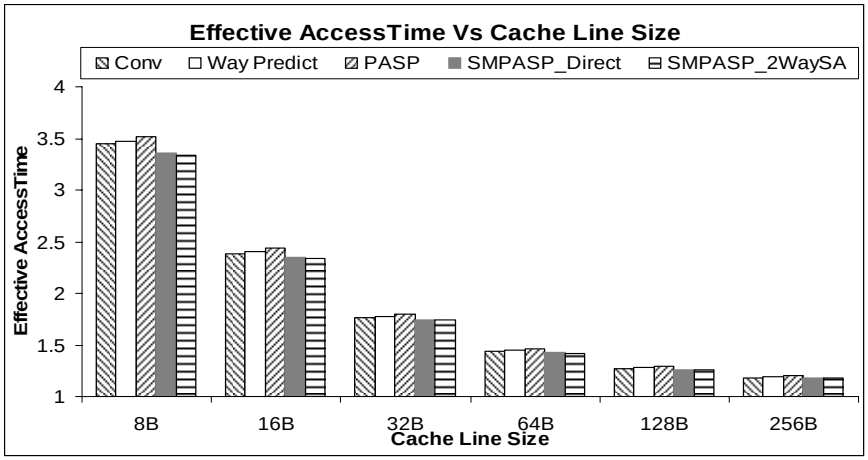

Fig. 5.41: Effective access time Vs Cache line size for an 8KB, 4-way set-associative cache with context switch duration of 500 references

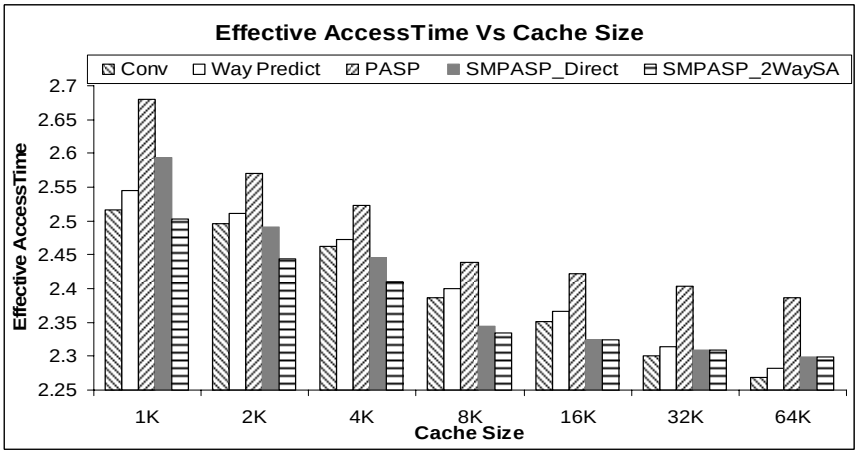

Fig. 5.42: Effective access time Vs Cache size for a 4-way set-associative cache with 16 Byte cache line size and context switch duration of 500 references

The ECAT vs context switch duration for a 4-way set-associative cache for various SPEC95 benchmark program sets running as processes with data sharing among them is

shown in Figure 5.43. The results confirm that the context switching does not affect the ECAT performance of the SMPASP architecture. This is because context switching does not affect the cache hit rate and first cycle hit rate of the SMPASP architectures. The ECAT of the PASP reduces with increase in the context switch duration. If the context switch interval is high, then the PASP cache performs as if only one process is executing (without shared data) which results in an improved cache hit performance for the PASP architecture. The ECAT performance of the conventional and way prediction cache architectures varies with change in context switch duration due to the use of global replacement strategy, poor prediction hit and poor cache hit.

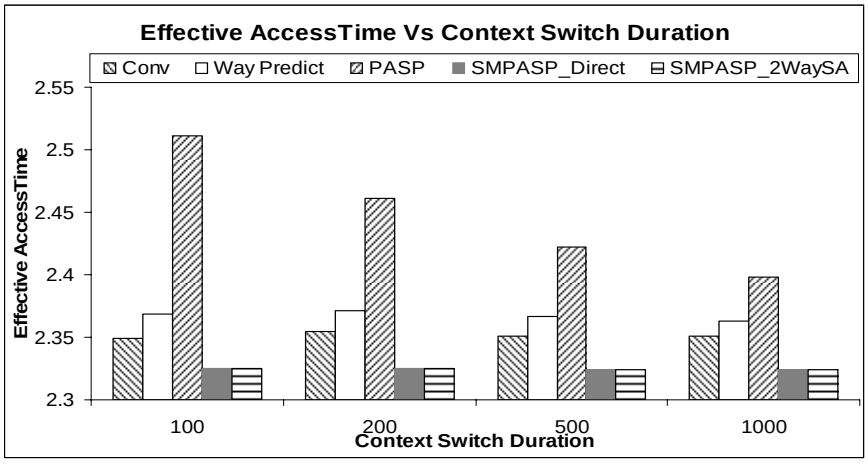

Fig. 5.43: Effective access time Vs Context switch duration for an 16K, 4-way set-associative cache with 16 Byte cache line size

5.7.2.3 Comparative Figures

The cache hit rate of the conventional and way prediction cache architecture is always the same for all the cache configurations irrespective of the level of sharing among the processes. The cache hit rate of the PASP cache, SMPASP cache with direct-mapped shared set, SMPASP cache with 2-way set-associative shared set is always the same for all the configurations, when the processes have no sharing among them. The cache hit rate of the PASP / SMPASP cache architecture varies from -2% to +2% of that of the conventional / way prediction cache architectures for all the cache configurations and benchmark program sets, when processes have no shared data among them. The performance of the PASP cache architecture degrades for the shared benchmark program sets. The cache hit rate of the PASP cache, SMPASP cache with direct-mapped shared

set, and SMPASP cache with 2-way set-associative shared set architecture varies from -5% to +0.1%, -1.3% to +1.85% and -0.5% to +2.1% respectively of that of the conventional / way prediction cache architectures for all the cache configurations and benchmark program sets when the processes exhibit data sharing among them.

The first cycle hit rate of the PASP cache, SMPASP cache with direct mapped shared set, SMPASP cache with 2-way set-associative shared set is always the same for all configurations when the processes do not share data among them. The first cycle hit rate of the PASP / SMPASP cache architecture improves from 1% to 10% of that of the way prediction cache architectures for all the cache configurations and benchmark program sets, when processes have no shared data among them. However, the PASP cache architecture degrades its performance for shared benchmark program sets. The first cycle hit rate of the SMPASP cache with 2-way set-associative shared set is higher than that of the PASP cache and SMPASP cache with direct-mapped shared set for all the cache configurations and benchmark program sets, when processes demonstrate data sharing among them. The first cycle hit rate of the PASP cache, SMPASP cache with direct-mapped shared set, and SMPASP cache with 2-way set-associative shared set cache architecture varies from -3.5% to +5%, +0.75% to +9.75% and +1% to +10% respectively of that of the way prediction cache architecture for all the cache configurations and benchmark program sets, when processes exhibit data sharing among them.

The tag comparison count of the PASP cache, SMPASP cache with direct-mapped shared set, and the SMPASP cache with 2-way set-associative shared set is always the same for all configurations when the processes do not share any data among them. The tag comparison count of the PASP / SMPASP cache architecture reduces by 50% to 75% of that of the conventional cache architecture for all the cache configurations and benchmark program sets, when processes have no shared data among them. The number of tag comparisons of the PASP / SMPASP cache architecture reduces by 1.2% to 9% of that of the way prediction architecture for all the cache configurations and benchmark program sets. The PASP cache architecture performance degrades in the case of shared benchmark program sets. The number of tag comparisons of the SMPASP cache with direct-mapped shared set is lesser than that of the PASP cache and SMPASP cache with

2-way set-associative shared set for all the cache configurations and benchmark program sets, when processes share data among them. The tag comparison count of PASP cache, SMPASP cache with direct-mapped shared set, and SMPASP cache with 2-way set-associative shared set architecture varies from +48% to +72.5%, +51% to +73% and +43% to +66% respectively of that of the conventional cache architecture for all the cache configurations and benchmark program sets, when processes have data sharing among them. The number of tag comparisons of PASP cache, SMPASP cache with direct-mapped shared set, and SMPASP cache with 2-way set-associative shared set architecture varies from -4.8% to +2.75%, +2% to +10% and -20% to -5.7% respectively of that of the way prediction cache architecture for all the cache configurations and benchmark program sets, when processes share data among them.

The ECAT of the PASP cache, SMPASP cache with direct-mapped shared set, and the SMPASP cache with 2-way set-associative shared set is always the same for all configurations when the processes have no sharing among them. The ECAT of the PASP / SMPASP cache architecture varies from -6.85% to +3.6% of that of the conventional cache architecture for all the cache configurations and benchmark program sets. The ECAT of the PASP / SMPASP cache architecture varies from -4.8% to +4.8% of that of the way prediction architecture for all the cache configurations and benchmark program sets. The PASP cache architecture performance degrades for shared benchmark program sets. The effective cache access time of SMPASP cache with 2-way set-associative shared set is less than that of the PASP cache and SMPASP cache with direct-mapped shared set for all the cache configurations and benchmark program sets, when processes have data sharing among them. The effective cache access time of the PASP cache, SMPASP cache with direct-mapped shared set, and SMPASP cache with 2-way set-associative shared set architecture varies from -10% to +0.1%, -3.5% to +4.4% and -1.4% to +5% respectively of that of the conventional cache architectures for all the cache configurations and benchmark program suite. The effective cache access time of the PASP cache, SMPASP cache with direct-mapped shared set, and SMPASP cache with 2-way set-associative shared set architecture varies from -9% to +1.4%, -1.9% to +4.8% and -0.6% to +5.5% respectively of that of the way prediction cache architecture for all the cache configurations and benchmark program sets.

5.7.3 Energy Consumption Measurement

Here, the energy components for analysis are obtained by using the power estimation model, the eCACTI cycle simulator [Mamidipaka 2004]. The eCACTI simulates any specified cache architecture and measures the dynamic and leakage energy of different cache components for various nanometer technologies. These dynamic and leakage energy component values are then used to find the dynamic, leakage and total energy consumption of different architecture implementations such as the conventional cache, way prediction cache, PASP cache, SMPASP cache with direct-mapped shared set and SMPASP cache with 2-way set-associative shared set. With the help of the above experimental results such as the cache hit rate, prediction hit rate, victim hit rate, shared hit rate for a given input program set (both for shared and independent processes), the dynamic, leakage and total energy of the cache, while executing these programs is obtained.

The dynamic, leakage and the total power consumption of various cache architectures for different nanometer technologies while running SPEC95 benchmark program sets as independent processes is shown in Figure 5.44, 5.45 and 5.46 respectively. The configuration used for the given analysis is an 8K, 16B, 4 – way set-associative cache with a context switching duration of 500 traces and the input program set 6, as given Table 5.1. From the results, it is evident that the PASP and SMPASP architectures save significant amount of power when compared to the conventional and way prediction schemes. The power components for the PASP, SMPASP with direct-mapped shared set and SMPASP with 2 – way set-associative shared set are equal for the instance of independent processes, as here, the shared set of the SMPASP architecture is completely shutdown.

The dynamic, leakage and total power consumption of the PASP and SMPASP cache architectures executing independent processes is always lesser than that of its conventional and way prediction counterparts, irrespective of nanometer technology in use. The dynamic power consumption of the PASP and SMPASP cache architectures is less as 75% of the main cache is in sleep state throughout the execution and its high first cycle hit rate makes the victim cache usage minimal. The dynamic power saving of the

PASP and SMPASP architectures over the conventional and way prediction cache varies from 61.6% to 62.03% and 8.76% to 8.873% respectively. The way prediction cache saves 57.913% to 58.331% of dynamic power of that of the conventional cache architecture. It is also clear from the figure depicted below that the dynamic power consumption of each of the architectures reduces with advancement in technology. All the cache architectures reduce their dynamic power consumption by at least 8 times as the technology advances from 180nm to 70nm.

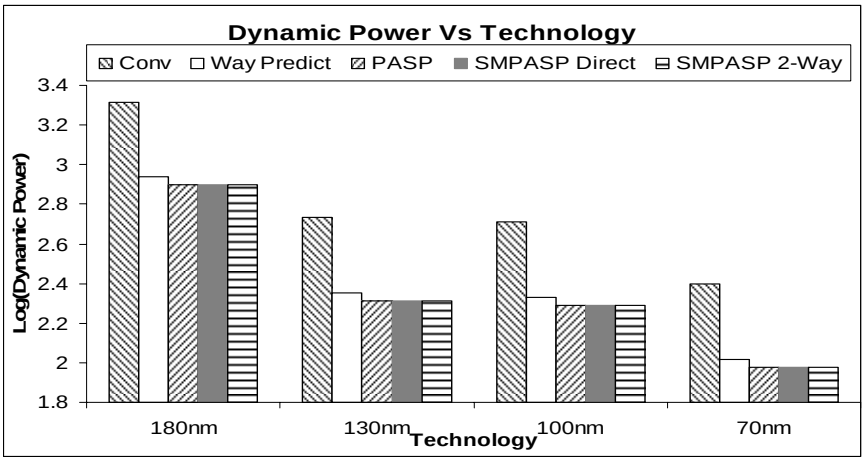

Fig. 5.44: Dynamic power consumption of various architectures Vs Technology for an 8K, 4-way set-associative cache with 16B line size and context switch duration of 500 references.

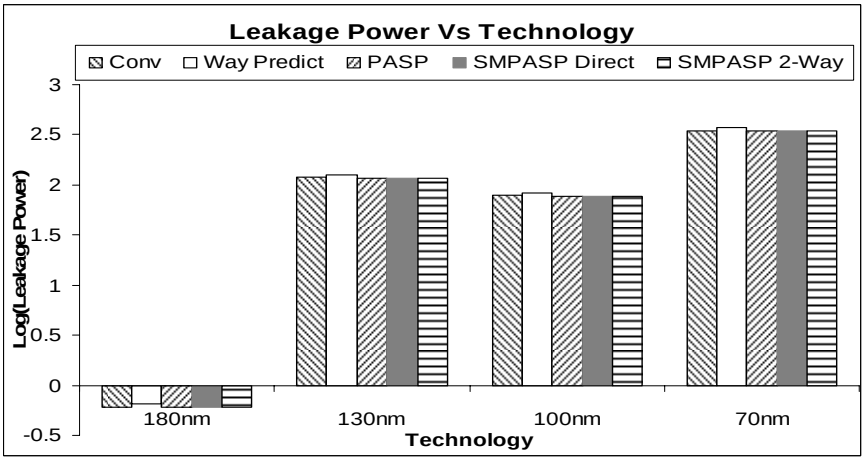

Fig. 5.45: Leakage power consumption of various architectures Vs Technology for an 8K, 4-way set-associative cache with 16B line size and context switch duration of 500 references

The leakage power consumption of the PASP and SMPASP cache architectures is lesser than that of the conventional and way prediction cache architectures due to its lower effective cache access time, high first cycle hit rate and high cache hit rate. The leakage power saving of the PASP and SMPASP architectures over their conventional and way prediction counterparts is around 0.78% and 7.44% respectively. The leakage power consumption of the conventional cache is lesser than that of the way prediction cache by 7.2%. This is because of the extra cycle penalty during a prediction miss and poor first cycle and cache hit rates. It is also observed that the leakage power consumption of each of the architectures increases with advancement in technology. All the cache architectures increase their leakage power consumption by around 572 times as the technology moves from 180nm to 70nm.

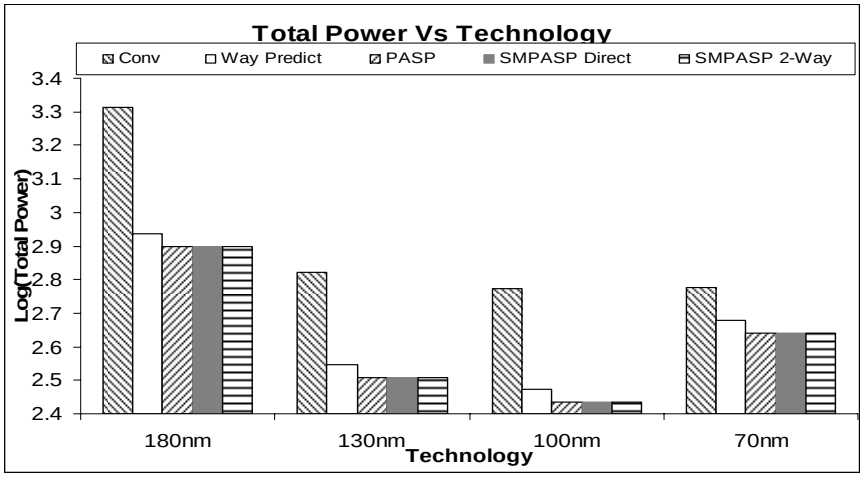

Fig. 5.46: Total power consumption of various architectures Vs Technology for an 8K, 4-way set-associative cache with 16B line size and context switch duration of 500 references

The total power consumption of cache architecture is the sum of its dynamic and leakage power components. So the same trend mentioned above is followed by total power consumption as well. The total power consumption of the PASP and SMPASP cache architectures is lower than that of the conventional and way prediction cache architectures. It is evident from the results that initially (from 180nm to 100nm) the total power consumption decreases with advancement in technology. This performance improvement is mainly attributed to the dynamic power consumption reduction. But the advancement in technology causes the leakage power consumption to rise for all the

cache architectures, which results in the increasing total power consumption with further technology advancement. This fact can be very well observed as the technology advances from 100nm to 70nm, where the total power consumption is pretty high compared to that of the 100nm technology. This increase in total power consumption is thus due to the huge increase in leakage power consumption with technology advancement. The total power saving of the PASP and SMPASP architectures over the conventional and way prediction cache architectures varies from 26.494% to 61.5829% and 7.75% to 8.76% respectively. The way prediction cache saves 20.32% to 57.893% of the total power of conventional cache architecture.

The dynamic, leakage and the total power consumption of various cache architectures for different nanometer technologies while running SPEC95 benchmark program sets as processes with data sharing among them is shown in Figure 5.47, 5.48 and 5.49 respectively. The configuration for this analysis is an 8K, 16B, 4 – way set-associative cache with a context switching duration of 500 traces and input SPEC program sets as set no. 6 shown in Table 5.1 wherein 26.35% of the data is shared. From the figures, it is evident that the PASP and SMPASP architectures save significant amount of power compared to the conventional and way prediction schemes. The dynamic and total power consumption of the PASP and SMPASP cache with direct mapped shared set and SMPASP with 2-way set-associative shared set architectures executing processes exhibiting data sharing among them is always lesser than that of the conventional and way prediction cache architectures, irrespective of the nanometer technology in use.

It is clear from the results that the dynamic power consumption of each of the architectures reduces with the advancement in technology. All the cache architectures reduce their dynamic power consumption by at least 8 times, as the technology moves from 180nm to 70nm. The dynamic power consumption of the SMPASP cache with direct mapped shared set is lower than that of any of the other cache architectures. The dynamic power consumption of the SMPASP cache with direct mapped shared set performs slightly better than the SMPASP cache with 2-way set-associative shared set because of the extra power required for the functioning of the multiplexer, comparators and data output. The dynamic power saving of the SMPASP cache with direct-mapped shared set architecture over the conventional, way prediction, PASP and SMPASP cache

with 2-way set-associative shared set varies from 65.1% to 65.5%, 16.86% to 16.92%, 11.67% to 11.74%, and 0.15% to 0.45% respectively. The dynamic power saving of the SMPASP cache with 2-way set-associative shared set architecture over the conventional, way prediction and PASP varies from 64.93% to 65.41%, 16.48% to 16.8%, and 11.33% to 11.56% respectively. The dynamic power saving of the PASP cache architecture over the conventional and way prediction varies from 60.45% to 60.89%, and 5.81% to 5.92% respectively.

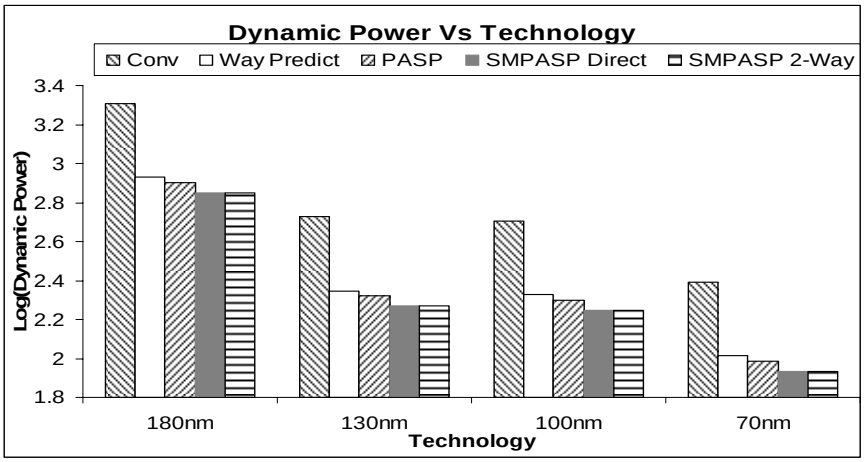

Fig. 5.47: Dynamic power consumption of various architectures Vs Technology for an 8K, 4-way set-associative cache with 16B line size and context switch duration of 500 references

The leakage power consumption of the conventional cache architecture is lower than that of the way prediction, PASP and SMPASP cache architectures. The SMPASP cache architecture shows extra leakage power consumption because of the additional shared and victim sets, while the PASP cache architecture shows extra leakage power consumption because of the victim set. The leakage power consumption of the conventional cache architecture is lower than that of the way prediction cache because of the extra cycle overhead to handle a prediction miss in a way prediction cache. The leakage power consumption of the PASP cache architecture is lower than that of the way prediction cache architecture. This is because the victim set has very less leakage power consumption in comparison to the effective cache access time reduction over that of the way prediction cache. The SMPASP cache with 2-way set-associative shared set has lower leakage power consumption in comparison to that of the SMPASP cache with

direct-mapped cache because of the reduction in the effective cache access time. Each of the cache architectures increases its leakage power consumption by around 572 times, as technology advances from 180nm to 70nm. The leakage power savings of the conventional cache architecture over way prediction, PASP, SMPASP with direct mapped shared set and SMPASP with 2-way set associative shared set cache is around 6.61%, 2.08%, 9.72% and 9.2% respectively. The leakage power consumption of the PASP cache architecture is lower than that of the way prediction, SMPASP with direct mapped shared set and SMPASP with 2-way set-associative shared set cache architectures by 4.62%, 7.8% and 7.27% respectively. The leakage power consumption of the way prediction cache architecture is lower than that of the SMPASP with direct-mapped shared set and SMPASP with 2-way set-associative shared set cache architectures by 3.33%, and 2.77% respectively. The leakage power consumption of the SMPASP with 2-way set-associative shared set cache architecture is lower than that of the SMPASP with direct-mapped shared set cache architecture by around 0.57%.

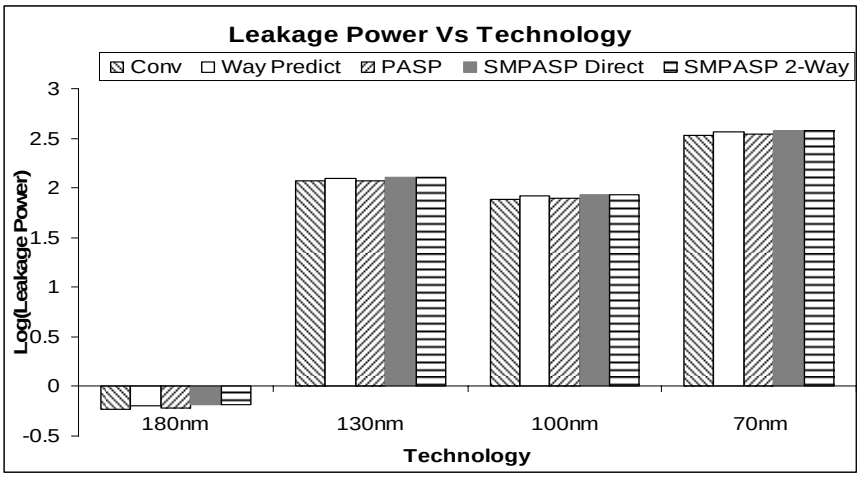

Fig. 5.48: Leakage power consumption of various architectures Vs Technology for an 8K, 4-way set-associative cache with 16B line size and context switch duration of 500 references

The total power consumption of the SMPASP cache with direct-mapped shared set is better than that of all the other cache architectures for technologies older than 180nm. For cache technologies greater than 100nm and less than 180nm, the SMPASP cache with 2-way set-associative shared set architecture is better than all the other architectures. For technologies like 70nm, the PASP cache architecture gives lesser total power

consumption than the other architectures. The total power consumption increases for SMPASP cache because of the huge increase in leakage power with advancement in technology. For 70nm technology, the majority of power consumption (58% in conventional cache, 78% in way prediction and PASP, 82% in SMPASP) is due to the leakage power component. It is observed from the results that initially (from 180nm to 100nm), the total power consumption decreases with advancement in technology. This performance improvement is mainly because of the reduction in the dynamic power consumption. But the advancement in technology increases the leakage power consumption of all the cache architectures, which results in increased total power consumption with technology advancement. This can be seen in 70nm technology, where the total power consumption increases when compared to that of the 100nm technology.

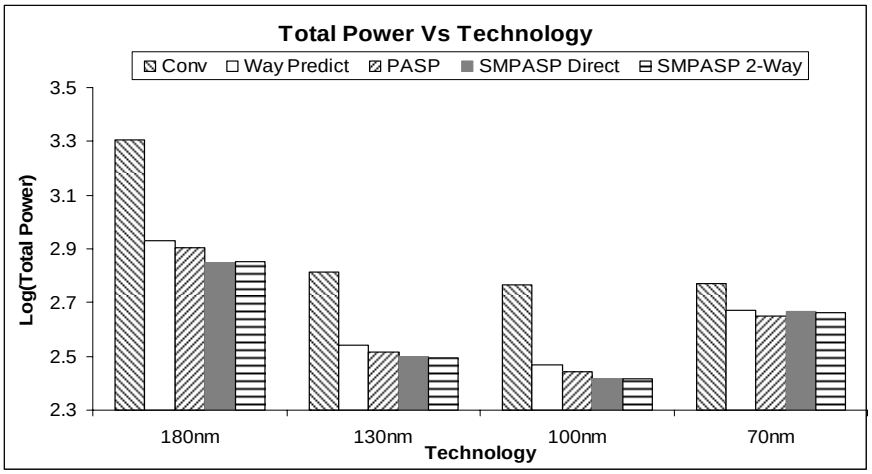

Fig. 5.49: Total power consumption of various architectures Vs Technology for an 8K, 4-way set-associative cache with 16B line size and context switch duration of 500 references

The total power savings of the SMPASP cache with 2-way set-associative shared set cache architecture over the conventional, way prediction, PASP and SMPASP cache with direct mapped shared set cache architectures varies from 51.9% to 64.9% (21.58% for 70nm), 9.74% to 16.47% (1.46% for 70nm), 4.53% to 11.32% (-3.63% for 70nm) and -0.45% to 0.15% respectively. The performance of the SMPASP with direct-mapped shared set is almost the same as SMPASP cache with 2-way set-associative shared set cache architecture. The PASP cache saves 49.62% to 60.43% (24.33% for 70nm), 5.46% to 5.81% (4.91% for 70nm) of the total power consumed by the conventional and way

prediction cache architectures respectively. The way prediction cache saves 46.72% to 57.99% (20.42% for 70nm) of the total power consumed by the conventional cache architecture.

5.8 CONCLUSION

Here, the two proposed schemes – the process aware selective placement (PASP) and the shared memory process aware selective placement (SMPASP) caching scheme that efficiently handles shared data among processes – are discussed in detail. They are also evaluated experimentally and compared with the existing cache architectures with respect to different performance metrics and the results are analyzed with prime focus on reduction in energy consumption and cache hit rate performance enhancement. Table 5.2 presents a comprehensive summary of this evaluation in the form of desired performance characteristics and the corresponding choice of the appropriate cache architectures.

Table 5.2: Ready Reckoner for choice of cache architecture

Desired performance characteristic	**Suitable cache architecture**
Low dynamic power consumption for data access	SMPASP
High cache hit rate	SMPASP / Conventional cache
Low leakage energy for data accesses	Conventional cache
Low Effective Cache Access Time > 32KB <=32KB	 SMPASP Conventional Cache
Low total energy consumption for only non-shared data	PASP / SMPASP
Low total energy consumption for shared and non-shared data access	SMPASP

CHAPTER 6

ENERGY EFFICIENT TASK SCHEDULING

6.1 INTRODUCTION

This chapter discusses various scheduling algorithms which will reduce the preemptions (thus context switches) in a real-time schedule. The main objective behind reducing preemptions is to reduce power consumption due to preemptions. Our approach leads to platform-independent scheduling algorithms with reduced power consumption. This is achieved by applying heuristics to reduce the number of preemptions. We present a static algorithm (IntFragment) and two dynamic priority algorithms (EDFRCS and RMRCS). The latter are variants of the Earliest Deadline First (EDF) and Rate Monotonic (RM) algorithms respectively. We also present a rigorous evaluation of these scheduling algorithms.

6.2 ASSUMPTIONS

1. All tasks are periodic and preemptible.
2. For each task type T, the arrival time of the first job is time 0.
3. For each task type T, the period, denoted by period(T) is known and the period of job J_i, where job J_i is the i^{th} instance of T, denoted by period(J_i) = period(T).
4. All tasks in the task set are in phase. The arrival time of job J_i, denoted by arrTime(J_i) = (i-1)*period(T).
5. For each task type T, the worst case execution time, denoted by exeTime(T) is known. The worst case execution time of job J_i, denoted by exeTime(J_i) = exeTime(T).
6. For each task type T, the deadline, denoted by deadline(T) is known and is the same as period(T). The deadline of job J_i, denoted by deadline(J_i) = i * period(T). This can also be expressed as deadline(J_i) = arrTime(J_i) + period(J_i).

6.3 A BRUTE – FORCE ALGORITHM FOR MINIMIZING PREEMPTIONS

We present an offline algorithm that inspects valid schedules to find one with minimum preemptions. This takes O(H!) time, where H is the hyper-period and we use it as a standard to measure the effectiveness of other algorithms.

6.3.1 BRUTE – FORCE ALGORITHM

Input: Hyper-period H and a list of job records, Jobs (ordered based on arrTime)

Output: A feasible schedule**,** if it exists and the least number of preemptions.

Steps

1. Generate all schedules P of Jobs i.e. divide each job into sub-jobs of unit execution time and compute all schedulable permutations of the list of sub-jobs.
2. MinConSw = H; CurSchedule=first schedule in P.
3. For each permutation P_i in P,
    a. Check if P_i is feasible (All jobs in P_i meet its deadline)
    b. If yes, then count the number of preemptions, say m';
        If (m' < MinConSw) , then MinConSw=m'; CurSchedule=P_i.
4. If (MinConSw = H) then output 'Infeasible'
5. Else output CurSchedule and MinConSw.

Finding all permutations**,** though offline**,** for a given task set (S) is impractical**,** if H is large. Some of the optimization conditions like 'only one job of a task will be available in the ready queue at a time', 'if any job misses its deadline, the schedule is not feasible' etc**.** can be introduced to reduce the number of feasible permutations. This requires a good understanding of the assumptions made for the real-time scheduling. Another way of reducing the complexity is to introduce heuristics to simplify the search and obtain a near-optimal solution. This may adversely affect efficiency and schedulability.

Required:

An algorithm which can compute a feasible schedule (if one exists) in polynomial time**,** such that the number of preemptions in the schedule is low. One such heuristic to find a feasible schedule with minimum number of preemptions is IntFragment scheduling algorithm.

6.4 INTFRAGMENT ALGORITHM

The IntFragment is an offline, hard real-time scheduling algorithm for a periodic task set that reduces the number of preemptions. The heuristic used for minimizing the number of preemptions is to minimize the fragmentation of schedulable intervals. Under the assumptions in section 6.2, given a list of task types L**,** ordered by their periods, one can pre-compute the following easily:

Hyper-period H and a list of job records Jobs, lexicographically ordered by the key (p(t(j)), t(j)), where each record has a job identifier j, task type t(j), deadline d(j), arrival time a(j) and execution time e(j). The key (p(t(j)), t(j)) orders tasks based on the period p(t(j)). If the period of tasks is the same, then the task number (which is unique) is assigned as the key. The basic idea behind the IntFragment algorithm is to provide the maximum fragment size for the greater period tasks to execute. This is achieved by executing an even instance of the task (even job: job having value of odd variable as false) and its immediate next odd instance of the same task (odd job) as distant in time as possible. The execution of the previously mentioned odd instance of the task (odd job) and its immediate next even instance of the same task (even job) will be as close as possible. This allocation results in creating a big fragment in the schedule. With the help of an example we discuss the working of IntFragment algorithm.

Table 6.1: Task list for the schedule

Task	Arrival Time	Period	Exec. Time
A	0	2	1
B	0	8	4

Table 6.2: Job list derived from table 6.1

Job	Arrival Time	Deadline	Exec. Time
A1	0	2	1
A2	2	4	1
A3	4	6	1
A4	6	8	1
B	0	8	4

Table 6.1 provides the task set S. Table 6.2 provides the arrival time, deadline and execution time of all the jobs corresponding to each of the tasks in Table 6.1. In this case**,** IntFragment heuristic is applied as follows:

Step 1 Take Task A's jobs from Table 6.2 (A1, A2, A3, and A4). Assign odd as true.

Step 1.1 The variable odd is true, so schedule A1 in the first feasible slot (i.e. time 0 to 1) and A4 in the last feasible slot (i.e. time 7 to 8) as shown in Figure 6.1. Change odd to false and continue.

Step 1.2 The variable odd is false, so schedule A2 in the last feasible slot (i.e. time 3 to 4) and A3 in the first feasible slot (i.e. time 4 to 5) as shown in Figure 6.2. As all the jobs in Task A have been scheduled, move to Task B.

Step 2 Take Task B's job from Table 6.2. As Task B has only one job, schedule job B in the first feasible slot (time 1 to 3 and time 5 to 7), as shown in Figure 6.3. At the end of time slot 3, it has to be preempted and rescheduled at the end of time slot 5. The resultant schedule has one preemption.

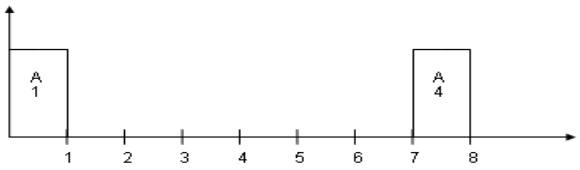

Fig. 6.1: Gantt chart after step 1.1

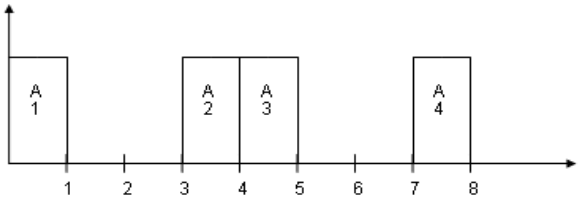

Fig. 6.2: Gantt chart after step 1.2

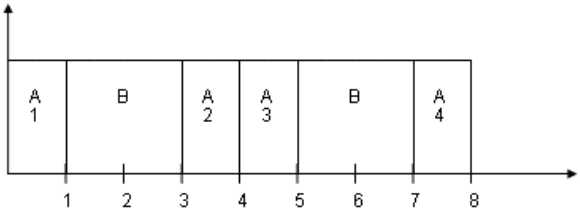

Fig. 6.3: Gantt chart after step 2

This idea can be written as IntFragment algorithm as follows:

6.4.1 INTFRAGMENT ALGORITHM

Input: Hyper-period H and a list of job records J (ordered as above)

Output: A feasible schedule**,** if it exists and the number of preemptions.

Steps

1. odd = true
2. Let J_i, J_{i+1}, … J_k be all the jobs of task T
   a. If (odd) then schedule J_i, J_k in the first and last feasible slots respectively
   b. Else schedule J_i, J_k in the last and first feasible slots respectively
   c. odd = !odd
   d. i=i+1; k=k-1;
   e. Repeat steps 2.a to 2.d until k <= i
   f. If (k==i), then schedule J_i in the first feasible slot
3. Repeat steps 1 and 2, until no more tasks are left
4. Output the schedule and the number of preemptions in the schedule

The main advantage of this scheme is the simplicity of its heuristics. The resultant IntFragment schedule is likely to reduce preemptions**,** as it reduces fragmentation of intervals**,** thereby allowing jobs to fit into these intervals, thus executing without preemptions. An indirect**,** but equally important benefit is improving the cache impact. As the IntFragment algorithm combines jobs of the same task together, the amount of cache flushes in the schedule is reduced. The resultant schedule produced by IntFragment algorithm helps in saving time and power by reducing preemptions and cache flushes.

6.4.2. SCHEDULABILITY ARGUMENTS FOR INTFRAGMENT ALGORITHM

Algorithm IntFragment may fail to find a feasible schedule for some inputs that admit feasible schedules.

For instance, consider the input from Table 6.3. Table 6.4 gives the arrival time, deadline and execution time of all the jobs corresponding to each of the tasks in Table 6.3.

Table 6.3: Task list for which IntFragment algorithm fails to find a valid schedule

Task	Arrival Time	Period	Exec. Time
A	0	3	1
B	0	5	3

Table 6.4: Job list derived from table 6.3

Job	Arrival Time	Deadline	Exec. Time
A1	0	3	1
A2	3	6	1
A3	6	9	1
A4	9	12	1
A5	12	15	1
B1	0	5	3
B2	5	10	3
B3	10	15	3

In this case**,** IntFragment heuristic is applied as follows:

Step 1 Take Task A's jobs from Table 6.4 (A1, A2, A3, A4 and A5). Assign odd as true.

Step 1.1 The variable odd is true**.** So**,** schedule A1 in the first feasible slot (i.e. time 0 to 1) and A5 in the last feasible slot (i.e. time 14 to 15), as shown in Figure 6.4. Change odd to false and continue.

Step 1.2 The variable odd is false**.** So**,** schedule A2 in the last feasible slot (i.e. time 5 to 6) and A4 in the first feasible slot (i.e. time 9 to 10), as shown in Figure 6.5. Change odd to true and continue.

Step 1.3 As pointers first and last point to the same location, schedule A3 in the first feasible slot (i.e. time 6 to 7), as shown in Figure 6.6. As all the jobs in Task A have been scheduled**,** move to Task B.

Step 2 Take Task B's jobs from Table 6.4 (B1, B2, and B3). Assign odd as true.

Step 2.1 The variable odd is true**.** So**,** schedule B1 in the first feasible slot (i.e. time 1 to 4) and B3 in the last feasible slot (i.e. time 11 to 14), as shown in Figure 6.7. Change odd to false and continue.

Step 2.2 The variable odd is false. So, schedule B2 in the first feasible slot. B2 needs 3 units of execution time between time 5 and 10. But A2, A3 and A4 are occupying one unit each between time 5 and 10, which makes B2 unschedulable as shown in Figure 6.8. The scheduling algorithm failed to provide a valid schedule**,** though there exists one.

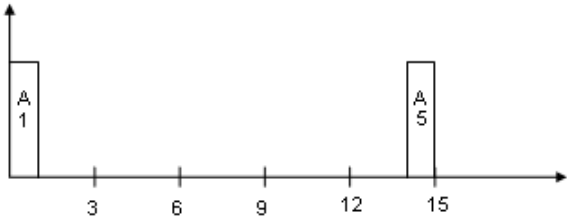

Fig. 6.4: Gantt chart after step 1.1

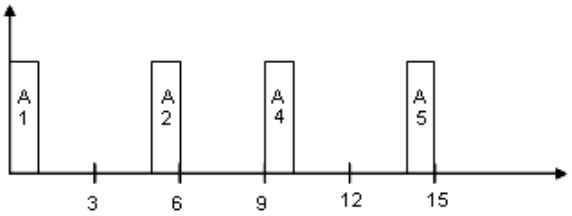

Fig. 6.5: Gantt chart after step 1.2

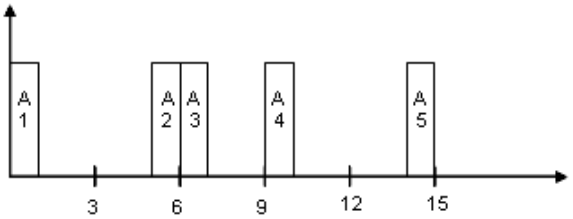

Fig. 6.6: Gantt chart after step 1.3

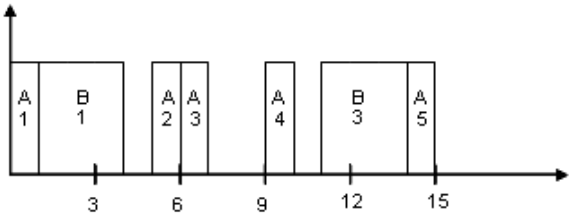

Fig. 6.7: Gantt chart after step 2.1

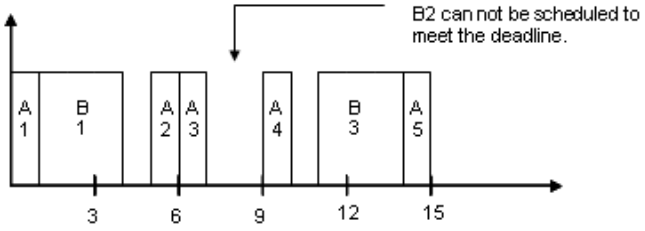

Fig. 6.8: Gantt chart after step 2.2

Such failures happen typically when the utilization is high. The following schedulability test states a sufficient condition for the algorithm to find a feasible schedule.

Given a set of N independent tasks on a uniprocessor under the assumption in 6.2, if for each task i, $(p(i) - \Sigma_{j<i} (\lceil p(i) / p(j) \rceil * e(j))) <= e(i)$ → (6.1) where p(i) is the period of task i and e(i) is the execution time of task i, then a feasible schedule (if one exists) can be found using IntFragment algorithm.

6.4.3. CORRECTNESS OF INTFRAGMENT ALGORITHM

Given a set of N independent tasks on a uniprocessor under the assumption in 6.2, Algorithm IntFragment generates a feasible schedule, if one exists and if the schedulability test is satisfied.

6.4.4. PROOF OF CORRECTNESS

By induction on the number of jobs per task T_i.

Algorithm makes sure all jobs of higher frequency tasks already scheduled

In each step of the algorithm, there are two cases to consider:

Step 1 Two selected jobs are scheduled in the first and last feasible slots of the interval. Then the other jobs must be schedulable in the remaining part of the interval.

(i) For the job executing in the first feasible slot, the algorithm follows task level fixed priority scheduling (RM) and will be feasible if it is RM feasible.

(ii) For the job executing in the last feasible slot, the algorithm guarantees a feasible execution if no job of lower frequency tasks has deadline between this job's arrival time and deadline.

Step 2 Two selected jobs are scheduled such that they split the interval into two intervals. The remaining jobs must still be schedulable, albeit requiring one extra preemption.

By inductive hypothesis**,** if the schedulability of the smaller job list(s) is assumed to be true, then by an inductive step, the full job list is schedulable.

End of Proof

6.4.5. ANALYSIS OF INTFRAGMENT ALGORITHM

6.4.5.1. Complexity of the Algorithm

Claim: The worst case time complexity of Algorithm IntFragment is

$$p_{max} * \Sigma_{T \in Tasks} H/p(T) \quad \rightarrow \quad (6.2)$$

where p_{max} is the maximum among the periods of all tasks, H is the hyper-period, and p(T) is the period of task T.

Proof: The outer loop in Algorithm IntFragment executes n times where n is the number of tasks. The inner loop executes k (T) times where k(T) = H/p(T) is the number of jobs or instances of task T. So, the total number of iterations is $\Sigma_{T \in Tasks}$ H/p(T). And in each iteration**,** the amount of work done to find a feasible slot is less than p(T). Thus the total work done is p_{max} * $\Sigma_{T \in Tasks}$ H/p(T).

End of Proof

6.4.5.2. Quality of the Schedule

Since the objective is to minimize the number of preemptions**,** the algorithm is evaluated using the metric described below and compared with other algorithms.

Consider the example of Tables 6.1 and 6.2. Algorithm IntFragment produces a schedule with one preemption as shown in Figure 6.3. In comparison, for the same input, both the RM algorithm and the EDF algorithm produce a schedule with 3 preemptions as shown in Figure 6.9.

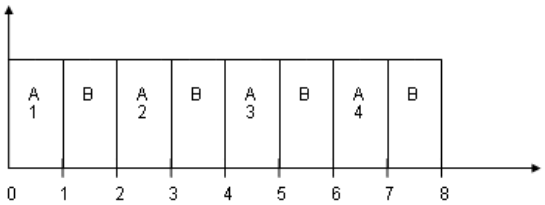

Fig. 6.9: Schedule obtained by Rate Monotonic and Earliest Deadline First Algorithms

Although Algorithm IntFragment typically fares better than other real-time priority-based dynamic scheduling algorithms in reducing preemptions, it does not necessarily produce a schedule with the least number of preemptions. This is explained with the help of the following example.

Table 6.5: Task list for which IntFragment algorithm performs better than the other scheduling algorithms like Rate Monotonic and EDF

Task	Arrival Time	Period	Exec. Time
A	0	2	1
B	0	10	4

Table 6.6: Job list derived from table 6.5

Job	Arrival Time	Deadline	Exec. Time
A1	0	2	1
A2	2	4	1
A3	4	6	1
A4	6	8	1
A5	8	10	1
B	0	10	4

Consider the input from Table 6.5. Table 6.6 gives the arrival time, deadline and execution time of all the jobs corresponding to each of the tasks in Table 6.5.

For this input, Algorithm IntFragment produces a schedule with 2 preemptions as shown in Figure 6.10**,** while there exists a feasible schedule with one preemption as shown in Figure 6.11. Thus, Algorithm IntFragment is an approximation algorithm.

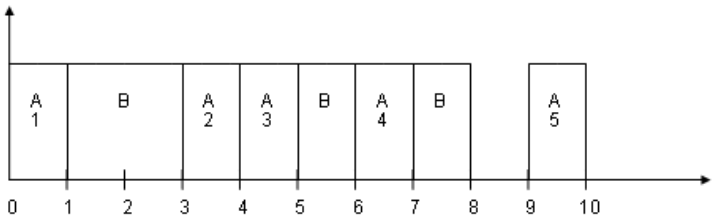

Fig. 6.10: Schedule obtained by the IntFragment algorithm

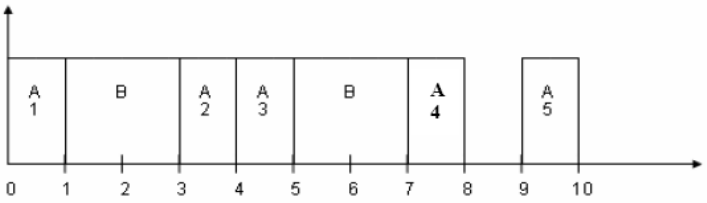

Fig. 6.11: Schedule obtained by the Brute-Force Technique

Worst-case Approximation Claim:

Let I be an input of T tasks. If I admits a feasible schedule and satisfies the Schedulability Test 6.2.4, then Algorithm IntFragment will produce a feasible schedule, with at most $O(T^2)$ preemptions.

Although this may imply that a schedule produced by Algorithm IntFragment is infinitely worse compared to Brute-Force minimal preemption schedule, in practice, it produces far fewer number of preemptions in most cases.

6.5 REDUCED CONTEXT SWITCH (RCS) SCHEDULING ALGORITHMS (EDFRCS AND RMRCS)

6.5.1 ALGORITHMS

This approach to preemption (context switch) reduction is similar to the one used in MLLF [Oh 1998] and MMUF [Vahid 2005]: defer the preemption of an active process when it can be guaranteed that any process that is delayed will not miss its deadline. But the heuristic in MLLF / MMUF is weak: preemption is deferred only when there is a tie in the priority (i.e. laxity in this case): the possibility of delaying a higher priority process without the delayed process missing the deadline is not considered.

We adopt a more aggressive approach that considers deferrals in the preemption of an active process even in the presence of higher priority processes – without affecting the schedulability of the delayed processes. The adopted heuristic below maximizes the extension period of the active process by considering the deadlines of all processes in the ready queue whose priority is same as or higher than the active process. Based on this heuristic, a scheduling algorithm (RCSS) is developed, parameterized by a priority function. By choosing the appropriate priority function, variants of EDF (named EDFRCS) and of RM (named RMRCS) are obtained from this RCSS algorithm. The schedulability of tasks is preserved by these variants i.e. the variant (say RMRCS) is optimal if and when the original algorithm (say RM) is optimal.

The following notation is used in the RCSS algorithms:

readyQ(t) : the ready queue at time t, ordered by priority.

priority(J) : the priority of Job J.

extension_time(J,t): maximum possible extension time for job J at time t.

deadline(J) : deadline of a job J.

period(J) : period of a job J (i.e period of task T, where J is an instance of T).

execution_time(J) : execution time of a job J.

remaining_time(J,t) : execution time of a job J, still remaining at time t.

slack(J,t) : deadline(J) – t – remaining_time(J,t)

Reduced Context Switches Scheduling (RCSS) Algorithm

Input: A list L of tasks T1, T2, … Tn, their periods and execution times and
A priority function *priority* that is job-level fixed.

Output: A feasible schedule for L or failure.

begin

Let Cur be the job with the highest priority; schedule Cur;

For every time unit t when there is at least one arrival or a departure or a deferred switch

Let J be the job with the highest priority in readyQ(t) .

if (Cur is to depart)

then Cur = J ; schedule Cur;

else if (priority(Cur) >= priority(J))

then continue with Cur;

else ExtensionTime_Cur = extension_time(Cur, t);

if (ExtensionTime_Cur==0)

then preempt Cur; Cur = J; schedule Cur;

else if (ExtensionTime_Cur > 0)

then mark a deferred switch at t + ExtensionTime_Cur;

continue with Cur upto t + ExtensionTime_Cur;

else fail;

end RCSS

function int extension_time(current_job, t)

begin

Let j1, j2, …, jm be the jobs in readyQ(t) such that

priority(j1) >= priority(j2) >= … >= priority(jm) >= priority(current_job)

return $\min_i$[slack(j_i, t) – $\Sigma_{k<i}$(remaining_time(j_k,t)+ceil((deadline(j_i)-

deadline(j_k))/period(j_k)) * execution_time(j_k))];

end extension_time

Variants of Rate Monotonic (RM) scheduling algorithm and Earliest Deadline First (EDF) are easily obtained from the above algorithm (RCSS) by specifying the appropriate priority function as listed below.

Rate Monotonic with Reduced Context Switches (RMRCS) Algorithm

Input: A list *L* of tasks *T1, T2, ... Tn*, their periods and execution times.

Output: A feasible schedule if L is RM-schedulable, failure otherwise.

begin

(1) Define the priority function as priority(J) = H / period(T) where, T is a task in L, J is a job (instance) of T and H is the hyper-period for L.

(2) Execute RCSS.

end RMRCS

Earliest Deadline First with Reduced Context Switches (EDFRCS) Algorithm

Input: A list *L* of tasks *T1, T2, ... Tn*, their periods and execution times.

Output: A feasible schedule if L is schedulable, failure otherwise.

begin

(1) Define the priority function as priority (J) = -1 * deadline(J) for any job J.

(2) Execute RCSS.

end EDFRCS.

Note on EDFRCS:

1. Observe that in this case, the *fail* statement in Algorithm RCSS will never be reached if the input *L* has a feasible schedule because the scheduling decisions align with EDF.
2. The extension time for any job may be increased further by replacing the use of *ceil* with the use of *floor* in the function *extension_time(),* without affecting feasibility.

End of Note

The working of algorithms (EDFRCS) and (RMRCS) is explained with the help of an example. Consider the following list of tasks given in Table 6.7. Table 6.8 gives the arrival time, execution time and deadline of all the jobs corresponding to each of the tasks in Table 6.7. The jobs in Table 6.8 are arranged in the order of deadline and when the deadlines are the same, then in the order of arrival – as this is the likely arrangement of a queue.

Table 6.7: Task List (L)

Task	Arrival Time	Period	Execution Time
T1	0	4	1
T2	0	5	2
T3	0	20	7

Table 6.8: Job list corresponding to L in Table 6.7 (Hyper-period = 20)

Job (Task)	Arrival Time	Execution Time	Deadline
J1 (T1)	0	1	4
J2 (T2)	0	2	5
J3 (T1)	4	1	8
J4 (T2)	5	2	10
J5 (T1)	8	1	12
J6 (T2)	10	2	15
J7 (T1)	12	1	16
J8 (T3)	0	7	20
J9 (T2)	15	2	20
J10 (T1)	16	1	20

The working of EDFRCS is illustrated with the above list of jobs. The resultant schedules till time t=4 and till time t=7 is shown in Figure 6.12 and Figure 6.13 respectively. The final schedule for this example will be as in Figure 6.14.

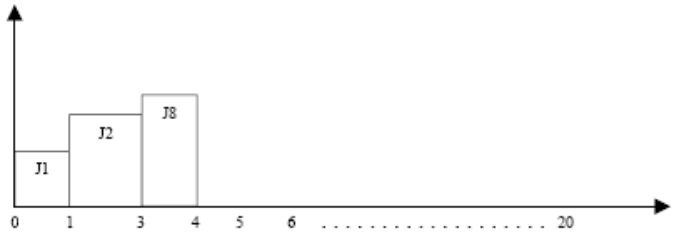

Fig. 6.12: Intermediate schedule up to t = 4

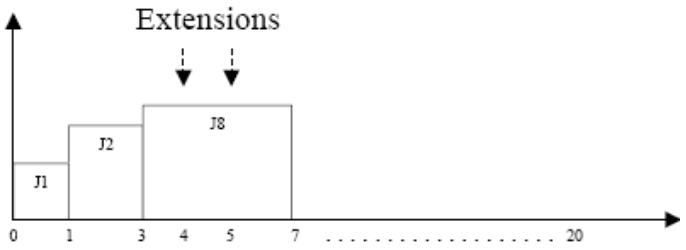

Fig. 6.13: Intermediate schedule up to t = 7

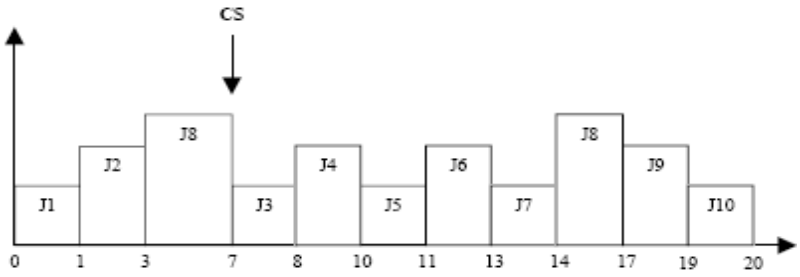

Fig. 6.14: Schedule by EDFRCS for task list in Table 6.7

Observe that EDFRCS outputs a schedule with just one preemption. As opposed to this, RM produces the schedule in Figure 6.15 (number of preemptions = 5), EDF produces the schedule in Figure 6.16 (number of preemptions = 3) and MLLF produces the schedule in Figure 6.17 (number of preemptions = 3). Furthermore, the minimum possible number of preemptions in a feasible schedule is 1 for this task set (this can be verified easily).

For this particular example, RMRCS also outputs the same schedule i.e. the number of preemptions is 1. The extension points are shown in Figure 6.18 and the final schedule is as in Figure 6.19.

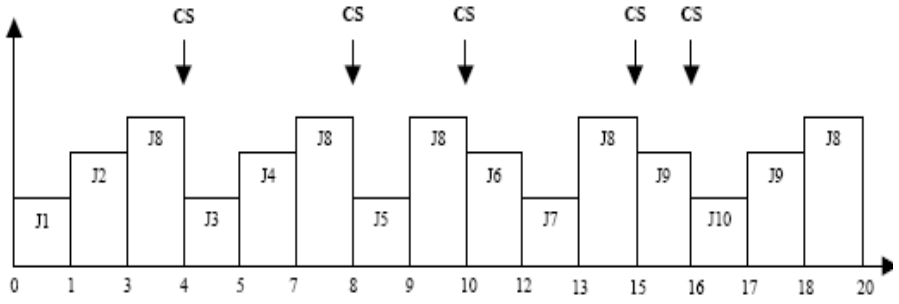

Fig. 6.15: Schedule by RM for task list in Table 6.7

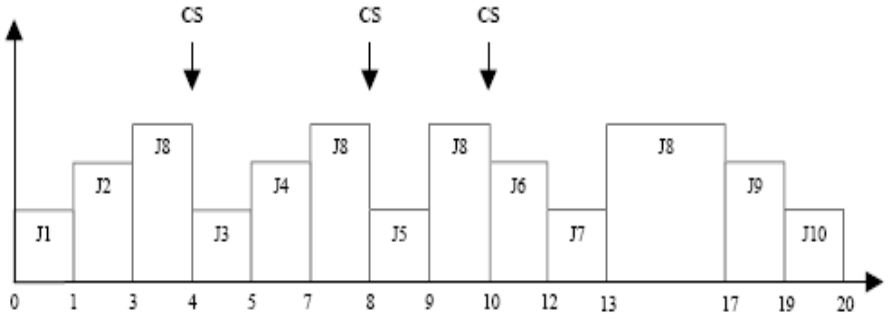

Fig. 6.16: Schedule by EDF for task list in Table 6.7

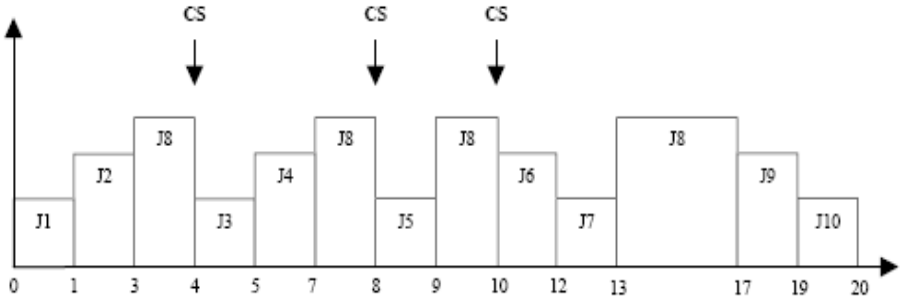

Fig. 6.17: Schedule by MLLF for task list in Table 6.7

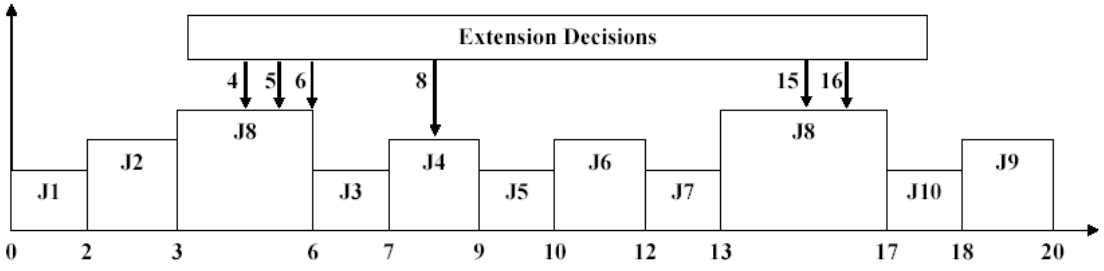

Fig. 6.18: Schedule with extension decision points for RMRCS for task list in Table 6.7

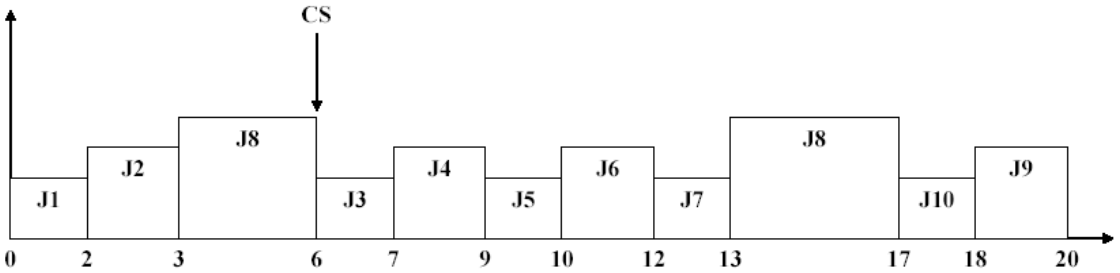

Fig. 6.19: Schedule by RMRCS for task list in Table 6.7

This example demonstrates that RMRCS and EDFRCS are aggressive in eliminating preemptions whenever possible, while the other algorithms are not. Our experimental results in Section 6.5 confirm this argument.

6.5.2 SCHEDULABILITY OF THE ALGORITHMS

The adopted heuristic preserves schedulability of scheduling decisions i.e. RMRCS and EDFRCS are schedulable if and when RM and EDF (respectively) are schedulable.

Some notations used in the proofs below:

- A schedule *S* is a sequence of runs, where each run is a tuple of an identifier, a start time and an end time.
- To identify some runs of a schedule but ignore others, the notation used is:
  ((R1, t1, u1), S1, (R2,t2,u2), S2, …, Sn-1, (Rn,tn,un), Sn)
  where, each tuple *(Ri, ti, ui)* is a run starting at *ti* and ending at *ti+ui* and each *Si* is a sequence of runs – possibly empty – occurring between runs *(Ri-1,ti-1, ui-1)* and *(Ri,ti,ui)*.
- Given a schedule *S*, we use $N_{CS}(S)$ to denote the number of preemptions in *S*.

The following lemma is used in proving the theorems stated below and it can be informally stated as:

The extension step in RCSS – the step that continues the active process – does not affect schedulability.

Lemma 1:

Let *S* be a feasible schedule: *((R1,t1,u1), S1, (R2,t2,u2), S2, …, Sm-1, (Rm,tm,um), Sm)* where, all *Ri, 1 <= i <= m* are runs of the same job *B*. Assume *S* was generated by a priority scheduling algorithm.

Let *U* be the schedule *(((R1,R2...Rk-1,Rk'), t1, u1+u2+…+uk-1+uk'), S1, S2, Sk-1, (Rk'',tk+uk', uk-uk'), Sk, … (Rm,tm,um), Sm),* where, some runs of *B* at the beginning of *S* have been merged into a single run, and one run (*Rk*), has been partly merged.

Assume *extension_time(B, t1+u1) >= (u2+u3+…uk-1+uk')* →(6.3)

Then *U* is feasible and $N_{CS}(U) <= N_{CS}(S)$

Proof:

Let *C* be any job such that *priority(C) >= priority(B)*.

Define *P(C,t) = { D | D* is in *readyQ(t)* and *priority(D) >= priority(C) }*

Define *I(C,t) = { T_D | D* is in *P(C,t)* and *D* is an instance of task T_D *}*

Let *Rem(C,t)* be the time required for all remaining runs - at time *t* - of all jobs of the tasks in *I(C,t)*.

Then by definition of the function *extension_time*,

$slack(C, t) - Rem(C,t) >= extension\_time(B, t)$ →(6.4)

Let S_{ij} be any run of a job *F*, such that $(S_{ij}, t_{ij}, u_{ij}) \varepsilon S_i$, for some *i* s.t. $1 <= i <= k$.

and *priority(F) >= priority(B)*

Then by (**6.4**),

$t_{ij} + u_{ij} + extension\_time(B, t_1+u_1) <= t_{ij} + u_{ij} + slack(F, t_1+u_1) - Rem(F, , t_1+u_1)$

i.e. $t_{ij} + u_{ij} + extension\_time(B, t_1+u_1) <= t_{ij} + u_{ij} + slack(F, t_1+u_1) - Rem(F, t_1+u_1)$

i.e. $t_{ij} + u_{ij} + extension\_time(B, t_1+u_1) <= t_{ij} + u_{ij} + slack(F, t_{ij} + u_{ij}) - Rem(F, t_{ij}+u_{ij})$

which, by definition of *slack* results in

$t_{ij} + u_{ij} + extension\_time(B, t_1+u_1) <= deadline(F)$ →(6.5)

So, by assumption (**6.3**) and **(6.5)**,

$tij+uij+(u2+u3+...+uk-1+uk') <= deadline\ of\ F$.

Thus, any delayed run S_{ij} in schedule *S*, will not cause any run of any higher priority job *F* to miss its deadline i.e. *U* is feasible. Furthermore, observe that the numbers of runs of jobs other than *B* remain unchanged from *S* to *U*; whereas the number of runs of *B* may be reduced, i.e. $\boldsymbol{N_{CS}(U) <= N_{CS}(S)}$

End of Proof

Corollary 1 is a special case of Lemma 1, where all the runs of a particular job are merged into one.

Corollary 1:

Let *S* be a feasible schedule *((R1, t1, u1), S1, (R2,t2,u2), S2, ..., Sm-1, (Rm,tm,um), Sm)* where all *Ri, 1 <= i <= m* are runs of the same job *B*. Assume *S* was generated by a priority scheduling algorithm.

Let *U'* be the schedule *(((R1,R2...,Rm), t1, u1+u2+...+um), S1, S2, Sm)* where all runs of *B* in *S* have been merged into a single run.

Assume that *extension_time(B, t1+u1)>= (u2+u3+…um).*

Then U' is feasible and $N_{CS}(U') <= N_{CS}(S)$

Proof:

By Lemma 1, with k=m and uk = uk'.

End of Proof

Theorem 1:

If a task set is RM-schedulable, then it is RMRCS-schedulable and RMRCS outputs a schedule with no more preemptions than the schedule output by RM.

Proof:

Let *S* be the job set corresponding to the given task set. We need to prove that the schedule generated by RMRCS for *S* is feasible, if there is a feasible schedule generated by RM for *S. In each iteration of the loop in Step (2)of RCSS*, the scheduling decision is

- either the same as the decision RM would take
- or a decision that does not affect feasibility, but may reduce preemptions.

Assume in this case, RCSS is executed with the priority function set by RMRCS.

There are five branches of the *if-then-else* statement in each iteration of the loop in Step(2) of RCSS.

Branch 1: *Cur* terminates; RCSS schedules the highest priority job from *readyQ*. So would RM.

Branch 2: *priority(Cur)* is at least as high as the priority of any job in *readyQ*. RCSS continues to run *Cur*. So would RM.

Branch 3: *extension_time(Cur,t)* returns 0. *Cur* cannot be continued without affecting the schedulability of other jobs. RCSS preempts *Cur* and schedules the highest priority job from *readyQ*. So would RM.

Branch 4: *Cur* can be extended up to *extension_time(Cur,t)*. RCSS extends the execution of *Cur*. This leads to two possibilities:

(a) Some future runs of *Cur* are merged into the current run possibly including a partial run. This is equivalent to transforming *S*, (a feasible schedule output by RM)

((R1, t1, u1), S1, (R2,t2,u2), S2, …, Sm-1, (Rm,tm,um), Sm)

where all *Ri* are runs of *Cur* and *R1* is the current run into a schedule *U*

(((R1,R2...Rk-1,Rk'),t1, u1+u2+…+uk-1+uk'),S1,…,Sk-1, (Rk'',tk+uk',uk-uk'), Sk, … (Rm,tm,um),Sm)

where some subsequent runs of *Cur* in *S* have been merged into the current run *R1*.

By *Lemma 1*, feasibility is invariant under this transformation and $N_{CS}(U) <= N_{CS}(S)$.

(b) All future runs of *Cur* are merged into the current run. This is equivalent to transforming *S*, a feasible schedule output by RM

((R1, t1, u1), S1, (R2,t2,u2), S2, …, Sm-1, (Rm,tm,um), Sm)

where all *Ri, 1 <= i <= m* are runs of *Cur* and *R1* is the current run, into a schedule *U'*

(((R1,R2...,Rm), t1, u1+u2+…+um), S1, S2, Sm)

where all subsequent runs of *Cur* in *S* have been merged into the current run *R1*.

By *Corollary 1*, feasibility is invariant under this transformation and $N_{CS}(U') <= N_{CS}(S)$.

Branch 5: *extension_time(Cur,t)*<0 (i.e.) the current run has consumed a part of the runtime of a higher priority job. RCSS fails but in this case**,** the rest of the jobs would not be schedulable by RM either.

Thus, we have shown that a single iteration of the loop in Step (2) of RCSS makes a decision that is as feasible as RM. Hence, *by induction on the number of iterations of the loop*, we conclude that RMRCS outputs a schedule that is feasible, if RM outputs a feasible schedule for the same input.

Furthermore, each iteration of the loop in Step (2) of RCSS will introduce no additional preemptions than RM would. In fact, as argued above, Branches 1, 2, 3 and 5 agree with a decision RM would make, and Branch 4 may reduce the number of preemptions in comparison with RM.

End of Proof

Theorem 2: If a task set is EDF-schedulable, then it is EDFRCS-schedulable and EDFRCS outputs a schedule with no more preemptions than the schedule output by EDF.

Proof:

Similar to the proof for Theorem 1 with the assumption *priority(J) = -1 * deadline(J)* for any job *J*.

The scheduling decisions in each iteration of the loop in Step(2) of RCSS either agree with the scheduling decisions of EDF (because of the priority assumption) or perform a transformation on an EDF schedule (which preserves feasibility as per *Lemma 1* or *Corollary 1*).

End of Proof

6.5.3 ALGORITHMIC COMPLEXITY

Every scheduling decision of RCSS is either a priority decision or an extension decision. In the former case, the time taken for a scheduling decision is *O(logN),* where *N* is the number of tasks. The *O(logN)* factor arises because the ready queue is at any time t, sorted by priority**.** When a scheduling decision requires the computation of extension period, the time taken is *O(logN + m*m),* where m is the number of jobs of higher priority than the active job. The worst case value for *m* is *O(N)*. Thus the worst case response time of our scheduling algorithm is *O(N*N)*. But in practice, the value of m is more likely to be less than *N*. Particularly for high priority jobs, the value of *m* will be much less than *N*.

6.6 PARAMETERS FOR COMPARISON

We evaluated our algorithms IntFragment, RMRCS and EDFRCS and compared them with other priority scheduling algorithms like RM, EDF, LLF and MLLF. We use the approach in [Buttazzo 2005] in our evaluation. In this section, we briefly review the parameters used for evaluation and their significance.

The task set listed in Table 6.7 is used to illustrate the metrics. The job set corresponding to the above task set along with the deadlines is given in Table 6.8. Also, assume that Figure 6.16 is the resultant schedule by which these jobs are to be executed.

6.6.1 RESPONSE TIME

For all real-time systems, the Response Time of a job is an important quality metric of a schedule.

For a periodic task T, the Maximum Response Time is evaluated over all jobs**:**

Maximum Response time of T = max_i (responseTime(T_i))

where responseTime(T_i) = finish-time(T_i) – arrival-time(T_i)

where T_1, T_2, … are the jobs corresponding to task T.

For example, the Maximum Response Time of T2 is max (3, 2, 2, 4) = 4.

For the purpose of evaluation, the Maximum Response Time values are normalized with respect to execution times of the corresponding tasks, so that a normalized value of 1 corresponds to the least possible response time.

6.6.2 RESPONSE TIME JITTER

In real-time control systems, predictability of the output (or responses) is an important issue. Predictability is affected by variation in the response time among jobs of the same task. This is referred to as Response Time Jitter [Buttazzo 2005] [Marti 2002].

Two different jitters are usually defined - Absolute Response-Time Jitter (ARJ) and Relative Response-Time Jitter (RRJ).

$ARJ(T) = max_i(responseTime(T_i)) – min_i(responseTime(T_i))$

$RRJ(T) = max_i(|(responseTime(T_{i+1}) – responseTime(T_i))|)$

where T_1, T_2, … are the jobs corresponding to task T.

For example, from the above task set, for task T2 with 4 instances (J2, J4, J6 and J9),

ARJ(T2) = max (3, 2, 2, 4) – min (3, 2, 2, 4) = 2.

and RRJ(T2) = max (1, 0, 2, 1) = 2

For the purpose of evaluation, the Jitter values are normalized with respect to the periods of the corresponding tasks, so that a normalized value of 1 corresponds to the worst possible jitter in an optimal schedule.

6.6.3 LATENCY

In real-time control systems, input-output latency of a job is another important metric of task schedules [Buttazzo 2005] [Cervin 2003].

For a task T, the maximum input-output latency is defined as

$L(T) = max_i(finish\text{-}time(T_i) – start\text{-}time(T_i))$

where T_1, T_2, … are the jobs corresponding to task T.

For example, for Task T2 above with 4 instances (J2, J4, J6, and J9),

L(T2) = max{(3–1), (7–5), (12–10), (19–17)} = 2

For the purpose of evaluation, the Latency values are normalized with respect to execution times of the corresponding tasks, so that a normalized value of 1 corresponds to the least possible latency.

6.6.4 SCHEDULING COMPLEXITY

The algorithmic complexity of a priority scheduling algorithm is the product of the number of decision points (i.e. number of scheduling decisions) and the complexity per scheduling decision.

Given a set of tasks S with hyper-period H, the algorithmic complexity for algorithm A is

$T_A(H,S) = decisions(A,H, S) * T_{dec}(A,H, S)$

where *decisions* defines the number of decision points to be made by the scheduling algorithm and T_{dec} defines the time complexity of a single decision.

For the purpose of evaluation**,** the number of decision points is normalized with respect to the number of jobs. The complexity of a scheduling decision depends on implementation issues such as whether the algorithm can be realized using a fixed number of priority levels or that the number of priority levels changes dynamically. In the latter case, the time taken for activities such as insertion or deletion of jobs in the priority queue affects the complexity. The time taken for insertion or deletion is a function of the length of the priority queue [Gooch 1998]. The length of the queue is bounded by the number of tasks for any periodic task set.

6.6.5 PREEMPTION COUNT

Preemptive scheduling in real-time systems has merits and demerits: no online algorithm can be optimal without using preemptions [Liu 2000] [Mok 1983] but on the other hand, preemptions can add significant overhead to the schedule as such [Tan 2002]**.** Preemptions result in both time and energy overheads [Buttazzo 2005][Mok 1983] [Gopalakrishnan 1996] but the precise evaluation of such overheads - particularly the energy overhead - has been lacking. In this work, preemptions initiated due to tasks blocking on their own - say for resource requirements – are ignored and only preemptions introduced by scheduling are considered exclusively, as focus here, is on the evaluation of scheduling algorithms. The impact of preemptions on utilization and energy consumption is not straightforward – particularly due to data movement within the

memory hierarchy. All preemptions may not cause data movement but whenever data movement results, this part of the overhead may be much more significant than that caused by just a switching of tasks. This implies that the time spent in switching contexts may be variable and in turn may increase response time and response time jitter for tasks. The measurement of the overhead due to data movement within the memory hierarchy is beyond the scope of this work. Thus it is restricted to counting the preemptions in the schedule output by an algorithm. This is similar to the approach in [Gopalakrishnan 1996], but here, focus is on experimental evaluation. In the above example (Table 6.7), for the given schedule, the preemption count is 3 (each of which is marked as CS in Figure 6.16).

Liu [Liu 2000] argues that 2*N, where N is the number of jobs, is the upper bound for the number of preemptions in a schedule, decided by job-level priorities. The reasoning is simple: each job may cause a switch once when it starts and once when it ends, observing that if a job is preempted in between, then this preemption gets attributed to the beginning or end of some other job. Of course, this does not apply for fully online algorithms like LLF: for instance, two jobs J1 and J2 arriving at time 0 with the same deadlines 2t and execution times t will be switched every two time units and rather unnecessarily at that. On the other hand, the theoretical upper bound may not be a close estimate of the actual number of preemptions for the other algorithms. For the purpose of evaluation, preemption counts are normalized with respect to the number of jobs i.e. preemptions are considered as a function of the total number of jobs in a schedule. This makes particular sense when the preemption overhead is compared with the overhead due to scheduling decisions, as the number of scheduling decisions usually depends on the number of jobs, among other factors.

The impact of the number of preemptions on utilization (i.e. on time) is theoretically analyzed in [Gopalakrishnan 1996], based on various preemptive models. Here, this impact is evaluated using experimental measurements.

6.6.6 ENERGY CONSUMPTION

As real-time embedded systems often operate under limited battery power, the amount of energy consumed is a critical issue. So, task-scheduling algorithms for such systems need

to be energy-conscious. Scheduling algorithms impact energy consumption in two ways: by the time taken to schedule tasks and by the number of preemptions they induce [Buttazzo 2005][Mok 1983]. Four different components of energy consumption have been attributed to scheduling [Tan 2002]: Timer Interrupt Energy, Scheduling Energy, Context Switch Energy and Signal Handling Energy. The sample experimental values obtained on Linux are cited in [Tan 2002].

Table 6.9: Sample experimental values for energy consumption of Linux OS on arm architecture from Tan et al. [Tan 2002]

Component	Energy (in nJ)
Context Switch Energy	12500
Timer Interrupt	450
Scheduling Energy	1200
Signal Handling Energy	3200
malloc call	123
file *open* system call	2351

Table 6.9 lists energy consumption for a few common operations for the purpose of comparison: even scheduling energy is significant in comparison with calls to *malloc* or *open,* although context switch energy may be an order of magnitude larger than that of scheduling energy. Signal Handling Energy and Timer Interrupt Energy are specific to the platform (i.e., the architecture and the operating system), but fairly independent of scheduling algorithms. It is important to observe that Scheduling Energy and Context Switch Energy are identified separately because each scheduler invocation may not result in a preemption (i.e., a context switch). The energy value cited above under the Context Switch Energy component includes energy consumed by data movement in the memory hierarchy. Isolated measurements of energy consumption are harder because data movements are dependent on many factors including nature of application, input profile, allocation policies etc.

A simpler measure, with a compromise on accuracy, is used in this analysis: the number of preemptions in a schedule. Lesser preemptions always result in less energy

consumption, but may or may not result in significant energy reduction as data movement may still not be reduced.

Another factor in measuring preemption overhead depends upon the task model on a platform: whether tasks are lightweight (ala threads) or heavy weight (ala processes). Thread switching usually causes less overhead than process switching. For instance, [Acquaviva 2003] reports an average of 860 nJ as the energy consumed per thread switch in an eCOS system running on a StrongArm 1100 processor at its peak frequency (221.2MHz). This is an order of magnitude smaller than the 12500 nJ listed above in Table 6.13. Again, using preemption count as the metric abstracts away from these details and allows for the comparison of scheduling algorithms. This comparative analysis of scheduling algorithms uses two metrics for energy consumption: the time spent by the algorithm on scheduling decisions and the number of preemptions in the resultant schedule.

6.7 COMPARATIVE EVALUATION

In this section, we present the results of our evaluation. Experimental measurements for each of the metrics – discussed in section 6.6 - are summarized; along with analysis and comparison. The experimental setup includes simulations of all the seven algorithms and various test suites randomly generated under certain conditions: each test suite is characterized by either a fixed number of tasks with utilization varying from low (50%) to high (100%) or by a fixed utilization with the number of tasks varying from 2 to 20. Each test suite includes 100 different task sets of varying hyperperiods – from 100 to 32000. The results obtained are then averaged over these 100 test suites as appropriate.

Schedulability is not included as a metric here in this evaluation as schedulability analyses of RM, EDF and LLF have been dealt with extensively in the literature, whereas RMRCS, EDFRCS, and MLLF are optimality-preserving variants of RM, EDF, and LLF respectively. The schedulability of IntFragment algorithm is discussed in Section 6.4 extensively. In the cases of RM and RMRCS, only those task sets that are RM-schedulable are considered for the following analyses. This does introduce a bias - albeit a predictable one - in favor of these two algorithms. Wherever this is an issue, this fact is addressed explicitly in the analysis.

6.7.1 RESPONSE TIME

For the purpose of evaluation, here, the response times are normalized over execution times of the corresponding tasks and the tasks are ordered by decreasing frequency. The following plots in Figures 6.20, 6.21 and 6.22 shows the variation of response times of tasks for the various algorithms for task set of a particular size and at a fixed utilization.

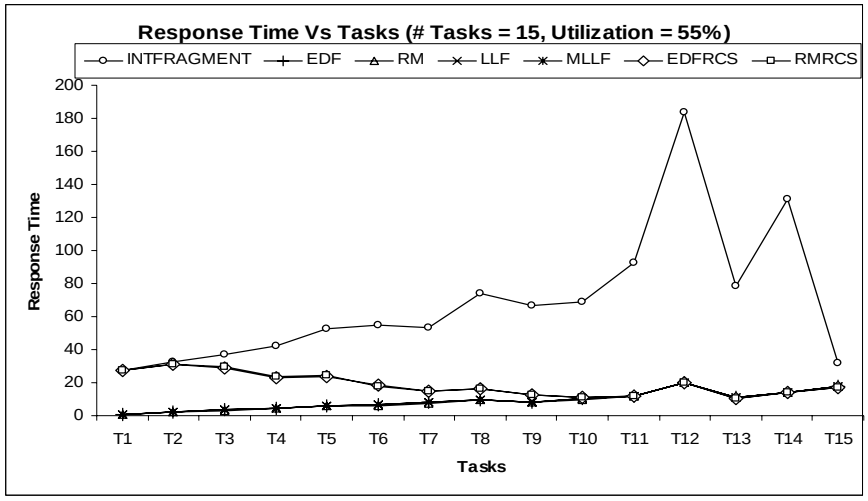

Fig. 6.20: Response Time per Task (# Tasks = 15, Utilization = 55%)

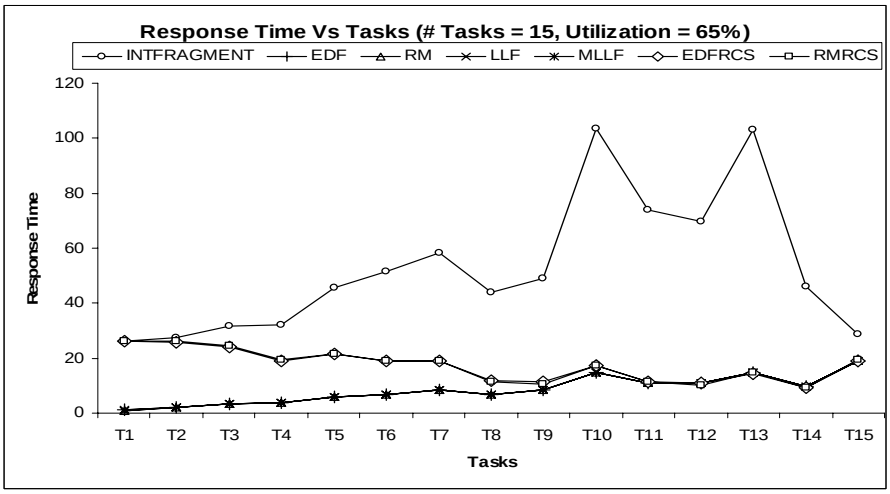

Fig. 6.21: Response Time per Task (# Tasks = 15, Utilization = 65%)

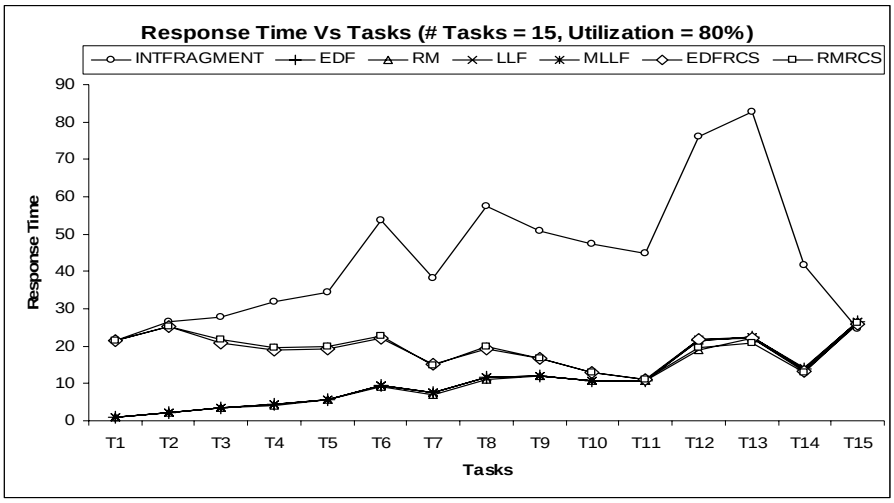

Fig. 6.22: Response Time per Task (# Tasks = 15, Utilization = 80%)

It is observed that RM's response time for the highest frequency task is always 1, irrespective of the number of tasks and the utilization. This behavior is attributed to the fact that RM is a task-level fixed priority scheduling algorithm where the period of a task decides its priority. The traditional algorithms (RM, EDF, and LLF) do not show any real difference – among themselves – in response time behavior and the response time increases exponentially with decreasing frequency of the tasks. Among the preemption reduction algorithms, MLLF results in schedules with same response times as those produced by LLF. But the RCS algorithms (RMRCS and EDFRCS) result in schedules where the response times of higher frequency tasks are much larger than in schedules produced by RM and EDF respectively. This increase in response times can be attributed to the fact that a lower priority task may continue execution in preference to a higher priority task due to the preemption reduction heuristic used by RMRCS and EDFRCS, thus resulting in a higher response time for the higher frequency tasks (than their traditional counterparts) and a lower response time for the lower frequency tasks. This heuristic thus explains the trend exhibited by the RCS algorithms (i.e.) decreasing response time of tasks with decreasing frequency. IntFragment gives the highest response time for all tasks, except for the lowest frequency task at very high utilizations. This highest response time for tasks is owing to the fact that IntFragment schedules the execution of tasks in a way that minimizes the fragmentation of schedulable intervals. It means that it schedules a task's instances in the currently available first and last feasible slots (i.e.) as far as possible within the period, and thus this task scheduled for the last

feasible slot gives rise to the highest response time of tasks. But for the case of the lowest frequency task, scheduling the higher frequency tasks as far as possible within the period creates the maximum fragment for the lower frequency tasks to execute with the least possible number of preemptions. This results in the lowest frequency task executing well in advance of its deadline, which considerably reduces the finish time, resulting in the least response time of the lowest frequency task at high utilizations. This behavior can be noted at 100% utilization in Figure 6.23.

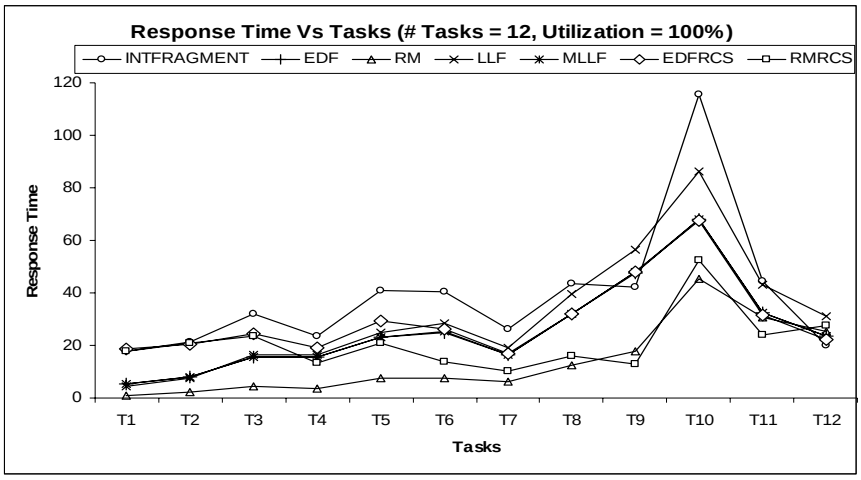

Fig. 6.23: Response Time per Task (# Tasks = 12, Utilization = 100%)

From Figures 6.20, 6.21, and 6.22, the difference in the response time behavior of algorithms with respect to utilization can also be studied. With increase in utilization, in the case of the traditional algorithms and MLLF, the response time of tasks increases. For these algorithms, at low utilizations, all tasks are scheduled at the earliest feasible slots and hence the response time is lower. As the utilization increases, the increased execution time of tasks contributes to a greater finish time limit and hence the increase in response time at high utilizations. But for IntFragment, the response time decreases with increase in utilization. In IntFragment, the heuristic applied causes the response time of the highest frequency task to be a fixed value (determined by its period). It schedules the adjacent two instances of the immediate lower frequency task in the currently available first and last feasible slots, as mentioned in 6.4. Thus, the response time of that instance which is scheduled for the last feasible slot is affected by the execution time of the highest frequency task. Continuing this trend, it is inferred that as the frequency decreases, the response time of tasks is influenced by the execution times of tasks having

a higher frequency. So, therefore, this explains the decreasing response time of tasks (except the highest frequency task) for IntFragment with increase in utilization, as the execution time of a task increases with increased utilization.

The RCS algorithms show a decrease in the response times of higher frequency tasks with increased utilization and an increase in response times, in the case of lower frequency tasks. This is because the increased execution time causes the extension time of lower frequency tasks to be reduced, resulting in the earlier preemption of the lower frequency tasks and eventually in their increased response time. This earlier preemption of lower frequency tasks also causes the higher frequency tasks to execute sooner than would be possible at a lower utilization and hence the reduced response time of higher frequency tasks with increasing utilization.

Studying the response time behavior of tasks against a combination of a particular range of utilizations and frequencies, the trends observed are ploted as below.

- For high and intermediate frequency tasks,

  $RM < EDF \leq MLLF < LLF < EDFRCS \leq RMRCS < IntFragment$

The response times of higher and intermediate frequency tasks exhibited by RCS algorithms can be attributed to the fact that a lower priority task may continue execution in preference to a higher priority task due to the preemption reduction heuristic used by RMRCS and EDFRCS. But these response times are lesser than those exhibited by IntFragment, as IntFragment schedules the execution of tasks in a way that minimizes the fragmentation of schedulable intervals. It schedules the higher and intermediate frequency tasks as far as possible within their period, thus creating the maximum fragment for the lower frequency tasks to execute. This results in the higher and intermediate frequency tasks having a greater response time. This is shown in Figure 6.24. It is also observed that this behavior does not change much with change in utilization from 55% to 70%.

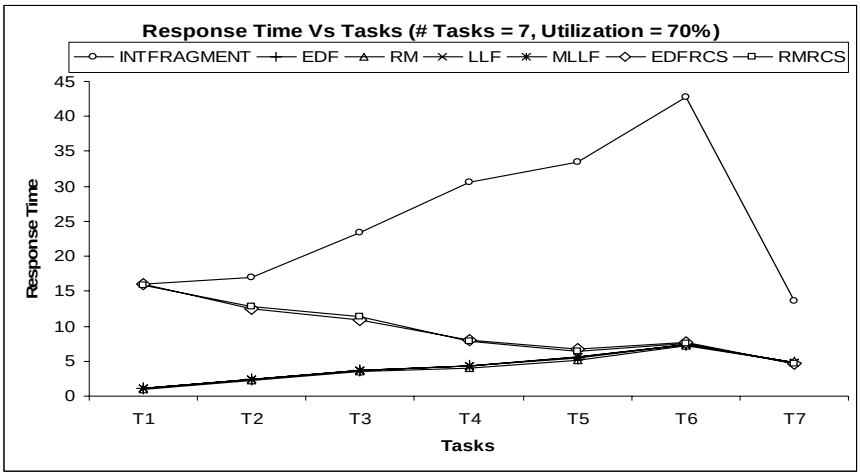

Fig. 6.24: Response Time per Task (# Tasks = 7, Utilization = 70%)

It is also noted that for tasks with intermediate frequency, at high utilizations (>75%) in the case of large task set sizes (>8), LLF sometimes gives greater response time than the RCS algorithms, but less than IntFragment. This trend can be noted in Figure 6.23, presented earlier. This increase in the response times of tasks in the case of LLF may again be attributed to the nature of LLF as a job-level varying priority algorithm, i.e., it is possible that at times, an instance of an intermediate frequency task may have a lesser priority (greater slack time) when compared to a lower frequency task and hence, the increase in response time of intermediate frequency tasks when compared to that for the RCS algorithms.

For lower frequency tasks

- At low utilizations - The trend is the same as for higher and intermediate frequency tasks.
- At high utilizations
  - For some of the lower frequency tasks,

    $RM \leq RMRCS < EDFRCS < EDF \leq MLLF < LLF < IntFragment$

    This trend can be noted in Figures 6.23 and 6.25.

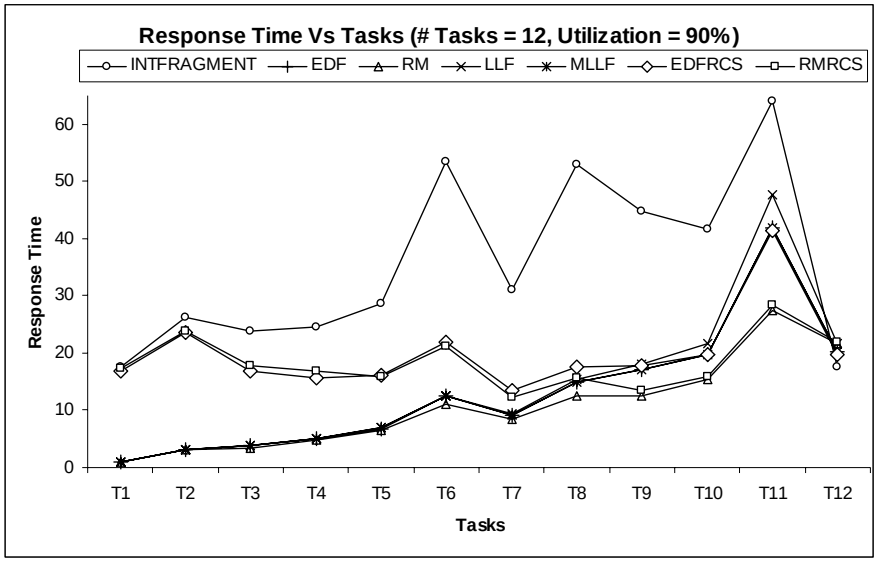

Fig. 6.25: Response Time per Task (# Tasks = 12, Utilization = 90%)

Here, EDFRCS algorithm shows a lesser response time than LLF, MLLF and even EDF owing to its preemption heuristic which favors the delayed preemption of lower frequency tasks, resulting in their lesser response times. At higher utilizations, many task sets fail to be RM-schedulable and in turn, RMRCS-schedulable also and as only schedulable tasks are considered in this analysis, the results turn out to be biased toward RM and RMRCS.

- For the lowest frequency task
  - In case of smaller task sets (excluding 100% utilization)

    $$RMRCS \leq EDFRCS < RM < EDF \leq MLLF < LLF < IntFragment.$$

  - IntFragment gives the least response time for the lowest frequency task at 100% utilization, irrespective of the task set size. For task set sizes greater than 12, this fact holds true even for high utilizations like 90% onwards (Figures 6.23 and 6.25). It is also noted that as the number of tasks increases, this fact is true even for a utilization of 80%.

Figure 6.26 is used to study the average response time of tasks against utilization for all the algorithms. The average response time at a particular utilization described here is evaluated as the mean of the response times of all the tasks in the task set at that fixed utilization and hence, as mentioned in the response time evaluation of tasks, the average response time of tasks also increases slowly or remains constant with increased utilization

for all the algorithms, except IntFragment, where the average response time decreases linearly with utilization. For all the algorithms except IntFragment, at low utilizations (implying lesser execution time), all tasks are scheduled at the earliest feasible slots (priority-based algorithms) and hence the response time is lower. As the utilization increases, the increased execution time of tasks contributes to a greater finish time limit and hence the increase in response time at high utilizations. In IntFragment, the decrease in average response time of tasks with increase in utilization is closely associated with the fact that the response time of tasks decreases with utilization. At low utilizations, the response time is high while the execution time is low resulting in a high normalized response time, while at higher utilizations, the response time reduces and execution time increases, thus contributing to a lower normalized response time. Hence the average response time of tasks also reduces with increased utilization. The RCS algorithms and IntFragment show consistently higher average response times than the traditional algorithms due to their preemption reduction heuristics. The average response time behavior of tasks is not showing any significant change with increase in number of tasks.

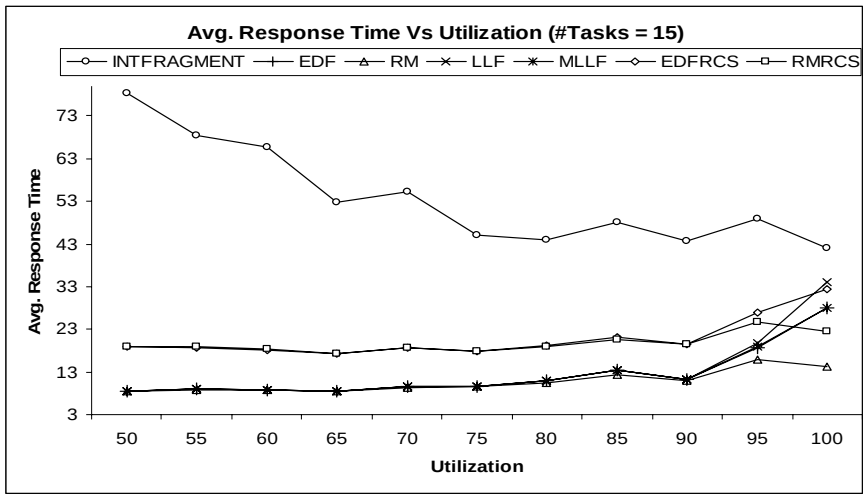

Fig. 6.26: Average Response Time per Utilization (# Tasks = 15)

Comparing the average response time of tasks against the number of tasks for a fixed utilization (Figure 6.27 and 6.28), it is seen that an increase in the number of tasks results in a linear increase (for #tasks ≤15) in the average response time for IntFragment, while all of the other algorithms grow slowly with the number of tasks. The traditional algorithms (RM, EDF and LLF), MLLF and RCS algorithms schedule tasks in the

earliest feasible slots (i.e., they are priority-based algorithms), thus causing the average response time to be a function of the sum of the execution times (lesser than their periods) of the tasks involved. This also explains the fact that the average response time of tasks in the case of these algorithms is very low at low utilizations and considerably high at higher utilizations. For IntFragment, in an aim to minimize the number of preemptions, tasks are scheduled as distant as possible within their periods, thus rendering the average response time of tasks for IntFragment to be a function of the periods of tasks (usually greater than the execution times) resulting in the average response time of IntFragment to be significantly greater than that for the other algorithms. Moreover, with increase in the number of tasks while keeping the utilization fixed, the execution time of tasks is reduced. Hence, this explains that with increase in the number of tasks, there is an exponential increase in the average response time for IntFragment, while all of the other algorithms grow slowly with the number of tasks.

The following trend is thus generally noted in the performance of the algorithms with respect to average response time of tasks, as seen in all the Figures above.

$$IntFragment > EDFRCS \geq RMRCS > LLF > MLLF \geq EDF > RM$$

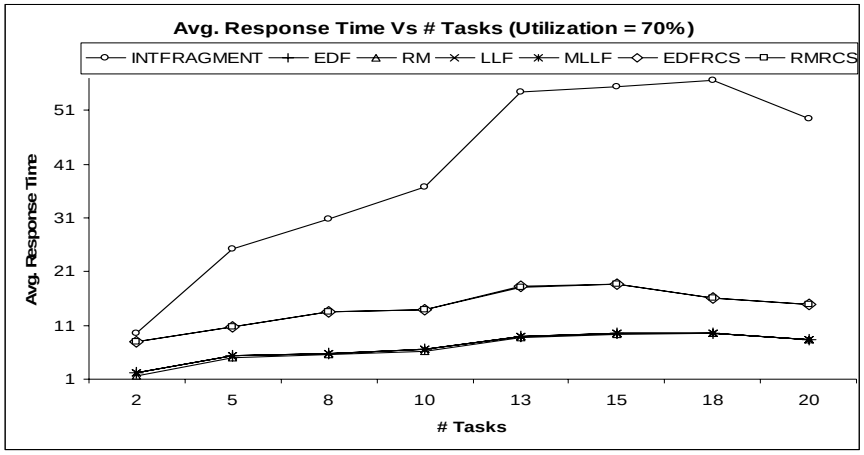

Fig. 6.27: Average Response Time per Number of Tasks (Utilization = 70%)

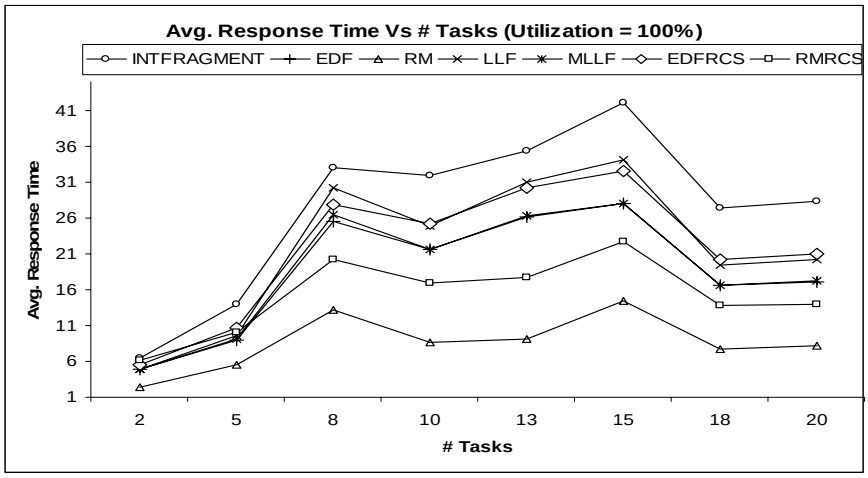

Fig. 6.28: Average Response Time per Number of Tasks (Utilization = 100%)

6.7.2 RESPONSE TIME JITTER

The jitter values are normalized over periods of the corresponding tasks and the tasks are ordered by decreasing frequency. The Absolute Response Time Jitter (ARJ) and Relative Response Time Jitter (RRJ) for different tasks for fixed utilization values are measured and the Figures 6.29 and 6.30 show the ARJ values per task for a fixed number of tasks in a task set and at a fixed utilization.

It can be observed that the RM algorithm gives the least jitter for almost all tasks. RM's jitter for the highest frequency task is always 0, irrespective of the number of tasks and the utilization as RM's priority function is based on the period of the task. Among the traditional algorithms (RM, EDF and LLF) and MLLF, the EDF, LLF, and MLLF algorithms exhibit large jitter for the higher frequency tasks than RM at high utilizations. Still, it is observed that the jitter of these algorithms (RM, EDF, LLF and MLLF) for higher frequency tasks is negligible, when compared with that of the RCS algorithms and IntFragment.

It is noted that the absolute jitter remains low (as a proportion of the period) for the lower frequency tasks for all the scheduling algorithms. On the other hand, for the higher frequency jobs, the RCS algorithms and IntFragment exhibit very high jitter (close to one full period in some cases). This above mentioned behavior of RCS algorithms can be explained by the fact that an instance of a higher frequency task may be scheduled early due to its high priority (frequency or deadline) or may be scheduled late due to continued

execution of lower frequency job(s) to avoid preemption, thereby resulting in the variation (i.e. jitter) in response times. The reason attributed to the above mentioned behavior of IntFragment can be that for IntFragment, in an aim to minimize the number of preemptions, the higher frequency tasks are scheduled as distant as possible within their periods, thus causing the response time of higher frequency tasks to vary from just their execution time up to their one full period. This variation accounts for the high response time jitter values of higher frequency tasks for IntFragment. From these observations, it is obvious that the jitter exhibited by the RCS algorithms and IntFragment reduces greatly with decrease in frequency.

The following trend is generally noted for these algorithms with respect to response time jitter of tasks:

$$RM < EDF \leq MLLF \leq LLF < EDFRCS \leq RMRCS < IntFragment$$

This trend is observed in Figure 6.29. However, at a higher utilization (Figure 6.30), some of the algorithms show marked deviation from the above behavior for lower and intermediate frequency tasks (i.e.) the trend changes to

$$RM < EDF \leq MLLF < RMRCS \leq EDFRCS < LLF < IntFragment$$

Figure 6.30 shows the normalized ARJ values per task at 100% utilization. It is observed that in this case of a fully loaded system, all the other algorithms exhibit higher jitter compared to RM and RMRCS for lower frequency tasks and LLF shows a pronounced increase in jitter for these tasks compared to all the other algorithms. It must be observed that these results appear to be different from those in [Buttazzo 2005] because we consider only RM-schedulable task sets for RM and RMRCS. If all the task sets are included in this analysis, then at high utilizations, RM and RMRCS would exhibit higher response times for low frequency tasks as compared to EDF and EDFRCS respectively. This behavior is in agreement with the results reported in [Buttazzo 2005], but such performance analysis may not be relevant for tasks that are missing the deadline anyway.

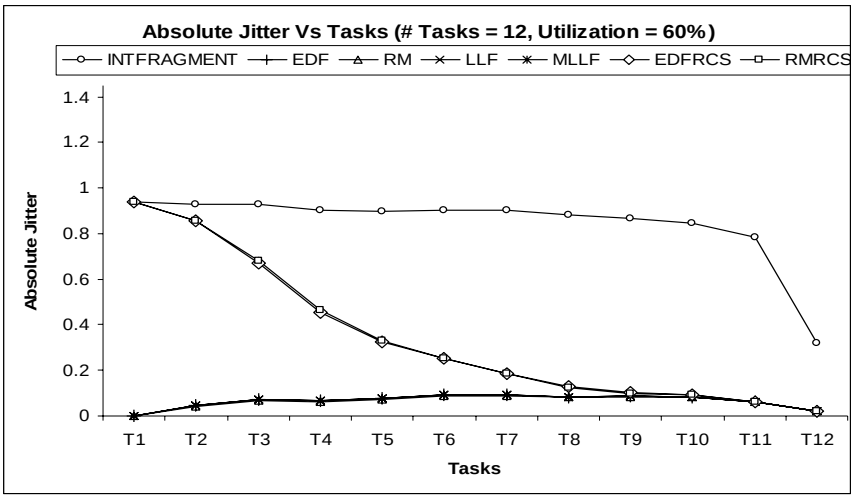

Fig. 6.29: Absolute Jitter per Task (# Tasks = 12, Utilization = 60%)

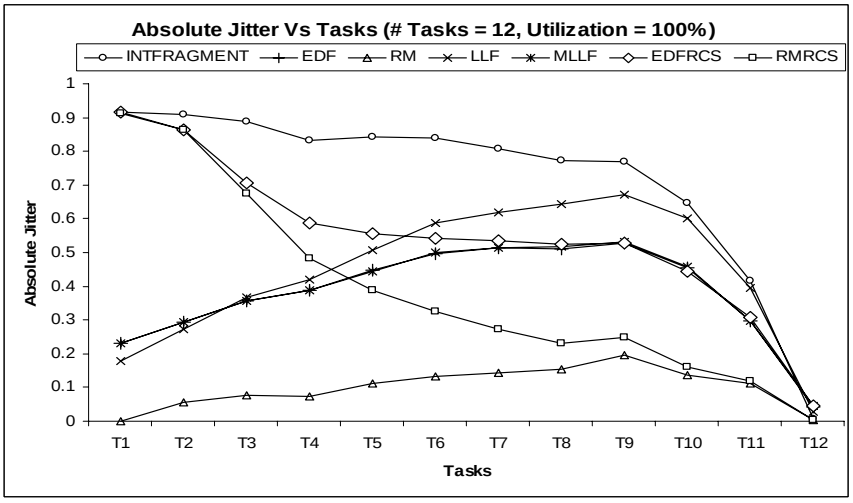

Fig. 6.30: Absolute Jitter per Task (# Tasks = 12, Utilization = 100%)

In the plots shown in Figures 6.29 and 6.30, the lowest frequency task is an anomalous case, because often there is exactly one instance of such a task within a hyperperiod and hence jitter (i.e. variation in response times) is non-existent.

Also, our experiments showed no difference in comparative behavior among these algorithms with respect to absolute jitter versus relative jitter as shown in Figures 6.31 and 6.32.

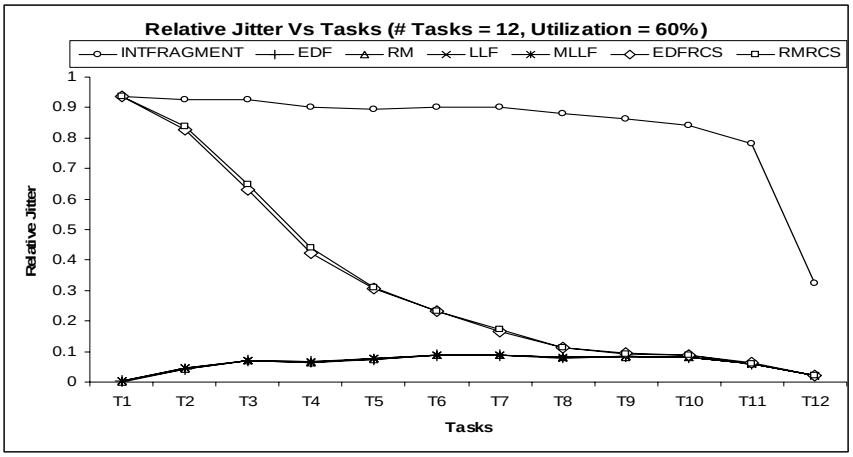

Fig. 6.31: Relative Jitter per Task (# Tasks = 12, Utilization = 60%)

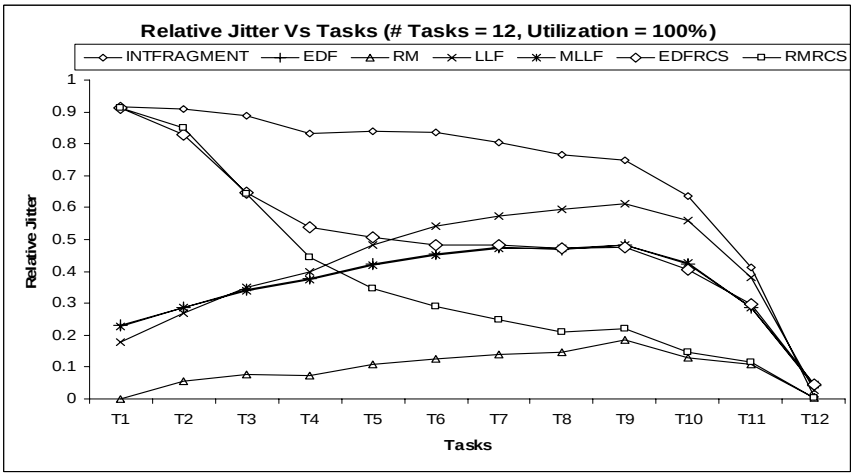

Fig. 6.32: Relative Jitter per Task (# Tasks = 12, Utilization = 100%)

Attempting to note the variation (jitter) shown in response times of tasks by various algorithms against utilization, it is found that the absolute response time jitter per task increases or remains constant with utilization for all the scheduling algorithms except IntFragment, where the absolute response time jitter of tasks decreases linearly with increased load. All the algorithms except IntFragment are priority-based algorithms which spare no idle time for the processor as long as a job is ready for execution and hence the response time of jobs of the same task do not tend to vary too much except at very high utilizations, where the response time jitter increases exponentially. Among the traditional algorithms, LLF shows a greater increase as it a job-level varying priority algorithm which causes the response time of jobs of a task to fluctuate and hence the increased jitter. The above noted trend of decrease in response time jitter with increased

utilization can be explained as: IntFragment schedules tasks such that the response time of the highest frequency task is a fixed value (determined by its period) at a particular utilization. It schedules the instances of the immediate lower frequency task in the currently available earliest and last feasible slots (which none of the other algorithms do). Thus, the response time of that instance which is scheduled for the last feasible slot is affected by the execution time of the highest frequency task and its response time reduces with increase in the execution time of the highest frequency task. On the other hand, the response time of that instance which is scheduled for the earliest feasible slot increases with increase in the execution time of the highest frequency task resulting in the absolute jitter of this second highest frequency task reducing with increase in the execution time (caused by increase in load) of the highest frequency task. This fact remains true as the frequency decreases and hence the decrease.

With focus on the average variation (jitter) in the response time behavior of tasks for the different scheduling algorithms, the following plots (Figures 6.33, 6.34 and 6.35) show the variation of average response time jitter of tasks for the various scheduling algorithms against the number of tasks in a task set for a fixed utilization.

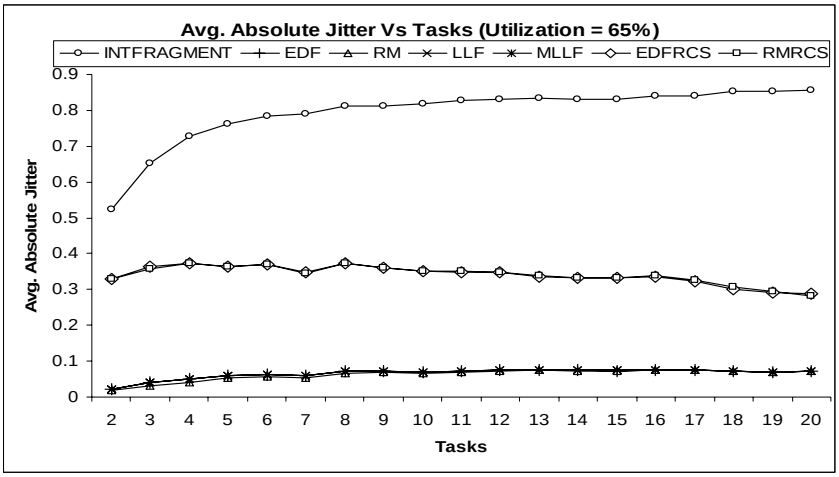

Fig. 6.33: Average Absolute Jitter per Number of Tasks (Utilization = 65%)

It is seen that the average response time jitter for IntFragment increases linearly with increase in the number of tasks, while for the other algorithms, the average jitter remains almost constant. Among the traditional algorithms, the trend observed is RM < EDF ≤ MLLF ≤ LLF. The RCS algorithms exhibit considerably greater average response time jitter than these traditional algorithms, with EDFRCS giving a slightly greater average

response time jitter than RMRCS at high utilizations. With increase in the number of tasks for a fixed utilization, IntFragment gives a greater average response time jitter than the RCS algorithms. For IntFragment, the average response time jitter of tasks increases exponentially with increasing number of tasks up to a task set size of 6. From Figures 6.33, 6.34 and 6.35, it can also be seen that with increase in utilization, the average response time jitter for all the algorithms increases except IntFragment, where the effect is reverse, that is, the average response time jitter decreases with increase in utilization. The slightly anomalous trend observable between 90% and 100% utilization is due to the fact that the trend lines for RM and RMRCS include only task sets that are RM-schedulable.

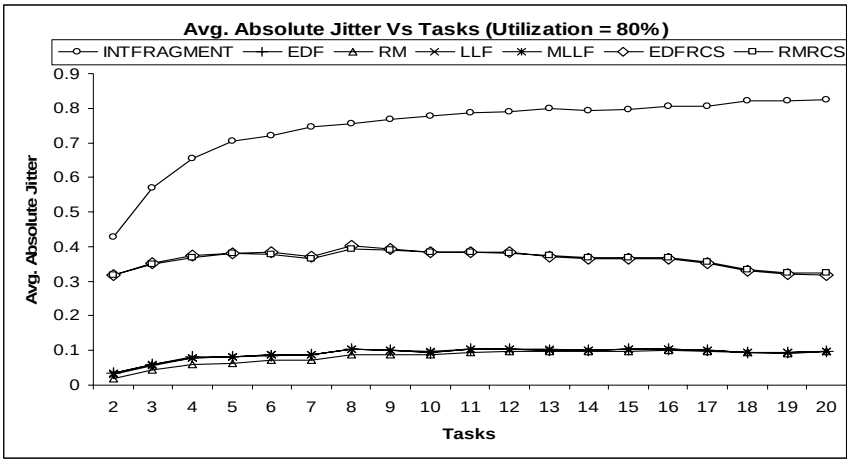

Fig. 6.34: Average Absolute Jitter per Number of Tasks (Utilization = 80%)

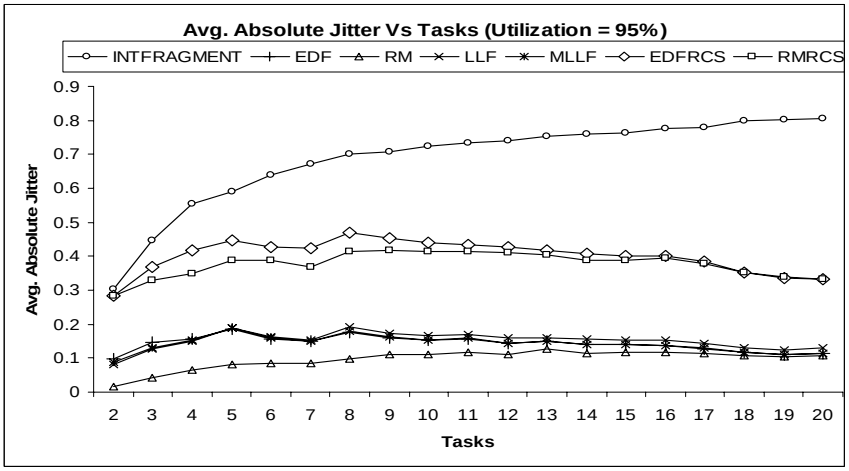

Fig. 6.35: Average Absolute Jitter per Number of Tasks (Utilization = 95%)

The above mentioned facts about average absolute jitter of tasks holds true for average relative jitter of tasks also, for each scheduling algorithm under consideration.

6.7.3 LATENCY

The following plots (Figures 6.36 and 6.37) show the latency of tasks in task set schedules generated by the various algorithms for a fixed task set size and at a fixed utilization. The latency values are normalized over execution times of the corresponding tasks and the tasks are ordered by decreasing frequency.

It is noted that at low utilizations (≤70), with decrease in frequency, all algorithms except the RCS algorithms increase the latency of tasks linearly or remain constant for higher frequency tasks and reduces latency slightly for lower frequency tasks, irrespective of the number of tasks. This slight reduction for lower frequency tasks observed predominantly in the case of IntFragment, may be due to the fact that IntFragment aims at providing the maximum for the lower frequency tasks to execute in order to reduce preemptions. This trend at low utilizations can be observed in Figure 6.36. The RCS algorithms show a linear increase throughout with decreasing frequency due to the preemption reduction logic. It is also observed that at high utilizations, there is an exponential increase in the latency of tasks with decreasing frequency of tasks. This fact holds true for each algorithm under consideration, irrespective of the task set size and utilization. Irrespective of the scheduling algorithm, more often, at high utilizations, the execution of higher frequency tasks is carried out at a cost of preemptions of the lower frequency tasks thus resulting in an increase in the finish time of the lower frequency tasks and hence the increase in latency with decrease in frequency. This exponential increase in latency with decrease in frequency can be noted in Figure 6.37. In addition, it is noted that with increase in utilization (Figure 6.37), this exponential growth in the latency of lower priority tasks becomes more pronounced.

Here, it is also noticed that almost all the algorithms are providing a normalized latency value that is close to 1 (the least possible latency value) for the highest frequency task, irrespective of the number of tasks and utilization. The RCS algorithms (EDFRCS and RMRCS) consistently exhibit the least latencies at all utilizations for all tasks (except for the lowest frequency task at high utilizations) because of its preemption reduction logic.

For the lowest frequency task, at high utilizations (excluding 100% utilization), IntFragment provides the least latency due to its heuristic of minimizing the fragmentation of schedulable intervals. At 100% utilization, the 0% idle time of the processor results in multiple internal fragments which cause preemption of the lower frequency tasks more frequently and hence, IntFragment comes second to EDFRCS in providing the least latency for the lowest frequency task. This characteristic is noted in Figure 6.37. The traditional algorithms (RM, EDF and LLF) and MLLF do not fair too well at this metric owing to the increased number of preemptions in their schedules. LLF gives the worst latencies in almost all the frequencies irrespective of the task set size and utilization owing to its laxity metric and hence, increased number of preemptions as can be observed in Figure 6.37.

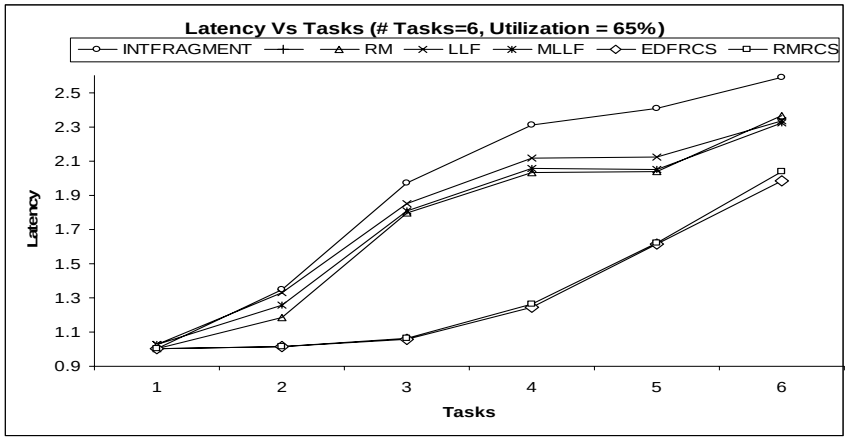

Fig. 6.36: Latency per Task (# Tasks = 6, Utilization = 65%)

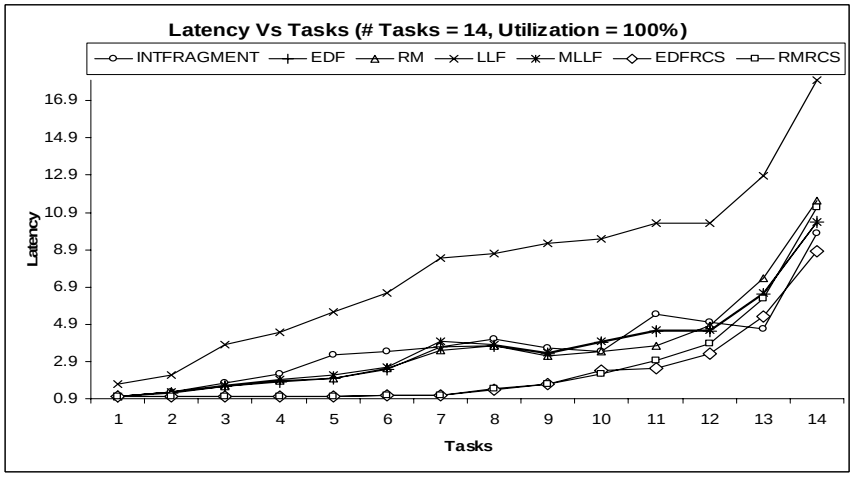

Fig. 6.37: Latency per Task (# Tasks = 14, Utilization = 100%)

Comparing the latency of tasks against utilization, it is observed that the latency of tasks increases with utilization levels for all algorithms, irrespective of the task set size. With increase in utilization, the execution time increases and hence, the possibility of increased number of preemptions arises, thus contributing to increased latency.

Figure 6.38 shows the average latencies of tasks against utilization levels for the various algorithms. As noted above, the average latency also increases with utilization (≤ 85) for all algorithms. The RCS algorithms consistently provide the least average latencies. The preemption reduction algorithms perform slightly better than their counterparts in all cases: EDFRCS vs. EDF, RMRCS vs. RM, and MLLF vs. LLF. Among the traditional algorithms, LLF has the worst latencies particularly at high utilizations. Both of these observations can be explained by the fact that reduced preemptions increase the chances of continued execution of any job, thereby reducing its latency. This difference between the traditional and the preemption-reducing algorithms is more pronounced at higher utilizations because the traditional algorithms (RM, EDF and LLF) induce more preemptions at higher utilizations. The trend followed by the algorithms with respect to average latencies of tasks for low utilizations (< 80), irrespective of the task set size is:

$$IntFragment > LLF > MLLF > RM > EDF > RMRCS \geq EDFRCS$$

At higher utilizations (≥80), LLF induces more preemptions and hence gives the worst average latencies at high utilizations.

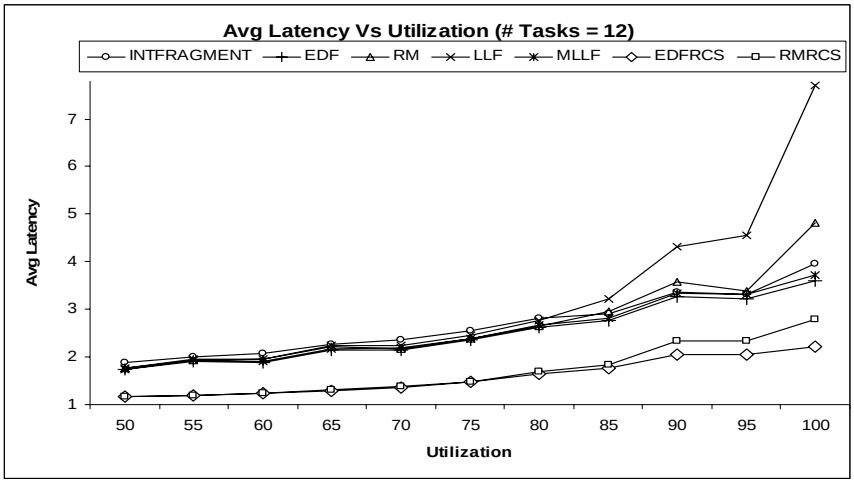

Fig. 6.38: Average Latency per Utilization (# Tasks = 12)

The plots in Figures 6.39 and 6.40 show the average latency of tasks against the number of tasks at a fixed load. It is observed that an increase in the number of tasks results in an

increase in the average latency for all scheduling algorithms at all loads due to an increase in the number of preemptions. The increase for LLF is more pronounced at high utilizations as it is a job-level varying priority algorithm which causes an increased number of preemptions.

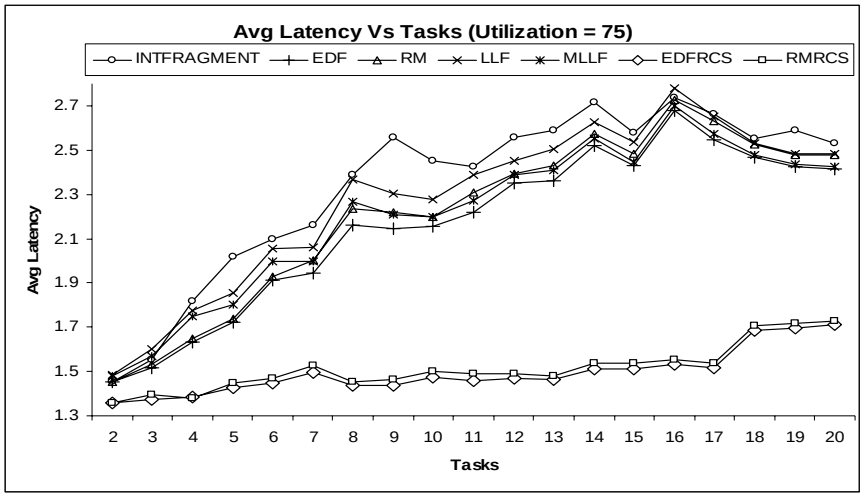

Fig. 6.39: Average Latency per Number of Tasks (Utilization = 75%)

IntFragment is designed with the aim of minimizing the fragmentation of schedulable intervals, which causes the average latency of tasks to be very low (next to RCS algorithms) for smaller task sets (Figure 6.40). With increase in the task set size, the internal fragmentation caused induces increased number of preemptions, thus increasing the finish time and hence, the increase in average latency.

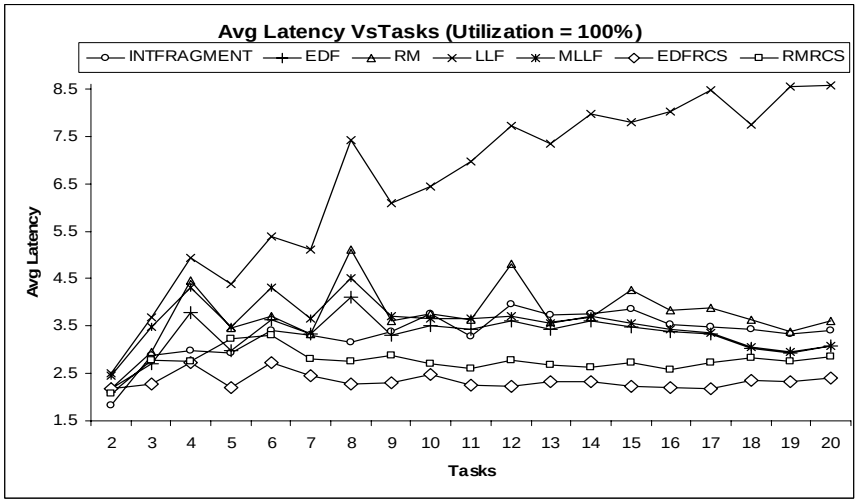

Fig. 6.40: Average Latency per Number of Tasks (Utilization = 100%)

6.7.4 SCHEDULING COMPLEXITY

The worst case time complexity of Algorithm IntFragment is

$$p_{max} * \Sigma_{t \in Tasks} (H / p(t))$$

where, p_{max} is the maximum among the periods of all tasks, H is the hyper-period, and p(t) is the period of task t.

As mentioned in Section 6.4.4, the time complexity of an online scheduling algorithm A can be expressed as:

$$T_A(H,S) = decisions(A,H, S) * T_{dec}(A,H, S)$$

For RM and EDF, the number of decision points is 2*N, where N is the number of jobs for the task set S, because RM and EDF consider every job arrival and departure. For LLF and MLLF, the number of decision points is H, the hyper-period, because these are totally on-line algorithms, by definition [Mok 1983]. The RCS algorithms also behave like RM and EDF, except when they defer a preemption. Each deferred switch adds another decision point. It is not easy to theoretically estimate the number of deferred switches, as this depends on various parameters of the task set, but the upper bound on the number of deferred switches is a small fraction of N, the number of jobs, and it grows with |S|, the number of tasks in the input task set. Thus the number of decision points for the RCS algorithms is (2+d)*N in the worst case, where d is a slowly growing function of |S|. These complexity measures are summarized in Table 6.10.

The time taken per scheduling decision is dependent on the way in which priority-based selection is implemented. For RM, it is possible to implement this selection in O(1) time by assigning a fixed priority to each task (as the inverse of its period). For EDF, a fixed priority implementation is not possible and the time for selection is dependent on whether or not the data structure used for the ready queue is ordered. If an ordered queue is maintained (i.e., a priority queue), then the time taken per EDF scheduling decision would be O(log m). Else, it would be O(m), where m is the queue length [Gooch 1998]. Although the value of m is not easily predictable, it has an upper bound of |S|.

The complexity of a single scheduling decision in LLF or MLLF varies on whether or not the priority (i.e. the slack time) of the current job changes – w. r. t. the highest priority job in the queue – in between two decisions. If there is a relative priority change, then

this new priority value has to be compared with the updated priorities of all the other processes in the queue. In this case, the time complexity of a single scheduling decision is O(m), where m is the queue length. If the priority does not change, the time taken for the scheduling decision is just the comparison between the current job and the highest priority job in the queue. In the latter case, the complexity of the scheduling decision is O(1).

For the RCS algorithms, the time taken for a single scheduling decision is dominated by the extension function [Raveendran 2006] with complexity O(p*p), where p is the number of jobs of priority higher than the current job. The upper bound for p is |S|. These complexity measures are summarized in Table 6.10. The last column of Table 6.10 lists the scheduling complexity of each algorithm.

Table 6.10: (Worst Case) Time Complexity of Scheduling Algorithms

Algorithms	**Decision Points**	**Per Decision Complexity**	**Scheduling Complexity**
RM	2*N	O(1)	O(2*N)
EDF	2*N	O(log m)	O(2*N*log|S|)
LLF	H	O(m)	O(H*log|S|)
MLLF	H	O(m)	O(H*log|S|)
RMRCS	(2+d)*N where d grows very slowly w.r.t |S|	O(p*p)	O((2+d)*N*|S|*|S|)
EDFRCS	(2+d)*N where d grows very slowly w.r.t |S|	O(p*p)	O((2+d)*N*|S|*|S|)

where H is Hyper-period, N is number of jobs, |S| is number of tasks, m is queue length, p is number of higher priority jobs and d is number of deferred switches.

The average case complexities are harder to estimate theoretically. For instance, the average value of m (i.e. queue length) is not easily predicted. The queue length over a large number of test cases has been experimentally measured. The average queue length for all the algorithms is approximately 1.25*log|S|. Thus, the average case time complexity of EDF is O(2*N*log(1.25*log|S|)).

For LLF and MLLF, although the number of decision points is H, the number of points at which the priorities have to be updated is significantly less. As already discussed, priorities are to be updated only when the priority of the current job changes with respect

to the highest priority job in the queue. These values have been estimated experimentally to be approximately 2.37*N and 2.35*N for LLF and MLLF respectively. Again, as already discussed, updating priorities takes time proportional to m, the length of the queue, and the average value for m as mentioned above is 1.25*log|S|. Thus, the average case time complexities for LLF and MLLF would be O(H+2.96*N*log|S|) and O(H+2.94*N*log|S|) respectively.

For the RCS algorithms, the experimentally estimated average number of decision points are 2.06*N. The experimentally estimated average value of p, the number of higher priority jobs, is approximately 0.6*log|S|. Thus, the average case time complexity of both RMRCS and EDFRCS is O(0.74*N*log|S|*log|S|). These average case complexity measures are summarized in Table 6.11.

Table 6.11: Estimated(Average Case) Time Complexity of Online Scheduling Algorithms

Algorithms	**Decision Points**	**Per Decision Complexity**	**Scheduling Complexity**
RM	2*N	O(1)	O(2*N)
EDF	2*N	O(log(1.25*log|S|))	O(2*N*log(1.25*log|S|))
LLF	H decisions and 2.37*N updates	O(1.25*log|S|)	O(H+2.96*N*log|S|)
MLLF	H decisions and 2.35*N updates	O(1.25*log|S|)	O(H+2.94*N*log|S|)
RMRCS	2.06*N	O(0.36*log|S|*log|S|)	O(0.74*N*log|S|*log|S|)
ECFRCS	2.06*N	O(0.36*log|S|*log|S|)	O(0.74*N*log|S|*log|S|)

6.7.5 PREEMPTION COUNT

The preemption count values obtained are normalized over the number of jobs and the tasks are ordered by decreasing frequency. The trend observed among the algorithms with respect to the number of preemptions introduced in the resulting schedules is as follows:

$$EDFRCS \leq RMRCS < IntFragment < EDF < RM < MLLF < LLF$$

The plots in Figures 6.41 and 6.42 represent preemption count against utilization for task sets with a fixed number of tasks. It is observed that the number of preemptions increases with increase in utilization, irrespective of the number of tasks due to increase in execution time and schedulability constraints.

It is seen that the RCS algorithms show significant reduction in preemptions compared to their traditional counterparts, irrespective of the utilizations. For instance, at 50% utilization, EDF results in an average of 0.27 preemptions per job, whereas EDFRCS results in an average of 0.0725 preemptions per job, which is about a 73% reduction. For a fully loaded system (100% utilization), EDF results in an average of 0.662 preemptions per job, whereas EDFRCS results in 0.22 preemptions per job, which is about a 66% reduction. The reductions are similar for RMRCS versus RM. IntFragment also establishes its objective of minimizing the number of preemptions, though second to the RCS algorithms. In the same test case listed above, (i.e.) at 50% utilization, IntFragment results in an average of 0.126 preemptions per job, which is about a 53% reduction over EDF. Also, for a fully loaded system, IntFragment results in an average of 0.353 preemptions per job, which is about a 47% reduction over EDF. In the case of IntFragment, reduced number of preemptions occurs when the periods of tasks are in multiples. If not so, then the multiple internal fragments caused results in an increased number of preemptions. No such behavior would be noted in the case of RCS algorithms even if the periods of the lower frequency tasks are not multiples of those of the higher frequency tasks. MLLF on the other hand, reduces preemptions significantly with respect to LLF at high utilizations. But it is important to note that MLLF still results in more preemptions than EDF or RM.

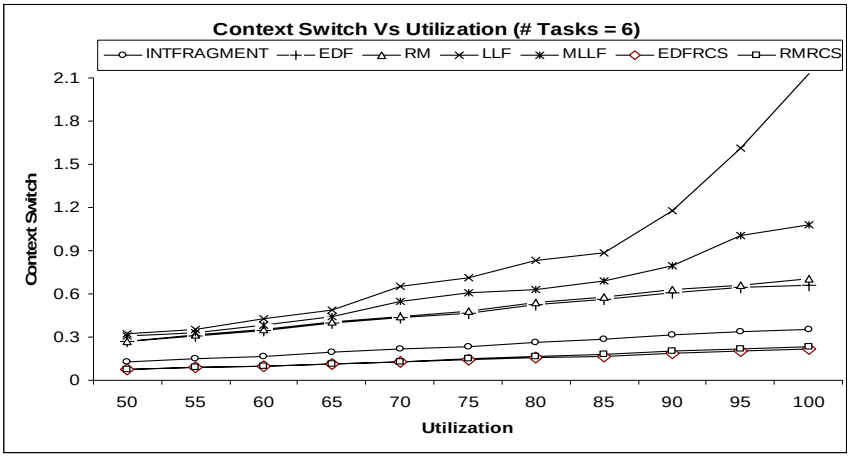

Fig. 6.41: Preemptions per Utilization (# Tasks = 6)

In [Buttazzo 2005], it is shown that EDF results in lesser preemptions than RM. The obtained results confirm this: EDF results in about 1% to 6% lesser preemptions than RM

on an average. Similarly, EDFRCS also results in lesser preemptions than RMRCS, but the difference is often small with a maximum of 2% in the chosen test cases. It is also noted that the increase in the number of preemptions in EDFRCS and RMRCS with the increase in utilization is very less due to the regress preemption reduction heuristic. In case of IntFragment, it is seen that with the increase in utilization, the number of preemptions increases more steeply than EDFRCS and RMRCS, but less compared to the other traditional algorithms. Among the traditional algorithms, LLF's preemption count increases very steeply with increase in utilization and performs very poorly at high utilizations (≥ 80%) due to frequent priority changes of jobs owing to the laxity metric.

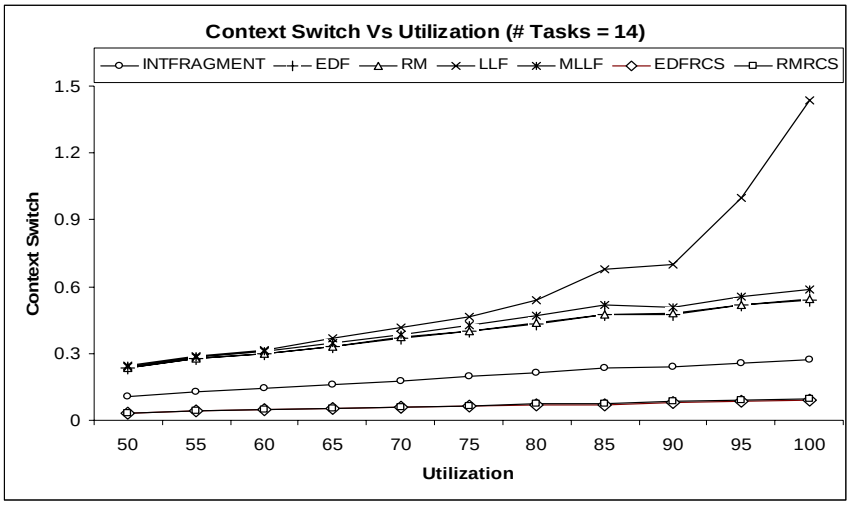

Fig. 6.42: Preemptions per Utilization (# Tasks = 14)

The plots below (Figures 6.43 to 6.48) represent preemption count against the number of tasks in a task set with a fixed utilization. It is observed that for each scheduling algorithm under consideration, the number of preemptions decreases with increase in the number of tasks at a fixed utilization, irrespective of the utilization. This behavior is attributed to the fact that an increase in the number of tasks at a fixed utilization causes the execution time per task to reduce, resulting in reduced preemptions between completions of jobs. As the number of tasks in the task set increases, preemption reduction is more pronounced for the RCS algorithms and IntFragment: EDFRCS and IntFragment reduce the preemption count by at least 82% and 48% respectively with respect to EDF at all utilization levels, when the task set size is 20. MLLF shows significant reductions over LLF in preemption count when the number of tasks is smaller,

but this is partly explained by the fact that the preemption count for LLF itself is higher when the task set is smaller.

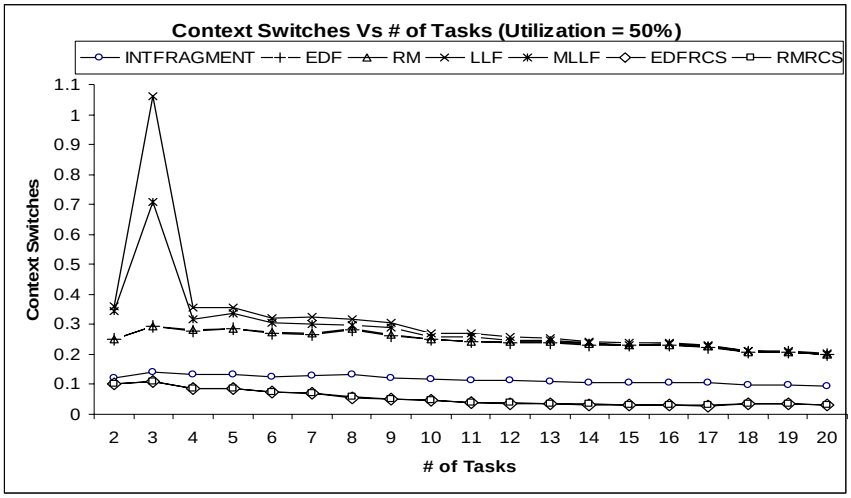

Fig. 6.43: Preemptions per Number of Tasks (Utilization = 50%)

It is also observed that for smaller sets, the number of preemptions introduced by MLLF is very large in comparison to RM and EDF, but as the number of tasks increases, MLLF produces preemptions close to EDF and RM. Evidently, LLF and MLLF are not suitable for generating power-aware schedules of smaller task sets. The preemption reduction brought about by MLLF over LLF (almost 50%) is presented more clearly in Figures 6.46 and 6.48.

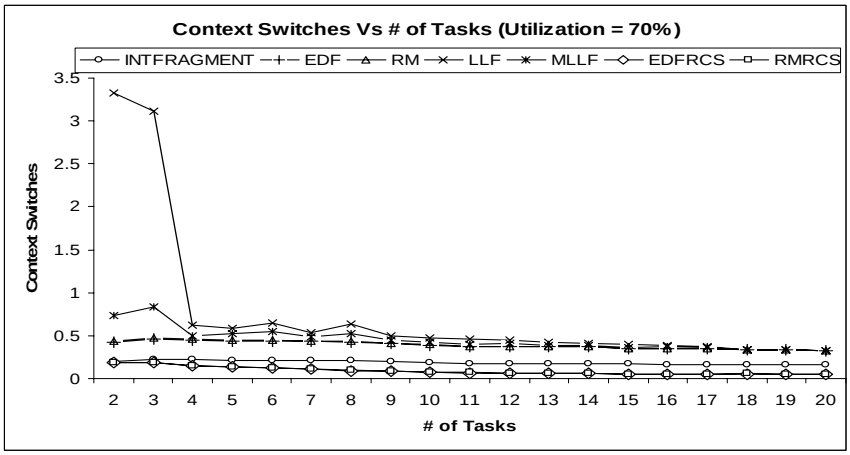

Fig. 6.44: Preemptions per Number of Tasks (Utilization = 70%)

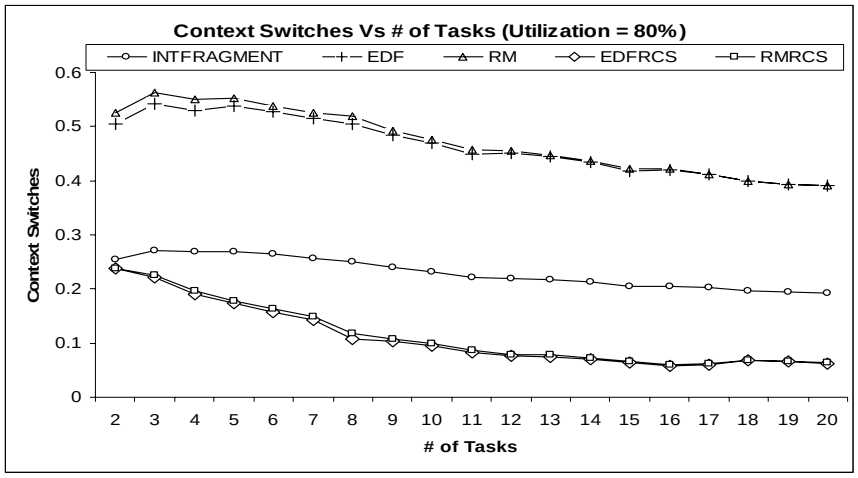

Fig. 6.45: Preemptions per Number of Tasks (Utilization = 80%)

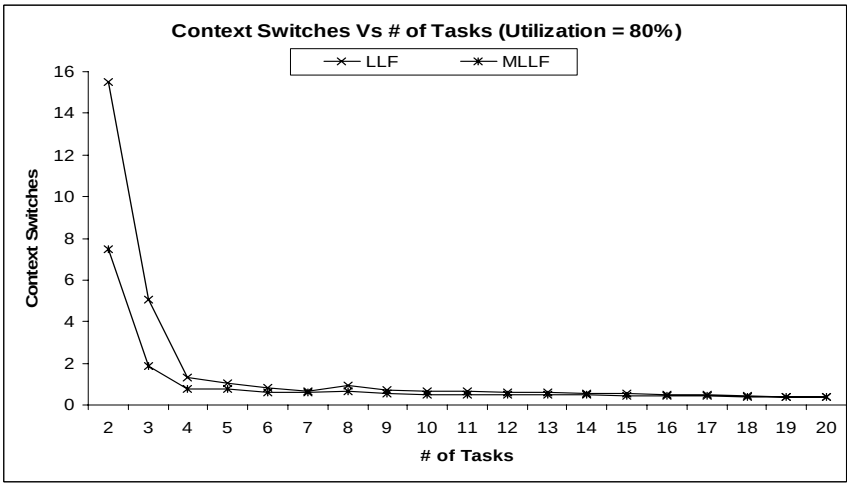

Fig. 6.46: Preemptions per Number of Tasks (Utilization = 80%)

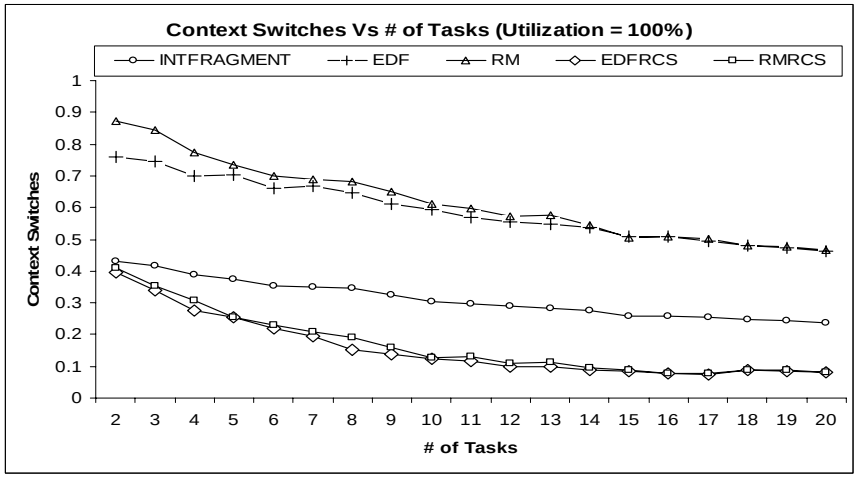

Fig. 6.47: Preemptions per Number of Tasks (Utilization = 100%)

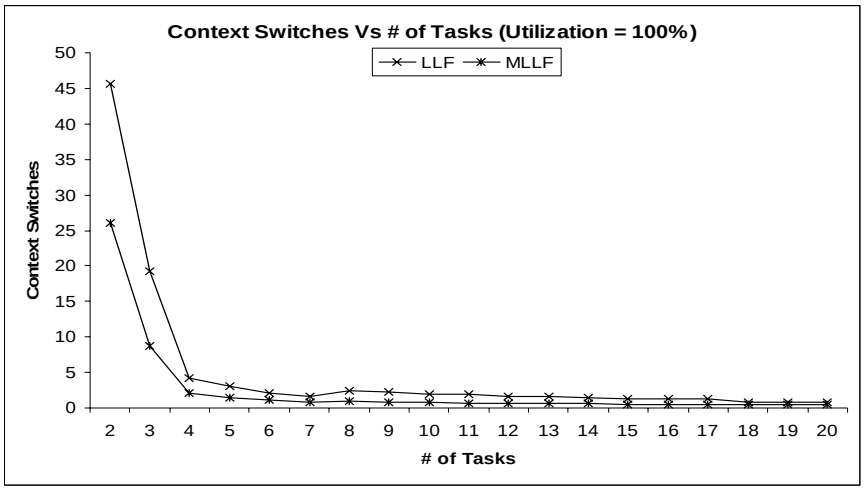

Fig. 6.48: Preemptions per Number of Tasks (Utilization = 100%)

It is observed that the average number of preemptions per job grows monotonically with increasing utilization for a fixed number of tasks for each algorithm under consideration. Except in the case of LLF, this growth is often linear. Furthermore, the average preemption count decreases with increasing number of tasks in the task set. More precisely, it is estimated here, that the normalized average preemption count is a linear function of 1/|S|, where S is the task set. Based on these observations, the estimated average pre-emption counts for all the online algorithms except LLF are summarized in Table 6.12.

Table 6.12 Estimated Average Preemption Count

Algorithm	Preemption Count
RM	O((0.42+2.7/|S|)*N*U
EDF	O((0.45+2.1/|S|)*N*U
MLLF	O((10.9/|S|)*N*U
RMRCS	O((1.9/|S|)*N*U
EDFRCS	O((1.9/|S|)*N*U

N – number of jobs |S| - number of tasks U – utilization (between 0 and 1)

6.7.6 ENERGY CONSUMPTION

As observed in Section 6.6.6, energy consumption overhead due to scheduling depends on two factors: scheduling decisions and preemptions. Among online scheduling algorithms, based on the time spent on scheduling decisions, RM appears to be the least

energy-consuming algorithm closely followed by EDF. Based on the number of preemptions, the RCS algorithms turn out to be the least energy consuming algorithms followed by IntFragment. Although the number of preemptions in a schedule depends on various characteristics, the RCS algorithms invariably offer a significant reduction in the number of preemptions. For the purpose of a head-to-head comparison of the energy consumption impact among the online scheduling algorithms, the time spent on scheduling decisions and the time delay introduced due to preemptions, both amortized over N (the number of jobs) are compared. Since the scheduling complexity may depend on |S|, the size of the task set, whereas the preemption count may depend on |S| as well as U, the utilization as a fraction between 0 and 1, the comparison for specific values of |S| and U is performed. For instance, the average time complexity amortized per job for EDF is 2*log(1.25*log|S|) and the average number of preemptions produced by EDF amortized per job is (0.45+2.1/|S|)*U. Assuming that E_s is the energy consumed per unit time of scheduling and E_c is the average energy consumed per preemption, the above two formulae are evaluated for specific values of |S| and U, say for 10 and 0.5 respectively as follows:

Energy Spent on Per Job Scheduling Decision for EDF

$= 2*\log(1.25*\log(10))*E_s = 4.33E_s$ and

Energy Spent on Per Job Preemption for EDF

$= (0.45+2.1/10)*0.5*E_c = 0.33E_c$

So, in the case of EDF, the total energy impact of scheduling can be formulated as $4.33E_s+0.33E_c$ per job for task sets of size 10 and for a utilization level of 50%. Table 6.13 below summarizes these values for different combinations of |S| and U values.

Table 6.13 can be used a ready reckoner for energy consumption tradeoffs: for instance, for a E_c/E_s ratio of 50, one can observe that preemption energy is likely to dominate and hence the RCS algorithms are likely to offer energy savings of the order of 50% or more compared to RM or EDF; whereas if the E_c/E_s ratio is around 10, then one can observe that the RCS algorithms may still offer energy savings for small task set but for large task sets they spend more energy on scheduling decisions than they save on preemptions.

Table 6.13: Estimated Energy Consumption due to Scheduling Decisions and Preemptions (normalized per job)

|S|	**U**	**RM**	**EDF**	**MLLF**	***RCS**
5	0.5	2Es+0.48Ec	3.04Es+0.44Ec	H`+6.83Es+1.09Ec	3.99Es+0.19Ec
5	0.7	2Es+0.67Ec	3.04Es+0.61Ec	H`+6.83Es+1.53Ec	3.99Es+0.27Ec
5	0.9	2Es+0.86Ec	3.04Es+0.78Ec	H`+6.83Es+1.96Ec	3.99Es+0.34Ec
10	0.5	2Es+0.35Ec	4.33Es+0.33Ec	H`+9.77Es+0.55Ec	8.13Es+0.10Ec
10	0.7	2Es+0.48Ec	4.33Es+0.46Ec	H`+9.77Es+0.76Ec	8.13Es+0.13Ec
10	0.9	2Es+0.62Ec	4.33Es+0.59Ec	H`+9.77Es+0.98Ec	8.13Es+0.17Ec
20	0.5	2Es+0.28Ec	5.28Es+0.28Ec	H`+12.71Es+0.27Ec	13.82Es+0.05Ec
20	0.7	2Es+0.39Ec	5.28Es+0.39Ec	H`+12.71Es+0.38Ec	13.82Es+0.07Ec
20	0.9	2Es+0.50Ec	5.28Es+0.50Ec	H`+12.71Es+0.49Ec	13.82Es+0.09Ec

*RCS denotes RMRCS or EDFRCS.

where H` is a constant multiple of hyperperiod H, E_s is energy consumption per unit time spent in scheduling decisions and E_c is energy consumption per preemption.

Given the earlier reports on energy measurements [Acquaviva 2003][Tan 2002], context-switching energy E_c is an order of magnitude larger than scheduling energy and as each scheduling decision is likely to require several instructions, an E_c/E_s value close to 100 is expected. However, this value also depends on several factors including architectural constraints, operating system implementation issues, whether thread switching or process switching is involved, and how much data movement is involved in a preemption. Thus, this table is useful in practice only in conjunction with specific experimental measurements of energy consumption on the target platform. But the table does give a good indication of when preemption reduction is worth considering and when it is not.

Also, it can be observed that the MLLF algorithm performs very badly with respect to energy consumption as it always spends more time on scheduling decisions than the RCS algorithms and the number of preemptions produced by MLLF is no better than that produced by EDF.

IntFragment is a static scheduling algorithm with O(1) online decision making time complexity. The number of preemptions saved in a schedule contributes to energy saving

directly. The saving of energy per unit time (one second) for IntFragment with respect to EDF and LLF algorithms when measured with 10 different task sets (utilization varying from 50% to 70%) is presented in Table 6.14.

Table 6.14: Energy saved by IntFragment algorithm compared to EDF and LLF

Exp. No	Energy Saved w.r.t. EDF		Energy Saved w.r.t. LLF	
	Min (uJ)	Max (mJ)	Min (uJ)	Max (mJ)
1	482.22	3.584	502.74	3.736
2	376.2	2.796	376.2	2.796
3	461.7	3.431	502.74	3.736
4	372.78	2.77	372.78	2.77
5	307.8	2.287	307.8	2.287
6	485.64	3.609	485.64	3.609
7	485.64	3.609	506.16	3.762
8	478.8	3.558	478.8	3.558
9	543.78	4.041	543.78	4.041
10	478.8	3.558	478.8	3.558

6.8. CONCLUSION

We have described the IntFragment, EDFRCS and RMRCS algorithms along with analysis and experimental evaluation of the same. We have also compared these algorithms with EDF, RM and LLF on various metrics. A comprehensive summary of this evaluation is presented (Table 6.15) in the form of desired performance characteristics and the corresponding choice of scheduling algorithms. The dynamic priority scheduling algorithms like EDF and RM along with their energy efficient variants EDFRCS and RMRCS were implemented in RTLinux. Four implemented scheduling algorithms were verified with many test cases and varying attributes of tasks. The implementation results support the simulation results discussed in section 6.7.

Table 6.15: Ready Reckoner for choice of Scheduling Algorithm

Desired Performance Characteristic	**Choice of Scheduling Algorithm**
Low Response Time	RM under low load and EDF under high load
Low Response Time Jitter	RM, if jitter is critical only for high frequency tasks RM, under low load and EDF under high load
Low Latency	EDFRCS
Ease of Implementation	RM under low load and EDF under high load
Low Energy Consumption	EDFRCS for small task sets Ec/Es ≥50 EDFRCS (Irrespective of task set size) Ec/Es around 10 RM under low load and EDF under high load

CHAPTER 7

CACHE CONSCIOUS SCHEDULING

7.1 INTRODUCTION

This chapter discusses a cache conscious dynamic priority-based real-time scheduling algorithm which reduces the cache impact caused by a real-time schedule. The main objective behind reducing the cache impact caused by a real-time schedule is to optimize the energy consumption due to data movements across the memory hierarchies. This work aims at providing a platform-independent scheduling algorithm which yields a reduced cache impact and increased power savings. This chapter explains a branch and bound approach to determine the schedule that causes the least cache impacts. This chapter also proposes a dynamic priority – based Reduced Cache Impact (RCI) modification to the Earliest Deadline First (EDF) real-time scheduling algorithm.

The assumptions made in section 6.2 are applicable here as well. In this work, the cache impact is measured as the total number of switching between jobs of different tasks. This is because the jobs of the same task will share the entire code area and a part of the data area, which makes the cache impact very minimal. This work also assumes that all the jobs require as much memory space as to cause the entire previous job's content to be flushed out of the cache completely.

7.2 A BRUTE – FORCE ALGORITHM FOR MINIMIZING CACHE IMPACT

We describe an offline algorithm that inspects all valid schedules to find one with minimum cache impact. This takes O(H!) (where H is the hyper-period) time and we use it as a standard to measure the effectiveness of other algorithms.

7.2.1 BRUTE-FORCE ALGORITHM FOR MINIMUM CACHE IMPACT

Inputs: hyper-period H, a list of job records Jobs

Output: A feasible schedule, if one exists, with minimum possible cache impact.

Steps:

1. Generate all schedules P of Jobs i.e. divide each job into sub-jobs of unit execution time and compute all permutations of the list of sub-jobs.

2. m=H; feasible =FALSE; cur = the first schedule in P.

3. for each permutation P_i in P,

a. check if P_i is feasible

b. if yes, then feasible = TRUE,

Compute the cache impact count m′;

if (m′ < m), then m = m′; cur=P_i.

4. if (feasible = FALSE) then output 'infeasible'

5. else output cur and m.

Finding all permutations**,** though offline**,** for a given task set (S) is impractical**,** if H is large. Thus, this stresses the need for an online scheduling algorithm which can compute a feasible schedule, if one exists in polynomial time with minimum cache impact.

7.3 REDUCED CACHE IMPACT (RCI) REAL-TIME SCHEDULING ALGORITHM

The objective of this work is to design an online dynamic, priority-based scheduling algorithm for hard real-time systems such that the generated schedule has the minimum possible cache impact. This work considers only periodic tasks.

7.3.1 INTRODUCTION

A preemption results in storing the context of the currently running process and loading the context of the next job to be executed. Most of the newly selected process' contents may not be available in the cache, which results in initiating data movement across the memory hierarchy. As the size of the cache is very small, almost all the preemption points are also cache impact points. The cache impact caused at preemption point is a subset of the total cache impact caused by the resultant schedule. The preemption points are the points where majority of the cache impacts take place. This premise puts forth an efficient solution that to reduce the cache impact, it is required to reduce the number of preemptions. The factors affecting the number of preemptions caused in a schedule include the number of tasks and the system utilization. Another solution would be to combine similar jobs together. This can be achieved by combining consecutive instances of the same task together as they share the code area and a portion of their data areas. However, combining consecutive instances of many tasks may result in an increase in the number of preemptions due to the fragmentation of the time frame, and thus leads to a

greater cache impact. The experimental results given in Table 7.1 justify this argument. Though the preemption count of the schedule increases while trying to combine more number of tasks, yet the schedule gives a better cache impact than that caused by the EDF schedule as shown in Table 7.2. The preemption count and the cache impact obtained from the schedule while combining instances of the 'N' higher frequency tasks is compared with those obtained from the schedule generated by EDF. In this experiment, N is varied from 1 to 10 for test cases with 20 tasks. From Tables 7.1 and 7.2, it is observed that for all the test cases, combining instances of the highest frequency task alone results in a significant preemption reduction and hence, greater cache impact saving. This motivates the design of an energy efficient scheduling algorithm called the Reduced Cache Impact (RCI) scheduling algorithm, which reduces both the number of preemptions and the cache impact as compared to the EDF. This is achieved by combining the maximum possible instances of the highest frequency task.

Table 7.1: Preemption variation (% saving caused by combining instances of 'N' higher frequency tasks) as compared to the EDF schedule

Utilization	**N=1**	**N=3**	**N=5**	**N=8**	**N=10**
50	+7.49	–7.55	–14.04	–16.79	–17.11
55	+6.92	–8.24	–15.10	–18.37	–18.74
60	+7.22	–7.91	–15.23	–18.52	–19.05
65	+6.81	–9.21	–16.54	–20.18	–20.82
70	+7.05	–9.42	–17.23	–21.35	–22.23
75	+6.57	–10.54	–18.65	–23.22	–24.09
80	+7.27	–10.36	–18.39	–23.20	–24.25
85	+7.06	–11.00	–20.27	–25.24	–26.47
90	+7.23	–10.96	–20.16	–25.66	–26.73
95	+7.34	–11.49	–21.00	–27.03	–28.05
100	+6.85	–12.00	–21.69	–27.58	–28.76

Table 7.2: Cache Impact variation (% saving caused by combining instances of 'N' higher frequency tasks) as compared to EDF schedule

Utilization	N=1	N=3	N=5	N=8	N=10
50	+7.04	+5.10	+3.99	+3.42	+3.34
55	+7.58	+5.29	+4.00	+3.25	+3.16
60	+8.42	+5.80	+4.20	+3.36	+3.22
65	+9.02	+5.74	+3.98	+2.93	+2.76
70	+9.72	+5.98	+3.91	+2.65	+2.40
75	+10.29	+5.95	+3.63	+2.20	+1.94
80	+11.17	+6.22	+3.70	+2.06	+1.73
85	+11.72	+6.22	+3.13	+1.32	+0.93
90	+11.52	+6.14	+3.19	+1.29	+0.94
95	+12.20	+6.17	+2.94	+0.72	+0.37
100	+12.39	+6.11	+2.69	+0.52	+0.12

RCI is an online dynamic priority-based real-time scheduling algorithm designed to minimize the cache impact of a given schedule. The proposed scheduling algorithm aims at reducing the amount of data transferred across the memory hierarchy with the least number of cache block replacements. This work adopts the heuristic of executing instances of the highest frequency task together, if possible, to minimize the cache impact. The execution of all the other task instances follows EDF scheduling policy. The jobs (instances) of the highest frequency task can be numbered as odd or even. This heuristic defers the execution of all the even jobs of the highest frequency task to the maximum possible extent and schedules all the odd jobs of the highest frequency task for execution as early as possible. The scheduling algorithm guarantees the successful execution of all the other jobs (if utilization is less than or equal to 100%) while deciding upon deferral and immediate execution of the instances of the highest frequency task. Basically, the system follows EDF with 100% schedulability, i.e., instances of all the tasks except those of the highest frequency task follow the EDF strategy for scheduling decisions. The proposed algorithm is described below.

7.3.2 REDUCED CACHE IMPACT (RCI) ALGORITHM

The following notation is used in the RCI algorithm:

readyQ(t) : the ready queue at time *t*, ordered by deadline.

deadline(J) : deadline of a job *J*.

execution_time(J) : execution time of a job *J*.

Algorithm Reduced Cache Impact Scheduling (RCIS)

Input: A list L of tasks T_1, T_2, … T_n, their periods and execution times and

A priority function *priority*, that is job-level fixed (-1 *deadline(J)).

Output: A feasible schedule for L or failure.

begin

Let the deferred switch be initialized as the Hyper period

Let *Cur* be the job selected by findNextJobToExecute(*readyQ(t), t) function at time 0 and* deferred switch be the value set by findNextJobToExecute(*readyQ(t), t)function at time 0.*

For every time unit *t* when there is at least one ***arrival*** or a ***departure*** or a ***deferred switch***

if (*Cur* is to depart OR new Job Arrived OR t = deferred switch time) then

Cur1 = findNextJobToExecute(*readyQ(t), t)* ;

if (Cur1 = = Cur) then ***schedule*** *Cur;*

else

preempt *Cur*; *Cur* = *Cur1*; ***schedule*** *Cur*;

end RCIS

function Job findNextJobToExecute(*readyQ(t), t)*

begin

Let *Cur* be the job with the highest priority in *readyQ(t)*

if (*Cur* is the even instance of the highest frequency task) then

deferredSwitchTime_Cur = maxDeferredTime(Cur,t);

if (deferredSwitchTime_Cur > 0) then

find the next highest priority Job J in readyQ(t),

if no other jobs exist then J=Cur;

if (J <> Cur) then

deferred switch = t + deferredSwitchTime_Cur; return J;

return Cur;

end findNextJobToExecute

```
function int maxDeferredTime(job, t)
begin
        return deadline(job) – t – execution time(job);
end maxDeferredTime
```

7.3.3 RCI ALGORITHM EXPLANATION

The working of the proposed algorithm is explained with the following task set in Table 7.3.

Table 7.3: Task list for the schedule

Task	Arrival Time	Period	Execution Time	Deadline
T0	0	5	1	5
T1	0	15	6	15
T2	0	30	12	30

Table 7.4: Job list Jobs, derived from Table 7.3 (Hyperperiod = 30)

Job	Arrival Time	Deadline	Execution Time
J0 (T0)	0	5	1
J1 (T0)	5	10	1
J2 (T0)	10	15	1
J3 (T0)	15	20	1
J4 (T0)	20	25	1
J5(T0)	25	30	1
J6(T1)	0	15	6
J7(T1)	15	30	6
J8(T2)	0	30	12

The RCI scheduling algorithm schedules the task set i.e., the corresponding job set given in Table 7.3 and Table 7.4. The ready queue is maintained in a sorted fashion according to the priority of jobs, where priority of a job J is defined as (-1*deadline (J)).

Figure 7.1 shows the resultant schedule generated by the RCI scheduler. The number of cache impact points in this schedule is 8 and the number of preemption points in the schedule is 3. Figures 7.2 and 7.3 show the resultant EDF and RM schedules respectively for the same task set. The number of cache impact points and the number of preemption points in the EDF schedule is 12 and 4 respectively.

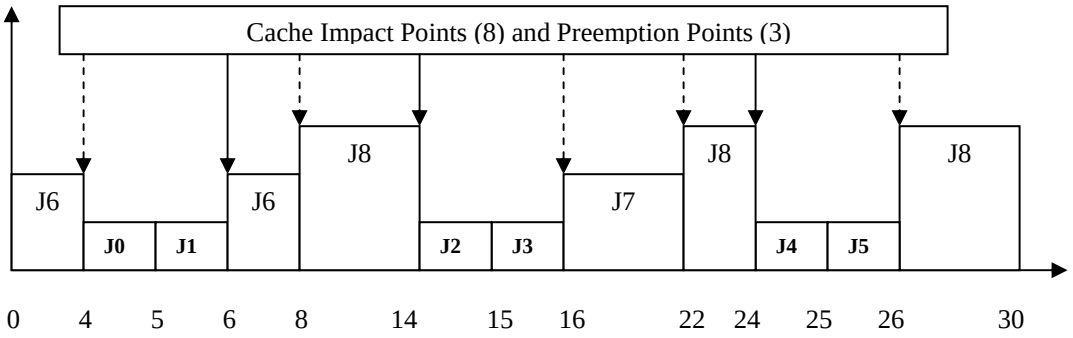

Fig. 7.1: Resultant RCI schedule for the task set given in Table 7.3

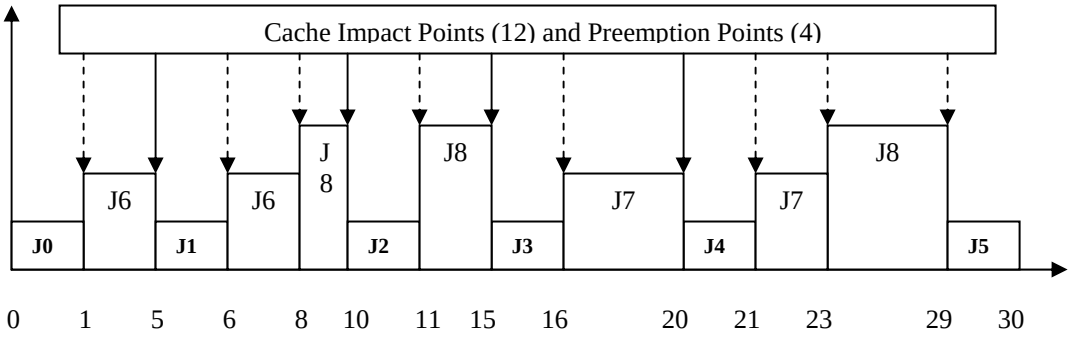

Fig. 7.2: Resultant EDF schedule for the task set given in Table 7.3

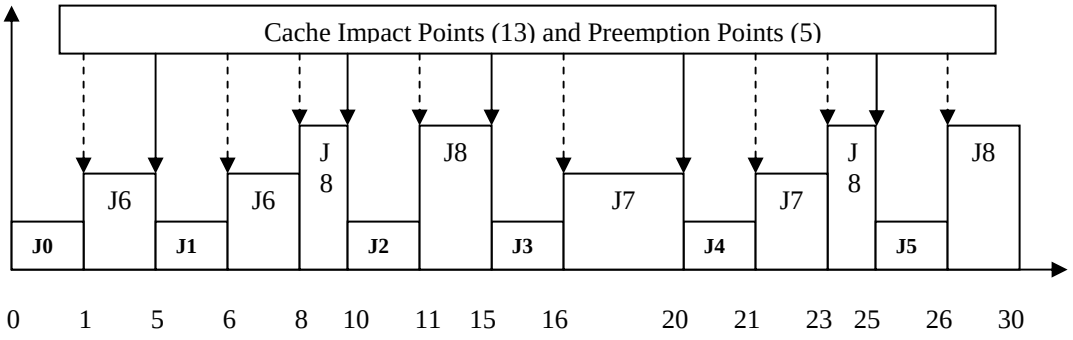

Fig. 7.3: Resultant RM schedule for the task set given in Table 7.3

The number of cache impact points and the number of preemption points in the RM schedule is 13 and 5 respectively. From this, one can observe that the EDF and RM scheduling algorithms perform badly when compared to the RCI scheduling algorithm

both in terms of the cache impact and preemption count. The brute-force result for the same task set is given in Figure 7.4. The brute-force approach gives the number of cache impact points as 8 and the number of preemptions as 1. From this, one can observe that RCI scheduling algorithm provides near optimal solution for cache impact points in almost all the cases.

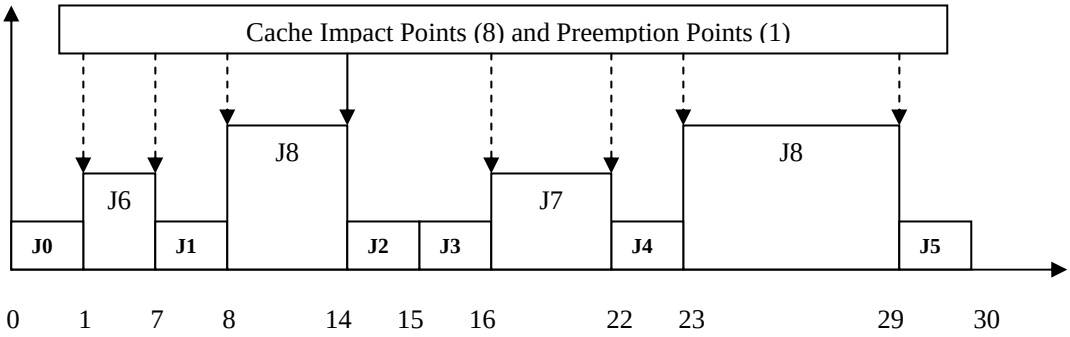

Fig. 7.4: Resultant Brute-Force schedule for the task set given in Table 7.3

7.3.4 SCHEDULABILITY OF THE RCI ALGORITHM

The adopted heuristic preserves the optimality of scheduling decisions i.e., RCI is optimal if and when EDF is optimal. As EDF is schedulable for all the periodic task sets with tasks having their deadlines equal to their respective periods and the task set utilization <=1, the RCI is also schedulable.

Some notations used in the proofs presented below:

- A schedule *S* is a sequence of runs, where each run is an instance of a task or a portion of it.
- To identify some runs of a schedule the following notation used is:

  (A1, B1,C1, ..., S1, A2, S1, T1, ...S2, ..., Sm)

  where, Ai to Ti are the i^{th} instances of the different tasks in a task set.
- Given a schedule *S*, $N_{CI}(S)$ denotes the number of cache impact points in *S*.

The following lemma is used in proving the theorems stated below and it can be informally stated as:

The deferral step in RCI – the step that postpone the execution of an even instance of the highest frequency task – does not affect schedulability.

Lemma 1:

Let *S* be a feasible schedule: *(A1, B1, C1, ….. S1, A2, …..… Sm)* where, A1 to S1 are the first instance of the tasks in the task set and A2 to Sm are the second instance to m^{th}

instance of the tasks in the task set. Assume *S* was generated by a priority scheduling algorithm.

Let *U* be the schedule *(B1, C1, ….. S1, A1, A2, …..… Sm),* where, the instance A1 of the highest frequency task is delayed to the maximum possible extent while ensuring that each of these jobs does not miss its deadline, thus schedulability is maintained and *U* is feasible with $N_{CI}(U) <= N_{CI}(S)$

Proof:

Let *A1* be the highest priority job in the ready queue, i.e., *priority(A1) >= priority(Ji) for all i*. Let t1 be the time when A1 is the highest priority job in the ready queue and t2 the deadline(A1). The maximum time one can postpone the execution of this job in a schedule is deferred time, given by

deferred time = (deadline(A1) – execution_time(A1) – current time)

During this time period for which the execution of A1 is deferred, jobs whose priority less than A1 are scheduled for execution. Each of the jobs executing between time t1 and time t2 thus advances its execution time by execution_time(A1). Then A1 executes from (t2 – execution_time(A1)) to t2.

This provides a valid schedule without any deadline miss. The resultant schedule also has lesser or atmost an equal number of cache impact points as compared to the previous schedule as similar jobs A1 and A2 are combined together at t2.

Thus, any delayed run of the highest priority job in schedule *S*, will not cause any run of any other lower priority job *F* to miss its deadline. So *U* is feasible. Furthermore, observe that the number of cache impact points differs by 1 between S and U; i.e. $\boldsymbol{N_{CI}(U) <= N_{CI}(S)}$

End of Proof

Theorem 1:

Given a set of N independent, preemptable and periodic tasks on a uniprocessor such that their relative deadlines are equal to their respective periods. Algorithm RCI generates a feasible schedule, if one exists and if the task set is EDF-schedulable. The RCI outputs a schedule with no more cache impact points than that in the EDF schedule.

Proof:

Let *S* be the job set corresponding to the given task set. The objective is to prove that the schedule generated by RCI for *S* is feasible, if there is a feasible schedule generated by EDF for *S.* In each iteration of the loop, RCI algorithm calls **findNextJobToExecute** function to determine the next job to execute in the CPU. The findNextJobToExecute function calls maxDeferredTime function only if the highest priority job in the ready queue is an even instance of the highest frequency task. So the scheduling decision is

- Either the same as the decision EDF would take, i.e., the highest priority job (in other words, the shortest deadline job) will be the next job to execute in the CPU.
- Or if the highest priority job is an even instance of the highest frequency task, then execute the next available highest priority job for 'k' units of time, where k is the maximum deferrable duration of the highest priority job without affecting any job's deadline. This decision does not affect the feasibility and optimality of the schedule, moreover, it may reduce the number of cache impact points.

There are five possible cases in RCI scheduling. They are:

Case 1: The highest priority job in the ready queue is not an instance of the highest frequency task; RCI schedules the highest priority job from the *ready queue*. So would EDF.

Case 2: The highest priority job in the ready queue is an instance of the highest frequency task but is an odd instance of the highest frequency task; RCI schedules the highest priority job from the *ready queue*. So would EDF.

Case 3: The highest priority job in the ready queue is an instance of the highest frequency task and is an even instance of the highest frequency task. But the maxDeferredTime of the job is zero; RCI schedules the highest priority job from the *ready queue*. So would EDF.

Case 4: The highest priority job in the ready queue is an instance of the highest frequency task and is an even instance of the highest frequency task. The maxDeferredTime of the job is greater than zero but there is no other job available in the ready queue; RCI schedules the highest priority job from *ready queue*. So would EDF.

Case 5: The highest priority job in the ready queue is an instance of the highest frequency task and is an even instance of the highest frequency task. The maxDeferredTime of the

job is greater than zero and there is more than one job available in the ready queue; RCI schedules the second highest priority job from the *ready queue* for a duration of maxDeferredTime duration or until a new arrival or departure occurs. This leads to combine adjacent instances of the highest frequency task, if possible, which results in reducing the cache impact points by one, or in the worst-case, the number of cache impact points is the same as that by EDF without affecting the schedulability of the task set. By *Lemma 1*, feasibility is invariant under this transformation and $N_{CI}(U) <= N_{CI}(S)$. Thus, it is shown that a single iteration of the loop in RCI results in a decision that is as feasible as one by the EDF. Hence, *by induction on the number of iterations of the loop*, one can conclude that the RCI outputs a schedule that is feasible, if EDF outputs a feasible schedule for the same input.

Furthermore, each iteration of the loop in RCI introduces no additional cache impact points than the EDF would. In fact, as argued above, possibilities 1, 2, 3 and 4 agree with a decision EDF would make, and possibility 5 may reduce the number of cache impact points in comparison with EDF.

End of Proof

7.3.5 ANALYSIS OF THE RCI ALGORITHM

7.3.5.1 Quality of the RCI schedule

The quality of the RCI scheduling algorithm is analyzed with the following example. The task set is given in Table 7.5 and the job list derived from the given task set is given in Table 7.6. The resultant schedule produced by EDF, RM, EDFRCS, RCI and Brute-force approach is shown in Figure 7.5, 7.6, 7.7, 7.8, and 7.9 respectively.

Table 7.5: Task list for the schedule

Task	Arrival Time	Period	Execution Time	Deadline
T0	0	5	1	5
T1	0	7	3	7
T2	0	35	13	35

From the resultant schedule it is evident that the RCI scheduling algorithm performs far better than the EDF and RM scheduling algorithms. The EDF scheduling algorithm produces a schedule with 6 preemptions and 18 cache impact points. The RM scheduling

algorithm, which always produces greater or an equal number of preemptions and cache impact points as compared to the EDF schedule, produces 8 preemptions and 20 cache impact points.

Table 7.6: Job list Jobs, derived from Table 7.5 (Hyperperiod = 35)

Job	Arrival Time	Deadline	Execution Time
J0 (T0)	0	5	1
J1 (T0)	5	10	1
J2 (T0)	10	15	1
J3 (T0)	15	20	1
J4 (T0)	20	25	1
J5(T0)	25	30	1
J6(T0)	30	35	1
J7(T1)	0	7	3
J8(T1)	7	14	3
J9(T1)	14	21	3
J10(T1)	21	28	3
J11(T1)	28	35	3
J12(T2)	0	35	13

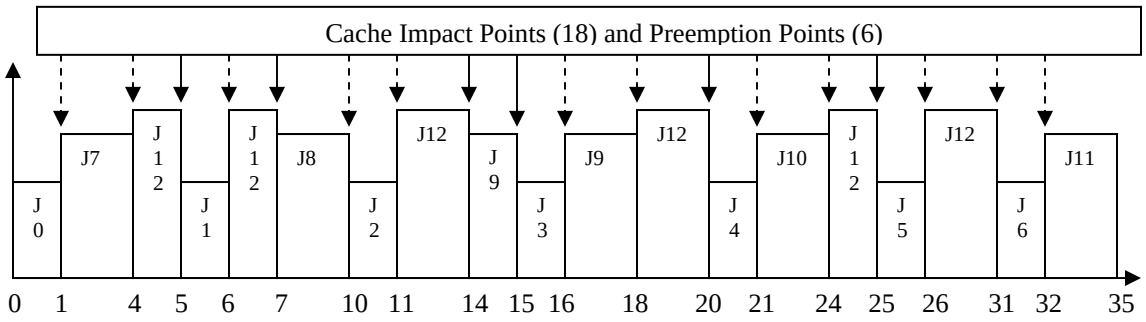

Fig. 7.5: Resultant EDF schedule for the task set given in Table 7.5

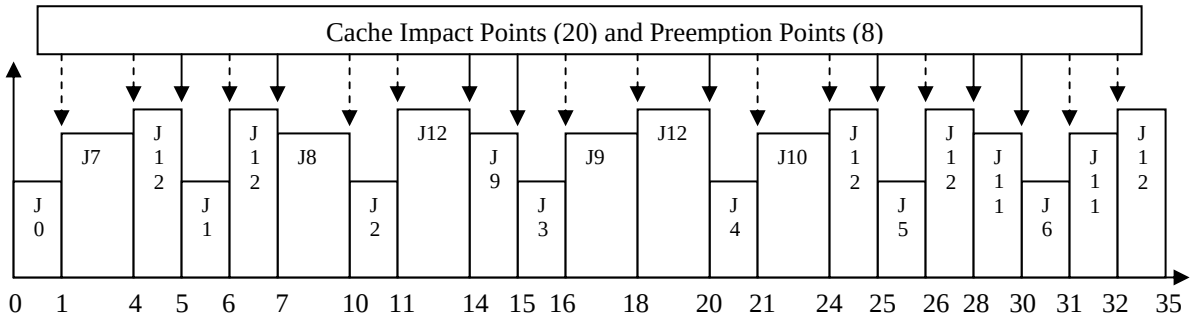

Fig. 7.6: Resultant RM schedule for the task set given in Table 7.5

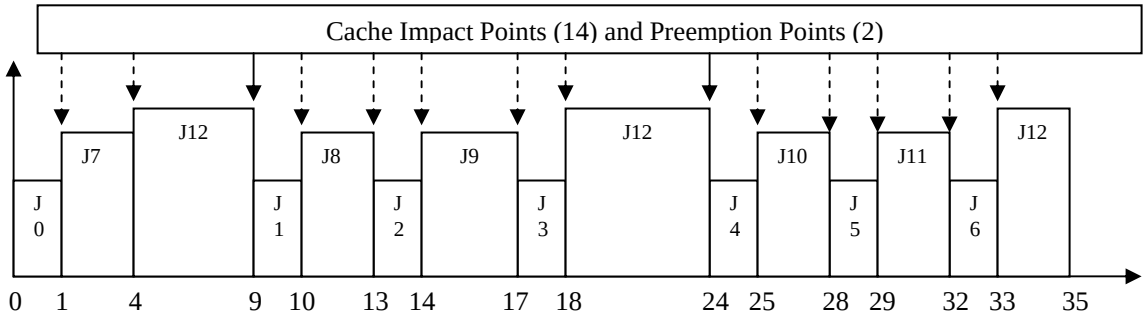

Fig. 7.7: Resultant EDFRCS schedule for the task set given in Table 7.5

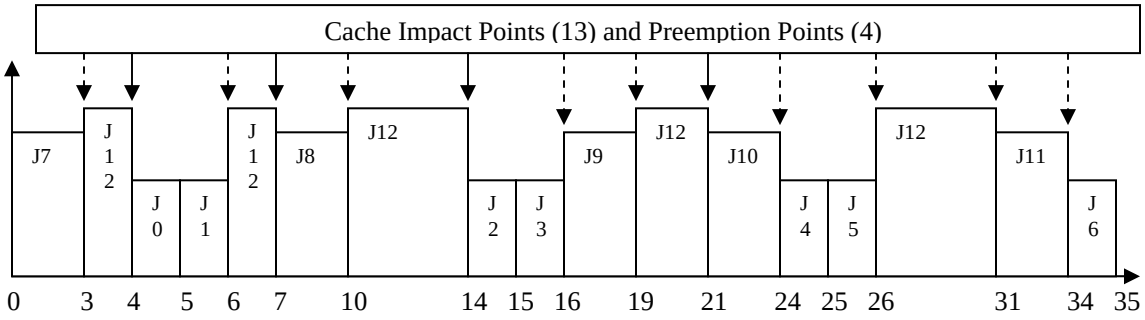

Fig. 7.8: Resultant RCI schedule for the task set given in Table 7.5

The resultant RCI schedule performs far better than the EDF and RM schedule both in terms of number of the preemptions and cache impact points. For the same task set, the RCI scheduling algorithm produces 4 preemptions and 13 cache impact points. Though the EDFRCS scheduling algorithm produces lesser number of preemptions as compared to the RCI scheduling algorithm, it still produces greater or an equal number of cache impact points in comparison with the RCI schedule. For the given task set, the EDFRCS produces only 2 preemptions whereas, the RCI produces 4 preemptions. But the number of cache impact points in EDFRCS schedule is 14, which is greater than the 13 cache impact points produced by the RCI schedule.

Though the RCI scheduling algorithm produces a schedule with lesser number of cache impact points when compared to other dynamic scheduling algorithms like EDF, RM, LLF, MLLF, and EDFRCS, it is not the optimal schedule. Figure 7.9 shows the Brute-force schedule with an optimal number of cache impact points. The number of cache impact points in Brute-force algorithm is 12. This proves that the RCI is a near-optimal heuristic for reducing the number of cache impact points in a schedule.

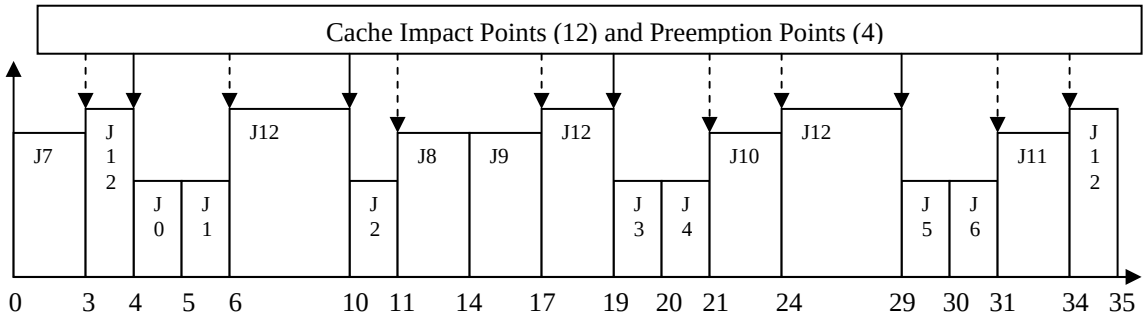

Fig. 7.9: Resultant Brute-force schedule for the task set given in Table 7.5

7.3.5.2 Complexity of the RCI Algorithm

The RCI scheduling algorithm calls the findNextJobToExecute function to identify the next job to execute in the CPU. For all cases in the ready queue except for an even instance of the highest frequency task, the RCI algorithm (findNextJobToExecute function) performs similar to the EDF and the complexity of the scheduling algorithm is O(N). This is because the selection of the next job to run in the CPU takes O(1) time, if the ready queue is a priority queue. In this case, maintaining the ready queue such that it remains sorted always takes O(N) complexity. If the ready queue is not a priority queue then searching the highest priority ready-to-run job in the unsorted list takes O(N) time.

The special case encountered by the RCI scheduler is when the highest priority job in the ready queue is an even instance of the highest frequency task. In this case, the findNextJobToExecute function in the RCI scheduler calls the maxDeferredTime function to calculate the maximum possible deferred time. Based on the maximum deferred time, the findNextJobToExecute module selects the highest priority or the second highest priority job for execution. If the second highest priority job is selected, then the deferred time is set as an additional scheduler invocation point. The worst possible scenario occurs when it is required to execute both the findNextJobToExecute

and maxDefferedTime functions for finding the next job to run in the CPU. In this case the complexity of the selection is the summation of the time complexity of findNextJobToExecute and maxDeferredTime functions. The complexity of the findNextJobToExecute function is O(1), as it checks only the head of the queue to find whether the highest priority task is an even instance of the highest frequency task. The maxDeferredTime function also has a time complexity of O(1) as the maximum deferred time of the highest priority task is based on only the deadline, execution time and the current time of the task. As the worst case time complexity for finding the next job to execute in the CPU is O(1), and the worst case time complexity for inserting a job into the ready queue (priority queue) is O(N), the resultant time complexity of the RCI scheduling algorithm is O(N) which is same as that of the EDF scheduling algorithm.

7.4 COMPARATIVE EVALUATION

In this section, the evaluation and comparison of the RCI scheduling algorithm against various dynamic priority-based scheduling algorithms like EDF, RM and LLF for periodic real-time tasks is carried out. This section compares the quality of schedules of all these algorithms in terms of the number of cache impact points and the number of preemption points. The experimental measurements for each of the metrics are shown; the results for the different algorithms are analyzed and compared. The experimental setup includes a simulation of all the algorithms and different test suites generated under certain conditions: each test suite is characterized by either a fixed number of tasks with utilization varying from low (50%) to high (100%) or by a fixed utilization with the number of tasks varying from 2 to 20. Each test suite includes 100 different task sets of varying hyperperiods – from 100 to 32000. The results obtained are then averaged over these 100 test suites as appropriate. Schedulability is not included as a metric in this evaluation as RCI is schedulable if EDF schedulable.

7.4.1 NUMBER OF CACHE IMPACT POINTS

The cache impact count values obtained during the experimentation are normalized over the number of jobs and the tasks are ordered by non-increasing frequency. The trend observed among the algorithms with respect to the number of cache impact points introduced in the resulting schedules is RCI <=EDF <=RM <= LLF.

The charts below (Figure 7.10 to 7.13) represent the cache impact count against utilization for task sets with a fixed number of tasks. It is observed that the number of cache impact points increases with increase in utilization, irrespective of the number of tasks due to the increase in execution time and schedulability constraints.

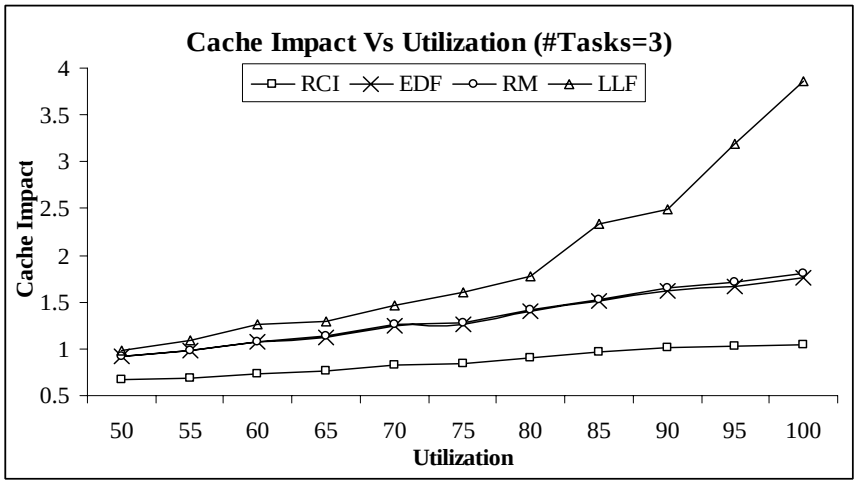

Fig. 7.10: Cache Impact points per Utilization (# Tasks = 3)

It is seen that the RCI algorithm exhibits a significant reduction in the number of cache impact points when compared to their traditional counterparts, irrespective of the utilization level, thus fulfilling the purpose of their design. For instance, at 50% utilization, EDF results in an average of 1.143 cache impact points per job, whereas RCI results in an average of 1.01 cache impact points per job, which is a 13% reduction. For a fully loaded system, EDF results in an average of 1.6 cache impacts per job, whereas RCI results in 1.28 cache impact points per job, which is about a 25% reduction.

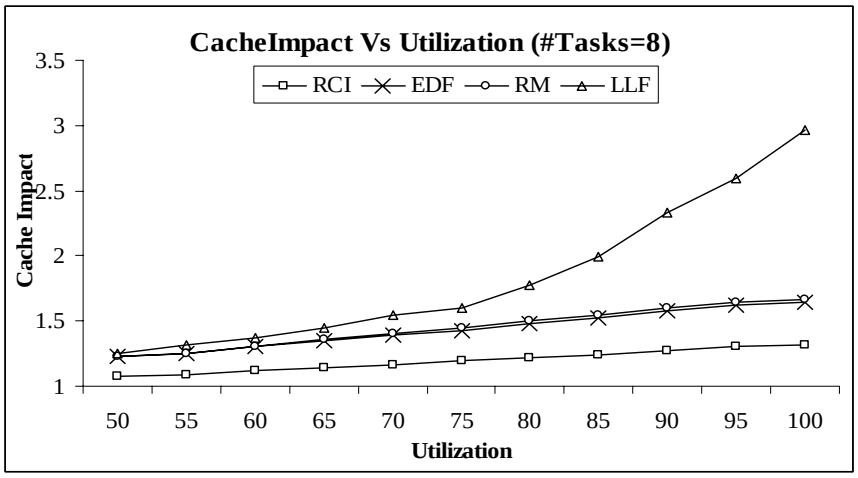

Fig. 7.11: Cache Impact points per Utilization (# Tasks = 8)

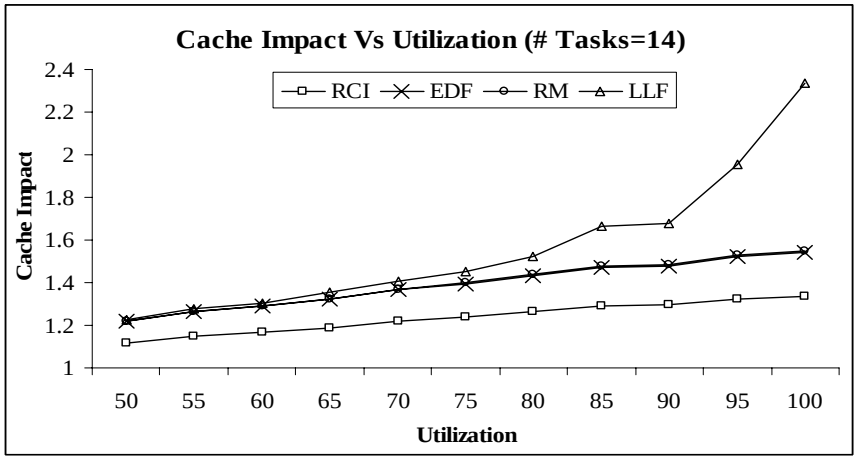

Fig. 7.12: Cache Impact points per Utilization (# Tasks = 14)

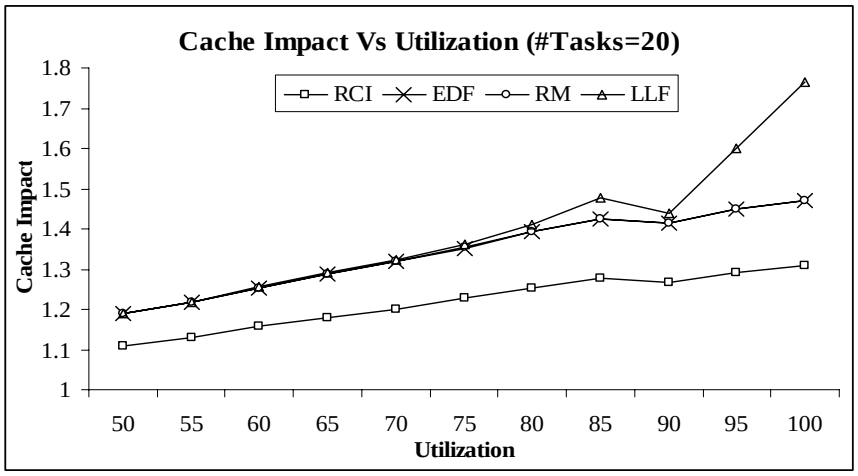

Fig. 7.13: Cache Impact points per Utilization (# Tasks = 20)

The result also confirms that the EDF produces lesser number of cache impact points than RM and LLF. EDF outputs schedules that have about 1% lesser cache impact points than RM on an average. Similarly, LLF introduces greater cache impact points compared to the RCI, EDF and RM. The RCI scheduling algorithm eliminates around 15% and 35% of the cache impact points in an LLF schedule when the utilization is 50% and 100% respectively. Among the traditional algorithms, LLF's cache impact count increases drastically with increase in utilization and it performs poorly at high utilizations (>= 80%), due to the frequent changes in the priority of jobs.

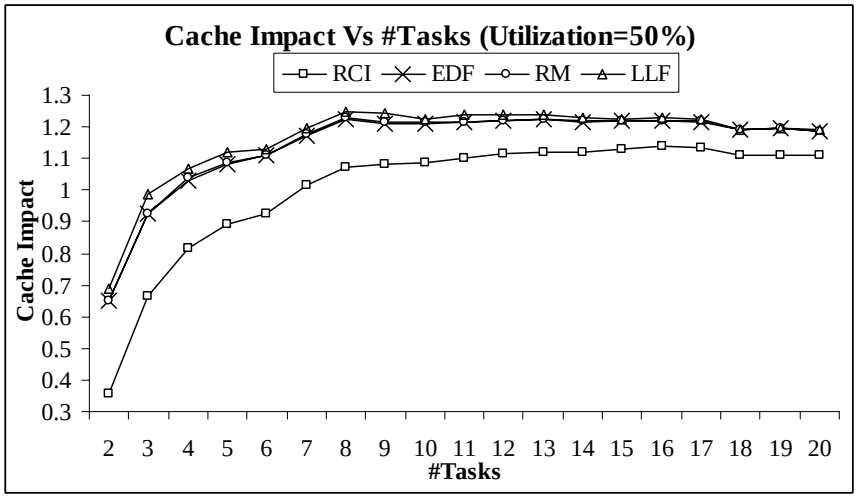

Fig. 7.14: Cache Impact points per Number of Tasks (Utilization = 50%)

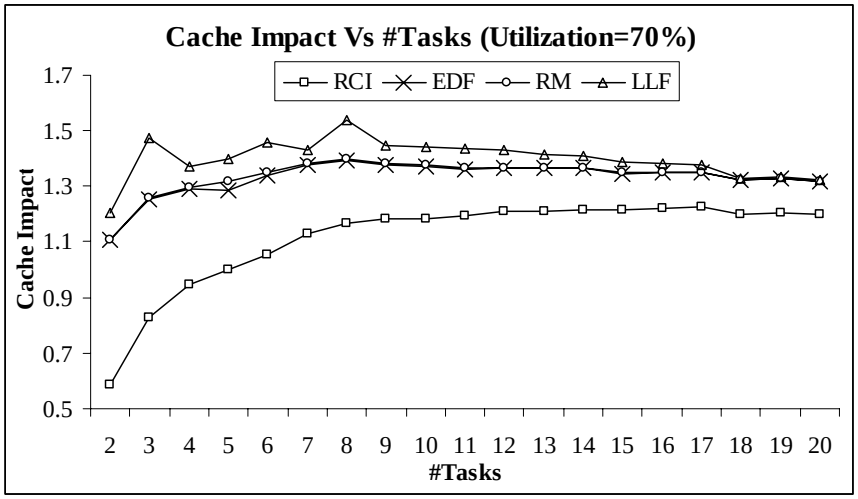

Fig. 7.15: Cache Impact points per Number of Tasks (Utilization = 70%)

The charts below (Figure 7.14 to 7.17) represent the cache impact count against the number of tasks in a task set with a fixed utilization. It is observed that for each scheduling algorithm under consideration, the number of cache impact points initially increases with increase in number of tasks (till number of tasks <=8) and then stabilizes in and around that value.

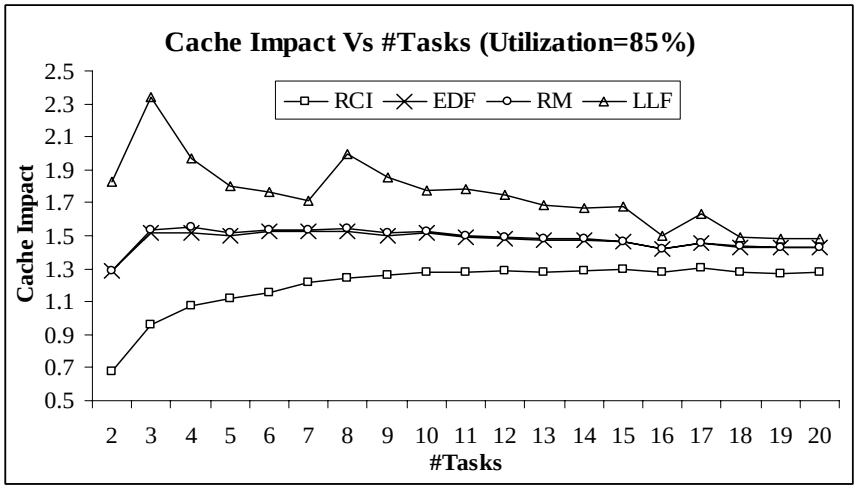

Fig. 7.16: Cache Impact points per Number of Tasks (Utilization = 85%)

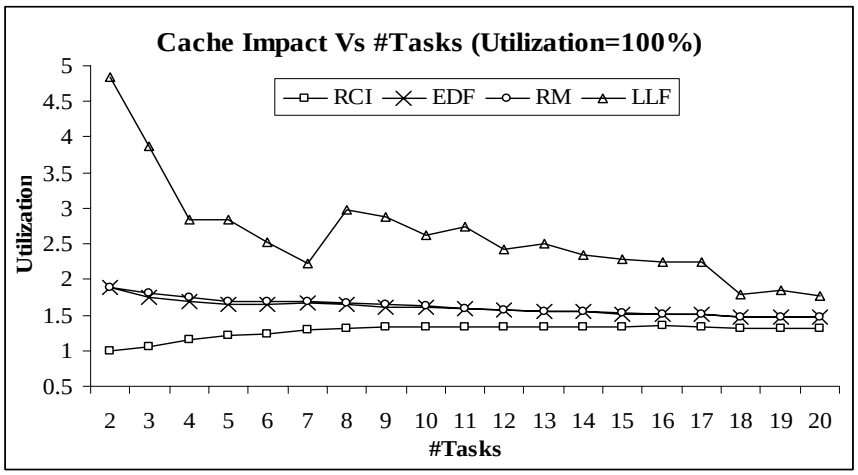

Fig. 7.17: Cache Impact points per Number of Tasks (Utilization = 100%)

7.4.2 NUMBER OF PREEMPTIONS

The preemption count values obtained are normalized over the number of jobs and the tasks are ordered by decreasing frequency. The trend observed among the algorithms with respect to the number of preemptions introduced in the resulting schedules is as follows: RCI <=EDF <=RM <= LLF

The charts below (Figure 7.18 to 7.21) represent the preemption count against utilization for task sets with a fixed number of tasks. It is observed that the number of preemptions increases with increase in utilization, irrespective of the task set size, due to the increase in execution time and schedulability constraints.

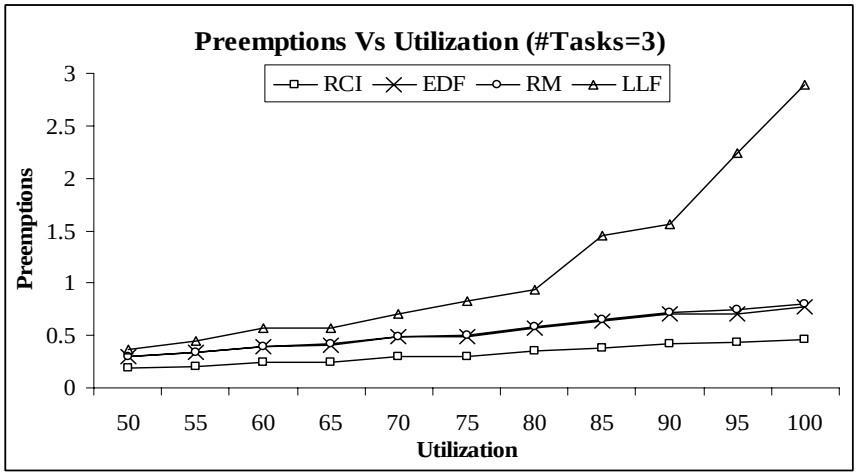

Fig. 7.18: Preemptions per Utilization (# Tasks = 3)

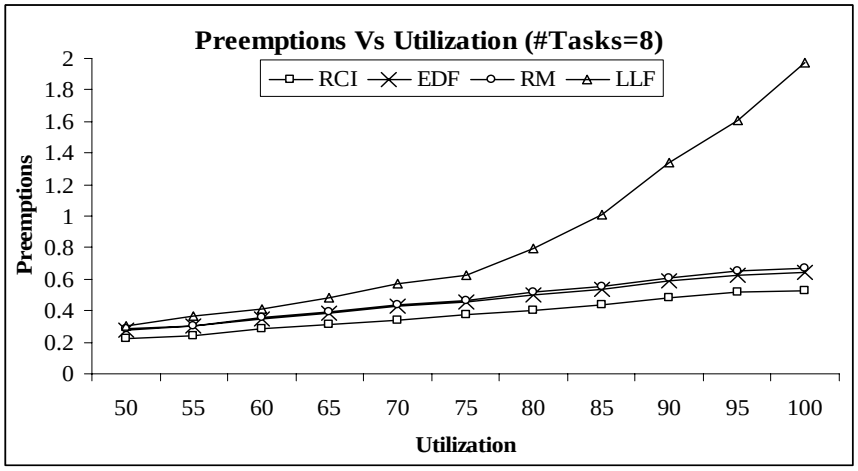

Fig. 7.19: Preemptions per Utilization (# Tasks = 8)

It is seen that the RCI algorithms show a significant reduction in the preemption count when compared to their traditional counterparts, irrespective of the utilizations. For instance, at 50% utilization, EDF results in an average of 0.25 preemptions per job, whereas RCI results in an average of 0.20 preemptions per job, which is about a 25% reduction. For a fully loaded system (100% utilization), EDF results in an average of 0.602 preemptions per job, whereas RCI results in 0.49 preemptions per job, which is about a 22% reduction. The preemption reduction by RCI when compared to that by the RM and LLF at 50% utilization is 23% and 33% respectively. The preemption reduction by the RCI when compared to that of the RM and LLF at 100% utilization is 26% and 229% respectively. LLF's preemption count increases drastically with an increase in utilization and it performs poorly at high utilizations (>= 80%) due to the frequent

changes in the priority of jobs owing to the laxity metric. A detailed analysis of the EDF, RM, and LLF is presented in Chapter 6. The charts below (Figure 7.22 to 7.25) represent the preemption count against the number of tasks in a task set with a fixed utilization. It is observed that for each scheduling algorithm under consideration, the number of context switches decreases with an increase in the number of tasks at a fixed utilization, irrespective of the utilization. This behavior is attributed to the fact that an increase in the number of tasks at a fixed utilization causes the execution time per task to reduce, resulting in reduced preemptions before the completion of jobs.

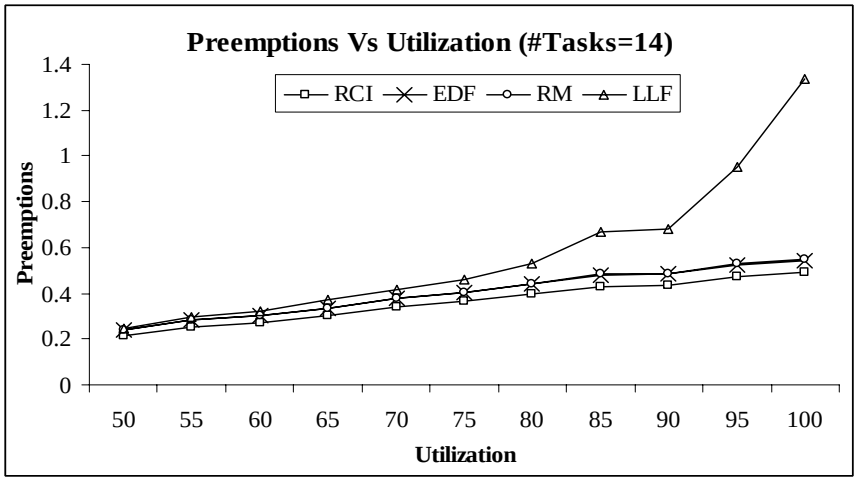

Fig. 7.20: Preemptions per Utilization (# Tasks =14)

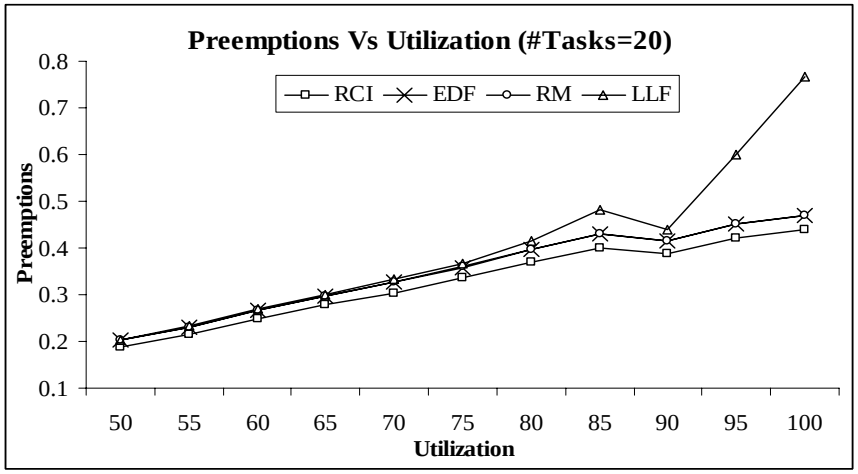

Fig. 7.21: Preemptions per Utilization (# Tasks = 20)

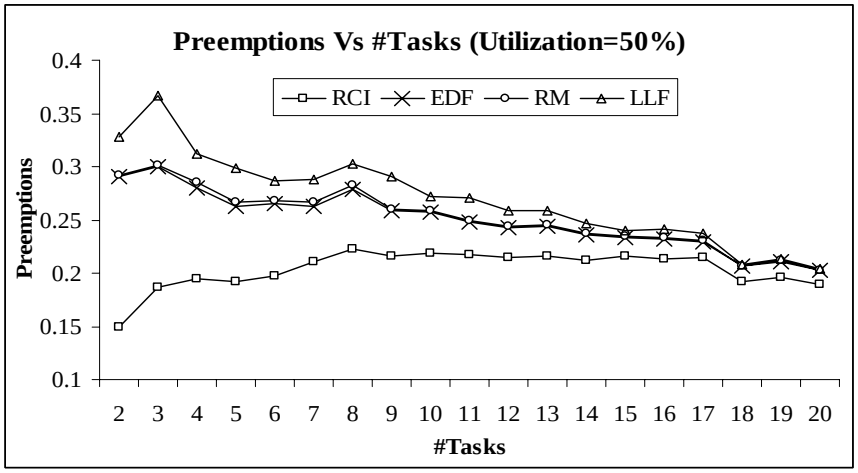

Fig. 7.22: Preemptions per Number of Tasks (Utilization = 50%)

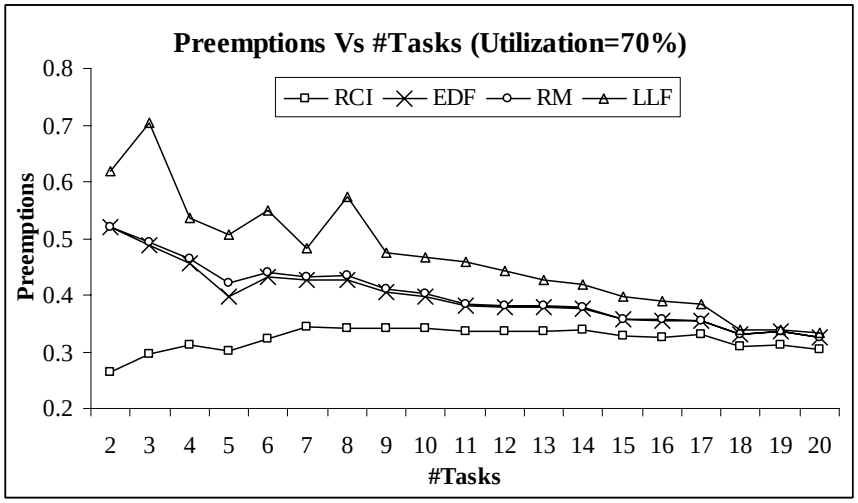

Fig. 7.23: Preemptions per Number of Tasks (Utilization = 70%)

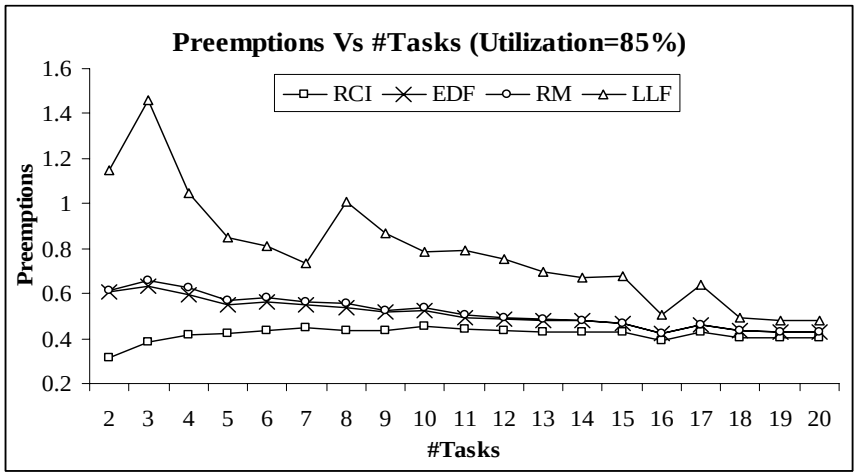

Fig. 7.24: Preemptions per Number of Tasks (Utilization = 85%)

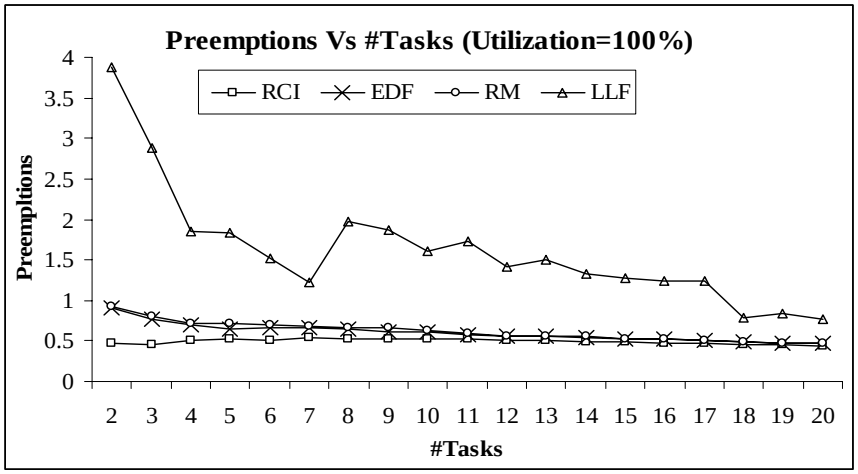

Fig. 7.25: Preemptions per Number of Tasks (Utilization = 100%)

As the number of tasks in the task set increases, preemption reduction is less pronounced for the RCI algorithm. This is because all the algorithms, except the RCI reduce the preemption count with increase in the number of tasks in the task set, but RCI performs consistently, irrespective of number of jobs. Thus preemption reduction is less pronounced for the RCI algorithm with a larger number of tasks. A detailed analysis of the EDF, RM, and LLF with regard to preemption reduction is presented in Chapter 6.

7.5 CONCLUSION

In this chapter, initially, a discussion on the general Brute-Force algorithm that determines the schedule with the minimum possible cache impact points has been presented, following which the newly designed cache impact point reduction-specific scheduling algorithm (i.e.) RCI algorithm has been discussed theoretically and then evaluated experimentally too using various task sets and by varying the different parameters of the number of preemptions and number of cache impact points. The results thus obtained have been analyzed and compared against the traditional algorithms for different performance metrics. The RCI scheduling algorithm reduces the number of cache impact points up to 25%, 26% and 35% of those generated by the EDF, RM and LLF respectively. The RCI scheduling algorithm also reduces the number of preemptions up to 22%, 26% and 229% of those generated by the EDF, RM and LLF respectively, thus establishing its objective.

CHAPTER 8

CONCLUSION

This thesis addresses the energy consumption issues in multi-tasking real-time embedded systems at the architecture level and the operating system level. This is achieved by proposing various techniques to reduce cache and scheduling-related energy consumption. This chapter summarizes the major contributions made and points out some of the possible extensions of the work.

In this thesis, architecture level energy efficiency is achieved by improving the cache hit rate by modifying the LRU replacement strategy, reducing the internal activity, hardware complexity, prediction miss rate and the effective cache access time of a way prediction cache, which is an energy efficient cache architecture and by designing process aware cache architectures that achieve reduced energy consumption while running multi-tasking real-time applications. The operating system level energy efficiency is achieved by designing static and dynamic real-time scheduling algorithms resulting in reduced preemptions and by designing a dynamic real-time scheduling algorithm causing reduced cache impacts. Chapter 2 summarized the major work done in the area of energy efficient cache architecture and real-time task scheduling. Chapters 3, 4 and 5 addressed different cache memory-related techniques to reduce energy consumption. Chapter 6 dealt with reduction of scheduling-related energy consumption due to preemptions and Chapter 7 elaborated on the technique based on cache-conscious scheduling to reduce the cache-related energy consumption caused by the scheduler.

Chapter 3 of this thesis focused on the design, implementation and analysis of a new variant of LRU replacement strategy called the LLRU, which helps in increasing the life span of shared cache lines with the help of compiler information. The cache hit rate for various cache configurations employing the LRU and LLRU replacement strategies was measured using a software simulator with Simplescalar benchmark address traces. The hardware implementations of the LRU and LLRU based on the square matrix and counter designs were carried out using the tools ModelSim and Leonardo Spectrum. The layouts of these hardware architectures were also obtained from the tool IC station. The results

thus obtained, were analyzed, evaluated and compared on the basis of parameters like area, clock frequency, critical path delay, number of transistors and the cache hit rate. The LLRU hardware implementation offers a higher clock frequency and lesser critical path delay, but it demands more number of transistors and area. The experimental analysis reveals that with minimal extra hardware, the LLRU improves cache performance significantly.

Chapter 4 dealt with designing energy efficient cache architecture. The work explains and evaluates a modification of the way prediction scheme – way predictive placement scheme – to improve the cache performance in terms of power, access time, prediction hit rate and cache hit rate with lesser hardware complexity. This is achieved by replacing the MRU table in way prediction scheme with a register i.e., local prediction in way prediction scheme is modified with global prediction in way predictive placement scheme. To improve the prediction hit rate and cache hit rate in way predictive placement scheme, a modified placement / replacement strategy called the ALRU replacement strategy was proposed. The conventional cache, way prediction cache and the way predictive placement cache were evaluated using the Simplescalar 2.0 cache simulator with SPEC95 benchmark suite. Based on evaluation results given in Table 4.2, one can comprehend that the way predictive placement cache performs better than the way prediction cache. Way predictive placement cache reduces the hardware complexity from $k * \log_2 N$ bits to $\log_2 N$ bits where k is the number of sets and N is the associativity.

The concept of process aware energy efficient cache design for Embedded Systems is oriented towards achieving energy efficiency in Embedded Systems. Chapter 5 presents two new process aware energy efficient caching schemes for an N-way set-associative cache: (i) a process aware selective placement scheme (PASP) with a victim set and (ii) a shared memory process aware selective placement (SMPASP) scheme with small shared and victim sets. These schemes work based on the cache – operating system – compiler interaction. These two schemes aim at bringing down the power consumption while improving the cache hit rate of the process aware cache. Here, the proposed schemes were assessed and compared with the conventional set-associative and way – prediction cache with respect to the first cycle hit rate, cache hit rate, number of tag comparisons,

effective cache access time, dynamic power consumption and leakage power consumption for various cache configurations. This performance evaluation was carried out with independent processes and with processes which exhibit a considerable amount of data sharing among them. A cache simulator CACHEMEM 1.0 with configurable cache size, cache line size and context switching duration was implemented. The simulator used SPEC95 benchmark address traces for evaluation. The dynamic and leakage power consumption for the various caching schemes were obtained using the eCACTI cycle-based power estimation model. A comprehensive summary of this evaluation is presented in Table 5.2, in the form of desired performance characteristics and the corresponding choice of the appropriate cache architectures.

Chapter 6 discussed the techniques adopted to reduce power consumption at the operating system level. Here, various scheduling algorithms which reduce the number of preemptions in a real-time schedule were discussed. This work proposed and implemented a platform independent static scheduling algorithm called IntFragment and two platform independent dynamic scheduling algorithms called EDFRCS and RMRCS. These algorithms reduce the power consumption by aggressively reducing preemptions without extensive computations. All these scheduling algorithms were discussed theoretically and evaluated experimentally using various task sets by varying different parameters like task set size, utilization and hyper-period. The results obtained were analyzed and compared with EDF, RM, LLF and MLLF against the different performance metrics of response time, response time jitter, latency, scheduling complexity, preemption count and energy consumption. A summary of this evaluation is presented in Table 6.16, in the form of desired performance characteristics and the corresponding choice of scheduling algorithms has been put forth. The EDF, RM, EDFRCS and RMRCS scheduling algorithms were implemented and tested in the RTLinux real-time operating system.

Chapter 7 of this thesis talked about cache-conscious dynamic priority-based real-time scheduling algorithm called the RCI which reduces the cache impact of a real-time schedule. This work decreases the energy consumed by reducing the cache impact, i.e., reduction in data movements across memory hierarchies demanded by preemptions. The

RCI, a modification of EDF, was discussed theoretically and evaluated experimentally too, using various task sets by varying different parameters like task set size, utilization and hyper-period. The results obtained were analyzed and compared with the EDF, RM and LLF against the number of preemptions and number of cache impact points. The RCI scheduling algorithm reduces the number of cache impact points by up to 25%, 26% and 35% of those caused by EDF, RM and LLF respectively. The RCI scheduling algorithm also reduces the number of preemptions by up to 22%, 26% and 229% of those caused by the EDF, RM and LLF algorithms respectively.

Further analysis and experimentation in the area of energy efficient embedded systems includes an extension of the EDFRCS and RMRCS, i.e., applying preemption reduction heuristics to platform-dependent algorithms like DVS and DFS for better power saving. Nowadays, the requirements of an embedded system are varied at different points of time. Sometimes, the embedded system demands high response time and at some other times, it should work with minimum energy consumption or minimum latency. Achieving all of these with a single scheduling algorithm is a cumbersome task. This requires a new combo design with multiple scheduling algorithms as a part of the scheduler. Future work in this area is oriented towards designing such a scheduler which can select a suitable scheduling algorithm on-the-fly according to the requirements, thus achieving the required performance. The design of an optimal energy efficient dynamic scheduling algorithm with optimal cache impact and preemptions supporting DVS / DFS is a potential future research area. The entire research is based on the assumption that the Embedded System is a uniprocessor machine. In the future, some of the system side assumptions like a uniprocessor model to a multicore one and distributed systems and other real-time task set assumptions like the task period is equal to its deadline, which implies only one instance of a task is available in the ready queue at any point of time, etc. can be relaxed. In the area of cache architecture, further work includes the modification of PASP and SMPASP cache architectures to allocate variable number of ways to a process based on some parameters like priority, cache usage and the cache hit rate of a process. This will improve the utilization of the cache when less number of processes are executing. Later, one can also explore the possibility of adopting these techniques in multi-threaded multi-core architectures.

ANNEXURE A – COMPARISON CHART OF CACHE REPLACEMENT ALGORITHMS

Replacement Algorithms	Cache miss rate w.r.t. LRU	Speedup (ECAT)	Action on cache hit	Action on cache miss	Additional Energy Required	Hardware Complexity	Additional instructions needed	L1 DC / IC / DC & IC / L2
LRU	REFERENCE	REFERENCE	Update LRU counters	Update LRU counters	N $\log_2$ N bits per set and associated circuitry	N $\log_2$ N bits per set	NIL	L1 DC & IC / L2
OPT	Up by 70%	MIN	NA	NA	NA	NA	NIL	L1 DC & IC / L2
RAND	Less by 22%	Average 22% slower	NONE	Update a register	$\log_2$ N bits and Pseudo Random generator	Pseudo Random generator + $\log_2$ N bits	NIL	L1 DC & IC / L2
FIFO	Less by 20%	Average 20% slower	NONE	Update FIFO counter	N $\log_2$ N bits per set and associated circuitry	N $\log_2$ N bits per set	NIL	L1 DC & IC / L2
LFU	Less by 18%	Average 18% slower	Update LFU counter	Update LFU counter	N $\log_2$ X bits per set and associated circuitry	N $\log_2$ X bits per set	NIL	L1 DC & IC / L2
LFUDA	Less by 15%	Average 15% slower	Update LFU counters + shift LFU counters if reference counter = MAX	Update LFU counter + shift LFU counters if reference counter = MAX	$\log_2$ X bits reference counter and shifting operation after every MAX references	N $\log_2$ X bits per set + $\log_2$ X bits for reference counter	NIL	L1 DC & IC / L2
MRU	Less by at least - 100%	At least - 100%	Update MRU counters	Update MRU counters	N $\log_2$ N bits per set and associated circuitry	N $\log_2$ N bits per set	NIL	L1 DC & IC / L2
EELRU	Up by 30%	Best case 30% faster	Update counters	Update counters	K $\log_2$ X + 2 $\log_2$ X bits per set, associated circuitry and feedback logic	K $\log_2$ X bits per set + 2 $\log_2$ X for total no. of page ref per recency region + feed back circuit	NIL	L2
PLRU	Up by 2.5%	Best case 2.5% faster	Update MRU bits	Update MRU bits	N bits per set and associated circuitry	N bits per set	NIL	L1 DC & IC / L2

MLRU	Up by 12%	12% faster	Update LRU counters	Update LRU counters and non-temporal (nt) bit	N $\log_2$ N + N bits per set and associated circuitry	N $\log_2$ N + N bits per set	Additional cache hint instructions to transfer non-temporal hint	L1 DC
CACHE / RC	Up by 60.9%	12% faster	Update LRU counters	Update LRU counters and lock/ release bit	N $\log_2$ N + N bits per set and associated circuitry	N $\log_2$ N + N bits per set	Additional instructions for lock and release operations	L1 DC
PRL	Up by 12%	12% faster	Update LRU counters and temporal bits	Update LRU counters and temporal bits	N $\log_2$ N + N bits per set, its associated circuitry and profiling	N $\log_2$ N + N bits per set	Modified Load instructions to set temporal bit and non-temporal instructions to reset temporal bit	L2
ORL	Up by 20%	20% faster	Update LRU counters, temporal bits and locality table	Update LRU counters, temporal bits and locality table	N $\log_2$ N + N bits per set, locality table and associated circuitry	N $\log_2$ N + N bits per set + locality table	NIL	L2
NTS	Up by 53%	5KB NTS cache is 12.6% faster than 8KB DM cache	Update non temporal data detection Unit and LRU counters in FA / NT bit DM.	Update LRU counters and temporal bits	N bits for NT in DM + non temporal data detection unit + N $\log_2$ N bits for FA and associated circuitry for all these.	N bits for NT in DM + non temporal data detection unit + N $\log_2$ N bits for FA	NIL	L1 DC
SOFTWARE ASSISTED LRU	Up by 36%	36% faster (Improvement in cycles is 14.37%)	Update LRU counters, kill bit and keep bit	Update LRU counters, kill bit and keep bit	N $\log_2$ N + 2N bits per set and associated circuitry for Kill, Conditional Kill, flexible Keep and fixed Keep	N $\log_2$ N + 2N bits per set	Kill, Conditional Kill, flexible Keep and fixed Keep instructions	L1 DC & IC
EMLRU	Up by 21%	16% faster	Update LRU counters and temporal bits	Update LRU counters and temporal bits	N $\log_2$ N + N bits per set and its associated circuitry	N $\log_2$ N + N bits per set	Additional cache hint instructions to transfer nt hint	L1 DC & IC / L2

LLRU	Up by 30%	Best case 30% faster	Update LRU counters	Update LRU counters and shared bit	N $\log_2$ N + N bits per set and associated circuitry	N $\log_2$ N + N bits per set	Additional instruction to set and clear shared bit	L1 DC
SCLRU	Up by 24%	Best case 24% faster	Update LRU counters, AI tags and some state bits	Update LRU counters, AI tags, HLE, HMRU, SD, LRUV and MHT bits	[N*(16 bit AI + 1 bit HLE+ 1 bit HMRU)] + [28 bit SD + 16 bit AI + 1 bit LRUV] and associated circuitry	[N*(16 bit AI + 1 bit HLE+ 1 bit HMRU)] + [28 bit SD + 16 bit AI + 1 bit LRUV]	NIL	L2
LRU-SEQ	Varies from -0.4% to +0.05%	0.36%	Update LRU counters and Pway	Update LRU counters, Pway and Pline	Reduction of energy by 23% w.r.t. LRU	N $\log_2$ N + 2 $\log_2$ N bits per set	NIL	L1 DC & IC
SFLRU	Up to 6.3% (DC) Up to 9.3% (IC)	6.3% (DC) and 9.3% (IC) faster	Update LRU & LFU registers	Update LRU & LFU registers	N $\log_2$ N + N $\log_2$ X bits per set and associated logic	N $\log_2$ N + N $\log_2$ X bits per set	NIL	L1 DC & IC
IGDR	Up to 46.1% (19.8% avg), 16 cycles cache miss time	48.9% (12.9% avg) faster	Stores IRG of the cache block and updates the block information	Replacement occurs in main directory (and in ghost directory if block does not exist), and update all the counters (CL, LA, RC and SC)	Main directory, ghost directory, all the counters and the associated circuitry	42.5KB for 512KB cache	NIL	L2

DTTM → Data Transfer Time from Main Memory, N → Number of Ways, X → Maximum count value,

K → Number of blocks in Main Memory / Number of sets in Cache, CL → Current Memory Block Class

LA → Virtual Time of the Last Reference, RC → Reference Count, SC → Number of Consecutive References

DC → Data Cache, IC → Instruction Cache, DC&IC → Applicable to Data and Instruction Cache

Annexure B – IMPLEMENTATION OF SCHEDULING ALGORITHMS IN RTLINUX

This section explains in detail about the modifications done in RTLinux in order to modularize the scheduler and implement various priority based real-time scheduling algorithms like EDF, RM, EDFRCS and RMRCS.

B.1. MODULARIZING THE SCHEDULER

The RTLinux scheduler implements both the dispatching and scheduling algorithms in the same function which makes the modifications hard. In order to implement and test various scheduling algorithms effectively under the same framework, the scheduler needs to be modularized. In this work, modularization of the scheduler is done by abstracting away the scheduling algorithm from the dispatching function written in *rtl_schedule()*. The default *rtl_schedule()* function consists of following steps:

1. Cycle through all tasks and add pending signal bits to the tasks.
2. Find the new task to be scheduled.
3. Find a preemptor for the new task.
4. If scheduler is running in one-shot mode, set the timer.
5. Dispatch the new task (and handle its signals as well).

We implemented the scheduler as two functions - *find_new_task()* and *find_preemptor()* in a file named rtlinux/schedulers/rtl_sched_fixp.c. This abstraction provides the flexibility of implementing any new scheduling algorithms without affecting dispatcher.

For the EDF implementation, an additional field named *deadline* representing deadline of the task is added in *rtl_thread_struct* data structure. For the RCS algorithms, the fields *remaining_time* and *execution_time* are added in *rtl_thread_struct* data structure. With the addition of *remaining_time*, the scheduler has the additional responsibility of updating it. This is done by adding a field *came_to_cpu_time* in *rtl_thread_struct* data structure. When the *rtl_schedule( )*function is about to return, it updates the *came_to_cpu_time* of the thread going to be scheduled.

B.2. IMPLEMENTATION OF PRIORITY-BASED ALGORITHMS IN RTLINUX

Two priority based schedulers namely EDF and RM were implemented during the course of this work. The default RTLinux scheduler is a fixed priority scheduler, in which the scheduling decision is based on *sched_param.sched_priority*.

In priority based schedulers, deadline field of the *rtl_thread_struct* is modified by *rtl_schedule()* function. Whenever a new job arrives in the system (detected by *now >= t→resume_time*), the *rtl_schedule()* function sets the deadline to *resume_time + period*. This function also checks whether the current time (now) is greater than the deadline. For all the tasks whose current time is greater than deadline, the function checks for existence of any task with non-zero *remaining_time*. A Non-zero *remaining_time* when the current time is greater than *deadline* concludes that the task missed its deadline and requires debugging of scheduling logic.

B.2.1. Implementation of EDF in RTLinux

EDF scheduler selects the tasks based on deadlines i.e. lower the deadline higher the priority. The file *rtl_sched_edf.c* implements functions *find_preemptor()* (Figure B.1) and *find_new_task()* (Figure B.2). In the case of EDF, the find_new_task() function takes decisions based on deadlines. If no real time thread is found in the system, then it schedules the Linux thread or any other aperiodic task (based on priority). The source files rtl_sched_edf.c and rtl_sched.c are compiled to create rtl_sched_edf.o.

B.2.2. Implementation of RM in RTLinux

Rate monotonic scheduler schedules the tasks based on periods i.e. lower the period higher the priority. Figures B.1 and B.2 show the functions *find_preemptor()* and *find_new_task()* which are used for finding the smallest period real-time task if there exist any. If no such task is found, it schedules the Linux thread or any other aperiodic task (based on priority). The source files *rtl_sched_rm.c* and *rtl_sched.c* are compiled to create *rtl_sched_rm.o*.

```
/*Function to find preemptor for EDF / RM scheduler*/
struct rtl_thread_struct* find_preemptor(struct rtl_thread_struct *new_task)
{
    struct rtl_thread_struct *preemptor = 0;
    struct rtl_thread_struct *preemptor1 = 0;
    struct rtl_thread_struct *t = 0;
    schedule_t* sched=&rtl_sched[rtl_getcpuid()];
    hrtime_t now=sched->clock->gethrtime(sched->clock);
    for(t = sched->rtl_tasks; t; t = t->next)
    {
            /**** The Comparison if EDF Scheduler ****/
            if((!preemptor1 || (t->deadline < preemptor1->deadline)) &&
                t!=new_task)      preemptor1=t;

            /****The Comparison if RM Scheduler
            if((!preemptor1 || (t->resume_time < preemptor1->resume_time)) &&
                t!=new_task)      preemptor1=t;
            ****/

            if(now < t->resume_time)
                    if(!preemptor ||
                        ((t->sched_param.sched_priority >= preemptor->
                                sched_param.sched_priority) &&
                                (t->resume_time < preemptor->resume_time)) ) {
                            preemptor=t;
                    }
    }
    if(preemptor1)
            return preemptor1;
    return preemptor;
}
```

Fig. B.1: *find_preemptor()* function for EDF and RM Schedulers

```
  /* Scheduling algorithm specific task comparison (deadline / period) */
int task_compare_sched(struct rtl_thread_struct *t, struct rtl_thread_struct * new_task)
{       schedule_t *s=LOCAL_SCHED;
        if(t= =&(s->rtl_linux_task)) return 0;
        else if (new_task ==&(s->rtl_linux_task)) return 1;
                return (t->deadline < new_task->deadline); // For EDF Scheduler
                //return (t->period < new_task ->period);     // For RM Scheduler
}
/* Task comparison based on priority */
int task_compare(struct rtl_thread_struct *t1, struct rtl_thread_struct *t2)
{
 return (t1->sched_param.sched_priority > t2->sched_param.sched_priority);
}
/* This function finds out the lowest priority (deadline / period task */
struct rtl_thread_struct* find_new_task(struct rtl_thread_struct *linux_thread)
{       struct rtl_thread_struct *new_task = 0;
        struct rtl_thread_struct *new_task1 = 0;
        struct rtl_thread_struct *t = 0;
        int cpu_id=rtl_getcpuid();
        schedule_t* sched=&rtl_sched[cpu_id];
        for(t = sched->rtl_tasks; t; t = t->next)
        {       if(t==linux_thread)
                        continue;
                if(t->deadline>0) {                             // For EDF Scheduler
                //if(t->period>0) {                             // For RM Scheduler
                        if ((t->pending & ~t->blocked) && (!new_task1 ||
                            task_compare_sched(t,new_task1))) {
                                new_task1 = t;
                        }
                }
                if ((t->pending & ~t->blocked) && (!new_task ||
task_compare(t,new_task)))   new_task = t;
        }
        if(new_task1)
        {
                new_task=new_task1;
        }
        if(!new_task && (~linux_thread->blocked))
                new_task=linux_thread;
        return new_task;
}
```

Fig. B.2: *find_new_task()* function for EDF and RM Schedulers

B.3. IMPLEMENTATION OF ENERGY EFFICIENCY PRIORITY-BASED REAL-TIME SCHEDULERS

As discussed in section 6.5, the RCSS algorithm is used to reduce the number of preemptions in a schedule. For the RCSS implementation, we added *remaining_time* and *execution_time* fields to the *rtl_thread_struct* data structure.

The RCS algorithm discussed in section 6.5 is coded as a separate function in *rtl_sched.c* file, and is called by *rtl_schedule()* function. This function finds the maximum possible extension time up to which currently running task can continue its execution without any deadline miss. If the possible extension time is greater than zero then set the *future preempt_time* of the currently running task accordingly and allow it to continue. The *rtl_sched.c* is renamed to *rtl_sched_rcs.c*.

B.3.1. Implementation of EDFRCS in RTLinux

The implementation of EDFRCS is almost similar to EDF. The only difference is the addition of *find_permissible_extension_time( )* function (Figure B.3) in *rtl_sched.c*. The source file of the new scheduler is named as *rtl_sched_rcs.c*. All the common implementations (with EDF) are accessed from *rtl_sched_edf.c*. The source files *rtl_sched_edf.c* and *rtl_sched_rcs.c* are compiled to create *rtl_sched_edf_rcs.o*.

B.3.2. Implementation of RMRCS in RTLinux

The implementation of RMRCS is almost similar to RM. Like in EDFRCS, the only difference is the addition of *find_permissible_extension_time( )* function (Figure B.3) in *rtl_sched.c*. The source file of the new scheduler is named as rtl_sched_rcs.c. All the common implementations (with RM) are accessed from *rtl_sched_rm.c*. The source files *rtl_sched_rm.c* and *rtl_sched_rcs.c* are compiled to create *rtl_sched_rm_rcs.o*.

Figure B.4 shows the skeleton of modified *rtl_schedule()* with call to *find_permissible_time()*, i.e., reduced context switch algorithm included. In addition to all these, a new function call *pthread_set_worst_exec_time( )* is added to the system. Through this function call a module can set worst case execution times.

```
hrtime_t find_permissible_extension_time()
{
/*we have to run a loop for each task which is of higher priority than s->rtl_current_task,each loop will again have a loop to calculate the time which this higher priority task can donate to currently running task, depending upon the time which is supposed to be snatched from it due to even more higher priority tasks*/

        pthread_t t1,t2;
        schedule_t * s=LOCAL_SCHED;
        hrtime_t permissible_extension=HRTIME_INFINITY,slack,snatch;
        hrtime_t now=clock_gethrtime(CLOCK_REALTIME);
        /*we have to run outer loop for every task which is runnable and is of higher priority than current task*/
        for(t1=s->rtl_tasks;t1;t1=t1->next)
                {
                if(t1==s->rtl_current) continue;
                if( ! (  (t1->pending & ~t1->blocked ) && task_compare_sched(t1,s->rtl_current)  )  ) continue;

                slack=t1->deadline-now-t1->remaining_time;
        //      if(slack < 50000000 ) rtl_printf("\ntask %x: slack=%lld: deadline=%lld , now=%lld , remaining time=%lld",t1,slack,t1->deadline,now,t1->remaining_time);
                snatch=0;
                for(t2=s->rtl_tasks;t2;t2=t2->next)
                        {
                        if(t2==s->rtl_current || t2==t1) continue;
                        if(!( (t2->pending & ~t2->blocked ) && task_compare_sched(t2,t1) ) )  continue;
                        snatch+=t2->remaining_time + ceiling((t1->deadline - t2->deadline),t2->period )*t2->execution_time;
                        }
                 permissible_extension=min(permissible_extension,slack-snatch);
                }

        if(permissible_extension < 0 )          return 0;
        else return permissible_extension;
}
```

Fig. B.3: find_possible_extension_time()

First of all we define certain variables of type schedule_t and rtl_thread_struct.
Disable interrupts.
Then, we get time by calling gethrtime() function.
If current task is real time,
set remaining_time=remaining_time – (now- came_to_cpu_time)
If CONFIG_PC_TIMERS
Then,Check if timer has expired for each task by (now>=t->expires.it_value) ? 1 : 0;
If expired then, Send notification by
signal_generation(t->owner,t->signal.sigev_signo);
And update timer calling UPDATE_TIMER() function:
This function checks if the timer is periodic, if yes, increase timer by hr_interval value else DISARM_TIMER().
Now, for each task:
If it's deadline has been missed, report it.
Test if RTL_THREAD_TIMERARMED is set for this task.
If yes and now > expiry time, then expire the timer and find preemptor.
Set deadline=expiry time+period
Set remaining time as worst case execution time
Add a pending signal.
Check if preemptor has earlier deadline than this task, and set it appropriately.
Find the new task as highest-priority-runnable task by calling find_new_task ( ) function
If new task is not equal to current task and current task has time remaining according to it's already set worst case execution_time, then find permissible extension.
If(extension_time > 0) set new task as current task
Else { Find preemptor for new task
/*new task is the task foubd by the find_new_task( ) function call*/
If preemptor found , Set preempt_time=preemptor's next arrival time
/*which is represented by resume_time*/
}
If no new task was found, disable interrupts, and goto_idle.
If extension time was permitted, set preempt time as now plus extension time
If CONFIG_OC_PTIMERS,
For each timer in timer list,
Recalculate preempt-time, if there is a process with sched_priority greater than new-tasks and it is ARMED.
/*Now, we have the new task selected. */
If we have a preemptor set the timer to the preemptors expire time otherwise set it to the virtual linux timer interrupt viz. (HRTICKS_PER_SEC/HZ)/2.
If current task is same as new task, then do nothing but return.
If the new task is real time, set appropriate bits to make rt system active/inactive.
Then switch the context
Delay the switching for FPU (if support in kernel) by first checking if the new process needs the fpu and doesn't have it, call rtl_fpu_save for previous process and rtl_fpu_restore for current.
Find mask as pthread_self()->pending and restore interrupts.
Call do_signal() for any pending signals of new process.
If defined CONFIG_OC_PSIGNALS
Signal handlers are non re-entrant, i.e. first finish current handler execution and then manage the rest of pending signals.
Check if a handler execution is in progress and some are pending, do nothing.
Check if some signals are pending and the thread is blocked , call do_user_signal() for it.
If RTL_SIGNAL_READY is not a member of pending signals, goto idle.
Set current task's "came to cpu time" as current time
Call trace and return mask.

Fig. B.4: rtl_schedule()

B.4. DYNAMIC LOADING OF SCHEDULER

We implemented four dynamic priority scheduling algorithms, namely EDF, EDFRCS, RM, and RMRCS as modules in RTLinux. One of the major issues in testing these algorithms is the overhead of compiling the kernel again and again in order to insert the appropriate scheduling algorithm. The required scheduling module has to be inserted into the kernel while loading RTLinux. The simplest way to resolve this issue is to compile all the scheduler implementations in advance and choose the required scheduler at the time of RTLinux startup. The selection of a scheduler can be achieved with the help of a configuration file.

RTLinux boots up when user executes *rtlinux start* in the command prompt, which fires a shell script named *rtlinux*. This shell script scans the /usr/rtlinux-3.2/modules directory and acquires the module list and loads them. User modules can be loaded later (manually). At startup, the shell script reads the configuration file provided by the user to insert the appropriate scheduler module into the kernel.

B.5. VERIFICATION OF SCHEDULING ALGORITHM IMPLEMENTATION

The kernel module creates a periodic thread which executes an infinite loop i.e., it keeps on iterating until the kernel module is unloaded. The C code of the task in RTLinux is given below. The kernel module serves as the task and one iteration of the while loop simulates one job of the periodic task.

```
while (1) {
        for(i=0;i<1000000;i++)
            clock_gethrtime(CLOCK_REALTIME);
        jobno++;
        pthread_wait_np ();
}
```

If the period of the real-time task is very small, then the real-time task will occupy the CPU for almost all the time which results in starvation of general-Linux task. The priority scheduling algorithms like EDF and RM can be verified with the help of the above mentioned C program along with the addition of *deadline* field in *rtl_thread_struct* data

structure. The verification of EDFRCS and RMRCS can be carried out with the help of C program along with the addition of *deadline*, *remaining_time*, *execution_time* and *came_to_CPU* fields in *rtl_thread_struct* data structure. The remaining execution time of the thread is managed with the help of *came_to_CPU* field. The worst case execution time of the thread has to be pre-declared with *pthread_set_worst_case_exec_time( )* function before making the thread into a periodic thread.

B.5.1. Verification of Implementation

Table B.1 shows the task set used for experimentally evaluating the RTLinux scheduler with EDF, EDFRCS, RM and RMRCS as scheduling algorithms. The task set in Table B.1 has 3 periodic tasks. This experimentation uses three modules (one per task) T1, T2 and T3. These tasks were designed with the assumption that deadline of the task is equivalent to its period. The utilization of the tasks is 12.5%, 3.33% and 60% for T1, T2 and T3 respectively. In literature and in previous discussions we assumed that all tasks in the task set are inphase and running the experimentation till hyper-period will guarantee a valid schedule. For the RTLinux scheduler evaluation these assumptions does not hold true as all tasks cannot be released at the same time because of the uniprocessor constraint. Thus in this experimentation, we took the schedule of the first 40 seconds (hyper-period = 30 seconds) for analysis.

Table B.1: Test case with three tasks

Task	Arrival time (sec)	Period (sec)	Worst case Execution time (sec)	Deadline (sec)
T1	0	2	0.25	2
T2	0	3	0.1	3
T2	0	5	3	5

For the given task set (Table B.1) with 75.83% utilization, EDF and RM produced valid schedules with 10 preemptions where as EDFRCS and RMRCS produced valid schedules with 3 preemptions. This shows the efficiency of RCS algorithms in reducing preemptions.

REFERENCES

[Acquaviva 2003] A. Acquaviva, L. Benini and B. Ricco, "Energy Characterization of Embedded Real-time Operating Systems", in Proceedings of the Workshop on Compilers and Operating Systems for Low Power, pp. 53 – 73, December 2003.

[Agarwal 1988] A. Agarwal, J. Hennesy and M. Horowits, "Cache Performance of Operating Systems and Multiprogramming", ACM Transactions on Computer Systems, vol. 6, issue 4, pp. 393 – 431, 1988.

[Agarwal 1993] A. Agarwal and S. D. Pudar, "Column Associative Caches: A Technique for Reducing the Miss Rate of Direct mapped Caches", in Proceedings of 35th Annual International Symposium on Computer Architecture, pp. 179 – 190, May 1993.

[Aguilar 2004] J. Aguilar and E. L. Leiss, "An Adaptive Coherence – Replacement Protocol for Web Proxy Cache Systems", Communication and Systems, vol. 8, issue 1, pp. 1 – 14, 2004.

[Al-Zoubi 2004] H. Al-Zoubi, A. Milenkovic and M. Milenkovic, "Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite", in Proceedings of the 42nd Annual Southeast regional conference, pp. 267 – 272, April 2004.

[Albonesi 1999] D. H. Albonesi, "Selective Cache Ways: On-demand Cache Resource Allocation", in Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture, pp. 248 – 259, November 1999.

[Alghazo 2004] J. Alghazo, A. Akaaboune and N. Botros, "SF-LRU Cache Replacement Algorithm", In Proceedings of the Records of the 2004 International Workshop on Memory Technology, Design and Testing, pp. 19 – 24, August 2004.

[Aly 2003] R. E. Aly, B. R. Nallamilli and M. A. Bayoumi, "Variable-way Set Associative Cache Design for Embedded System Applications", in Proceedings of the 46th IEEE International Midwest Symposium on Circuits and Systems, pp. 1435 – 1438, December 2003.

[Aydin 2004] H. Aydin, D. Mosse, P. Mejı´a-Alvarez, “Power-Aware Scheduling for Periodic Real-Time Tasks”, IEEE Transactions on Computers, vol. 53, issue 5, pp. 584 – 600, 2004.

[Bajwa 1997] R. S. Bajwa, M. Hiraki, H. Kojima, D. J. Gorny, K. Nitta, A. Shridhar, K. Seki and K. Sasaki, “Instruction Buffering to Reduce Power in Processors for Signal Processing”, IEEE Transactions on Very Large Scale Integrated System, vol. 5, issue 4, pp. 417–424, 1997.

[Bannon 1995] P. Bannon and J. Keller, “Internal Architecture of Alpha 21164 microprocessor”, in Proceedings of the 40th IEEE Computer Society International Conference, pp. 79 – 87, March 1995.

[Basumallick 1994] S. Basumallick and K. Nilsen, “Cache Issues in Real-Time Systems”, in First ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems, June 1994.

[Batson 2001] B. Batson and T. N. Vijaykumar, “Reactive Associative Caches”, in Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques, pp. 49 – 60, September 2001.

[Bechade 1994] R. Bechade, R. Flaker, B. Kauffmann, S. Kenyon, C. London, S. Mahin, K. Nguyan, D. Pham, A. Roberts, S. Ventrone and T. Vonreyn, “A 32b 66 MHz 1.8 W Microprocessor”, in Proceedings of the International Solid-State Circuits Conference, pp. 208 –209, February 1994.

[Bellas 1999] N. Bellas, I. Hajj and C. Polychronopoulos, “Using Dynamic Cache Management Techniques to Reduce Energy in a High-Performance Processor”, in Proceedings of the 1999 international Symposium on Low Power Electronics and Design, pp. 64 – 69, August 1999.

[Benini 2000] L. Benini and G. D. Micheli, “System-Level Power Optimization: Techniques and Tools”, ACM Transactions on Design Automation of Electronic Systems, vol. 5, issue 2, pp. 115–192, April 2000.

[Benini 2003] L. Benini, A. Macii and M. Poncino, “Energy-Aware Design of Embedded Memories: A Survey of Technologies, Architectures, and Optimization Techniques”, ACM Transactions on Embedded Computing Systems, vol. 2, issue 1, pp. 5–32, February 2003.

[Bunda 1994] J. Bunda, W.C. Athas and D. Fussell, “Evaluating Power Implication of CMOS Microprocessor Design Decisions”, in Proceedings of the 1994 International Workshop on Low Power Design, pp. 147 – 152, April 1994.

[Burger 1997] D. Burger and T. M. Austin, “The Simplescalar Tool Set, Version 2.0.”, Technical report, University of Wisconsin, Madison, Computer Science Department, 1997.

[Buttazzo 2005] G. C. Buttazzo, “Rate Monotonic vs. EDF: judgement day”, Real Time Systems, Kluwer Academic Publishers, vol. 29, issue 1, pp. 5 – 26, January 2005.

[Calder 1996] B. Calder, D. Grunwald and J. Emer, “Predictive Sequential Associative Cache”, in Proceedings of the 2nd International Symposium on High-Performance Computer Architecture, pp. 244–253, February 1996.

[Cervin 2003] A. Cervin, “Integrated Control and Real-time Scheduling”, Doctoral Dissertation, ISRN LUTFD2/TFRT-1065- SE, Department of Automatic Control, Lund, 2003.

[Chandrakasan 1995] A. P. Chandrakasan and R. W. Brodersen, “Low Power Digital CMOS Design”, Kluwer Academic Publishers, USA, 1995.

[Chang 1987] J. H. Chang, H. Chao and K. So, “Cache Design of a Sub Micron CMOS System/370”, in Proceedings of the 14th Annual International Symposium on Computer Architecture, pp. 208 – 213, June 1987.

[Chang 2004] Y. Chang, C-L. Yang and F. Lai, “Value-Conscious Cache: Simple Technique for Reducing Cache Access Power”, in Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, pp. 16 – 21, February 2004.

[Chen 1995] T. F. Chen and J. L. Baer, “Effective Hardware-Based Data Prefetching for High Performance Processors”, IEEE Transactions on Computers, vol. 44, issue 5, pp. 609 – 623, May 1995.

[Collerell 2002a] S. Collerell and F. Vahid, “Tuning of Loop Cache Architectures to Programs in Embedded System Design”, in Proceedings of the 15th International symposium on System Synthesis, pp. 8 – 13, October 2002.

[Collerell 2002b] S. Collerell and F. Vahid, “Synthesis of Customized Loop Caches for Core-Based Embedded System”, in Proceedings of the 2002 IEEE/ACM international Conference on Computer-aided design, pp. 655 – 662, November 2002.

[Cyrix 1998] "Cyrix Cyrix 6X86MX Processor.", 1998.

[Cyrix 1999] "Cyrix. Cyrix MII Databook.", 1999.

[Dertouzos 1974] M. L. Dertouzos, “Control Robotics: the Procedural Control of Physical Processes,” Information Processing 74, North-Holland Publishing Company, pp. 807 – 813, 1974.

[Deville 1992] Y. Deville and J. Gobert, “A Class of Replacement Policies for Medium and High-Associativity Structures”, ACM SIGARCH Computer Architecture News, vol. 20, issue 1, pp. 55-64, March 1992.

[Dick 2000] R. P. Dick, G. Lakshminarayana, A. Raghunathan and N. K. Jha, “Power Analysis of Embedded Operating Systems”, in Proceedings of the 37th Annual ACM/ IEEE Design Automation Conference, pp. 312 – 315, June 2000.

[Dittman 2004] B. Dittman, “Strategied for Minimizing Context Switch Times in Large Register set Environment with Primary Focus on the PowerPC Architecture with Floating Point and AltiVec Extensions”, Quadros Systems. http://www.rtxc.com/pdf/article_esd-conference_05-08-2004.pdf, August 2004.

[Dropsho 2002] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magkis and M. L. Scott, “Integrating Adaptive On-chip Storage Structures for Reduced Dynamic Power”, in Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques, pp. 141 – 152, 2002.

[Dudani 2002] A. Dudani, F. Mueller and Y. Zhu, “Energy Conserving Feedback EDF Scheduling for Embedded Systems with Real-Time Constraints”, in Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems, vol. 37, issue 7, pp. 213 – 222, June 2002.

[Dulong 1998] C. Dulong, “The IA-64 Architecture at Work”, IEEE Transactions on Computers, vol. 31, issue 7, pp. 24 – 32, July 1998.

[Efthymiou 2002] A. Efthymiou and J. D. Garside, “An Adaptive Serial-parallel CAM Architecture for Low Power Cache Blocks”, in Proceedings of the International Symposium on Low Power Electronics and Design, pp. 136 – 141, August 2002.

[Gonzalez 1996] R. Gonzalez and M. Horowitz, “Energy Dissipation in General Purpose Microprocessors”. IEEE Journal of Solid-State Circuits, vol. 31, issue 9, pp. 1277–1284, 1996.

[Gooch 1998] Richard Gooch, “Linux Scheduler Benchmark Results”, http://www.atnf.csiro.au/people/rgooch/benchmarks/linux-scheduler.html , September, 1998.

[Gopalakrishnan 1996] R. Gopalakrishnan and G. M. Parulkar, “Bringing Real-time Scheduling Theory and Practice Closer for Multimedia Computing”, ACMSIGMETRICS Performance Evaluation Review, vol. 24, issue 1, pp. 1-12, 1996.

[Gruian 2001] F. Gruian, “Hard Real-time Scheduling for Low Energy using Stochastic Data and DVS Processors”, in Proceedings of the International Symposium on Low-Power Electronics and Design, pp. 46 – 51, August 2001.

[Gupta 1990] R. Gupta and C. Chi, “Improving Instruction Cache Behavior by Reducing Cache Pollution”, in Proceedings of the 1999 ACM/IEEE Conference on Supercomputing, pp. 82 – 91, November 1990.

[Hallnor 2004] E. G. Hallnor and S. K. Reinhardt, “A Compressed Memory Hierarchy using an Indirect Index Cache” in Proceedings of the 3rd workshop on Memory Performance issues: in Conjunction with the 31st International Symposium on Computer Architecture, pp. 9 - 15 , 2004.

[Hasegawa 1995] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki and P. Biswas, “Sh3: High Code Density, Low Power”, IEEE Micro, vol. 15, issue 6, pp. 11 – 19, 1995.

[Hennessy 2007] J. L. Hennessy and D. A. Patterson, “Computer Architecture: A Quantitative Approach”, 4th Edition, Morgan-Kaufmann Publishing Co., 2007.

[Hildebrandt 1999] J. Hildebrandt, F. Golatowski and D. Timmermann, “Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in Hard Real-Time Systems”, in Proceedings of 11th Euromicro Conference on Real-Time Systems, pp. 208 – 215, June 1999.

[Hong 1997] I. Hong and M. M. Potkonjak, “Power Optimization Using Divide-and-Conquer Techniques for Minimization of the Number of Operations”, in Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, pp. 108-113, November 1997.

[Huang 2001] M. Huang, J. Renau, S. Yoo and J. Terrellas, “L1 Data Cache Decomposition for Energy Efficiency”, in Proceedings of the International Symposium on Low Power Electronics and Design, pp. 10 – 15, August 2001.

[Inoue 1999] K. Inoue, T. Ishihara and K. Murakami, “Way-predicting Set-Associative Cache for High Performance and Low Energy Consumption”, in Proceedings of the 1999 International Symposium on Low Power Electronics and Design, pp. 273–275, August 1999.

[Inoue 2001] K. Inoue, “High-Performance Low-Power Cache Memory Architectures”, Ph.D. Thesis, Kyushu University, January 2001.

[Inoue 2002] K. Inoue, V.G. Moshnyaga and K. Murakami, “A History-Based I-Cache for Low-Energy Multimedia Applications” in Proceedings of the 2002 International Symposium on Low Power Electronics and Design, pp. 148 – 153, August, 2002.

[Ishihara 2005] T. Ishihara and F. Fallah, “A Non-Uniform Cache Architecture for Low Power System Design”, in Proceedings of the 2002 International Symposium on Low Power Electronics and Design, pp. 363 – 368, August 2005.

[Jain 2001] P. Jain, S. Devadas, D. Engels and L. Rudolph, “Software-Assisted Cache Replacement Mechanisms for Embedded Systems”, in Proceedings of the International Conference on Computer-Aided Design, pp. 119-126, November 2001.

[Jeong 1999] J. Jeong and M. Duhois, “Optimal Replacements in Caches with Two Miss Costs”, in Proceedings of the 11th Annual ACM symposium on Parallel Algorithms and Architectures, pp. 155 – 164, June 1999.

[Jianli 2005] Z. Jianli and C. Chaitali, “System-Level Energy-Efficient Dynamic Task Scheduling”, in Proceedings of the 42nd Annual Conference on Design Automation, pp. 628 – 631, June 2005.

[Jouppi 1990] N. P. Jouppi, “Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers”, in Proceedings of the 17th Annual International Symposium on Computer Architecture, vol. 18, issue 3, pp. 364 – 373, May 1990.

[Ju 2007] L. Ju, S. Chakraborty and A. Roychoudhury, “Accounting for Cache-Related Preemption Delay in Dynamic Priority Schedulability Analysis”, in Proceedings of the Conference on Design, Automation and Test in Europe, pp. 1623-1628, April 2007.

[Juan 1996] T. Juan, T. Lang and J. J. Navarro, “The Difference-bit Cache”, ACM SIGARCH Computer Architecture News, vol. 24, issue 2, pp. 11 – 120, 1996.

[Kalla 2003] P. Kalla, X. S. Hu and J. Henkel, “LRU-SEQ: A Novel Replacement Policy for Transition Energy Reduction in Instruction Caches”, in Proceedings of the 2003 IEEE/ACM International Conference on Computer-aided design, pp. 518 – 522, November 2003.

[Kampe 2004] Martin Kampe, Per Stenstrom and Michel Dubois, “Self-Correcting LRU Replacement Policies”, in Proceedings of the 1st Conference on Computing Frontiers, pp. 181 – 191, April 2004.

[Kandemir 2003] M. Kandemir, G. Chen, W. Zhang and I. Kolcu, “Data Space Oriented Scheduling in Embedded Systems”, in Proceedings of the Conference on Design, Automation and Test in Europe – Volume 1, pp. 10416 - 10421, March 2003.

[Kessler 1998] R. Kessler, "The Alpha 21264 Microprocessor: Out-Of-Order Execution at 600 Mhz", http://www.hotchips.org/archives/hc10/2_Mon/HC10.S1/HC10.1.1.pdf, Hot Chips, 1998.

[Kessler 1999] R. E. Kessler, E. J. McLellan and D.A. Webb, “The Alpha 21264 microprocessor architecture”, Technical report, http://www.compaq.com/AlphaServer/download/ev6chip.pdf, November 1999.

[Kin 1997] J. Kin, M. Gupta and W. H. Mangione-Smith, “The Filter Cache: An Energy Efficient Memory Structure”, in Proceedings of 30th Annual International Symposium on Microarchitecture, pp. 184 – 193, December 1997.

[Kirk 1989] D.B. Kirk, “SMART (Strategic Memory Allocation for Real-Time) Cache Design”, in Proceedings of the 10th Real-Time Systems Symposium, pp. 229-237, December 1989.

[Krishna 2000] C. M. Krishna and Y. H. Lee, “Voltage-clock-scaling techniques for low power in hard real-time systems”, in Proceedings of the IEEE Real-Time Technology and Applications Symposium, pp. 156-165, May 2000.

[Lai 2001] A. Lai, C. Fide and B. Falsafi, “Dead-block prediction & dead-block correlating prefetchers”, ACM SIGARCH Computer Architecture News, vol. 29, issue 2, pp. 144 – 154, 2001.

[Leback 2000] A. Lebeck, X. Fan, H. Zeng and C. Ellis, “Power Aware Page Allocation”, ACM SIGOPS Operating Systems Review, vol. 34, issue 5, pp. 105–116, 2000.

[Lee 1998] C-G. Lee, J. Hahn, Y-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee and C. S. Kim, “Analysis of cache-related preemption delay in fixed-priority preemptive scheduling”, in Proceedings of IEEE Transactions on Computers, pp. 700–713, June 1998.

[Lee 1999] S. Lee, S. L. Min, C. S. Kim, C. G. Lee and M. Lee, “Cache-Conscious Limited Preemptive Scheduling”, Real-Time Systems, vol. 17, issue 2-3, pp. 257–282, 1999.

[Lee 2000] C. Lee, J. K. Lee and T. Hwang, “Compiler Optimization on Instruction Scheduling for Low-Power”, in Proceedings of the 13th International Symposium on System Synthesis, pp. 55-60, September 2000.

[Lee 2001a] D. Lee, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho and C. S. Kim, “LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies”, IEEE Transactions on Computers, vol. 50, issue 12, pp. 1352-1360, December 2001.

[Lee 2001b] C-G Lee, Y-M Seo, S. L. Min, S. Hong and C. S. Kim, “Bounding Cache-Related Preemption Delay for Real-Time Systems”, IEEE Transactions on Software Engineering, vol. 27, issue 9, pp. 805 – 826, September 2001.

[Liedtke 1997] J. Liedtke, H. HaÈrtig and M. Hohmuth, “OS-Controlled Cache Predictability for Real-Time Systems”, in Proceedings of 3rd Real-Time Technology and Applications Symposium, pp. 213 – 227, June 1997.

[Liu 1973] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in Hard Real- Time Environment”, Journal of ACM (JACM), vol. 20, issue 1, pp. 46 – 61, January 1973.

[Liu 2000] J. W. S. Liu, “Real Time Systems”, ISBN 9780130996510, Prentice Hall, March 2000.

[Luculli 1997] G. Luculli and M. D. Natale, “A Cache-Aware Scheduling Algorithm for Embedded Systems”, in Proceedings of the 18th IEEE Real- Time Systems Symposium, pp. 199 – 209, December 1997.

[Lyon 2002] T. Lyon, E. Delano, C. McNairy and D. Mulla, “Data Cache Design Considerations for the Itanium2 Processor”, in Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 356 – 363, September 2002.

[Ma 2001] A. Ma, M. Zhang and K. Asanovi´c, “Way Memoization to Reduce Fetch Energy in Instruction Caches”, in 28th ISCA Workshop on Complexity Effective Design, Gothenburg, June 2001.

[Maki 1999] N. Maki, K. Hoson and A. Ishida, “A Data-Replace-Controlled Cache Memory System and its Performance Evaluations”, in Proceedings of the IEEE Region 10 Conference, pp. 471 – 474, September 1999.

[Mamidipaka 2004] M. Mamidipaka and N. Dutt, “eCACTI: An Enhanced Power Estimation Model for On-chip Caches”, Technical report, Center for Embedded Computer Systems, Donald Dren School of Information and Computer Science, University of California, Irvine, http://www.cecs.uci.edu/technical_report/TR04-28.pdf, September 2004.

[Marti 2002] P. Marti, G. Fohler, K. Ramamritham and J. M. Fuertes, “Control Performance of Flexible Timing Constraints for Quality-of-Control Scheduling”, in Proceedings of the 23rd IEEE Real-Time System Symposium, pp. 91 – 102, December 2002.

[Mataix 1996] J.V. B. Mataix, J. J. S. Martin, R. Ors, P. Gil and A. Wellings, “Adding Instruction Cache Effect to Schedulability Analysis of Preemptive Real-Time Systems”, in Proceedings of the Second Real-Time Technology and Applications Symposium, pp. 204 – 213, June 1996.

[May 1994] C. May, E. Silha, R. Simpson and H. T. Warren, “The PowerPC Architecture: A Specification for a New Family of RISC Processors”, Morgan Kaufmann Publishers, Inc., 1994.

[Min 2004] R. Min, Z. Xu, Y. Hu and W. Jone, “Partial Tag Comparison: A New Technology for Power-Efficient Set-Associative Cache Designs”, in Proceedings of the 17th International Conference on VLSI Design, page 183 – 188, August 2004.

[Mok 1983] A. K. Mok, “Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment”, Ph.D.Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1983.

[Montanaro 1997] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, G. W. Hoeppner, D. Kruckemyer, T. H. Lee, P.C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stephany and S. C. Thierauf, “A 160MHz, 32-b, 0.5-W CMOS RISC microprocessor”, Digital Technical Journal, vol.9, issue 1, pp 49 - 62, 1997.

[Mudge 2000] T. Mudge, “Power: A First Class Design Constraint for Future Architectures”, in Proceedings of the International Conference on High Performance Computing, pp. 215 – 224, December 2000.

[Muller 1995] F. Muller, “Compiler Support for Software-based Cache Partitioning”, in Proceedings of the Second ACM SIGPLAN Workshop Languages, Compilers, and Tools for Real-Time Systems, pp. 125 – 133, June 1995.

[Negi 2003] H. Negi, T. Mitra and A. Roychoudhury, “Accurate Estimation of Cache-related Preemption Delay", in Proceedings of the 1st IEEE / ACM / IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 201-206, October 2003.

[O’Neil 1999] E. J. O’Neil, P. E. O’Neil and G. Weikum, “An Optimality Proof of the LRU-K Page Replacement Algorithm”, Journal of ACM, vol. 46, issue 1, pp. 92-112, 1999.

[Oh 1998] S. H. Oh and S. M. Yang, “A Modified Least-Laxity-First Scheduling Algorithm for Real-Time Tasks”, in Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications, pp. 31 – 36, October 1998.

[Panda 2001] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle and P. G. Kjeldsberg, “Data and Memory Optimization Techniques for Embedded Systems”, ACM Transactions on Design Automation of Electronic Systems, vol. 6, issue 2, pp. 149–206, April 2001.

[Panwar 1995] R. Panwar, and D. Rennels, “Reducing the frequency of Tag Compares for Low Power I-Cache”, in Proceedings of the 1995 International Symposium on Low Power Electronics and Design, pp. 57 – 63, April 1995.

[Park 2007] G-H. Park, K-W. Lee, T-D. Han and S-D. Kim, “Cooperative Cache System: A Low Power Cache System for Embedded Processors”, IEICE Transactions on Electronics, vol. E90–C, issue 4, pp. 708 – 717, April 2007.

[Patel 2006] K. Patel, L. Benini, E. Macii and M. Poncino, “Reducing Conflict Misses by Application-Specific Reconfigurable Indexing”, IEEE Transactions on Computer – Aided Design of Integrated Circuits and Systems, vol. 25, issue. 12, pp. 2626 – 2637, 2006.

[Pering 1998] T. Pering, T. D. Burd and R. W. Brodersen, “The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms”, in Proceedings of the 1998 International Symposium on Low Power Electronics and Design, pp. 76 – 81, August 1998.

[Pillai 2001] P. Pillai and K. G. Shin, “Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems”, in Proceedings of the 18th ACM Symposium on Operating Systems Principles, pp. 89 – 102, October 2001.

[Pouwelse 2000] J. Pouwelse, K. Langendoen and H. Sips, “Dynamic Voltage Scaling on a Low-Power Microprocessor”, UbiCom Technical Report 2000/3, Delft University of Technology, pp. 1 – 4, 2000.

[Powell 2001] M. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi and K. Roy, “Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping”, in Proceedings of the 34th International Symposium on Micro architecture, pp. 54 – 65, December 2001.

[Qureshi 2005] M. K. Qureshi, D. Thompson and Y. N. Patt, “The V-Way Cache: Demand-Based Associativity via Global Replacement”, in Proceedings of the 32nd International Symposium on Computer Architecture, pp. 544 – 555, June 2005.

[Ramaprasad 2006a] H. Ramaprasad and F. Mueller, “Bounding Preemption Delay within Data Cache Reference Patterns for Real-Time Tasks” in Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 71 – 80, April 2006.

[Ramaprasad 2006b] H. Ramaprasad and F. Mueller, “Tightening the Bounds on Feasible Preemption Points”, in Proceedings of the 27th IEEE International Real-Time Systems Symposium, pp. 212-224, December 2006.

[Raveendran 2006] B. Raveendran, S. Balasubramaniam, K. D. Prasad and S. Gurunarayanan, “Variants of Priority Scheduling Algorithms for Reduced Context Switches in Real Time System”, in Proceedings of the International Conference on Distributed Computing and Networking, pp. 466 – 478, December 2006.

[Rivers 1998] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson and M. Farrens, “Utilizing Reuse Information in Data Cache Management”, in Proceedings of the 1997 ACM International Conference on Supercomputing, pp. 449–456, July 1998.

[Robinson 1990] J. T. Robinson and M. V. Devarakonda, “Data Cache Management Using Frequency-Based Replacement”, in Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 134-142, May 1990.

[Sartor 2005] J. B. Sartor, S. Venkiteswaran, K. S. McKinley and Z. Wang, “Cooperative Caching with Keep-Me and Evict-Me”, in Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures, pp. 46 – 57, February 2005.

[Seznec 1993] A. Seznec and F. Bodin, “Skewed-associative Caches”, in Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe, pp. 304 – 316, June 1993.

[Smaragdakis 1999] Y. Smaragdakis, S. Kaplan and P. Wilson, “EELRU: Simple and Effective Adaptive Page Replacement”, in Proceedings of the 1999 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, pp. 122 – 133, May 1999.

[Smaragdakis 2004] Y. Smaragdakis, “General Adaptive Replacement Policies”, in Proceedings of the 4th International Symposium on Memory Management, pp. 108 – 119, October 2004.

[Smith 1982] A. J. Smith, “Cache memories”, Computer Survey, vol. 14, issue. 3, pp. 473-530, 1982.

[So 1988] K. So and R. N. Rechtshaffen, “Cache Operations by MRU Change”, IEEE Transaction on Computers, vol. 37, issue 6, pp. 700-707, 1988.

[SPEC95] “SPEC95 benchmark suite”, http://www.specbench.org/

[Spjuth 2004] M. Spjuth, M. Karlsson and E. Hagersten, “Low-Power and Conflict Tolerant Cache Design”, Technical Report 2004-024, Department of Information Technology, Uppsala University, 2004.

[Staschulat 2005a] J. Staschulat, S. Schliecker and R. Ernst, “Scheduling Analysis of Real-Time Systems with Precise Modeling of Cache Related Preemption Delay”, in Proceedings of the 17th Euromicro Conference on Real-Time Systems, pp. 41 – 48, July 2005.

[Staschulat 2005b] J. Staschulat and R. Ernst, “Scalable Precision Cache Analysis for Preemptive Scheduling” in Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded System, pp. 157 – 165, June 2005.

[Stewart 1991] D. B. Stewart and P. K. Khosla, “Real-Time Scheduling of Dynamically Reconfigurable Systems,” in Proceedings of the IEEE International Conference on Systems Engineering, pp. 139 – 142, August 1991.

[Stilliadis 1997] D. Stiliadis and A. Varma, “Selective Victim Caching: A Method to Improve the Performance of Direct-Mapped Caches”, IEEE Transactions on Computers, vol. 46, issue 5, 1997.

[Su 1995] C-L. Su and A. M. Despain, “Cache Design Trade-Offs for Power and Performance Optimization: A Case Study”, in Proceedings of the International Symposium on Low Power and Design, pp. 63 – 68, April 1995.

[Sudarshan 2004] T S B Sudarshan, R. Abbas and S. Vijayalakshmi, “Highly Efficient LRU Implementations for High Associativity Cache Memory”, in Proceedings of the 12th IEEE International Conference on Advanced Computing and Communications, pp. 87-95, December 2004.

[Sukumar 1993] R. A. Sukumar and S. G. Abraham, “Efficient Simulation of Caches Under Optimal Replacement with Application to Miss Characterization”, in Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer System, pp. 24-35, May 1993.

[SunMicrosystems 1997] SunMicrosystems (July 1997.). UltraSparc User’s Manual.

[Tan 2002] T. K. Tan, A. Raghunathan and N. K. Jha, “Embedded Operating System Energy Analysis and Macro-modeling”, in Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 515 – 522, May 2002.

[Tan 2004a] Y. Tan and V. Mooney, “Integrated Intra- and Inter-Task Cache Analysis for Preemptive Multi-tasking Real-time Systems”, in Proceedings of the International Workshop on Software and Compilers for Embedded Systems, pp. 182-199, September 2004.

[Tan 2004b] Y. Tan and V. Mooney, “Timing Analysis for Preemptive Multi-tasking Real-Time Systems”, in Proceedings of Design, Automation and Test in Europe, pp. 1034-1039, February 2004.

[Tan 2004c] Y. Tan and V. Mooney, “Timing Analysis for Preemptive Multi-tasking Real-time Systems with Caches”, Technical Report, GIT-CC-04-02, Georgia Institute of Technology, February 2004.

[Theobald 1993] K. B. Theobald, H. H. J. Hum and G. R. Gao, “A Unified Framework for Hybrid Access Cache Design and Its Applications”, ACAPS Technical Memo 65, December 1993.

[Tomiyamay 2000] H. Tomiyamay and N. D. Dutt, “Program Path Analysis to Bound Cache-Related Preemption Delay in Preemptive Real-Time Systems”, in Proceedings of the 8th International Workshop on Hardware/Software Codesign, pp. 67 – 71, May 2000.

[Vahid 2005] S. Vahid, T. Z. Saman and N. Muhmoud, “A Modified Maximum Urgency First Scheduling Algorithm for Real-Time Tasks”, Transactions on Engineering, Computing and Technology, http://enformatika.org/data/v9/v9-4.pdf, vol. 9, issue 4, pp. 19 – 23, 2005.

[Veidenbaum 1999] A. V. Veidenbaum, W. Tang, R. Gupta, R. Nicolau and X. Ji, “Adapting Cache Line Size to Application Behavior”, in Proceedings of the 13th International Conference on Supercomputing, pp. 145-154, June 1999.

[Wang 1999] Y. Wang and M. Saksena, “Scheduling Fixed-Priority Tasks with Preemption Threshold”, in Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications, pp. 328 – 335, December 1999.

[Wang 2000] Y. Wang and M. Saksena, “Scalable Real-time System Design Using Preemption Thresholds”, in Proceedings of the 21th IEEE Symposium on Real-Time Systems, pp. 25 – 34, November 2000.

[Wang 2002] Z. Wang, K. S. McKinley, A. L. Rosenberg and C. C. Weems, “Using the Compiler to Improve Cache Replacement Decisions”, in Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques, pp. 199 – 208, September 2002.

[Wang 2004] Z. Wang, “Cooperative Hardware/Software Caching for Next-Generation Memory Systems”, Ph.D.Thesis, Department of Computer Science, University of Massachusetts, Amherst, Massachusetts, February 2004.

[Wilton 1996] S. J. E. Wilton and N. P. Jouppi, “CACTI: An Enhanced Cache Access and Cycle Time Model”, IEEE Journal of Solid-State Circuits, vol. 31, issue 5, pp. 677 – 688, 1996.

[Wolfe 1994] A. Wolfe, “Software-Based Cache Partitioning for Real-Time Applications”, Journal of Computer and Software Engineering, vol. 2, issue 3, 1994.

[Wong 2000] W. A. Wong and J-L Baer, “Modified LRU Policies for Improving Second-level Cache Behavior”, in Proceedings of the 6th International Symposium on High-Performance Computer Architecture, pp. 49– 60, January 2000.

[Xu 2005] R. Xu, D. Mosse and R. Melhem, “Minimizing Expected Energy in Real-time Embedded Systems”, in Proceedings of the 5th ACM International Conference on Embedded Software, pp. 251 – 254, September 2005.

[Yang 2004] C-L Yang and C-H Lee, “HotSpot cache: joint temporal and spatial locality exploitation for I-cache energy reduction”, in Proceedings of the 2004 International Symposium on Low Power Electronics and Design, pp. 114 – 119, August 2004.

[Yang 2005] C-L Yang, H-W Tseng, C-C Ho and J-L Wu, “Software Controlled Cache Architecture for Energy Efficiency”, IEEE Transactions on Circuits and Systems for Video Technology, vol. 15, issue 5, pp. 634 – 644, 2005.

[Yongioon 1999] L. Yongioon and B-K, Chung, “Pseudo 3-Way Set-Associative cache: A Way of Reducing Miss Ratio with Fast Access Time”, in Proceedings of the IEEE Conference on Electrical and Computer Engineering, pp. 391 – 396, May 1999.

[Zhang 1997] C. Zhang, X. Zhang and Y. Yan, “Two Fast and High- Associativity Cache Schemes,” IEEE Micro Magazine, vol. 17, issue 5, pp. 40- 49, September 1997.

[Zhang 2003] C. Zhang, F. Vahid and W. Najjar, “A Highly-Configurable Cache Architecture for Embedded Systems”, ACM SIGARCH Computer Architecture News, vol. 31, issue 2, pp. 136 – 146, 2003.

[Zhang 2005] C. Zhang, F. Vahid, J. Yang and W. Najjar, “A Way-Halting Cache for Low-Energy High-Performance Systems”, ACM Transactions on Architecture and Code Optimization, vol. 2, issue 1, pp. 34 – 54, 2005.

[Zhang 2006] C. Zhang, “Balanced Cache: Reducing Conflict Misses of Direct-Mapped Caches through Programmable Decoders”, ACM SIGARCH Computer Architecture News, vol. 34, issue 2, pp. 155-166, 2006.

[Zhang 2007] C. Zhang, “Capacity Co-allocation Configurable Cache for Low Power Embedded Systems”, in Proceedings of International Conference on Computer Design, pp. 405 – 410, October 2007.

[Zhou 2006] X. Zhou and P. Petrov, “Low Power Cache Organization through Selective Tag Translation for Embedded Processors with Virtual Memory Support”, in Proceedings of the 16th ACM Great Lakes Symposium on VLSI, pp. 398 – 403, April 2006.

[Zolfaghari 2004] B. Zolfaghari, “A Dynamic Scheduling Algorithm with Minimum Context Switches for Spacecraft Avionics Systems”, in Proceedings of the 2004 IEEE Aerospace Conference, pp. 2618 – 2624, March 2004.

LIST OF PUBLICATIONS

JOURNAL PAPERS

1. **Biju Raveendran**, T S B Sudarshan, S Twinkle and S Gurunarayanan, "*Shared Memory Process Aware Selective Placement Data Cache for Low Energy Embedded Systems*", Journal of Systems Architecture, Elsevier The EURO Micro Journal (Accepted and to appear)

2. **Biju Raveendran**, T S B Sudarshan and S Gurunarayanan, "*Modified Way Predictive Set-Associative Cache for Energy Efficient Embedded Systems*", GESTS International Transactions on Computer Science and Engineering, vol. 51, issue 1, ISSN 1738-6438, pp. 93 – 98, December 2008.

3. **Biju Raveendran**, Sundar Balasubramaniam and S Gurunarayanan, "*Reduced Preemptions(RP): A Preemption Reduction Heuristic for Power Aware Off-line Scheduling"*, GESTS International Transactions on Computer Science and Engineering, vol. 51, issue 1, ISSN 1738-6438, pp. 167 – 172, December 2008.

4. **Biju Raveendran**, T S B Sudarshan and S Gurunarayanan, "*Cache Memory Design with Late Replacements for Embedded Systems*", International Journal of Lateral Computing, vol. 3, issue 1, ISSN 0973-208X, pp. 39-45, August 2006.

CONFERENCE PAPERS

5. **Biju Raveendran** , Sundar Balasubramaniam and S. Gurunarayanan, "*Evaluation of Priority Based Real Time Scheduling Algorithms: Choices and Tradeoffs*", in Proceedings of the 23rd Annual ACM Symposium on Applied Computing, pp. 302-307, March 2008.

6. **Biju Raveendran**, T S B Sudarshan, Avinash Patil, Komal Randive and S Gurunarayanan, "*Predictive Placement Scheme for Set-Associative Cache for Energy Efficient Embedded System*", in Proceedings of International Conference on Signal Processing, Communications and Networking, pp. 152-157, January 2008.

7. **Biju Raveendran**, T S B Sudarshan, Dlip Kumar, Priyanaka Tugudu and S Gurunarayanan, "*LLRU: Late LRU Replacement Strategy for Power Efficient Embedded Cache*", in Proceedings of 15th IEEE International Conference on Advanced Computing and Communications, pp. 339-344, December 2007.

8. **Biju Raveendran**, T S B Sudarshan, Avinash Patil, Komal Randive and S Gurunarayanan, "*An Energy Efficient Selective Placement Scheme for Set-Associative Data Cache in Embedded System*", in Proceedings of ESA'07- The 2007 International Conference on Embedded Systems and Applications, pp. 188 – 194, June 2007.

9. **Biju Raveendran**, J P Misra, Karan Bhatnagar and S Gurunarayanan, "*EFFS: Efficient Flash File System for Wireless Sensor Nodes*", in Proceedings of ESA'07- The 2007 International Conference on Embedded Systems and Applications, pp. 159 – 165, June 2007.

10. **Biju Raveendran**, T S B Sudarshan S Gurunarayanan, "*Selective Placement Data Cache for Low Energy Embedded System*", in Proceedings of 14th IEEE International Conference on Advanced Computing and Communications, pp. 473-476, December 2006.

11. **Biju Raveendran**, Sundar Balasubramaniam, K Durga Prasad and S. Gurunarayanan, "*Variants of Priority Scheduling Algorithms for Reduced Context Switches in Real Time System*", in Proceedings of the 8th International Conference on Distributed Computing and Networking, Lecture Notes in Computer Science, pp. 466-478, December 2006.

12. **Biju Raveendran** , Sundar Balasubramaniam , K Durga Prasad and S. Gurunarayanan, *"A Context-Switch Reduction Heuristic for Power-Aware Off-line Scheduling*", in Proceedings of the 11th Asia-Pacific Computer Systems Architecture Conference, Lecture Notes in Computer Science, pp. 404-411, September 2006.

13. **Biju Raveendran**, T S B Sudarshan, and S Gurunarayanan, "*Cache Memory Design with Late Replacements for Embedded Systems*", in Proceedings of 2nd International Conference on Embedded Systems, Mobile Communication and Computing, pp 76-90, August 2006.

BRIEF BIOGRAPHY OF CANDIDATE

Biju K R is a fulltime research scholar in Computer Science & Information Systems Group in Birla Institute of Technology and Science, Pilani since January 2004. Prior to this he worked as a Software Engineer at Cognizant Technology Solutions from 2001 to 2003. He obtained his Bachelors of Engineering (Electronics & Communication) from Madras University (MGR Engineering College, Chennai), and Masters of Technology (Information Technology) from IIIT – Pondicherry (Now a part of Pondicherry Engineering College), Pondicherry in 1999 & 2001 respectively. He is a Microsoft Research India Fellow since June 2004. His research interests are Real-Time Scheduling algorithms, Embedded Operating Systems, Energy Efficient File Systems, Energy Efficient Storage Systems, and Energy Efficient Architectures for Embedded Systems.

BRIEF BIOGRAPHY OF SUPERVISOR

Dr. S Gurunarayanan is Professor in Electronics and Instrumentation Group (Dean Admissions & Faculty Division II) in Birla Institute of Technology and Science, Pilani. He obtained Masters in Science (Physics) from Alagappa University, Karaikudi, Masters in Engineering (Systems & Information), from Birla Institute of Technology and Science, Pilani, and Ph.D. (Electronics) from Birla Institute of Technology and Science, Pilani in 1987, 1990 and 2000 respectively. He has several publications in National and International Journals. His research interests are Degital Design and Computer Architecture, VLSI Design, Embedded Systems.