New Approaches to Energy and Temperature Aware Scheduling Techniques for Real-time Multi-core Systems

Thesis submitted to the
Indian Institute of Technology Guwahati
for the award of the degree

of

Doctor of Philosophy
in
Computer Science and Engineering

Submitted by
Sanjay Moulik

Under the guidance of
Dr. Arnab Sarkar and Prof. Hemangee K. Kapoor

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
March, 2020

TH-2256_136101018

TH-2256_136101018

Abstract

A system is classified as real-time if it is characterized by a dual notion of correctness: logical as well as temporal. Proportional fair schedulers are often preferred in such real-time systems due to their inherent advantages such as the ability to provide temporal isolation to a task in the face of possible anomalous behaviour of other tasks, high resource utilization, seamless handling of dynamic task arrivals etc. Many real-time systems in devices like mobiles, laptops, PDAs, etc., depend upon battery as their primary source of energy. Therefore, efficient usage and management of energy while satisfying all temporal and resource constraints, has become a design parameter of paramount importance, in these devices. At the system level, two important strategies namely Dynamic Voltage/Frequency Scaling (DVFS) and Dynamic Power Management (DPM) are used to manage energy consumption. DVFS involves dynamic adaptation of a processor's operating voltage/frequency according to instantaneous requirements of the workload being handled at a given time. DPM on the other hand involves suspending parts of a system when processors are idling (due to low workloads), because energy consumed during the suspension period is negligible. Over the years, the industry is witnessing a significant shift in the nature of processing platforms in real-time embedded systems. The need to satisfy stringent performance requirements, often along with additional constraints on size, weight, power etc., has ushered in the era of heterogeneous processing platforms in todays complex embedded control systems. *This research work has delved towards the design of both DVFS and DVFS-cum-DPM based real-time scheduling strategies for homogeneous as well as heterogeneous multi-core platforms.* Apart from energy, temperature also often plays a critical role in the efficient performance of many real-time and embedded devices prevalent today. Uncontrolled rise in temperature beyond a safe threshold limit not only increases

TH-2256_136101018

cooling costs, but may also reduce a system's efficiency and life span. Some of the techniques used to reduce temperature at the system level includes workload balancing, frequency scaling, core suspension etc. *This thesis has also concentrated on the design of temperature-aware scheduling techniques which attempts to maximize resource utilization while ensuring that temporal as well as thermal constraints related to the system are satisfied.* We have not only used benchmark programs to test our proposed algorithms in real-life situations, but also carried out extensive simulation based experiments using synthetic task sets to validate the efficacy of the algorithms over varied scenarios that may be encountered.

TH-2256_136101018

Declaration

I certify that:

a. The work contained in this thesis is original and has been done by me under the guidance of my supervisors.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. I have followed the guidelines provided by the Institute in preparing the thesis.

d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Sanjay Moulik

TH-2256_136101018

TH-2256_136101018

Copyright

Attention is drawn to the fact that copyright of this thesis rests with its author. This copy of the thesis has been supplied on the condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author.

This thesis may be made available for consultation within the Indian Institute of Technology Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signature of Author.....

Sanjay Moulik

TH-2256_136101018

TH-2256_136101018

Certificate

This is to certify that this thesis entitled, **"New Approaches to Energy and Temperature Aware Scheduling Techniques for Real-time Multi-core Systems"**, being submitted by **Sanjay Moulik**, to the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, for partial fulfillment of the award of the degree of Doctor of Philosophy, is a bonafide work carried out by him under our supervision and guidance. The thesis, in our opinion, is worthy of consideration for award of the degree of Doctor of Philosophy in accordance with the regulation of the institute. To the best of our knowledge, it has not been submitted elsewhere for the award of the degree.

.....

Dr. Arnab Sarkar
Associate Professor
Department of Computer Science and Engineering
IIT Guwahati

.....

Prof. Hemangee K. Kapoor
Professor
Department of Computer Science and Engineering
IIT Guwahati

TH-2256_136101018

TH-2256_136101018

Dedicated to

Almighty GOD, my parents and all my respected teachers

Whose knowledge, blessing, love and inspiration paved my path of success

TH-2256_136101018

TH-2256_136101018

Acknowledgments

I wish to express my deepest gratitude to my supervisors, Prof. Arnab Sarkar and Prof. Hemangee K. Kapoor for their valuable guidance, inspiration, and advice. I feel very privileged to have had the opportunity to learn from, and work with them. Their constant guidance and support not only paved the way for my development as a research scientist but also changed my personality, ability, and nature in many ways. I have been fortunate to have such advisors who gave me the freedom to explore on my own and at the same time the guidance to recover when my steps faltered. Besides my advisors, I would like to thank the rest of my thesis committee members: Prof. S. Biswas, Prof. P. Bhaduri and Prof. A. Sahu, for their insightful comments and encouragement. Their comments and suggestions helped me to widen my research from various perspectives.

I would like to express my heartful gratitude to the director, the deans and other managements of IIT Guwahati whose collective efforts has made this institute a place for world-class studies and eduaction. I am thankful to all faculty and staff of Dept. of Computer Science and Engineering for extending their co-operation in terms of technical and official support for the successful completion of my research work. I thank IIIT Guwahati, especially Prof. Gautam Barua, for letting me pursue Ph.D. as a part-time research scholar from IIT Guwahati.

I am thankful to my friends Amarnath, Mansoor, Suryakant, Gitimoni, Rajesh, Piyoosh, Shounak, Satish and Sangeet for supporting and motivating to overcome any problems either in work and otherwise. The countless discussions, sharing ideas has improved our research.

Most importantly, none of this would have been possible without the love and patience of my family. I want to thank my parents, sister, brother-in-

TH-2256_136101018

law and Richa for being a constant source of love, concern, support, and strength all these years. Their patience and sacrifice will remain my inspiration throughout my life. Without their help, I would not have been able to complete much of what I have done and become who I am.

TH-2256_136101018

TH-2256_136101018

TH-2256_136101018

Contents

TH-2256_136101018

TH-2256_136101018

TH-2256_136101018

TH-2256_136101018

List of Figures

TH-2256_136101018

TH-2256_136101018

List of Algorithms

TH-2256_136101018

TH-2256_136101018

List of Tables

TH-2256_136101018

TH-2256_136101018

List of Symbols

T_i	i^{th} task
T	T= $\{T_1, T_2, \ldots, T_n\}$; Set of n tasks
a_i	Arrival time of task T_i
e_i	Execution time of task T_i
d_i	Relative deadline of task T_i (with respect to its arrival)
p_i	Fixed (Minimum) inter-arrival time for periodic task T_i
re_i	Remaining execution requirement of task T_i
rp_i	Remaining period of task T_i
V_j	j^{th} processor / processing core
V	$V = \{V_1, V_2, \ldots, V_m\}$; Set of m processors / processing cores
$e_{i,j}$	Execution time of task T_i on core V_j
f_{max}	The maximum operating frequency available at any core
P	Power consumed in the system
TS_k	k^{th} time-slice
shr_i	Execution share of task T_i in a time-slice

TH-2256_136101018

TH-2256_136101018

Basic Definitions

Formally, a real-time task can be characterized by the following parameters:

1. **Arrival time** is the time at which a task becomes ready for execution. It is also referred as *request time* or *release time* of the task.

2. **Start time** is the time at which a task starts its execution.

3. **Execution time** is the time required by the processor to finish the computational demand of a task without interruption.

4. **Finishing time** is the time at which a task finishes its execution in the system.

5. **Deadline** is the time before which a task should meet its execution requirement. If it is computed with respect to the system start time, it will be called an *absolute deadline*. If it is computed with respect to its arrival time, it will be called a *relative deadline*.

6. **Slack time** or *Laxity* is the maximum time a task can be delayed after its activation to complete within its deadline.

7. **Priority** is the importance given to a task in context of the schedule at hand.

A real-time task can be classified as periodic, aperiodic and sporadic based on the regularity of its activation [35]:

TH-2256_136101018

1. **Periodic** tasks consist of an infinite sequence of identical activities, called *instances* or *jobs*, that are always separated by a fixed inter-arrival time. The activation time of the first periodic instance is called *phase* (ϕ_i). The activation time of the k^{th} instance is given by $\phi_i + (k-1)p_i$, where p_i is the activation period (*fixed inter-arrival time*) of the task.

2. **Aperiodic** tasks also consist of an infinite sequence of identical jobs. However, their activations are not regularly interleaved.

3. **Sporadic** tasks consist of an infinite sequence of identical jobs with consecutive jobs separated by a *minimum inter-arrival time*.

Following are the three levels of constraint related to the deadline of a task:

1. *Implicit Deadline:* All task deadlines are equal to their periods ($d_i = p_i$).

2. *Constrained Deadline:* All task deadlines are less than or equal to their periods ($d_i \leq p_i$).

3. *Arbitrary Deadline:* All task deadlines may be less than, equal to, or greater than their periods.

We now provide a few other definitions related to tasks and taskset.

Utilization: The utilization of a (implicit deadline) task T_i is given by $u_i = e_i/p_i$. In case of constrained deadline, $u_i = e_i/d_i$.

Hyperperiod: It is the minimum interval of time after which the schedule repeats itself. For a set of periodic tasks (with periods $p_1, p_2, \ldots, p_n$) activated simultaneously at $t = 0$, the hyperperiod is given by the least common multiple of the periods.

Static and Dynamic Task System: In a static task system, the set of tasks that is executed on the platform is completely defined before it starts running the task set. In a dynamic task system, some tasks may experience modifications of their properties while other tasks leave or join the executed task set at run-time.

TH-2256_136101018

Processing Platform: The term *processor* refers to a hardware element in the platform which is able to process the execution of a task.

1. **Uniprocessor** system can only execute one task at a time and must switch between tasks.

2. **Multiprocessor** system will range from several separate uniprocessors tightly coupled using high speed network to multi-core. It can be classified as follows:

   (a) **Homogeneous:** The processors are *identical*, i.e., they all have same functional units, instruction set architecture, cache sizes and hardware services. The rate of execution of every task is same on all processors. Hence, the worst-case execution time of a task is not impacted by the particular processor on which it is being executed.

   (b) **Uniform:** The processors are *identical* - but they are running at different frequencies. Hence, all processors can execute all tasks but the speed at which they are executed and their worst-case execution time vary in function of the processor on which they are executing.

   (c) **Heterogeneous:** The processors are different, i.e., processors may have different configurations, frequencies, cache sizes or instruction sets. Some tasks may therefore not be able to execute on some processors in the platform, while their execution speeds (and their worst-case execution times) may differ on the other processors.

TH-2256_136101018

TH-2256_136101018

Chapter 1

Introduction

A system is classified as *real-time* if it is characterized by a dual notion of correctness: *logical* as well as *temporal* [35]. Safety-critical applications such as reactors in nuclear plants, anti-lock braking systems in vehicles, pacemakers in health-care, fly-by-wire in aircrafts, etc. are examples of real-time systems. Applications in real-time systems often consist of a possibly infinite sequence of *recurrent* tasks. A recurrent task typically represents few lines of code whose execution is triggered by external events that may happen in their operating environment and must complete within a stipulated time bound called deadline. Each execution of the task is referred to as a *task instance* or a *job*. A recurrent task is said to be *periodic* if any two consecutive instances of the same task are always separated by a fixed inter-arrival time. Periodic tasks may be executed either *preemptively* or *non-preemptively*. In preemptive execution, a job which is currently executing on a processor may be interrupted at any time and resumed later. On the contrary, no such interruption is allowed during non-preemptive execution. Given a set of real-time applications modeled as periodic tasks and a multi-core processing platform, *successfully satisfying all timing related specifications is ultimately a scheduling problem.*

Traditionally, scheduling for real-time tasks on multi-cores/multiprocessors make use of either a *partitioned* or *global* approach [47]. In a *fully partitioned* approach, every task is assigned to a single processor and each task is allowed to execute on the allotted processor only. This approach has the advantage of transforming the multiprocessor

TH-2256_136101018

scheduling problem to a set of uniprocessor scheduling ones. As each task runs only on a single processor, there is no penalty due to inter-processor task migrations. However, the process of *partitioning* is provably *NP-hard* both for homogeneous and heterogeneous multiprocessors [38]. Unlike partitioning, a *global scheduling* methodology allows migration of a task onto any available processors at every scheduling point. Hence, schedulers based on this approach may lead to very high migration overheads. Recently, researchers have started focusing towards a hybrid third approach called *semi-partitioned* [97, 98]. Schedulers based on such an approach can be classified into the following two categories. In the first category, processors in the system are divided into disjoint clusters based on stipulated specifications. Each task is allocated to a particular cluster and is only allowed to execute and migrate within the processors of that cluster. In the second category, the schedulers divide the timeline into intervals often called frames or time-slices. Tasks are executed in a partitioned fashion within every interval and they globally resynchronize at the interval boundary. A variety of semi-partitioned algorithms [78, 110, 112] have been proposed in literature.

Fair Scheduling: Many real-time multiprocessor embedded systems execute their applications in a proportional fair manner. Proportional fair or rate based execution guarantees are typically of the form, *complete X units of execution of application A out of every Y time units*. Such fairness guarantees are often useful in executing a mix of applications with various degrees of timeliness criticality such as online gaming, streaming audio and video, email, web browsing, etc. [113]. Proportional fair schedulers are often preferred in real-time embedded systems due to their inherent advantages such as ability to provide temporal isolation to a task in the face of possible anomalous behaviour of other tasks, high resource utilization, seamless handling of dynamic task arrivals etc.

Energy-Aware Scheduling: Apart from temporal constraints, energy-efficiency has become a primary design constraint in modern real-time computing systems like mobiles, laptops, PDAs, etc., which depend upon battery as their primary source of energy.

TH-2256_136101018

Hence, a lot of research has been conducted towards their power management at various levels of abstraction, starting from hardware and firmware to architectural, system and even application levels. Even as the hardware becomes more and more energy-efficient, the onus of extracting maximum efficiency out of the hardware very often falls on the software. Total power dissipation in a processor is caused by two major components: *dynamic* and *static*. At the operating system level, Dynamic Voltage and Frequency Scaling (DVFS) [18, 69, 115] technique is used to manage the dynamic power consumption. DVFS involves dynamic adaptation of a processor's operating voltage/frequency according to instantaneous requirements of the workload being handled at a given time. As energy dissipated per cycle in CMOS circuits scale quadratically with respect to the supply voltage, this strategy is able to provide large energy savings in DVFS enabled processors. On the other hand, static energy dissipation is due to leakage drain [71] from transistors and is sub-linearly proportional to the operating frequency. Static energy wastage is mainly controlled using Dynamic Power Management (DPM) [13, 21, 69] technique, where a processor is put into inactive low power suspension/sleep states by procrastinating task executions while simultaneously guaranteeing their timely completion.

Temperature-Aware Scheduling: The advent of deep sub-micron VLSI technologies have resulted in extremely dense multi-million gate chips, where power dissipation and thermal hotspots have become very serious design concerns [117]. The uncontrolled rise in temperature not only increases cooling costs but may also reduce a system's efficiency and life span. A study in [80] has shown that the lifespan of a chip may be reduced by upto 50% with a temperature increase of $10 - 15°C$ beyond normal operating temperature. Therefore, modern multi-core processors typically comes with a stipulated temperature threshold which must be adhered to for safe and efficient operation of the system. Dynamic Thermal Management (DTM) mechanisms, which enforce performance throttling in order to mitigate temperature hotspots, are triggered whenever the operating temperature in the system crosses the stipulated temperature threshold. Activation of

TH-2256_136101018

DTM involves steps such as powering-down processors, clock gating, dynamic supply voltage and frequency scaling, etc. These steps introduce unpredictability in a system's timing behavior. An important objective of real-time scheduler designs implemented on these platforms is, therefore, to ensure DTM-free operation over the entire schedule length. This can be achieved by always ensuring the operating temperature of the processors to be within a specified temperature threshold limit [99].

Scheduling techniques for heterogeneous platforms: Computing platforms with specialized components like multi-core CPUs with specialized graphics processing cores, specialized signal processing cores, specialized floating-point units, customizable FPGAs, etc., are called heterogeneous (or unrelated) processing platforms. On such platforms, the same piece of code may require different amounts of time to execute on different processing cores. For example, a task responsible for rendering images may take far less time to execute on a graphics processor compared to a general-purpose CPU, while number-crunching routines would execute more efficiently on CPUs. Development of efficient resource allocation strategies for real-time tasks on heterogeneous platforms has traditionally proved to be a challenging as well as a computationally expensive problem. In recent years, researchers have explored scheduling techniques for heterogeneous multi-core platforms [24, 45]. However, only a few works have focussed towards efficient energy-aware scheduling for generic heterogeneous platforms having an arbitrary number of processor types.

This research intends to study various types of scheduling strategies for real-time multi-core processing platforms and develop energy as well as temperature aware scheduling strategies for homogeneous and heterogeneous platforms.

TH-2256_136101018

1.1 Challenges

An efficient scheduling strategy must not only meet requirements of diverse applications but also support a variety of processing platforms in modern real-time systems. Some of the important challenges faced by such schedulers [35] are discussed below:

1. **Temporal requirements:**
   Computational activities in real-time systems are associated with temporal constraints, which have to be met in order to achieve the desired behaviour. A typical temporal constraint for a task is the deadline, which represents the time before which the task should complete its execution without causing any damage to the system. Another temporal feature for a real-time task can be associated with it's regularity of activation. In particular, tasks can be defined as *periodic*, or *aperiodic*, or *sporadic*. Scheduling strategies for real-time systems must be able to guarantee the temporal requirements (i.e., deadlines) associated with various types of tasks that co-exist in the system.

2. **Predictability:**
   Scheduling strategies for real-time systems must deliver guaranteed performance and co-execution behavior during online operation of the system, under pre-specified load and failure conditions. To achieve a desired level of performance, the system must be analyzable to predict the consequences of any scheduling decision. An important mechanism for enhancing predictability is to conduct static analysis [134] followed by offline schedule generation so that the performance of the system is guaranteed a priori even under worst-case task arrival behavior and service requirements.

3. **Repeatability:**
   Most of the devices based on embedded systems execute set of tasks which repeat themselves after sometime. As for example, reading data from sensors and then performing operations based on these readings. Repeatability is a property of a scheduler towards guaranteeing deadlines of all tasks where tasks may either repeat

TH-2256_136101018

after a fixed interval of time (periodic) or repeat with a minimum inter-arrival time constraint (sporadic).

4. **Resource constraints:**
   Real-time systems are implemented on platforms consisting of a limited number of processing elements, memory, network bandwidth, etc. For example, providing a lot of redundant hardware is not always possible in cost-sensitive real-time systems like cars, where a cost differential of even a hundred dollars can make a commercial difference [43, 72, 73, 108]. Software running on embedded systems are becoming more and more complex which need to run and meet their performance objective on the limited number of available processing elements. In addition, the nature of processing elements is also changing over the years. Specifically, single cores have slowly given way to multi-core platforms in order to cater to higher computation demands while adhering to restrictions on power/energy dissipation. Scheduling schemes designed for real-time systems must be able to effectively utilize the processing capacity of the underlying platform consisting of a limited number of processing resources, to satisfy the constraints associated with a given real-time task set.

5. **Platform Heterogeneity:**
   Nowadays, modern real-time systems are based on heterogeneous multi-core platforms, which help them efficiently cater to the diverse and high computation demands of the applications. However, devising efficient resource allocation strategies for real-time tasks on heterogeneous platforms has traditionally proved to be a challenging as well as a computationally expensive problem. As a consequence, today we face a severe dearth of low-overhead real-time scheduling techniques which are applicable to heterogeneous platforms.

6. **Energy Minimization:**
   If all timing and safety related constraints are satisfied, then the system designer can focus on further optimization, such as minimizing the overall energy consump-

TH-2256_136101018

tion to prolong the battery lifetime of systems or to cut the power bills in servers.

7. **Thermal Constraints:**
   Real-time systems implemented on multi-core platforms need to satisfy Thermal Design Power (TDP) thresholds used by chip manufacturers [1]. The rise in temperature beyond TDP may trigger Dynamic Thermal Management (DTM) in order to ensure thermal stability of the system. However, the application of DTM makes the system susceptible to higher unpredictability and performance degradations for real-time tasks [75, 99, 114]. This necessitates the development of schedulers that can guarantee adherence to a system level peak thermal constraint.

1.2 Motivation for this dissertation

Although the scheduling strategies discussed in literature attempt to enhance energy/temperature efficiency of the system, most of them do not consider fairness in the relative execution progress of tasks as an important parameter in their scheduling decisions. Proportional fair schedulers generally perform better at: i. Providing temporal isolations to a task from a possible anomalous behaviour of other tasks (which, for example, may misbehave by taking more time than they are stipulated to take), ii. Seamlessly handling dynamic arbitrary task arrivals at runtime, iii. Managing inherently rate-based tasks such as continuous multimedia applications. However, with more and more energy/temperature constrained embedded devices running interactive and QoS sensitive applications such as continuous media, gaming, etc. [113], the need for energy/temperature aware proportional fair scheduling algorithms is quickly gaining importance. In addition, the need to satisfy stringent performance requirements, often along with additional constraints on size, weight, power, etc., has ushered in the era of heterogeneous processing platforms in today's complex embedded control systems. Hence, in this dissertation, we propose a few novel resource usage efficient energy/temperature aware scheduling strategies for homogeneous as well as heterogeneous real-time multi-core platforms.

TH-2256_136101018

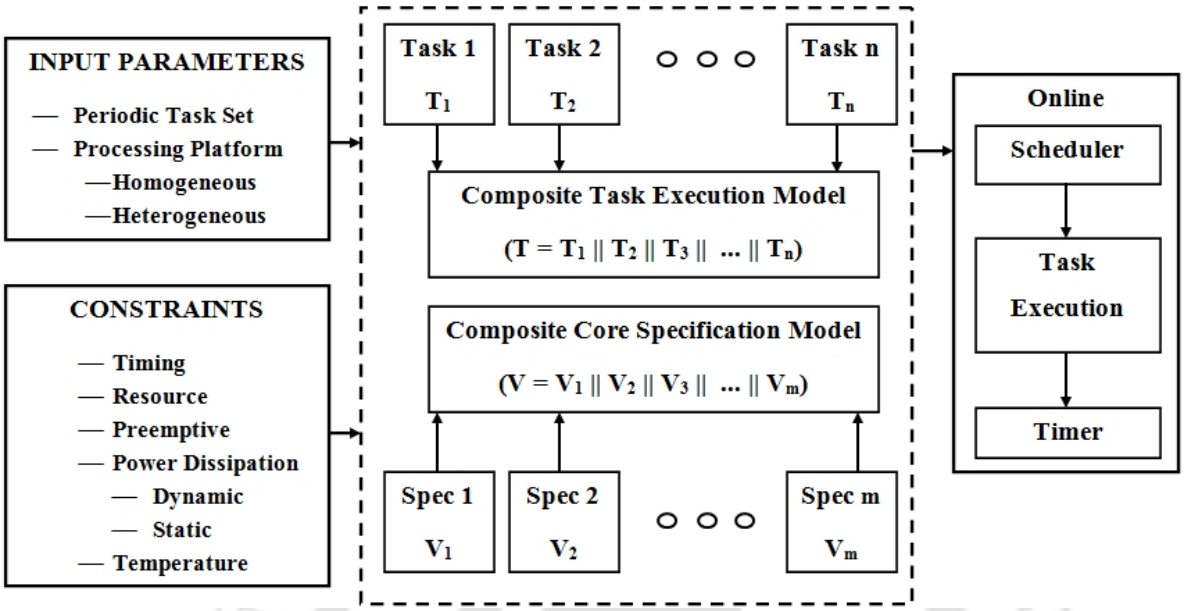


Figure 1.1: *Pictorial representation of the scheduling framework*

1.3 Proposed Framework

This research work prepares optimal online fair scheduling strategies for real-time systems. The pictorial representation of the proposed framework is presented in Figure 1.1. The proposed framework receives specifications regarding task set, processing platform and constraints, as input parameters. Here, task set consists of a set of n periodic tasks $T = \{T_1, T_2, ..., T_n\}$ to be scheduled on a set of m homogeneous / heterogeneous processing cores $V = \{V_1, V_2, ..., V_m\}$ which may operate on frequencies chosen from a discrete set. The set of frequencies are then normalized with respect to the maximum frequency to obtain the normalized set of frequencies represented as $F = \{f_1, f_2, ..., f_{max}\}$, such that, f_{max} represents the normalized frequency of 1 and other normalized frequencies lie between 0 and 1. Each instance of T_i has an execution requirement of e_i time slots and period/inter-arrival time p_i. The weight of any task T_i is defined as $wt_i = e_i/p_i$. At any given instant, re_i and rp_i denote the remaining execution requirement and remaining period of T_i. The constraints for schedulers can be timing, resource, power dissipation, temperature minimization, etc. Based on a given set of input parameters along with the

TH-2256_136101018

constraints, we propose different scheduling strategies for real-time systems.

1.4 Contributions

We have designed the following resource efficient low-overhead semi-partitioned schedulers for multiprocessor/multi-core systems:

1.4.1 EAFBFS: An Energy Aware Frame Based Fair Scheduler

In this work, we propose a new semi-partitioned scheduling strategy for hard real-time homogeneous multi-core systems called *Energy Aware Frame Based Fair Scheduling (EAFBFS)*, which combines the benefits of high resource utilization and restricted migrations as in the *Energy-Aware DPFair (EA-DPFair)* algorithm [57] while providing accurate as well as tunable proportional fairness for all tasks across all time slots. *EAFBFS* employs a semi-partitioning strategy with a two-level hierarchical scheduling scheme, where the outer level divides time into frames/slices, demarcated by the arrivals and departures of all tasks in the system. This approach helps it to significantly restrict migration and preemption overheads. The inner level scheduler supervises task executions within time-slices. Given a specific fairness deviation bound as input, the algorithm automatically adjusts itself to meet such a demand, albeit at the cost of possibly higher energy dissipation. Experimental results show that EAFBFS is able to achieve higher fairness accuracy (10 to 15 times on average) with respect to state-of-the-art [57] while saving almost the same percentage of energy on heavily loaded systems.

1.4.2 DPFair Scheduling with Slowdown and Suspension

A major class of real-time systems typically experience short bursts of heavy CPU activity interleaved with long durations of significantly lower workloads. The *DVFS based DP-Wrap* [57] algorithm provides an important scheduling solution for such resource constrained hard real-time systems due to its ability to provide optimal resource utilization, controlled migrations and minimized dynamic energy dissipation. However, DVFS based schedulers are only able to decrease dynamic energy consumption of the system

TH-2256_136101018

by reducing the voltage/frequency of a processor. With exponential growth in chip transistor densities over technology generations, static energy dissipation due to leakage drain from transistors has steeply increased over the years [14]. Since *DVFS based DP-Wrap* only focusses on reduction of dynamic energy, it allows significant static energy dissipation whenever system workloads are not high enough to demand operation above the critical frequency [69]. This work therefore proposes an integrated DVFS-cum-DPM based DPFair based scheduling strategy for homogeneous multi-core systems called, *DPFair with slowdown and suspension (DPFair-SS)*, in order to minimize overall system-wide energy consumption combining both static and dynamic energy dissipation. Experimental results show that our proposed scheduling technique DPFair-SS, exhibits appreciable energy savings over the state-of-the-art [57], in situations when the system experiences low workloads over significantly long durations.

1.4.3 A Cluster-Oriented Scheduling Technique for Heterogeneous Multi-cores

Development of efficient resource allocation strategies for real-time tasks on heterogeneous platforms has traditionally proved to be a challenging as well as a computationally expensive problem. However, strategies which can efficiently schedule real-time task sets on generic heterogeneous platforms having an arbitrary number of processor types, are rare. Most of the existing strategies [91,108] are oriented towards systems with restricted number of processing core types. Hence, this work proposes an effective low-overhead heuristic approach called *COST: A Cluster-Oriented Scheduling Technique for Heterogeneous Multi-cores*, for scheduling a set of periodic tasks executing on a heterogeneous multi-core system. The proposed technique works in three-phases namely, *Core Clustering*, *Task Partitioning*, and *Task Scheduling*. The *Core Clustering* step attempts to combine the available processing cores into a group of clusters. Each cluster consists of two cores and a disjoint subset of the given task set is assigned to it. The tasks assigned to a cluster are then allocated to the processing cores of the cluster in the *Task Partitioning* phase and scheduled fairly in the *Task Scheduling* phase. Experimental studies show that our proposed scheme provides high resource utilization by scheduling more

TH-2256_136101018

number of task sets with respect to state-of-the-art [96].

1.4.4 A Low Overhead Scheduler for Real-Time Periodic Tasks on Heterogeneous Multi-core Systems

This work proposes an effective low-overhead heuristic approach named *HETERO-SCHED*, for scheduling a set of periodic tasks executing on heterogeneous multi-core systems. The proposed approach first applies *deadline partitioning* [78] to obtain a set of discrete time-slices. Over each such time-slice, HETERO-SCHED conducts the following two phase operation: First, it determines the fractions of the computation demand of each task to be assigned onto the platform. Next, it assigns valid start and finish times to all tasks, according to the allocation prescribed in the first phase. Experimental studies show that our proposed scheduling mechanism is able to schedule significantly higher number of tasks sets, compared to the state-of-the-art [108].

1.4.5 An Energy-Aware Scheduler for Heterogeneous Multi-core real-time systems

Devising energy-efficient scheduling strategies for real-time periodic tasks on heterogeneous platforms is a challenging as well as a computationally demanding problem. As a consequence, today we face a scarcity of real-time energy-aware scheduling techniques which are applicable to heterogeneous platforms. Hence, this work proposes a low-overhead heuristic strategy called *HEALERS*, for DVFS enabled energy-aware scheduling of a set of periodic tasks executing on a heterogeneous multi-core system. The presented strategy first applies *deadline-partitioning* [78] to acquire a set of distinct time-slices. At any time-slice boundary, the following three phase operation is applied to obtain schedule for the next time-slice: First, it computes the fragments of the execution demands of all tasks on to each of the different processing cores in the platform. Next, it generates a schedule for each task on one or more processing cores such that the total execution demand of all tasks are satisfied. Finally, *HEALERS* applies DVFS on all processing cores so that energy consumption within the time-slice may be minimized while not jeopardizing execution requirements of the scheduled tasks. Experimental

TH-2256_136101018

results show that our scheme is not only able to achieve appreciable energy savings with respect to state-of-the-art [16] (5% to 42% on average) but also enables significant improvement in resource utilization (as high as 58%).

1.4.6 A Temperature-Aware Real-Time Semi-partitioned Scheduler

Modern multi-core systems, which execute complex functionalities at high frequencies on densely packed multi-million gate platforms, are often prone to unacceptable surges in core temperatures, if not effectively managed. Increase in temperature beyond stipulated thresholds not only results in high cooling costs but also leads to high leakage power dissipation [80, 117], along with reduced efficiency and lower life-span for the system. Given, a set of periodic real-time tasks to be executed on a thermally constrained multi-core system, proportional fair schedulers form an attractive scheduling alternative. This is because of their flexibility and the ability to deliver efficient resource utilization, which can potentially enable accurate control over the timeliness of all tasks as well as stipulated temperature upper bounds on all processing cores, over the entire schedule length. In this paper, we propose a low-overhead two-level hierarchical temperature-aware semi-partitioned proportional fair scheduler, called *Temperature-Aware Real-Time Semi-partitioned Scheduler (TARTS)*. The first level in TARTS partitions time into discrete slices based on task deadlines, such that accurate proportional fairness is maintained at all slice boundaries. The second level performs intra-slice scheduling with the objective of maximizing resource utilization while not breaching a stipulated temperature threshold. Our experimental results show that TARTS is able to perform appreciably under various realistic scenarios.

TH-2256_136101018

1.5 Organization of the Thesis

The thesis is organized into seven chapters. A summary of the contents in each chapter is as follows:

- **Chapter 1:** *Introduction*
  This chapter is introductory, discussing various challenges for schedulers in multi-core real-time systems and thus, presenting the motivation of our work. At the end of this chapter, we provide a brief summary of the work done in this thesis.

- **Chapter 2:** *Energy and Temperature Aware RT Scheduling: Background and State-of-the-Art*
  In this chapter, we give a brief description of different types of scheduling approaches for multi-core real-time systems. We also discuss about the advantages and disadvantages of each described approach.

- **Chapter 3:** *Energy-Aware scheduling on homogeneous multi-core systems*
  In this chapter, we propose two semi-partitioned energy-aware fair schedulers for homogeneous multi-core real-time systems namely, *Energy Aware Frame Based Fair Scheduling (EAFBFS)* and *DPFair Scheduling with Slowdown and Suspension (DPFair-SS)*. *EAFBFS* employs a DVFS based energy-aware scheduling strategy to combine the benefits of high resource utilization and restricted migrations while providing accurate as well as tunable proportional fairness for all tasks across all time slots. On the other hand, *DPFair-SS* proposes a combined slowdown-sleep strategy to minimize energy dissipation in hard real-time multi-core systems. DPFair-SS is able to perform significantly better than a purely DVFS based DPFair scheduler in situations when the system experiences low workloads over significantly long durations.

- **Chapter 4:** *Scheduling on heterogeneous multi-core systems*
  Research conducted in the fourth chapter deals with the problem of multi-core scheduling on heterogeneous platforms. Here, we present two semi-partitioned

TH-2256_136101018

heuristic schedulers named, *COST (A Cluster-Oriented Scheduling Technique for Heterogeneous Multi-cores)* and *HETERO-SCHED (A Low-overhead Heterogeneous Multi-core Scheduler for Real-time Periodic Tasks)*. *COST* works in three-phases to provide an effective low-overhead heuristic scheduling strategy and it has a restriction on the number of migrations for an individual task within a time-slice. On the other hand, *HETERO-SCHED* is a deadline partitioned based heuristic strategy which works in two phases to do the same and it allows unrestricted inter-core task migrations.

- **Chapter 5:** *Energy-Aware scheduling on heterogeneous multi-core systems*
  In this chapter, we propose a DVFS based fully-migrative energy-efficient strategy called *HEALERS*, for scheduling periodic real-time tasks on heterogeneous platforms. HEALERS is composed of two major components: i) *COMPUTE-SCHEDULE* and, ii) *COMPUTE-EA-SCHEDULE*. These two components work in unison to not only deliver appreciable energy savings but also very high resource utilizations.

- **Chapter 6:** *Temperature-Aware resource allocation strategy for real-time systems*
  This chapter proposes a two-level temperature-aware scheduling strategy for multi-core systems called *TARTS*. At the first level, time-slices are determined based on the deadlines of tasks and task execution proceeds time-slice by time-slice, in a proportional fair manner. At any time-slice boundary, shares of all tasks to be executed in the next time-slice, are determined. The second level performs intra-time-slice schedule generation.

- **Chapter 7:** *Conclusion and Future Works*
  The thesis concludes with this chapter. We discuss the work in progress, possible extensions and future work that can be done in this area.

TH-2256_136101018

Chapter 2

Energy and Temperature Aware RT Scheduling: Background and State-of-the-Art

Energy has become a first class design criterion in many of today's real-time embedded systems which are often operated by limited energy sources like batteries. Reduction of energy consumption is essential to prolong the battery life in these systems. Hence, a lot of research has been conducted towards their power management at various levels of abstraction, starting from hardware and firmware to architectural, system and even application levels. Apart from energy, temperature also often plays a critical role in the efficient performance of many real-time and embedded devices prevalent today. The uncontrolled rise in temperature beyond a safe threshold limit not only increases cooling costs but may also reduce system efficiency and life span.

In this chapter, we present a brief introduction to the definitions related to real-time systems and their task models. We first provide an overview on the structure of real-time systems. Then, various scheduling algorithms for both homogeneous and heterogeneous platforms are discussed. Next, we present some energy and temperature-aware scheduling algorithms for real-time systems.

2.1 Real-time Systems

Typically, real-time systems are composed of the following layers [101]:

TH-2256_136101018

- **An application layer**, which is composed of a set of applications that require execution in the system.
- **A real-time scheduler**, which takes the scheduling decisions and provides services to the application layer.
- **A hardware platform**, which includes the processors/cores (among other things such as memories, communication networks, etc.).

We will now present each of these layers in detail and introduce the theoretical models enabling researchers to analyze these systems and design efficient schedulers for real-time systems to schedule the application tasks on the hardware platform.

2.1.1 The Application Layer

The application layer contains a set of applications that the system needs to execute. In real-time systems, the applications are often composed of a set of *recurrent* tasks. Each such task may represent a piece of code (i.e., program) which is triggered by external events that may happen in their operating environment. Each execution of the task is referred to as a *task instance* or a *job*. We now present the set of definitions related to a real-time task.

2.1.1.1 A Real-time Task Model

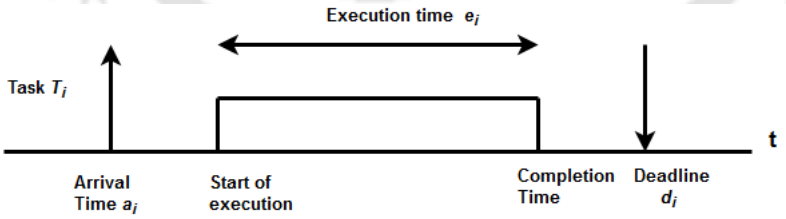


Figure 2.1: *Temporal Characteristics of real-time task T_i*

Formally, a real-time task (denoted by T_i; shown in Figure 2.1) can be characterized by the following parameters:

TH-2256_136101018

1. **Arrival time** (a_i) is the time at which a task becomes ready for execution. It is also referred as *request time* or *release time* of the task.

2. **Start time** is the time at which a task starts its execution.

3. **Execution time** (e_i) is the time required by the processor to finish the computational demand of a task without interruption.

4. **Finishing time** is the time at which a task finishes its execution in the system.

5. **Deadline** is the time before which a task should meet its execution requirement. If it is computed with respect to the system start time (at 0), it will be called an *absolute deadline*. If it is computed with respect to its arrival time, it will be called a *relative deadline*.

6. **Slack time** or *Laxity* is the maximum time a task can be delayed after its activation to complete within its deadline: $d_i - e_i$.

7. **Priority** is the importance given to a task in context of the schedule at hand.

A real-time task T_i can be classified as periodic, aperiodic or sporadic based on the regularity of its activation [35]:

1. **Periodic** tasks consist of an infinite sequence of identical activities, called *instances* or *jobs*, that are always separated by a fixed inter-arrival time. The activation time of the first periodic instance is called *phase* (ϕ_i). The activation time of the k^{th} instance is given by $\phi_i + (k-1)p_i$, where p_i is the activation period (*fixed inter-arrival time*) of the task.

2. **Aperiodic** tasks also consist of an infinite sequence of identical jobs. However, their activations are not regularly interleaved.

3. **Sporadic** tasks consist of an infinite sequence of identical jobs with consecutive jobs separated by a *minimum inter-arrival time*.

TH-2256_136101018

Following are the three levels of constraint related to the deadline of a task:

1. *Implicit Deadline:* All task deadlines are equal to their periods ($d_i = p_i$).
2. *Constrained Deadline:* All task deadlines are less than or equal to their periods ($d_i \leq p_i$).
3. *Arbitrary Deadline:* All task deadlines may be less than, equal to, or greater than their periods.

We now provide a few other definitions related to tasks and taskset.

Utilization: The utilization of a (implicit deadline) task T_i is given by $u_i = e_i/p_i$. In case of constrained deadline, $u_i = e_i/d_i$.

Hyperperiod: It is the minimum interval of time after which the schedule repeats itself. For a set of periodic tasks (with periods $p_1, p_2, \ldots, p_n$) activated simultaneously at $t = 0$, the hyperperiod is given by the least common multiple of the periods.

Static and Dynamic Task System: In a static task system, the set of tasks that is executed on the platform is completely defined before it starts running the task set. In a dynamic task system, some tasks may experience modifications of their properties while other tasks leave or join the executed task set at run-time.

2.1.2 A Real-time Scheduler

A real-time scheduler acts as an interface between applications and hardware platform. It configures and manages the hardware platform (e.g., manage hardware interrupts, hardware timers, etc.). More importantly, it schedules the tasks using a real-time scheduling algorithm. The set of rules that, at any time, determines the order in which tasks are executed is called a *scheduling algorithm*.

Given a set of tasks, $T = \{T_1, T_2, ..., T_n\}$, a *schedule* is an assignment of tasks onto available processors, so that each task is executed until completion. A schedule is said to

TH-2256_136101018

be *feasible* if all tasks can be completed according to a set of specified constraints. A set of tasks is said to be *schedulable* if there exists at least one algorithm that can produce a feasible schedule. A scheduling algorithm is said to be *optimal* if it is able to find a feasible schedule, if one exists. In many cases, optimal schedules are difficult to determine due to complex objectives and/or one or more complicated necessary conditions, especially when the number of tasks/resources become higher. Therefore, such complicated optimal solution strategies often do not scale for large systems or scenarios when only partial information about online task/performance behaviours are available. In many a times, for cases when optimal solutions are hard to derive, it is possible to obtain efficient solutions by analysing the problem structure and evolving a faster and greedier mechanism (compared to the optimal strategy) which strategically explores only a part of the overall solution space. Although, such strategies may not guarantee optimality or even bounds on the maximum deviation from optimality, they may provide good/satisfactory solutions in most practical situations. Such solution mechanisms are usually referred to as *heuristic* strategies. In a *recursive* scheduling algorithm, the overall task scheduling in the system is broken into group of time intervals and then schedules are prepared for each interval. In a *work-conserving* scheduling algorithm, the processor is never kept idle when there exists a task waiting for execution on the processor.

2.1.3 Processing Platform

The term *processor* refers to a hardware element in the platform which is able to process the execution of a task.

1. **Uniprocessor** systems can execute only one task at a time and must switch between tasks.

2. **Multiprocessor** systems range from several separate uniprocessors tightly coupled using high speed network to multi-core. It can be classified as follows:

   (a) **Homogeneous:** The processors are *identical*, i.e., they all have same functional units, instruction set architecture, cache sizes and hardware services.

TH-2256_136101018

The rate of execution of every task is same on all processors. Hence, the worst-case execution time of a task is not impacted by the particular processor on which it is being executed.

(b) **Uniform:** The processors are *identical* - but they are running at different frequencies. Hence, all processors can execute all tasks but the speed at which they are executed and their worst-case execution time vary based on the processor on which they are executing.

(c) **Heterogeneous:** The processors are different, i.e., processors may have different configurations, frequencies, cache sizes or instruction sets. Some tasks may therefore not be able to execute on some processors in the platform, while their execution speeds (and their worst-case execution times) may differ on the other processors.

2.2 A Classification of Real-time Scheduling Approaches

Preemptive Vs. Non-preemptive Scheduling: *Preemptive* schedulers are based on the assumption that the execution of a task may be interrupted and the processor directed to run a different piece of code after the interrupt. The unfinished portion of the interrupted task thus has to be re-allocated to may be, a different processor [52]. On the contrary, scheduling algorithms following a *non-preemptive* approach must allow a task to execute until completion. As a result, the response time to external events may be quite long if some tasks have a large execution time. However, many task systems are inherently atomic in the sense that task invocations must execute to completion without interruption once started. Preemptive schedulers are unusable for these task systems.

Online Vs. Offline Scheduling: In *offline* scheduling, the scheduler has a priori knowledge of the task set and its constraints, such as arrival times, execution times, precedence constraints, etc. The schedule is generated and stored at design time and dispatched later during runtime of the system. Offline scheduling is also referred to as *static* scheduling [46]. On the other hand, *online* scheduling algorithms make their

TH-2256_136101018

scheduling decisions at runtime based on the information about the tasks that have arrived so far. Although they are often flexible and adaptive, they may incur significant overheads because of runtime processing. However, they are a must in systems which do not have enough information before run-time to execute the scheduler statically. Online scheduling is also referred to as *dynamic* or *runtime* scheduling.

Clock-Driven Vs. Event-Driven Scheduling: In *clock-driven* schedulers, scheduling decisions are made at specific time instants which are chosen a priori before the system begins its execution [84]. Typically, in a system that uses clock-driven scheduling, all parameters of the job set are fixed and known. It is also called a *time-driven* scheduling approach. A *table-driven* scheduler is an example of a clock-driven approach. Here, the schedule is generated and stored in a table off-line. The system timer kicks off execution of a segment of the code of a task at each scheduling decision time by referring to the table at run time.

In the *event-driven* approach, scheduling points are defined by events such as job release or completion. Generally, these schedulers assign priorities to each task. At each scheduling instant, the currently highest priority task present in the ready queue gets hold of the resource (hence, they are also called *priority-driven schedulers*). These algorithms leave a resource idle only when no job requiring the resource is ready for execution. The *Rate Monotonic (RM)* [83,84] and *Earliest Deadline First (EDF)* [83,84] algorithms are examples of the event-driven approach. Event-driven schedulers are more proficient than clock-driven schedulers because they can feasibly schedule some task sets that clock-driven schedulers cannot. These are also more flexible because they can feasibly schedule sporadic and aperiodic tasks in addition to periodic tasks whereas clock-driven schedulers can only handle periodic tasks.

This thesis primarily deals with *real-time independent periodic* task-sets which are scheduled using variants of *online dynamic priority* scheduling policies on both *homogeneous* as well as *heterogeneous multi-core* systems.

TH-2256_136101018

Static Priority Vs. Dynamic Priority Scheduling: The distinction between *static priority* and *dynamic priority* scheduling is based on the priority management policy adopted by a priority-driven scheduler. In the static priority scheme, tasks are assigned an integer priority value that remains fixed for the lifetime of the task. Whenever a task is made ready to run, the active task with the highest priority commences or resumes execution, preempting the currently executing task if need be. Priority values may change at run time in case of dynamic priority schedulers. *Rate Monotonic (RM)* [83, 84] and *Deadline Monotonic (DM)* [6, 84] are examples of Static priority scheduling while *Earliest Deadline First (EDF)* [83, 84] and *Least Slack Time First (LST)* [84] are examples of dynamic priority scheduling.

Partitioning Vs. Global Scheduling: In the context of multiprocessor scheduling policies, a *global* scheduler is one which puts all the ready tasks in a single queue and selects the highest priority task at each invocation irrespective of which processor is being scheduled. Thus, a task is allowed to execute on any processor, even when resuming after having been preempted. In a purely *partitioned* approach, on the other hand, the set of tasks is partitioned into as many disjoint subsets as there are processors available, and each such subset is associated with a unique processor [37, 88, 122]. Thus, all instances of a task get executed on the same processor.

The main advantage of partitioning is that it allows the multiprocessor scheduling problem to be reduced to a set of uniprocessor ones. In each processor of this set a separate well known uniprocessor scheduler like *Rate Monotonic Analysis (RM)*, *Earliest Deadline First (EDF)*, etc. may be easily applied. In addition, the overheads of inter-processor task migrations and local cache misses are far smaller than global scheduling. Finally, because task-to-processor mapping (which task to schedule on which processor) need not be decided globally at each time-slot, the scheduling overhead associated with a partitioning strategy is lower than that associated with a non-partitioning strategy [10, 11, 37]. On the other hand, even though the generic global scheduling methodology may have a higher scheduling complexity and cause an unrestricted number of migrations and cache misses, it possesses many attractive features like flexible

TH-2256_136101018

resource management, dynamic load distribution, fault resilience, high system utilization, etc. [122]. Between these two extremes of no inter-processor migration and full migration, there is an intermediate class of algorithms that allow *restricted migration*. For example, different jobs of the same task may be allowed to execute on different processors. However, a single job may be constrained to execute on a particular processor.

2.3 A brief survey of scheduling algorithms

In this section, we present a brief survey of various scheduling algorithms proposed in the literature for real-time systems.

2.3.1 Partitioning Strategies

Optimal assignment of tasks to processors in partitioning is a bin-packing problem which can be stated as follows: given a list L of items of size $\{a_1, a_2, ..., a_n\}$, where $a_i \in (0, 1]$ (a_i represents the weight of task i), the problem of bin packing is to pack these items into a minimal number of unit capacity bins. The problem is known to be NP-hard and several polynomial time heuristics have been proposed to solve it.

The performance of any bin-packing algorithm is evaluated by a measure called *competitive-ratio(R)* which may be defined as follows:

$$R = \lim_{n\to\infty} \sup \frac{A(L)}{OPT(L)},$$

where, L is a list of items $< a_1, a_2, ..., a_n >$ of size n, $A(L)$ is the number of bins required by the bin packing algorithm A when list L is used and $OPT(L)$ is the best off-line number of bins required. It is easy to interpret that the use of an infinite sized list in the above measure gives us the worst-case performance ratio. However, there may be many other lists of smaller size that also gives us the worst-case ratio. We consider below some of the well-known approaches [87, 88, 122].

Next Fit(NF): This is one of the simplest of the known heuristics. It starts from the first bin and defines it as the active bin. If the next incoming item fits the bin, it places

TH-2256_136101018

it in that bin. Otherwise, it creates a new bin, makes it the new active bin, and packs the item into this bin. Thus, at any given time, there is only one active bin. The NF algorithm has a competitive-ratio of 2 [12].

First Fit(FF): Given a list of bins, the FF algorithm assigns the next item to the first bin that can accept it.

Best Fit(BF): The BF algorithm assigns the next item to such a processor that can accept the task and will have minimal remaining spare capacity after its addition.

Worst Fit(WF): WF is opposite to BF; it will find a bin which will fit a new item with the largest spare capacity left over. The algorithms FF, BF and WF discussed above have a competitive-ratio of 1.7 [12].

First Fit Decreasing(FFD): FFD is the same as FF, but the items are considered in non-increasing order of their sizes. In a similar fashion, *Best Fit Decreasing (BFD)* and *Worst Fit Decreasing (WFD)* can also be defined. All these algorithms FFD, BFD and WFD have a competitive-ratio of 1.22 [12]. Although, this competitive-ratio of 1.22 is the best among all the algorithms, the fundamental requirement of these algorithms, which is non-increasing order of items in list L, may not satisfy the criteria of On-line. Thus, all these algorithms (FFD, BFD and WFD) are generally used as off-line strategies.

2.3.2 Traditional Real-time Scheduling Strategies

As representative examples of traditional real-time scheduling approaches, we have chosen the *Rate Monotonic (RM)* and *Earliest Deadline First (EDF)* algorithms since they have proved to be two of the most widely used techniques over the years and form the foundation upon which most of the real-time scheduling theories have developed.

Rate Monotonic (RM) Algorithm: The RM algorithm [83, 84] is a preemptive, static priority scheduler applicable in a hard real-time environment. It assigns priorities

TH-2256_136101018

to tasks based on their periods; shorter the period, higher the priority. Because the rate of job releases of a task is the inverse of its period, the priority is directly proportional to the task's rate, and hence the name *rate monotonic*.

An important shortcoming of the RM algorithm (as shown by Liu and Layland in [83]) is that even on uniprocessor systems no more than 69% of the processor may be utilized to ensure scheduling feasibility of a set of tasks under rate-monotonic priority assignment in the worst case (when the number of tasks is large (∞)).

Earliest Deadline First (EDF) Algorithm: *EDF* [83,84] is a preemptive, dynamic priority algorithm. It assigns priorities to individual jobs of the tasks according to their absolute deadlines during run-time.

A variation of EDF scheduling is the *Least Laxity First (LLF)* [93] scheduler. In LLF, at each scheduling point, a laxity value ($\mathcal{L}$) is computed for each task and the task having the smallest laxity is assigned the highest priority. Here, the laxity value ($\mathcal{L}_i$) for a task T_i is defined as: $\mathcal{L}_i = d_i - (t - re_i)$, where d_i denotes the deadline of T_i, t the current time and re_i the remaining execution requirement of T_i.

Both EDF and LLF are optimal uniprocessor schedulers. That is, if a set of tasks is unschedulable under EDF or LLF, then no other scheduling algorithm will be able to schedule this task-set. The essential difference between EDF and LLF is that by incorporating the remaining execution requirement of a task in its scheduling decision, LLF also takes into consideration the available flexibility for scheduling a task. However, on multiprocessor systems both these algorithms (as well as RM) prove to be inefficient in terms of the achievable processor utilization in the worst-case [77].

2.3.3 Rate-based Resource Allocation Strategies

Under *rate-based resource allocation* strategies, each task is guaranteed to make progress according to a well-defined rate specification such as "process x subtasks per second" [4, 28, 65–68, 109, 122–124]. The evolution of rate based scheduling schemes can be traced to the problem of supporting multimedia computing, network link scheduling, and other soft and firm real-time applications. It is also emerging to be a promising solution in

TH-2256_136101018

the scheduling of tasks in today's embedded systems running a mix of different kinds of applications with different timeliness constraints such as streaming audio and video, telnet, email, web browsing, etc.

From the perspective of real-time CPU scheduling, three principal classes of rate based resource allocation methods have evolved: *fluid-flow allocation (proportional share schedulers)* [48, 102, 124, 125], *server-based allocation* [3, 4, 118, 119], and *generalized Liu and Layland style allocation* [66, 107]. We will first describe *server-based allocation* and *generalized Liu and Layland style allocation*. Then we will provide a detailed discussion and taxonomy of *fluid-flow allocation methods*, the principal background of our work.

2.3.3.1 Server-based Allocation

Server-based allocation has developed primarily from the necessity of scheduling real-time as well as non real-time aperiodic tasks in a combined fashion. Liu and Deng in [49] proposed a hierarchical scheduling mechanism to support systems consisting of hard, soft and non real-time dynamic tasks coexisting together. For each real-time application, it creates a dedicated *constant utilization server* [50]. All non real-time applications are executed by a separate server. At the top level, the operating system schedules the servers according to the *EDF* algorithm. At the low level, tasks in each application are scheduled by their respective server scheduler according to an algorithm chosen for the application.

In [4], Abeni and Buttazzo describe the *Constant Bandwidth Server (CBS)* scheduling methodology. In this methodology, a portion of the resource's capacity is reserved for processing aperiodic requests. When an aperiodic request arrives, it is assigned a suitable deadline (based on the rate at which the server can serve it) and scheduled using the EDF algorithm. If the server's capacity for executing this task gets exhausted, it is suspended until the next time the server is invoked.

2.3.3.2 Liu and Layland Style Allocation

Liu and Layland's periodic task execution model [83, 84] has been extended in a number of ways to make it more flexible, for example, to allow execution of sporadic tasks within

TH-2256_136101018

the framework. As an example of this allocation style, we discuss below the *Rate Based Execution (RBE)* [66] scheduling model.

Each task in the RBE model is uniquely characterized by a 3-tuple $< x, y, d >$. Here, y denotes an interval of time and each task is guaranteed to execute atleast x jobs within every interval of y time-slots. Each such job will be processed before a relative deadline d. Given the release time t_{ij} of a job J_{ij} (j^{th} job of the i^{th} task), the actual deadline $D_i(j)$ for the job is given by the following recursive definition:

$$D_i(j) = \begin{cases} t_{ij} + d_i & \text{if } 1 \leqslant j \leqslant x_i \\ \max(t_{ij} + d_i, D_i(j - x_i) + y_i) & \text{if } j > x_i \end{cases}$$

Thus, the deadline of a job is the larger of the sum of its release time (t_{ij}) and its relative deadline (d_i) or the sum of the deadline of the x^{th} previous job ($D_i(j - x_i)$) and the y parameter of the task. The first line of this definition implies that up to x consecutive jobs of a task may contend for the processor with the same deadline. The second line implies that deadline for jobs j and $j + x$ must be separated by atleast y time units.

2.3.3.3 Fluid-flow Allocation (Proportional Share Scheduling)

The ideal proportional share allocation model emulates a fluid-flow system based on the assumption that a task can be treated as an infinitely divisible fluid. Each task is assigned a weight (wt_i) determining the minimum bandwidth that the task must be provided. The actual share (shr_i) of the resource that a task T_i must be provided is defined as:

$$shr_i = \frac{wt_i}{\sum_{T_j \in A(t)} wt_j}$$

where $A(t)$ represents the set of active tasks. The definition says that during any time interval $[t_1, t_2]$, a task T_i must execute for $(t_2 - t_1)shr_i$ time units [67]. This ideal model is known as the *Generalized Processor Sharing (GPS)* [2] model. However, any practical system can only implement a discrete approximation of this ideal model because it is not possible to provide each task the exact amount of service that it is entitled

TH-2256_136101018

to receive. The difference between the amount of service that a task should ideally receive and the amount of service that the task has actually received is known as *lag*. It provides a measure of the allocation accuracy. One of the primary goals of any proportional share scheduler is to minimize this lag or provide a bound on its range of variance.

A Taxonomy of Fluid-flow Allocation Methods

Over the years, numerous fluid-flow scheduling algorithms have evolved. Each new algorithm has either been an improvement over its predecessors in terms of implementation ease, fairness accuracy, scheduling complexity, or, has been better equipped to suit a particular domain. Here, we will be discussing a few algorithms which have inspired or influenced this work.

Earliest Eligible Virtual Deadline First($EEVDF$): Given the share shr_i and worst-case execution time e_i of each active task T_i, EEVDF [125] first finds out the relative deadline d_i as follows:

$$d_i = \frac{e_i}{shr_i}$$

This deadline represents the time at which T_i should complete if it receives its exact share. Using these d_i's EEVDF schedules all the *eligible* clients using the *Earliest Deadline First* criterion. Here, a client is considered eligible at any time t provided its *lag* at t is non-negative. In this way, a client that has received more service time than its share is "slowed down", while giving the other active clients the opportunity to "catch up". Although optimal, EEVDF suffers from similar high scheduling complexities as EDF.

Weighted fair Queuing (WFQ): To describe WFQ [48], we first need to explain the terms *Virtual Time (vt(t))* and *Virtual Finish Time (vft(t))*. Virtual Time of a task is a measure of the degree to which it has received its proportional allocation relative to other tasks. Given the integral shares shr_j of each task T_j, the virtual time $vt_i(t)$ of a

TH-2256_136101018

task T_i at time t is defined as:

$$vt_i(t) = \frac{shrp_i(t)}{\sum_{T_j \in A(t)} shr_j}$$

where $shrp_i(t)$ denotes the part of the share that T_i has completed executing at time t and $A(t)$ represents the set of active tasks at time t.

Virtual Finishing Time (vft) is defined as the virtual time the task would have after executing for one time-slot. WFQ schedules tasks by selecting the one having the smallest vft. This is implemented by keeping an ordered queue of tasks sorted from smallest to largest vft, and then selecting the first task in the queue. After a task executes, its vft is updated and it is inserted back into the queue. Its position in the queue is determined by its updated vft. WFQ guarantees that the lag of any task at any point in the schedule is always less than 1. However, the scheduling overhead per time slot is atleast $O(lg\ n)$, where n is the number of tasks.

Lottery: The lottery scheduler [130] was proposed after WFQ mainly to provide an algorithm that would be easier to implement than WFQ. In lottery scheduling, each client is given a number of tickets proportional to its share. Tickets are randomly chosen and the task owning the selected ticket runs for one time quantum. Then the task is reinserted at a suitable position in the queue. In spite of being easier to implement, lottery scheduler also suffers from similar high overheads of atleast $O(lg\ n)$ as WFQ.

Virtual-Time Round-Robin ($VTRR$): $VTRR$ [102] is an amortized O(1) time algorithm providing fairly good proportional fairness in systems with infrequent task arrivals and removals. The algorithm places the tasks in a queue based on the descending order of their share values. Each task is run from the beginning of this queue for one time quantum in a round-robin manner. If a task has received more than its proportional allocation, the remaining tasks in the queue are skipped and tasks are again executed starting from the beginning of the queue. Because tasks with higher share values are kept first in the queue, they get more service than tasks having lower share values which are placed at the end of the queue. The primary problem with this algorithm is that

TH-2256_136101018

it considers only integral share values and cannot provide a bound on the maximum under-allocation that a task may suffer.

Stratified Round Robin (SRR): In SRR [106], the tasks are stratified into a finite number of classes, based on their weights (wt_i). A task T_i falls in class C_k if and only if:

$$\frac{1}{2^k} \leqslant wt_i \leqslant \frac{1}{2^{k-1}}.$$

Given a stratification of the task sets SRR works as a two step scheduler. The first step chooses a particular class to be allocated in the next time quantum based on the principle that if a class C_k has been scheduled in time t, its next scheduling interval will start at time $t + 2^k$. The tasks within a particular class are scheduled in simple round-robin manner. SRR is a low complexity scheduler providing bounded fairness accuracy.

Group Ratio Round-Robin (GR^3): The GR^3 scheme [103] classifies the tasks into groups in a manner similar to SRR. It is also a hierarchical scheduler consisting of two levels: the *inter group* scheduler and the *intra group* scheduler. The inter group scheduler sorts the groups in decreasing order of their weights and schedules them in a fashion similar to $VTRR$. The intra group scheduler allocates tasks within a particular group in round-robin fashion.

Pfair and ERfair Scheduling: The proportional fair schedulers [8, 9, 23, 27, 28, 122] may be considered a spin-off from the research with proportional share schedulers. The typical approach of all these algorithms is the following: each task is effectively divided into a sequence of quantum length sub-tasks. The l^{th} subtask of the j^{th} job of a task T_i is denoted $st_{i,j,l}$. Each subtask has a *pseudo-release* time $pr(st_{i,j,l})$ and *pseudo-deadline* time $pd(st_{i,j,l})$ defined as:

$$pr(st_{i,j,l}) = s_{i,j} + \left\lfloor \frac{l-1}{wt_i} \right\rfloor \quad and \quad pd(st_{i,j,l}) = s_{i,j} + \left\lceil \frac{l}{wt_i} \right\rceil,$$

TH-2256_136101018

where $s_{i,j}$ denotes the release time of the j^{th} job of T_i and wt_i denotes the weight $(\frac{e_i}{p_i})$ of T_i. The scheduling bandwidth or window $(win(st_{i,j,l}))$ for each subtask is given by:

$$win(st_{i,j,l}) \;=\; [pr(st_{i,j,l}),\ pd(st_{i,j,l})].$$

Thus, the window length, denoted by $|win(st_{i,j,l})|$ is:

$$|win(st_{i,j,l})| \;=\; pd(st_{i,j,l}) - pr(st_{i,j,l}).$$

For example, let us consider a task T_1 having execution requirement $e_1 = 8$ and period $p_1 = 11$. Therefore, its weight $\frac{e_1}{p_1}$ is $\frac{8}{11}$. The pseudo-release time and pseudo-deadline time of the third subtask of its second job, say, will be:

$$pr(st_{1,2,3}) = 11 + \left\lfloor \frac{3-1}{\frac{8}{11}} \right\rfloor = 13 \qquad and \qquad pd(st_{1,2,3}) = 11 + \left\lceil \frac{3}{\frac{8}{11}} \right\rceil = 16$$

respectively, and the corresponding window will be denoted by:

$$win(st_{1,2,3}) = [13, 16].$$

The length of its third subtask $|win(st_{1,2,3})|$ will hence be 3 $(16 - 13)$. (Without loss of generality, henceforth we will use the notations pr_{il} and pd_{il} instead of $pr(st_{i,j,l})$ and $pd(st_{i,j,l})$ respectively whenever the pseudo-release and deadlines of the l^{th} sub-task of the current job of a task T_i is referred to.)

At every time-slot, the subtasks are prioritized and executed on an *earliest pseudo-deadline first* basis. If there is more than one subtask having a pseudo-deadline equal to the earliest pseudo-deadline, the choice of the most appropriate subtask to execute in the next time-slot is made using tie-breaking rules. Pfair algorithms such as PF [27], PD [23], PD^2 [8] vary on the use of these tie-breaking rules. The most recent algorithm PD^2 is the most efficient and uses two tie-breaking rules as described below:

1. The first tie-breaking rule is a one bit flag, denoted $b(st_{i,j,k})$. The windows of consecutive subtasks of a task are either disjoint or overlap by one slot. The flag bit $b(st_{i,j,k})$ of a subtask $st_{i,j,k}$ is set to 1 if the windows of $st_{i,j,k}$ and $st_{i,j,(k+1)}$

TH-2256_136101018

overlap. Otherwise, $b(st_{i,j,k})$ is set to 0. Thus, $b(st_{1,1,1}) = 1$ and $b(st_{1,1,8}) = 0$. PD^2 favours subtasks having its b-bits equal to 1, because early execution of a subtask having b-bit = 1, potentially leaves more slots available for its next subtask.

2. The next tie-breaking rule called *group deadline* is required for systems with tasks having windows of length 2. Consider a set of consecutive subtasks of a task having overlapped windows of length 2. If any subtask in this sequence gets scheduled in its last slot, all the further subtasks in the sequence will be forced to execute in their last slots. The tie-breaking rule *group deadline* is applicable to the subtasks in such a sequence and denotes the earliest time by which the sequence ends. The sub-task $st_{1,1,3}$ has a group deadline at time 8. PD^2 favours subtasks with larger group deadlines.

In a Pfair-scheduled system, we define the fairness accuracy in terms of the *lag* between the amount of time allocated to a task and the amount of time that would be allocated to that task in an ideal system with a time quantum approaching zero. Formally, the lag of task T_i at time t, denoted $lag(T_i, t)$ [23], is defined as follows:

$$lag(T_i, t) = (e_i/p_i) * t - allocated(T_i, t),$$

where $allocated(T_i, t)$ is the amount of processor time allocated to T_i in $[0, t)$. A schedule is $Pfair$ iff:

$$(\forall\ T_i,\ t\ ::\ -1\ <\ lag(T_i, t) < 1) \tag{2.1}$$

Informally, the allocation error associated with each task must always be less than one time quantum.

The notion of early-release scheduling [9] is obtained from the definition of $Pfair$ schedule by simply dropping the -1 lag constraint. Formally, a schedule is early-release fair (ERfair) iff:

$$(\forall\ T,\ t\ ::\ lag(T, t) < 1) \tag{2.2}$$

Hence, in an ERfair system, a subtask becomes eligible for execution immediately after its previous subtask completes execution and the system never idles whenever there are

TH-2256_136101018

active tasks ready for execution. An ERfair system is thus work-conserving.

Practical Overheads of Proportional Fair Scheduling:

Pfair and ERfair algorithms are optimal and attain the maximum possible fairness that is practically achievable. However, it suffers from numerous overheads which turn out to be reasonably expensive, especially in real-time systems where time is at a premium.

1. **High scheduling complexity:** Pfair and ERfair schedulers are deadline based algorithms and maintain priority-queues to select the most appropriate subtasks in the next time-slot. This results in a scheduling complexity of at least $O(lg\ n)$ per processor per time-slot, where n is the number of tasks in the system.

2. **Ignorance of task-to-processor mutual affinities:** Proportional fair schedulers are usually ignorant of the affinities between tasks and their executing processors and this generally results in unrestricted inter-processor task migrations and preemptions, thus incurring high overheads.

   Task preemptions primarily result in delay suffered by resumed threads of execution due to compulsory and conflict cache misses while populating the caches with their evicted working sets. Therefore, a processor is affined to the task it executed last because its working set currently exist in cache (and are valid (non-dirty)) and hence, its execution results in cache hits [61, 120, 121, 127]. Although, even after intermediate preemptions, some traces of cache data of a task say T_i, (which executed previously on a given processor) may still remain valid (in that processor's cache), most of T_i's cache contents will typically be swapped out even by a single distinct task executing for just one time-slot between two consecutive executions of T_i. This will be true for all practical time slot lengths and sizes of cache [30, 100].

   Task migration related overheads refer to the time spent by the operating system to transfer the complete state of a thread from the processor where it had been executing to the processor where it will execute next after a migration. Obviously,

TH-2256_136101018

the more loosely-coupled a system, the higher will be this overhead. Task migrations may also effect some cache-miss related overheads, although in general, this is less in comparison to migration related overheads.

Alleviating Overheads of Proportional Fair Scheduling:

The expensive overheads discussed above underline the importance of devising suitable scheduling techniques that attempt to lower the context-switch/scheduling complexity related overheads while simultaneously providing high fairness and resource utilization. Algorithms like BF [140], LLREF [44], NVNLF [55], DP-Fair [78], etc., attempt to lower the number of context switches while concurrently preserving scheduling optimality by enforcing Pfair/ERfair constraints to be satisfied only at task period/deadline boundaries. The approach partitions time into slices, demarcated by the arrivals and departures of all the jobs in the system. Within a time-slice, each task is allocated a workload equal to its proportional fair share. Jobs within a slice may be scheduled using various techniques. For example, in the DP-Fair algorithm, the jobs get executed in the least laxity first (LLF) order. This approach proves to be useful in systems where strict fairness maintenance as done by Pfair/ERfair algorithms is not a necessity and dynamic task arrivals and departures during execution do not occur.

Lowering the $O(lg\ n)$ scheduling complexity barrier in proportional fair schedulers have proved to be a more daunting task. This barrier for uniprocessor systems has been removed through the $O(1)$ frame-based proportional fair algorithm FBPRR [111]. The basic idea behind FBPRR is to define a frame/window of a certain specific size (consisting of a certain number of time-slots) and to allocate shares (of time-slots) to each task in proportion to their weights $\frac{e_i}{p_i}$ within the frame. These shares are executed in VTRR [102] fashion within the frame, thus providing high proportional fairness accuracy within the frame. After execution inside a frame, each task is put in an appropriate future frame such that ERfairness [9] of the system remains preserved at frame boundaries. Experimental results [111] using this scheme showed that speedups of 5 to 20 times can be obtained (over $O(lg\ n)$ complexity schedulers) with high fairness accuracy.

TH-2256_136101018

2.3.4 Energy-Aware Scheduling strategies

Energy has now-a-days become a critical resource in all battery operated devices. Reduction of energy consumption is essential to prolong the battery life in these systems. With the advent of processors such as the 45 nm technology based Intel T9400 (dual core), the ARM Cortex-A9 MPCore (supporting 2 to 4 cores), Pluralitys Hypercore processor (capable of supporting 16 to 256 cores), etc., embedded systems such as the handheld devices like mobiles, PDAs, laptops, etc. are quickly moving into the (single chip) multiprocessor domain. As discussed earlier, the DVFS mechanism decreases dynamic energy consumption by reducing the voltage/frequency of a processor to the minimum value that is sufficient to handle a given workload. Authors in [86] apply DVFS as an energy harvesting mechanism that utilizes task slacks to lower operating frequencies in uniprocessor embedded systems. For multiprocessor platforms, a DVFS technique named Deterministic Stretch to Fit (DSF) based on the principle of inter-task slack shareability has been proposed in [32]. The authors have firstly proposed an online algorithm to reclaim energy by adapting to the variations in actual workloads of target application tasks. Secondly, they have extended the algorithm with an adaptive and speculative speed adjustment mechanism. This mechanism anticipates early completion of future task instances based on the information of their average workload. Thirdly, they have proposed a separate slowdown technique for situations in which number of tasks present in the ready queues of a system is less than or equal to the number of processors. An offline DVFS technique has been proposed in [54] for systems in which voltages and frequencies may be controlled uniformly (in a continuous fashion) and independently among the available processors. This technique needs the scaling to be controlled very minutely which may not be always possible. Authors in [82] have proposed task scheduling algorithms that leverage per core DVFS and achieve a balance between performance and energy consumption. They considered two task execution modes: the batch mode, which runs jobs in batches; and the online mode in which jobs with different time constraints, arrival times, and computation workloads co-exist in the system. Operating systems in smartphones employ interval-based dynamic voltage scaling algorithms to

TH-2256_136101018

reduce power consumption. To boost the effect of those algorithms, an application-level scheduling algorithm has been proposed in [74], which slows down the execution of applications deliberately and thus maintains low utilization rate of CPU. A hardware implementation of an energy-aware task-processor allocation algorithm for multiprocessor systems has been proposed in [104]. This algorithm endeavours to schedule and map a set of real-time precedence constrained tasks with the objective of temperature monitoring and control in safety critical applications. In [56], authors have presented an energy aware optimal scheduling approach based on the deadline partitioning oriented T-N plane abstraction technique. Authors have proposed a non-uniform multiprocessor frequency scaling algorithm for real-time systems called Growing Minimum Frequency (GMF) in [94]. It takes advantage of recent developments in multiprocessor scheduling when processors can be assigned different speeds. A task scheduling method using a combination of cultural and ant colony optimization algorithms for a cloud environment, has been proposed in [19]. In their work, the authors have tried to minimize the scheduling makespan as well as energy consumption of the system, using their hybrid approach. In [5], the authors propose an energy-aware scheduling algorithm called EAGS, which aims at minimising the computing-energy consumption in decentralised multi-cloud systems using DVFS. Shojafar et al. [116] have proposed a traffic engineering based adaptive approach to dynamically reconfigure the computing-plus-communication resources for real-time service data centers. Using this approach, the authors have tried to maximize the energy-efficiency of the system, while meeting requirements on the delivered transmission rate and processing delay.

With the exponential growth in chip transistor densities over technology generations, static energy dissipation due to leakage drain from transistors has steeply increased over the years [71]. Today, static energy dissipation has already become the major source of power wastage within a chip. Static energy wastage is mainly controlled using DPM, where a processor is put into inactive low power suspension/sleep states by procrastinating task executions while simultaneously guaranteeing their timely completion. DPM techniques try to minimize static energy dissipation in the system by putting a processor

TH-2256_136101018

in low-power suspension/sleep mode for as long as possible while still guaranteeing the tasks' timing constraints. Awan and Petters [14] proposed an energy efficient slack management approach to minimize leakage energy consumption for mixed-criticality uniprocessor systems. They presented this approach for dynamic priority systems with multiple sleep states. Bhatti et al. [31] have proposed a DPM based energy-aware strategy for global multiprocessor systems called Assertive Dynamic Power Management (AsDPM). AsDPM first determines the minimum number of active processors needed to fulfill the execution requirement of released jobs at runtime. Then it attempts to cluster the distributed idleness existing on a subset of the active processors into longer continuous idle intervals, so that these obtained intervals may be employed to switch some of the processors to deeper low-power states for a longer duration of time. The problem of online dynamic power management that provides hard real-time guarantees for multiprocessor systems has been considered in [41]. Legout et al. [76] presented a DPM based online scheduling scheme by extending the existing approach *Fixed Priority until Zero Laxity (FPZL)* in order to handle both hard real-time and mixed criticality systems.

There are also existing works which combine both DVFS and DPM techniques together to achieve better energy savings in the system. These approaches provide higher energy savings because the slowdown caused by DVFS also increases static leakage power. Therefore, for any given technology, there exists an optimal frequency called *critical frequency* [69] at which the processor should be clocked to minimize overall energy consumption of the system. In [42, 69], combined slowdown-suspension techniques for real-time systems with fixed-priority tasks have been considered. Total system-wide energy minimization mechanisms using dynamic slack reclamation has been proposed by Jejurikar et al. in [70]. In [40], a technique has been proposed to directly model the idle intervals of individual cores such that both DVFS and DPM can be optimized at the same time. Based on this technique, the energy optimization problem has been formulated by means of mixed integer linear programming. By utilizing both DVFS and DPM techniques, authors in [39] proposed a *Mixed Integer Linear Programming* (MILP) formulation for energy optimization in real-time multi-core systems. Ejiali et al. proposed

TH-2256_136101018

a standby-sparing based energy-aware fault-tolerant scheme for the scheduling of aperiodic tasks on dual-cores [51]. In order to reduce energy consumption, they employed DVFS for the main unit and DPM for the spare. Later, this work has been extended by Haque [63] et al. for periodic real-time applications. Recently, Guo et al. generalised the work presented in [63] for real-time systems deployed on platforms consisting of more than two processing cores [60]. However, all these works [32, 39, 51, 60, 63] have been targeted towards multi-core platforms consisting of *homogeneous* cores.

Over the years, homogeneous platforms have slowly given way to heterogeneous multi-core platforms in order to satisfy strict performance demands, often along with additional constraints on weight, size, power, etc. An energy-aware partitioning and scheduling algorithm for standby-sparing systems has been presented in [133]. It is essentially a primary-backup approach for dual-core platforms where the primaries and back-ups of tasks are always mapped to distinct cores. Given two-cores V_1 and V_2, the primaries of all tasks which require shared resources are mapped to one of the cores (say, V_1), while the primaries of the remaining tasks which do not require shared resources are mapped to the other core (say, V_2). Another scheduling technique for standby-sparing systems has been proposed in [91]. They presented a DVFS based energy-aware strategy to schedule fixed-priority real-time tasks. Although these strategies have attempted to address the problem of energy-aware scheduling in heterogeneous platforms, they restricted the number of processing core types to only two.

In recent years, many research works have been focused on generic heterogeneous platforms, having an arbitrary number of processor types. Authors in [128] proposed two heuristics based on the *Earliest Deadline First* (EDF) algorithm to minimize energy consumption in heterogeneous multi-core platforms. Another energy-aware partitioning strategy called the *Least Loss Energy Density* (LLED) Algorithm has been proposed in [15]. Although this strategy is able to use DVFS and DPM mechanisms to reduce energy consumption in the system, mapping of tasks onto processors is computationally expensive and this restricts its practical applicability. The problem of scheduling independent tasks with stochastic execution times and energy consumption on a heteroge-

TH-2256_136101018

neous platform under deadline and energy budget constraints, has been studied in [79]. A two-phased energy-aware scheduler for heterogeneous systems has been proposed in [16]. In the first phase, the authors presented a DVFS based task-to-core allocation strategy called *MaxMin* and in the next phase, they further refined the allocation to achieve sleep states and better energy savings. In [17], the authors have addressed the problem of task-to-core allocation onto heterogeneous multi-core platforms such that the overall energy consumption of the system is minimized. To this end, they proposed a DVFS-cum-DPM based two-phased approach. In the first phase, tasks have been allocated to cores such that dynamic energy consumption is reduced, while the second phase refines the prior task allocation in order to achieve better sleep states by trading off dynamic energy consumption with the reduction in leakage energy consumption.

Although, all the scheduling strategies discussed above attempt to enhance energy-efficiency of the system, most of them don't consider fairness in the relative execution progress of tasks as an important parameter in their scheduling decisions. However, with more and more energy constrained embedded devices running interactive and QoS sensitive applications such as continuous media, gaming, etc., the need for energy-aware proportional fair scheduling algorithms is quickly gaining importance.

2.3.5 Temperature-Aware Scheduling strategies

Thermal hotspots greatly influence reliability and performance of a system. Hence, thermal management has become a subject of great interest for researchers and practitioners during the past few years. Most of the work on this topic falls under the following two categories: i. Peak temperature minimization, and ii. Optimizing one or more parameters such as throughput maximization, energy dissipation, etc., under a stipulated thermal threshold. In [132], the authors have used a task partitioning algorithm called MinMin [53] in order to perform task-to-core assignments. Then, they presented two scheduling schemes to reduce hot-spots in multi-core embedded systems. In the first scheme, they use a task splitting strategy within each core, to divide each task into multiple subtasks and then interleave executions of generated subtasks based on their temperature characteristics. In the second scheme, they propose an adjustment

TH-2256_136101018

strategy over the first scheduling scheme to minimize imbalance in spatial temperature distribution among cores. Another partition oriented technique to minimize peak temperature in a multiprocessor system via task allocation and splitting, has been proposed in [139]. The proposed solution exploits thermal characteristics of tasks and processors to prepare a task-to-processor mapping and then applies a task splitting technique that enables alternation of task execution and idle slack time to reduce the peak temperature in the system. In [59], Ghahfarokhi et al. have proposed a partition based technique that reduces on-chip temperature gradients via task migration in distributed embedded systems, while satisfying real-time constraints. If the temperature of a processor exceeds a certain threshold at a given instant, all tasks from this overheated processor are swapped with tasks from the coldest processor, before execution is allowed to proceed further.

The above works have primarily attempted to minimize a system's operational processing core temperatures. Distinct from these, the second category of works is more directly targeted towards ensuring DTM free operation which necessitates the maintenance of the peak temperature within a stipulated threshold. Authors in [138] propose a temperature constrained partition oriented scheduling approach which attempts to minimize makespans of real-time tasks in multi-core environments. They presented a heuristic strategy, which determines assignment, replication, task execution sequence and processor operating frequencies. In [64], Huang et al. discuss two throughput enhancement approaches for single processor systems with stipulated thermal constraints. In the first approach, they shuffle the given processor between sleep and active modes to meet their objective. In the second approach, they recommend a scheduling strategy considering the task's thermal characteristics as well as the processors' active/sleep modes. A temperature threshold aware global multiprocessor scheduling approach, which tries to maximize the total number of completed jobs has been proposed in [34]. The proposed work also analyzes the competitive ratios of a large class of algorithms as a function of the cooling factors of processors. In [20], authors have addressed both the problems of temperature minimization and makespan minimization under a stipulated thermal con-

TH-2256_136101018

straint. They studied the temperature-aware scheduling problem with a model where jobs of unit-lengths and given temperature characteristics have to be scheduled on a set of identical processors. As discussed earlier, high chip temperatures result in higher power consumption as well as a reduced performance in systems. There are also a few works [81,92,136,137] showing the interdependence of operating temperatures and power consumption in a system.

Although, the above strategies have concentrated on different aspects related to temperature-aware scheduling but most of them have not adequately focussed towards resource usage efficiency/achievable resource utilization, which is a very critical design criterion towards the cost effective design of real-time systems.

2.4 Summary

This chapter started with a brief overview about the basic terms and definitions of real-time systems followed by literature survey of various kinds of scheduling algorithms for uniprocessor and multiprocessor real-time systems. These concepts and definitions may either be referred or reproduced appropriately later in this thesis. In the next chapter, we present two energy-aware scheduling strategies namely *EAFBFS* and *DPFair-SS*, developed by us for homogeneous multi-core systems. While EAFBFS applies the DVFS mechanism to reduce energy consumption in a system, DPFair-SS uses a DVFS-cum-DPM mechanism to achieve the same.

TH-2256_136101018

TH-2256_136101018

Chapter 3

Energy-Aware scheduling on homogeneous multi-core systems

With advancement of technology, newer classes of devices like mobiles, laptops, PDAs, etc. have emerged, which depend upon battery as their primary source of energy. Being battery operated, efficient energy usage has become a critical design constraint in all these devices. Hence, a lot of research has been conducted towards their power management at various levels of abstraction, starting from hardware and firmware to architectural, system and even application levels. As stated earlier, two primary mechanisms which are generally used at the operating system level to reduce energy consumption are: (1) Dynamic Voltage and Frequency Scaling (DVFS) [18, 69, 115] and, (2) Dynamic Power Management (DPM) [13, 21, 69]. The DVFS mechanism decreases dynamic energy consumption by reducing the voltage/frequency of a processor to the minimum value that is sufficient to handle a given workload. On the other hand, the DPM mechanism tries to minimize static energy dissipation in the system by putting a processor in low-power suspension/sleep mode for as long as possible while still guaranteeing the tasks' timing constraints. In this chapter, we propose two semi-partitioned energy-aware strategies for scheduling periodic tasks in homogeneous multi-core systems. While the first strategy named *Energy Aware Frame Based Fair Scheduler (EAFBFS)* tries to minimize dynamic energy consumption in a system by scaling down the frequencies of one or more core, the second strategy named *DPFair Scheduling with Slowdown and Suspension (DPFair-SS)* applies both DVFS and DPM in a coordinated manner to minimize

TH-2256_136101018

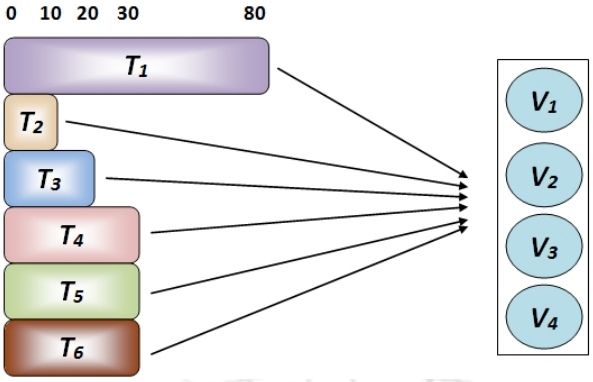


Figure 3.1: *Motivational Example*

both static and dynamic energy dissipation. The second strategy is more suitable in situations when the system experiences low workloads over significantly long durations. A few important terminologies used in the later sections of the chapter have been listed in Table 3.1. Next, we present our problem statement with help of an example.

3.1 Motivational Example

Consider a list of 6 tasks T_1, T_2, T_3, T_4, T_5 and T_6 having execution requirements 80, 10, 20, 30, 30 and 30, to be scheduled within 100 time-slots, in a system of $m = 4$ cores as shown in Figure 3.1. The scheduling strategy in the system should not only be able to allot and schedule tasks on the available cores but it has to do it in such a way that energy consumption is minimized, which makes it a challenging problem. In the following sections, we present the working principle of *EAFBFS* and *DPFair-SS* algorithms, which are able to propose solution for such problems.

3.2 Energy Aware Frame Based Fair Scheduling (EAFBFS)

In this work, we propose a new homogeneous multi-core scheduling strategy for hard real-time systems called *Energy Aware Frame Based Fair Scheduling* (EAFBFS). EAFBFS incorporates a DVFS mechanism into it's scheduling strategy to make the scheduling algorithm energy-aware. Being proportionally fair, the algorithm allows full resource

TH-2256_136101018

utilization similar to the ERFair [9] strategy. Given a set of n tasks, where each task T_i has an execution requirement e_i and period p_i, ERFairness is defined in terms of a parameter called *lag*, which is the difference between the amount of time that has been actually allocated to a task and the amount of time that would be allocated to it in an ideal system with a time quantum approaching zero. Formally, the *lag* of task T_i at time t is defined as follows:

$$lag(T_i, t) = \frac{e_i}{p_i} \times t - allocated(T_i, t) \tag{3.1}$$

where $allocated(T_i, t)$ is the amount of processor time allocated to T_i in $[0, t)$. Equation 3.1 has been taken from [18]. A schedule is said to be *ERFair* iff:

$$(\forall T, t :: lag(T, t) < 1) \tag{3.2}$$

That is, the under-allocation associated with each task must always be less than one time quantum. The algorithm guarantees that the above condition (Equation 3.2) can be satisfied whenever $\sum_{i=1}^{n} wt_i \leq m$, where m denotes the number of cores in the system. However, ERFair is a global scheduling algorithm and often incurs significantly high overheads due to inter-core task migrations and preemptions. On the other hand, by employing a semi-partitioned scheduling approach, EAFBFS is able to significantly restrict migration and preemption overheads compared to ERFair. In addition to it, EAFBFS also offers tunability against varying fairness accuracy demands.

In the next section, we explain the specifications needed to describe our work. Section 3.2.2 presents the proposed algorithm along with it's pseudo-code and illustrative examples. The theoretical analysis of the algorithm is discussed in Section 3.2.3 followed by experiment set up and results in Section 3.2.4.

3.2.1 Specifications

3.2.1.1 System Model

We have considered two types of homogeneous multi-core systems based on their frequency scaling abilities. *Type 1*: All cores must execute at the same system-wide frequency and modifications in frequency may only occur synchronously across all cores

TH-2256_136101018

Symbol	Description
n	Number of tasks
T_i	i^{th} task
s_i	Start time of i^{th} task
e_i	Execution requirement of i^{th} task
p_i	Period of i^{th} task
re_i	Remaining execution requirement of T_i
rp_i	Remaining period of T_i
wt_i	Weight of i^{th} task
shr_i	Share of i^{th} task
t	Current time
TS_r	r^{th} time-slice
U	Utilization Factor in the system
$fr_{critical}$	Normalized critical frequency
f_{max}	Maximum value of normalized operating frequency
f_i	i^{th} frequency level
fr_{T_i}	Minimum normalized frequency sufficient to complete T_i on a single core in a time-slice
TD	Set of tasks for which $fr_{T_i} > fr_1$
m	Number of cores in the system
V	Set of cores
V_k	k^{th} core
fr_1	Optimum operating frequency selected in a time-slice
fr_g	Global Operating frequency selected in a time-slice

Table 3.1: *Important Terminologies*

(e.g. the light weight processors ARC HS45D and HS47D from Synopsys ™). *Type 2*: Different cores are allowed to execute at distinct speeds and their operating frequencies may be modified asynchronously and independent of other cores (e.g. All modern processors allow operating frequencies to be modified asynchronously). Thus, if two cores V_1 and V_2 in a *type 2* uniform multi-core system execute at speeds f and f/k respectively, a task executing on V_1 completes k times the amount of work on V_2, within any given time interval.

3.2.1.2 Power Model

In a system that allows DVFS, the power consumed P is directly proportional to the operating frequency fr and the square of the supply voltage ν (i.e $P \propto fr \times \nu^2$). As the operating frequency is again linearly proportional to the voltage ν, so the expression for the power consumed may be approximately represented in the following simplistic form:

TH-2256_136101018

$$P_{fr} = c \times fr^3 \tag{3.3}$$

where, fr represents the operating frequency of the core, P_{fr} represents the power consumption in the system with operating frequency fr and c denotes the constant of proportionality. Fractional power saved in a DVFS based system operating on a frequency fr is given by:

$$P = \frac{P_{f_{max}} - P_{fr}}{P_{f_{max}}} \tag{3.4}$$

where, f_{max} is the maximum value of normalized frequency available in the system. Since, we consider normalized frequencies in our calculations, the power dissipation obtained is also in normalized form.

3.2.2 EAFBFS Scheduling Strategy

The *Energy Aware Frame Based Fair Scheduling (EAFBFS)* scheduling strategy starts by calling the basic EAFBFS function (Algorithm 1). The framework for the proposed strategy has been designed as a two-level hierarchical architecture. These two levels together attempt to allocate the CPU resources of a system to a given set of tasks in such a way that the cores may be executed at minimum possible frequencies while ensuring that the fairness in the execution rates of all tasks always remain above a pre-specified tolerance bound. At the outer level, time is partitioned into *frames*/*slices* (r^{th} time-slice is denoted by TS_r) demarcated by all the deadlines of all tasks in the system, similar to DPFair [57, 78].

The principal component at the outer level of the two-level split architecture of EAFBFS consists of the global *Frequency Allocation and Mapping* (*FAM*) module, which executes at the boundary of each frame/time-slice. At the inner level within each individual core, separate ERFair schedulers are applied to execute tasks within each core, inside a time-slice. We now discuss the EAFBFS algorithm in detail.

TH-2256_136101018

ALGORITHM 1: Algorithm EAFBFS

```
1 k ← 0; time ← 0
2 while true do
3 |   Determine length |TS_r| of the next time-slice TS_r
4 |   for each task T_i do
5 |   |   Determine shr_i using Equation 3.6
6 |   Call function FAM()
7 |   for each core V_k (1 ≤ k ≤ m) in parallel do
8 |   |   Call function Schedule()
9 |   k ← k + 1; time ← time + |TS_r|
```

3.2.2.1 Algorithm EAFBFS

At the start of the r^{th} time-slice TS_r, EAFBFS determines the size $|TS_r|$ of the ensuing time-slice. This is accomplished by finding the earliest among the remaining periods of all task instances.

$$|TS_r| = min\{rp_1, rp_2, ..., rp_n\} \tag{3.5}$$

Next, we find the execution shares that each task will have in the ensuing time-slice. This share for a task T_i is calculated as :

$$shr_i = \lceil \frac{re_i}{rp_i} \times |TS_r| \rceil \tag{3.6}$$

All tasks having a share of at least 1 are considered for execution in the current time-slice (Remaining tasks (if any) are considered for execution in appropriate future time-slices). The shares for all these tasks must be completed within the time-slice duration $|TS_r|$. Then algorithm EAFBFS calls the FAM function to partition the task set into the available cores and also determine the minimum frequency for each core, so that the workload imposed by the task set may be appropriately handled. Then EAFBFS calls a separate ERFair scheduler for each core. The local scheduler supervises the execution of tasks allocated to the core for the ensuing time-slice. The pseudo-code for the algorithm EAFBFS has been presented in Algorithm 1.

TH-2256_136101018

3.2.2.2 Frequency Allocation and Mapping (FAM)

The *Utilization factor* of the system in the time-slice can be calculated using the equation:

$$U = \frac{\sum_{i=1}^{N_{TS_r}} shr_i}{m \times |TS_r|} \tag{3.7}$$

where, N_{TS_r} represents the number of tasks to be executed in the commencing time-slice.

Task Share Adjustment: *If the value of* U *(Equation 3.7) exceeds 1 due to rounding of task shares (in Equation 3.6), then a subset of* η $(= \sum_{i=1}^{N_{TS_r}} shr_i - |TS_r|)$ *tasks with the highest rounding factors* $(r_i = shr_i - \lfloor (re_i/rp_i)|TS_r| \rfloor)$ *are selected and their shares are decremented by 1. This operation helps avoid overloads in the ensuing time-slice* TS_r *while ensuring maintenance of ERFairness at time-slice boundaries.*

Now, FAM determines the minimum frequency among the set of available optional frequencies that is sufficient to successfully complete the execution of the shares of all tasks within $|TS_r|$ time-slots. It may be may be derived as:

$$fr_1 = \overline{\sum_{i=1}^{|T|} shr_i/(m \times |TS_r|)} \tag{3.8}$$

Since $f_{max} = 1$ (considering normalized frequencies), the bar ($^-$) in the above equation denotes the nearest available frequency fr_1 (from the set F of available frequencies) higher than $\sum_{i=1}^{|T|} shr_i/(m \times |TS_r|)$.

The frequency fr_{T_i} that is sufficient to execute any task T_i when it executes alone on a dedicated core for the entire duration of the time-slice, is given by:

$$fr_{T_i} = \frac{shr_i}{|TS_r|} \tag{3.9}$$

where, shr_i is it's share at maximum frequency and $f_{max} = 1$.

All tasks T_i, whose fr_{T_i} value is higher than fr_1 will not be able to complete execution of it's entire share if the system runs at the minimum obtained frequency fr_1. However, in a *type 1* system all cores are restricted to operate at a single global operating frequency

TH-2256_136101018

ALGORITHM 2: Function FAM()

```
1  Find Utilization Factor U of TS_r using Equation 3.7
2  Adjust task shares, if required
3  Determine the minimum system level operating frequency fr_1 using Equation 3.8
4  if System is type 1 then
5  |  Find fr_g using Equation 3.10
6  else
7  |  Allocate dedicated cores to the set TD = {T_ρ1, T_ρ2, ..., T_ρ|TD|} of tasks for
   |    which fr_Tρi > fr_1
8  |  Determine fr_h, the execution frequency for the remaining m − |TD| cores
   |    using Equation 3.11
9  Update shares of each task using Equation 3.12
10 Call function Task_Partitioning() to allocate all tasks in a type 1 system
     (n − |TD| tasks in a type 2 system) onto the m (m − |TD|)
11 Adjust core frequencies to satisfy prescribed under-allocation bounds for fixed
     tasks in the presence of migrating tasks, using Equation 3.18
12 Adjust task shares accordingly
```

fr_g. This global frequency may therefore be obtained as:

$$fr_g = max\{fr_{T_1}, fr_{T_2}, ..., fr_{T_{N_{TS_r}}}, fr_1\} \tag{3.10}$$

A *type 2* system relaxes the above restriction of a single global operating frequency and allows cores to run at distinct frequencies. Therefore, each task T_i for which $fr_{T_i} > fr_1$ may now be allocated to a separate dedicated core which will operate at frequency fr_{T_i} for the entire time-slice duration $|TS_r|$. Let $TD = \{T_{\rho_1}, T_{\rho_2}, ..., T_{\rho_{|TD|}}\}$ represent the set of tasks for which $fr_{T_{\rho_i}} > fr_1$. When all tasks in TD are allocated to dedicated cores and the rest of the tasks are allotted the remaining $m - |TD|$ cores, the minimum frequency at which these $m - |TD|$ cores must execute is given by:

$$fr_h = \overline{\frac{\sum_{i=1}^{|T|} shr_i - \sum_{i=1}^{|TD|} shr_i}{(m - |TD|) \times |TS_r|}} \tag{3.11}$$

Since the shares of the tasks were initially calculated with respect to f_{max}, they must be recomputed with respect to the currently modified core frequencies. For a given modified core frequency fr_x and initial share shr_i of a task T_i, the updated share value of T_i may be obtained as:

$$shr_i = \overline{shr_i / fr_x} \tag{3.12}$$

TH-2256_136101018

The pseudo-code for the function FAM() has been shown in Algorithm 2.

Example 1: Consider the problem which was presented in the motivational example (Section 3.1). All the tasks have to be scheduled within a time-slice TS_r of size $|TS_r| = 100$. From Equation 3.8, $fr_1 = \lceil(80+10+20+30+30+30)/(4\times100)\rceil = 0.5$. However, for T_1, the task with the maximum share value, $fr_{T_1} = 0.8$ (refer to Equation 3.9). Hence, for a *type 1*, system $fr_g = 0.8$ (refer to Equation 3.10). Therefore, an estimate of the percentage fractional power saved in a core (from Equation 3.4) is given by: $P = (1 - 0.8^3) \times 100 = 48.8\%$. So, relative power saved in a DVFS based *type 1* system: $P = (48.8 + 48.8 + 48.8 + 48.8)/4 = 48.8\%$.

On the other hand, in a *type 2* system, since T_1 has a higher frequency demand ($fr_{T_1} = 0.8$) than the optimal operating frequency ($fr_1 = 0.5$), it is scheduled on a dedicated core V_1 with operating frequency 0.8. For rest of the cores, operating frequency (as derived in Equation 3.11) $fr_h = (10+20+30+30+30)/(3\times100) = 0.4$. Therefore, percentage of fractional power saved in V_1 (from Equation 3.4) is: $(1 - 0.8^3) \times 100 = 48.8\%$ while the savings in other cores is: $(1 - 0.4^3) \times 100 = 93.6\%$. Therefore, the overall percentage of fractional power saved in a *type 2* system is: $P = (48.8+93.6+93.6+93.6)/4 = 82.4\%$. ■

Task Partitioning: After the initial frequency calculation and assignment step, all tasks in a *type 1* system and those which have not already been allocated to dedicated cores in a *type 2* system are partitioned into the currently available cores using a two-phased scheme. The first phase partitions the task set into disjoint subsets using the *Worst-Fit Decreasing* (WFD) bin packing algorithm such that the sum of task shares in each such set is less than the time-slice size $|TS_r|$. These tasks are inserted into the priority queue of the cores, where they have been partitioned. Those tasks which cannot be accommodated into the remaining capacity of a single core are moved to a separate list $\wedge_{mgr}$ in *phase 1*. At the beginning of *phase 2*, the list $\wedge_{mgr}$ contains all tasks which

TH-2256_136101018

cannot be entirely executed by any single core in non-increasing order of their share values. These tasks must be appropriately split and executed using the combined capacities of more than one core. This forms the set of migrating tasks in the system. In the second phase, the splitting and allocation of migrating tasks are carried out. All the cores are maintained in non-increasing order of their remaining spare capacities. When a migrating task T_i from the list $\wedge_{mgr}$ cannot be fully accommodated within the core, say V_k, having the highest spare capacity sc_k among all available cores, T_i is allotted to V_k with core share sc_k and the remaining share of T_i is then tried to be allocated to the next core having the highest spare capacity. Each migrating task is inserted only into the priority queue of the core, where it is partitioned first. It helps to avoid parallel execution of migrating tasks at multiple cores. This process continues till T_i is fully allotted. The core where T_i is allotted last, may have some leftover spare capacity after the allocation of T_i. In that case, the allocation of the next task from the list $\wedge_{mgr}$ starts with this core. The pseudo-code for the function Task_Partitioning() has been shown in Algorithm 3.

Example 2: Continuing with the previous example, let us discuss the task partitioning and mapping scheme presented above, considering a *type 1* system. Before this step, the shares of the tasks are recalculated with respect to the system level operating frequency $fr_g = 0.8$ (as calculated using Equation 3.10 in Example 1). The updated share values obtained using Equation 3.12 are: $shr_1 = 100$, $shr_2 = 13$, $shr_3 = 25$ and $shr_4 = shr_5 = shr_6 = 38$. At the partitioning step, the tasks are arranged in non-increasing order of their respective share values and allocated to the cores using *WFD bin packing* algorithm. The task allocation using WFD may be enumerated as: $T_1 \rightarrow V_1$, $T_4 \rightarrow V_2$, $T_5 \rightarrow V_3$, $T_6 \rightarrow V_4$, $T_3 \rightarrow V_2$ and $T_2 \rightarrow V_3$.

In a *type 2* system, T_1 has been allotted the dedicated core V_1 with operating frequency of 0.8 (refer Example 1). It's modified share shr_1 is 100 (Equation 3.12). For the rest of the cores, the required operating frequency is 0.4. Hence, modified share values for the rest of the tasks are as follows (from Equation 3.12): $shr_2 = 25$, $shr_3 = 50$

TH-2256_136101018

ALGORITHM 3: Function Task_Partitioning()

```
1  {PC[m]: Array of core-ids sorted in non-decreasing order of spare capacities}
2  {cap[m]: Array of remaining core capacities}
3  {η: List of migrating tasks, η = {T_η1, T_η2, ...}}
4  {mflag_k: Flag which is set to 1 during the interval when a migrating task
     executes on V_k}
5  Allocate all fixed tasks into available cores using WFD bin packing algorithm
6  l ← 1; k ← PC[l]
7  for Each task T_ηi in η do
8      {ushr^r_ηi: Unallocated share of T_ηi in TS_r}
9      while ushr^r_ηi > cap[k] do
10         {TP_ηi: List of cores into which T_ηi gets partitioned}
11         TP_ηi ← TP_ηi ∪ V_k
12         if |TP_ηi| = 1 then
13             Set mflag_k ← 1 to indicate that a migrating task will execute on V_k
                 from start of time-slice
14         For all fixed tasks in V_k, calculate their weights when T_ηi executes on V_k
             (using Equation 3.20)
15         ushr^r_ηi ← ushr^r_ηi − cap[k]
16         l ← l + 1; k ← PC[l]
17     cap[k] ← cap[k] − ushr^r_ηi
18     TP_ηi ← TP_ηi ∪ V_k
19 Find UT^k_max for each core V_k
20 UT ← max{UT^1_max, UT^2_max, ..., UT^m_max}
```

and $shr_4 = shr_5 = shr_6 = 75$. The task mapper now allocates these shares into the remaining available cores. Using *WFD bin packing* algorithm, the allocation becomes: $T_4 \rightarrow V_2$, $T_5 \rightarrow V_3$, $T_6 \rightarrow V_4$. After this, none of the available cores have enough space left to allocate T_3. Hence, T_3 is added to $\wedge_{mgr}$. Again, we resume the task allocation process to allocate T_2 to V_2. Now in Phase 2, the task T_3 from $\wedge_{mgr}$ is split into two parts. The first part with share value 25 is allocated to the remaining capacity of V_3 and the second part with remaining share value of 25 is allocated to V_4. ■

Handling Potential Under-Allocation of fixed tasks: From the partitioning and mapping strategy presented above, it may be observed that a particular core in the system may contain a set of fixed tasks along with either: no other migrating tasks, one non-terminating migrating task, one terminating migrating task, one non-terminating

TH-2256_136101018

migrating task along with one terminating migrating task. Among these, in the scenarios in which a core contains a non-terminating migrating task, the fixed tasks may be transiently under-allocated within the time-slice. This happens because the migrating task must execute at it's own stipulated rate in all the cores it partially executes so that it can successfully complete execution of it's required share by the end of the time-slice.

Without loss of generality, assuming a core V_k to contain Z_k fixed tasks along with a migrating task T_1 which gets scheduled into $\#a$ cores within the ensuing time-slice, the fraction of T_1's share to be executed on V_k is : $(|TS_r| - \sum_{j=2}^{Z_k+1} shr_j)/shr_1$. Let shr_1^k denote the shares of T_1 allotted on V_k. The number of time-slots NTS_k by which T_1 must complete executing it's fraction of shares at V_k is given by:

$$NTS_k = |TS_r| \frac{|TS_r| - \sum_{j=2}^{Z_k+1} shr_j}{shr_1} = \frac{|TS_r| \times shr_1^k}{shr_1} \tag{3.13}$$

Since, T_1 executes on core V_k only after it finishes its execution sequentially on $V_1, V_2, ..., V_{(k-1)}$, the total time elapsed from the start of the time-slice to the termination of T_1 on V_k is given by:

$$TE_k = \frac{|TS_r|}{shr_1} \sum_{x=1}^{k} shr_1^x \tag{3.14}$$

Example 3: Continuing with the previous example, let us discuss the handling of the migrating and fixed tasks in the system. In the *type 1* system we considered, there were no migrating tasks, i.e. all tasks were allocated to single cores only. But in the *type 2* system, migrating task T_3 was allotted on V_3 and V_4 with $shr_3^3 = shr_3^4 = 25$. V_3 also has a fixed task T_5 allotted on it with a share of 75 in the current time-slice. The number of time-slots by which T_3 must complete executing its fraction of shares on V_3 (from Equation 3.13) is: $NTS_3 = (100 \times 25)/50 = 50$. Also, since V_3 is the first core on which T_3 will execute, $TE_3 = NTS_3 = 50$ and $TE_4 = 100(25 + 25)/50 = 100$. ■

As T_1's share on V_k is shr_1^k, all tasks in V_k would execute at their specified rates $shr_i/|TS_r|$ where $2 \leq i \leq Z_k + 1$, if T_1 would execute at a rate $shr_1^k/|TS_r|$ throughout the time-slice interval. At this rate, T_1 would complete $(shr_1^k \times TE_k)/|TS_r|$ time-slots of execution within the interval TE_k. However, T_1 actually executes at a faster rate

TH-2256_136101018

and completes executing its whole share shr_1^k in this interval. So, the extra time-slots executed by T_1 during TE_k is given by $shr_1^k - ((shr_1^k \times TE_k)/|TS_r|)$ which is equivalent to the sum of under-allocations suffered by the fixed tasks on V_k. It can be rewritten as:

$$UT_{sum}^k = shr_1^k - \frac{shr_1^k \times TE_k}{|TS_r|}$$

$$\Rightarrow UT_{sum}^k = shr_1^k[1 - \frac{\sum_{x=1}^{k} shr_1^x}{shr_1}] \tag{3.15}$$

The burden of this total under-allocation is proportionally shared by the fixed tasks on V_k. T_i's proportion of under-allocation among all the fixed tasks of V_k can be represented as $shr_i / \sum_{j=2}^{Z_k+1} shr_j$. Therefore, total under allocation suffered by T_i at the time when T_1 completes it's execution on V_k is:

$$UT_i^k = UT_{sum}^k \frac{shr_i}{\sum_{j=2}^{Z_k+1} shr_j}$$

$$\Rightarrow UT_i^k = \frac{shr_1^k \times shr_i}{\sum_{j=2}^{Z_k+1} shr_j}[1 - \frac{\sum_{x=1}^{k} shr_1^x}{shr_1}] \tag{3.16}$$

The maximum among all the under-allocations of the fixed tasks on V_k is then given by:

$$UT_{max}^k = \max_{i=2}^{Z_k+1}\{UT_i^k\} \tag{3.17}$$

In a *type 1* system, the system-wide maximum under-allocation is then computed as: $UT_{system} = \max_{k=1}^{m}\{UT_{max}^k\}$. As a *type 2* system has liberty of mutually distinct core frequencies, the actual frequency of a core may be calculated by separately considering it's own maximum under-allocation.

Example 4: Continuing with the scenario considered in the previous example, T_3 is the non-terminating migrating task on V_3 with $shr_3^3 = 25$ and $TE_3 = 50$. Also, T_5 is the only fixed task on V_3 with $shr_5 = 75$. Now, the sum of the maximum under-allocations obtained at the end of TE_3 time-slots (from Equation 3.15) is given by: $UT_{sum}^3 = 25(1 - 25/500) = 12.5$. This under-allocation is suffered only by T_5. Therefore,

TH-2256_136101018

$UT^3_{max} = UT^3_5 = 12.5.$ ■

The amount of unfairness occurring in a system using EAFBFS scheduling strategy within a time-slice can be judged from the value UT^k_{max}. In our algorithm, we restrict this unfairness within a permissible upper bound by increasing the operating frequency of the cores appropriately. Let us assume that the systems under consideration arrive with a pre-specified restriction Φ on the maximum under-allocation that may ever be suffered by any fixed task. Also, assume that T_i $(2 \leq i \leq Z_k + 1)$ is a fixed task suffering the highest under allocation UT^k_{max} $(UT^k_{max} > \Phi)$ during NTS_k and T_1 is the non-terminating migrating task on a given core V_k which operates at frequency fr_{V_k}. Number of instructions that will be executed on V_k during NTS_k is: $I_{V_k} = fr_{V_k} \times NTS_k$. Now, number of additional instructions that we need to execute to restrict the under-allocation within Φ is given by:

$$I_{extra} = (UT^k_{max} - \Phi) \times fr_{V_k}$$

Let V_k need to operate at frequency fr_{req} to complete the required number of instructions $I_{V_k} + I_{extra}$ within NTS_k. Therefore, we have:

$$fr_{req} \times NTS_k = fr_{V_k}(NTS_k + UT^k_{max} - \Phi)$$

$$\Rightarrow fr_{req} = \frac{(NTS_k + UT^k_{max} - \Phi) \times fr_{V_k}}{NTS_k} \tag{3.18}$$

Now, in a *type 1* system, fr_{req} is computed with respect to the maximum under-allocation in the entire system UT, whereas in a *type 2* system, it is computed with respect to the maximum under-allocation occurring at each core. If in any time-slice the value of fr_{req} is greater than f_{max} then we are only able to raise fr_{req} to f_{max}. In this case, although it allows unwanted under-allocation but that is unavoidable. All the share values are then recomputed according to fr_{req}. All the fixed tasks execute at a faster rate in the time interval $TE_k - NTS_k$ during which T_1 does not execute on V_k. Therefore, the total over-allocation of the fixed tasks on V_k during $TE_k - NTS_k$ is given

TH-2256_136101018

by:

$$OV^k_{sum} = \frac{shr^k_1}{|TS_r|} \times (TE_k - NTS_k)$$

$$OV^k_{sum} = \frac{shr^k_1 \times \sum_{x=1}^{k-1} shr^x_1}{shr_1} \tag{3.19}$$

We know that the fixed tasks have to execute at a lower rate during the execution of the migrating task T_1 on V_k. So, the number of time-slots executed by T_i during NTS_k can be calculated as $(UT^k_{sum} - OV^k_{sum}) \times shr_i / \sum_{j=2}^{Z_k+1} shr_j$. Hence the effective weight of T_i during NTS_k is given by:

$$wt^k_{me_i} = \frac{(UT^k_{sum} - OV^k_{sum}) \times shr_i}{\sum_{j=2}^{Z_k+1} shr_j \times NTS_k}$$

$$\Rightarrow wt^k_{me_i} = \frac{shr_i(|TS_r| - shr_1)}{|TS_r|(|TS_r| - shr^k_1)} \tag{3.20}$$

Example 5: Continuing with the previous example, let us discuss about the frequency required to restrict the under-allocation within a specified limit and the required weight of the fixed tasks during the execution of a non-terminating migrating task on the core. As we have seen, T_5 is the only fixed task which suffers under-allocation $UT^3_5 = 12.5$ (from Equation 3.17). Let, maximum permissible under-allocation in the system is 10. Then the required operating frequency for V_3 such that the extra under-allocation is avoided, can be calculated from Equation 3.18 as: $fr_{req} = ((50 + 12.5 - 10) \times 0.4)/50 = 0.42$. At this operating frequency, the under allocation constraint is satisfied at V_3. T_3 executes on V_3 from the start of the time-slice, so fixed tasks in V_3 are never over-allocated. Now, the effective weight of T_5 during NTS_3 (from Equation 3.20) is: $wt^3_{me_5} = (75 \times (100 - 50))/(100 \times (100 - 25)) = 0.5$.

Similarly on V_4, T_6 executes alone for NTS_3 time-slots and gets over-allocated. From Equation 3.19: $OV^4_{sum} = OV^4_6 = (25 \times 25)/50 = 12.5$. Now, the effective weight of T_6 after NTS_3 is (from Equation 3.20): $wt^4_{me_6} = (74 \times (100 - 50))/(100 \times (100 - 25)) = 0.5$.

TH-2256_136101018

At this rate, T_6 is able to complete it's allotted share before the completion of the time-slice. ■

3.2.2.3 Scheduling within an Individual Core

After all the tasks for the current time-slice are assigned to the available cores and the core frequencies and task weights have been finalized, the inner level of EAFBFS commences. At this level, a separate ERFair scheduler is employed within each core for the parallel intra-time-slice execution of tasks. Within each time-slice, all cores execute in parallel in a lock-step fashion. A separate ERFair scheduler is used within each core to schedule the tasks assigned to individual cores. After a non-terminating migrating task finishes it's execution of core shares at a given core then it is immediately inserted in the ready queue of the core where it is scheduled next. In this way, parallel execution of any migrating task at multiple cores is avoided. At the end of the time-slice, the size of the next time-slice is determined and the same EAFBFS scheduling strategy is continued till execution of all the tasks is not finished in the system. The pseudo-code for the function *Schedule()* has been shown in Algorithm 4.

3.2.3 Analysis of the Algorithm

In this section, we theoretically analyze the EAFBFS algorithm.

Theorem: *Algorithm EAFBFS satisfies ERFairness at time-slice boundaries and is optimal.*

Proof: (By Induction) At $t = 0$, all tasks have a *lag* of 0; the hypothesis is trivially true. We assume the truth of the hypothesis at the beginning of the r^{th} time-slice, that is at $t = \sum_{i=1}^{r-1} |TS_i|$. We have to establish it's truth at $t = \sum_{i=1}^{r} |TS_i|$. At the start of the r^{th} time-slice, the *shr* values for all the tasks are calculated using Equation 3.6 such that they do not get under-allocated at the end of the time-slice (i.e. at $t = \sum_{i=1}^{r} |TS_i|$). Hence, EAFBFS will be optimal if it is able to execute shares of all tasks for the time-slice TS_r. The *Task Share Adjustment* procedure (Section 6.6) guarantees that during

TH-2256_136101018

ALGORITHM 4: Function Schedule()

```
1  {count_i: Number of share of T_i remaining to be executed on V_k }
2  {mflag_k: Flag which is set to 1 during the interval when a migrating task
     executes on V_k}
3  while Priority queue of V_k is not empty do
4      if mflag_k = 1 then
5          Insert the migrating task T_mgi in priority queue of V_k
6          For all fixed tasks T_x in V_k, change their weight when T_mgi executes at V_k
             using Equation 3.20
7      Extract most urgent task T_i
8      Set count_i ← count_i − 1
9      if count_i = 0 then
10         {the share of T_i on V_k has exhausted}
11         if T_i is a migrating task then
12             Find next core of execution V_nxt from TP_i and set mflag_nxt ← 1
13             Set mflag_k to 0
14             For all fixed tasks T_x in V_k, change their weight when T_mgi executes at
                 V_k using Equation 3.20
15     else
16         Reinsert T_i in priority queue of V_k
```

transient pseudo-overloads caused by the ceiling taken in Equation 3.6, the overload is systematically overcome by decrementing the shares of the least urgent tasks by 1. These tasks which have lesser urgency in TS_r can compensate the deducted shares of 1 from TS_r in the later time-slices. It may be noted that this mechanism preserves the work conserving nature of the algorithm and hence ensures maintenance of ERFairness at time-slice boundaries. Additionally, the work conserving property of the algorithm also allows avoidance of overloads provided the total system load $\sum_{i=1}^{n} e_i/p_i$ remains within the total capacity m. As all the tasks complete their computed shares at the end of TS_r, we are able to claim that EAFBFS is optimal in it's resource utilization.

Analysis of Time Complexity: Let us analyze the complexity of each step of EAFBFS scheduling strategy. The global *Task Mapping and Frequency Allocation* (FAM) module is called at each time-slice boundary. In this module, the initial calculation for the time-slice size $|TS|$ (using Equation 3.5) and utilization factor (using Equation 3.7) as

TH-2256_136101018

well as finding the operating frequency demands for all the tasks (using Equation 3.9), takes O(n) time (where, n is the number of tasks). The calculation of the system level minimum operating frequency fr_1 (using Equation 3.8) takes a constant amount of time. In a *type 2* system, finding all the tasks with higher operating frequency demand than fr_1 can be done with a time complexity of O(n). Computation of the minimum common operating frequency (using Equation 3.10 and Equation 3.11) takes O(n) time. Then, we recompute the share values for all the tasks with respect to modified core frequencies, which can be done in O(n) time. Partitioning of tasks uses a two-phased scheme. In *phase 1*, the tasks are required to be kept in a sorted list for their partitioning using *Worst-Fit Decreasing* bin packing algorithm. In order to do so, we sort the tasks based on their shares in non-increasing order and then construct a priority queue of tasks. Here, the sorting of tasks take O($nlg\ n$) time and the construction of priority queue O(n) time. The tasks are then successively extracted from the priority queue and allocated to the core having maximum remaining capacity. Therefore, the overall allocation step for n tasks take O($nlg\ m$) time (where, m is the number of cores). In *phase 2*, we sequentially assign migrating tasks to available cores. This step takes O(n) time. Mitigating task under-allocation in each core by appropriately adapting their operating frequency takes O(n) time. Therefore, the overall scheduling complexity at the boundary of each time-slice is O($nlg\ n$) (=O($nlg\ n$) + O($nlg\ m$) + O(n)) (considering $n >= m$). A separate ERFair scheduler operates concurrently within each core inside a time-slice. Hence, the per core scheduling complexity at each time slot is O($lg\ n$). Therefore, the overall amortized time complexity of the EAFBFS scheduling strategy calculated over all time-slices during execution of a system is O(max($nlg\ n/|TS|, lg\ n$)) per time slot. If the minimum size of a time-slice is considered to be n/m ($|TS| = n/m$; n denotes the number of tasks), the amortized complexity of the algorithm becomes O($mlg\ n$) per time slot. Therefore, the time complexity of the EAFBFS scheduler is same as that of original ERFair.

TH-2256_136101018

3.2.4 Experiment and Results

We have implemented the EAFBFS algorithm and compared the same against EA-DPFair [57], a semi-fair optimal algorithm, using extensive simulation based experiments. The performance of both EAFBFS and EA-DPFair have been measured on various input data sets for two distinct multi-core platforms: *i*) where all cores are frequency locked together and frequency changes must occur synchronously (*type 1*) *ii*) where the core frequencies may be mutually distinct with asynchronous scaling allowed (*type 2*). EA-DPFair allows the cores to run at mutually distinct frequencies, which makes EA-DPFair better suited for *type 2* systems than for *type 1* systems. Hence, we have shown the results of EAFBFS on both *type 1* and *type 2* systems but have compared the results of EAFBFS with EA-DPFair on *type 2* systems only.

3.2.4.1 Experimental Set Up

The experimentation framework used is as follows: The data sets consist of randomly generated hypothetical periodic tasks whose execution requirements (e_i) and weights ($wt_i = \frac{e_i}{p_i}$) have been taken from normal distributions. Given the total number of tasks to be generated (n) and *Utilization Factor* (where *Utilization Factor* is defined as the ratio of the summation of weights of the tasks and the number of available cores) U, the task weights have been generated from a distribution with standard deviations ranging from $\sigma_{wt} = 0.1$ to $\sigma_{wt} = 0.5$ and mean $\mu_{wt} = 0.5$. The summation of the randomly generated task weights may not exactly be equal to U. However, making the summation of weights constant helps in the evaluation and comparison of the algorithms. Therefore, the individual weights have been scaled such that their summation becomes equal to U. All the task execution requirements have also been generated from a normal distribution having $\sigma_e = 20$ and $\mu_e = 100$. Now, different types of data sets have been generated by setting different values for the following parameters:

1. **Number of Tasks (n):** The size of task set was considered as 32, 64, 96, 128 and 160.

2. **Standard Deviation (σ_{wt}) for Task Weights:** Task weight distribution with

TH-2256_136101018

five distinct σ_{wt} values 0.1, 0.2, 0.3, 0.4 and 0.5 were considered.

3. **Number of Cores (m):** Multi-core systems consisting of 4, 8 12, 16, 20, 24 and 28 cores were considered.

4. **Utilization Factor (U):** We have considered utilization factors varying between 40% to 100%.

During experimentation, no slack has been provided between the periods of two consecutive instances of a task. This has been done to keep the total load on the system constant throughout the schedule. We ran our simulation for a total execution time of 100000 time-slots on the given systems. The fairness deviation of a task at any given time is denoted by it's under-allocation with respect to perfect ERFair allocation at that time. Because fairness deviations occur transiently within time-slices (with almost perfect fairness at time-slice boundaries) in our scheduling framework, the maximum value of the allowed fairness deviation has been calculated as a function of the time-slice size $|TS_r|$ and ranges from $0.01|TS_r|$ to $|TS_r|$. We developed a in-house simulator using C, to implement EAFBFS and EA-DPFair algorithms, where the simulation can be carried out for *type 1* and *type 2* systems. For each set of input parameters, we ran the simulation on 50 different test cases. The average of these 50 test cases has been considered as the final result.

Table 3.2: *Available levels of frequency*

Sl. No.	Frequency (in MHz)	Sl. No.	Frequency (in MHz)
1	384	7	1026
2	486	8	1134
3	594	9	1242
4	702	10	1350
5	810	11	1458
6	918	12	1512

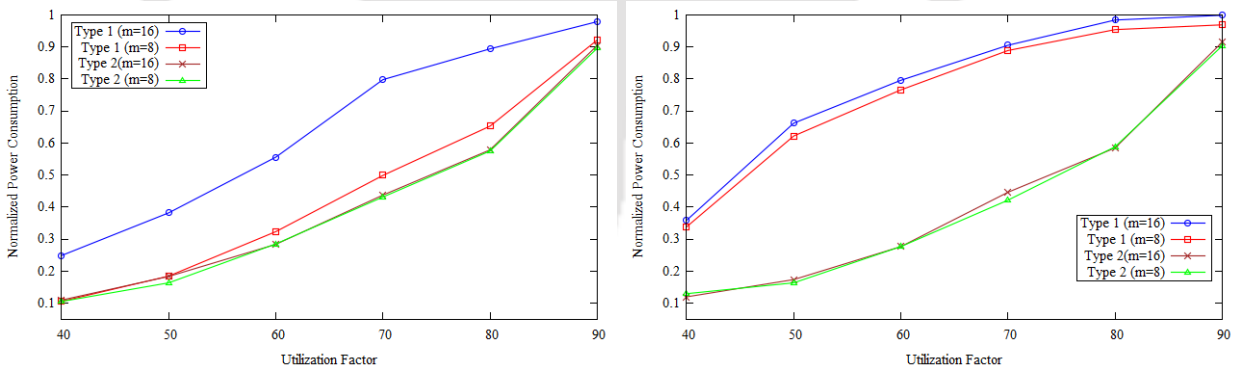
We also tested our algorithm for differently skewed task sets, each (say T) of which is obtained as a composition of two task subsets, say T_1 and T_2 ($|T| = |T_1| + |T_2|$). T_1

TH-2256_136101018

contains 80% of the total number of tasks in T, while T_2 contains the remaining 20%. Let the utilization factors of T_1 and T_2 be U_1 and U_2 respectively (that is, $U = U_1 + U_2$). Now, task sets with different degrees of skewness may be generated by varying the ratio $U_1 : U_2$. In this work, $U_1 : U_2$ has been varied from 1:4 (representing heavily skewed task sets) to 4:1 (representing task sets with fairly distributed weights). In the next section, we refer to task sets which are generated from a single uniform distribution as a *Normal Set* and those sets which are generated from two distinct task subsets as *Skewed Sets*. We carried out our experiments using the frequency set (Table 3.1) which is available in *Google Nexus 4* with quad-core *Snapdragon S4 Pro processor* [105].

3.2.4.2 Performance Evaluation of EAFBFS Algorithm

We have evaluated our algorithm with different input parameters and compared the results for *type 1* and *type 2* systems. The results of the evaluation has been discussed below:



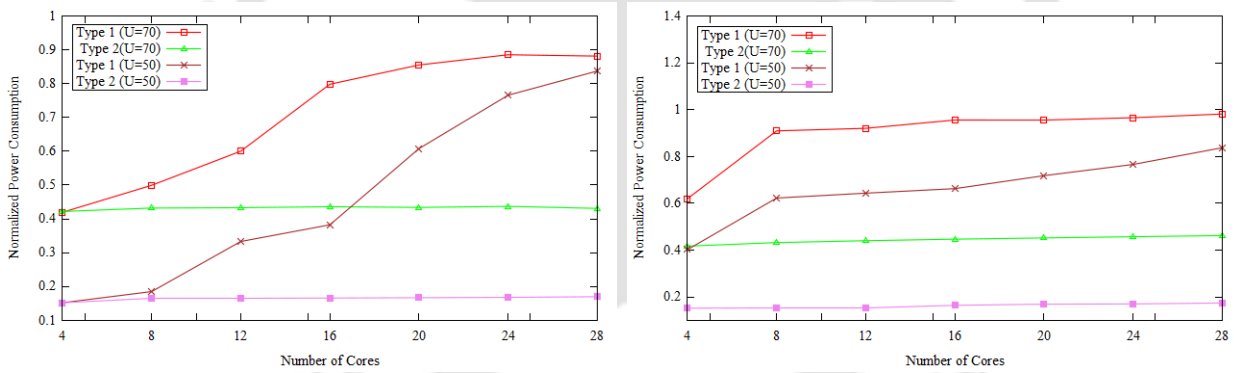
(a) *Normal Set* **(b)** *Skewed Set*

Figure 3.2: *Effect of varying utilization factor on power consumption ($n = 96$, $\phi = 0.1|TS_r|$ and $\sigma_{wt} = 0.3$)*

Experiment 1 (Effect of Utilization Factor): The utilization factor of a system represents the ratio of the summation of weights of the tasks and the number of available cores. In our experiments, we have varied the utilization factors between 40% and 100%. The system under consideration has $n = 96$, $\sigma_{wt} = 0.3$ and $\phi = 0.1|TS_r|$ with 8 or 16

TH-2256_136101018

cores. As we can see from Figure 3.2, the power consumption is directly proportional to the utilization factor of the system. As the utilization factor rises, the idling time for the cores within a time-slice reduces, thus allowing less scope for lowering operating frequencies, ultimately resulting in higher power consumption. In a *type 2* system, we can independently vary the operating frequency of individual cores. So, EAFBFS gives better results on a *type 2* system than on a *type 1* system. The average difference in power consumption between *type 1* and *type 2* systems were found to be 4.0% and 23.49% on 8 and 16 cores respectively (Figure 3.2a). To evaluate power consumption considering skewed task sets, the degree of skewness has been taken to be 1:4. In this case (Figure 3.2b), the relative difference between the weights of tasks is high; consequently, the frequency demand of the tasks also varies significantly. The average difference in power consumption were found to be 34.25% and 36.49% on 8 and 16 core systems, respectively. Hence, the difference in power consumption between *type 1* and *type 2* systems become more significant for skewed task sets. *Type 2* systems adapt well to the skewness among task weights.



(a) *Normal Set* **(b)** *Skewed Set*

Figure 3.3: *Effect of varying number of cores on power consumption ($n = 96$, $\phi = 0.1|TS_r|$ and $\sigma_{wt} = 0.3$)*

Experiment 2 (Effect of the number of cores): We evaluated our algorithm on systems having 4 to 28 cores with utilization factors of 50% and 70%. The rest of the system specifications are same as in the setup used for performance evaluation against changes in utilization factor (Experiment 1). Figure 3.3 shows that power consumption

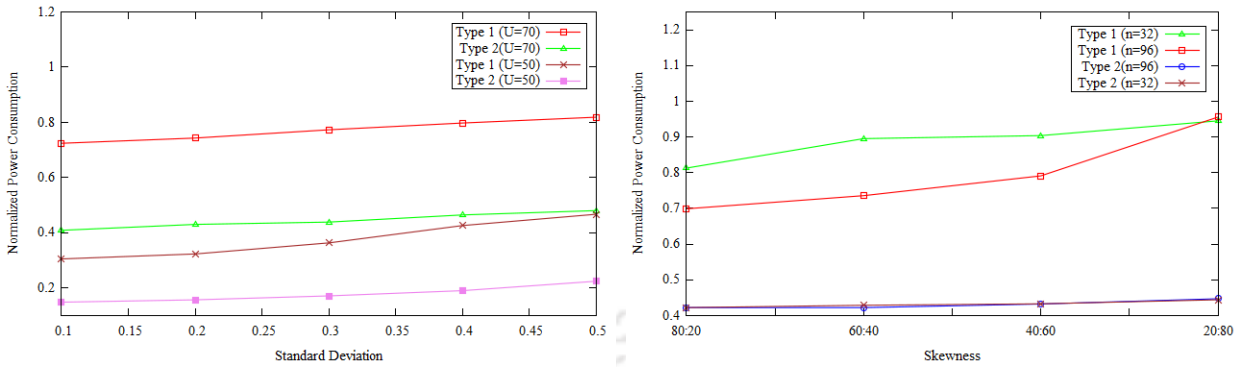
TH-2256_136101018

of *type 1* systems grow rapidly with increase in the number of cores, while for *type 2* systems, the growth is almost negligible. When the number of cores increase in a system, for a given utilization factor and number of tasks, the average individual task weights increase. Thus, we have more cases where the individual frequency demand (Equation 3.9) of a task is higher than the average system frequency (Equation 3.8). It may be observed that as the weight of a task increases in a system, the individual frequency demand also increases, which in turn results in increased power consumption. This leads to more wastage of power in a *type 1* system, as all cores in a *type 1* system are restricted to operate at the same frequency. Correspondingly, this effect is far milder in case of *type 2* systems as they are able to maintain an optimal frequency on all the cores. Thus the increase in number of cores has less impact on power consumption. In particular, when the number of cores is varied from 4 to 28 with 50% system utilization, the normalized power consumption in *type 2* systems remain almost same while it increases by 68.55% on *type 1* systems.

The average difference in power consumption between *type 1* and *type 2* systems for normal task sets having utilization factors 50% and 70% respectively, were found to be 31.26% and 28.44% (Figure 3.3a). The differences in power consumptions between *type 1* and *type 2* systems become more prominent when the task weights are skewed with respect to each other. This has been revealed in Figure 3.3b, where the values of the difference in power consumption between *type 1* and *type 2* systems for 50% and 70% utilization factors are 50.22% and 45.72%, respectively.

Experiment 3 (Effect of Skewness): We followed two approaches to evaluate and compare the power consumption of EAFBFS against skewness of a task set. In the first approach, we considered 16 core systems having utilization factors of 50% and 70%, with $n = 96$ and $\phi = 0.1|TS_r|$. We varied the standard deviation of the distribution for generating task weights from $\sigma_{wt} = 0.1$ to $\sigma_{wt} = 0.5$, keeping $(\mu_{wt}) = 0.5$ as constant. From Figure 3.4a, we observe that as the standard deviation among task weights increase, *type 2* systems are able to provide higher energy savings as compared to *type 1* systems. This may be attributed to the fact that increase in standard deviation in turn

TH-2256_136101018



(a) *Effect of variation in standard deviation of task* **(b)** *Effect of variation in skewness within Task Set weight*

Figure 3.4: *Effect of Skewness on Normalized Power Consumption*

increases the skewness among task weights, thereby also increasing the skewness among the summation of task weights within each individual core inside a time-slice. As a *type 2* system allows cores to run at mutually distinct frequencies, such skewness among the summation of task weights in different cores provide *type 2* systems with a better opportunity of extracting possible energy savings while meeting the fairness and deadline constraints. We observe from Figure 3.4a that as the utilization factor of the system increases from 50% to 70%, the difference in power consumption between *type 1* and *type 2* systems gets higher. This is due to the fact that higher utilization factors lead to higher average task weights which in turn lead to higher variation in frequency demand by the individual tasks. The average difference in the power dissipation between *type 1* and *type 2* systems was observed to be 19.48% and 32.69% for systems with utilization factor of 50% and 70%, respectively.

In the second approach, we considered 16 core systems having utilization factor of 70% with 32 and 96 tasks and $\phi = 0.1|TS_r|$. The task weights have been generated from normal distributions with standard deviation $\sigma_{wt} = 0.1$ and mean $(\mu_{wt}) = 0.5$. The generated task sets were divided into two subsets, the first containing 80% of the total number of tasks and the second containing the remaining 20%. For the first subset, the ratio of the summation of weights of the tasks with respect to the system's utilization

TH-2256_136101018

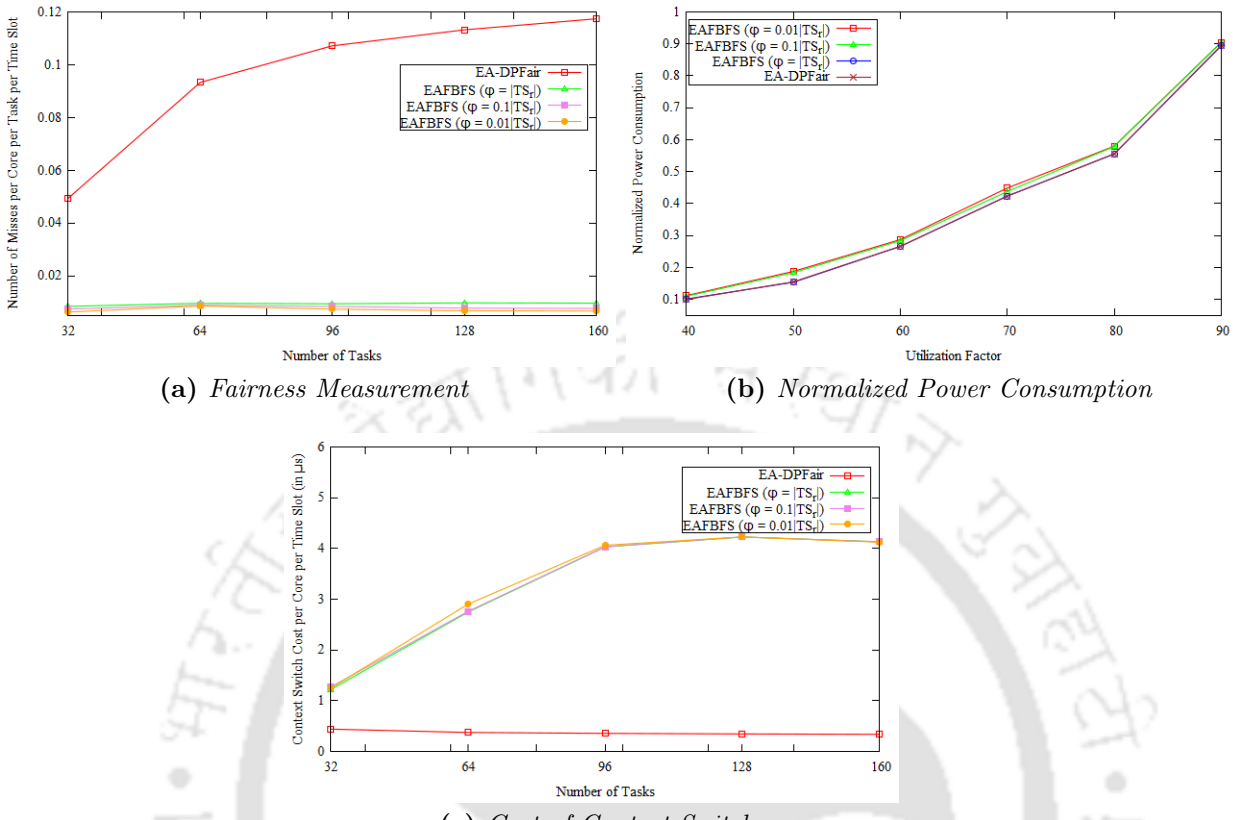
factor is varied from 1:5 to 4:5. Naturally, the ratio varies from 4:5 to 1:5 in the second subset. From Figure 3.4b, we observe that power consumption on *type 2* systems only varies slightly with respect to the variation in skewness. However, the variation is significant for *type 1* systems. It may be noted that for this task distribution, the task weights will be maximally uniform with least skewness when the ratio is 5:1. Hence, the power consumption of the EAFBFS algorithm in a *type 1*system has been observed to be lowest at point 80:20 (as shown in Figure 3.4b). As the uniformity among task weights decrease (i.e. skewness increase), the resultant power consumption rises in *type 1* systems. We further observe that power consumption in *type 1* systems is higher when the number of tasks is low ($n = 32$) than when it is high ($n = 96$). This is because as the number of tasks is increased while keeping the utilization factor constant, the average weight of individual tasks decrease. Thus, we have less cases where the individual frequency demand (Equation 3.9) of a task is higher than the average system frequency (Equation 3.8). In particular, when skewness is raised from 80:20 to 20:80 in *type 1* system with $n = 32$, the power consumption increased from 81.26% to 94.49%. The average difference in the consumption of power between *type 1* and *type 2* systems with $n = 32$ and $n = 96$ was found to be 45.71% and 36.46% respectively.

3.2.4.3 Performance Comparison with EA-DPFair Algorithm

We have compared the performance of EAFBFS with the EA-DPFair [57] on *type 2* systems algorithm against the following metrics: i.) Overall fairness achieved ii.) Power consumption iii.) Cost of context switches. All experiments were conducted on highly loaded 16 core systems with utilization factor $U = 90\%$, so that the concerned algorithms may be evaluated for stringent scenarios.

Experiment 4 (Fairness Measurement): In this work, we have measured fairness in terms of a parameter called *miss*. A subtask T_i at time t is said to have suffered a miss if $lag(T_i, t) > 1$. The degree of unfairness in the system has been computed by measuring the summation of such misses over the entire schedule length. This calculated sum is then normalized to obtain the average number of misses per core per task per

TH-2256_136101018



(a) *Fairness Measurement* **(b)** *Normalized Power Consumption*

(c) *Cost of Context Switch*

Figure 3.5: *Result Comparison: EAFBFS vs EA-DPFair*

time slot. We observe from Figure 3.5a that with it's ERFair-like scheduling strategy within time-slices, EAFBFS offers close to optimal fairness at all time-slots. On the other hand EA-DPFair, with it's EDF-like intra-time-slice scheduling strategy guarantees ERFairness only at time-slice boundaries while allowing arbitrary unfairness within the time-slices. Also, EA-DPFair's fairness decreases as the number of tasks increase. However, as the number of tasks become significantly high and correspondingly average individual task shares become considerably less, the fairness deviations stabilises progressively towards a certain value. In particular, when n is raised from 32 to 160 while keeping the utilization factor constant at 90%, EAFBFS is seen to achieve near perfect fairness (Fairness deviation ~ 0.007; $\phi = 0.1|TS_r|$) while EA-DPFair suffers a significant fairness deviation (~ 0.097) on 16 core *type 2* systems. Also, even when the value of ϕ is

TH-2256_136101018

relaxed from $0.01|TS_r|$ to $TS_r|$ (the length of an entire time-slice), the achieved fairness degrades only slightly. Our experimental results show that EAFBFS scheduling is able to achieve 10 to 15 times higher fairness (on average) than EA-DPFair. Hence, compared to EA-DPFair, EAFBFS may be considered to be more suitable for QoS sensitive applications which require execution progress at precise rates over short time scales.

Experiment 5 (Power Consumption): Power consumption of EAFBFS has been compared with EA-DPFair for data sets consisting of 96 tasks with different utilization factors. We observe from Figure 3.5b that power consumptions of EAFBFS and EA-DPFair follow a very similar trend, although EA-DPFair perform slightly better than EAFBFS. This is due to the fact that EAFBFS attempts to achieve the highest possible fairness for all tasks at all time-slots within time-slices. Additionally, it has to satisfy the constraint of limiting the maximum intra-time-slice transient unfairness of any task below a stipulated threshold ϕ in the presence of migrating tasks. EAFBFS achieves this by minimally increasing the operating frequency above the lowest frequency that is sufficient to guarantee schedulability of all tasks. EA-DPFair on the other hand allows unrestricted fairness deviations within time-slices and hence, do not require such frequency enhancement. Consequently EAFBFS's power consumption is slightly higher than EA-DPFair as observed. In particular, EA-DPFair was seen to outperform EAFBFS by 2.10%, 1.75% and 0.02% on 16 core *type 2* systems for $\phi = 0.01|TS_r|, 0.1|TS_r|$ and $|TS_r|$, respectively.

Experiment 6 (Cost of Context Switches): In our experiments, we have assumed the delay corresponding to a single context switch to be 5.24 μs [29], which may be considered to be a conservative assumption on most systems under typical workloads. We counted the total number of context switches in the system over the entire simulation duration. The total delay due to context switches was then obtained by multiplying the number of context switches with the delay caused by a single context switch ($5.24\mu s$ [29]). This value is then normalized by computing the average context switch

TH-2256_136101018

overhead (in μs) per core per time slot for both EAFBFS and EA-DPFair algorithms. As observed from Figure 3.5c, EA-DPFair incurs significantly fewer context switches compared to EAFBFS. This is because EA-DPFair uses an EDF based scheduling strategy within time slices, whereas EAFBFS uses ERFair. Higher proportional fairness as achieved by EAFBFS is essentially effected by appropriately switching task executions on cores (using preemptions and migrations) so that stipulated rates of progress may be guaranteed for all tasks across short time spans. To incorporate the cost of context switches into the overall power consumption, the operating frequencies of the individual cores were adjusted accordingly. For example from Figure 3.5c, we observe that for 64 tasks, EAFBFS suffers a context switch overhead of $\sim 3\mu s$ while EA-DPFair suffers a context switch overhead of only $\sim 0.5\mu s$. So, EAFBFS incurs an extra overhead of $2.5\mu s$ per core per task per time slot with respect to EA-DPFair. Given a time slot size of $1ms$, EAFBFS will therefore be able to complete as much work in $401ms$ as EA-DPFair will complete in $(1ms/2.5\mu s =)$ $400ms$. To overcome this additional context switching overhead, EAFBFS must execute at 401/400 times the operating frequency of EA-DPFair. As all our experiments have been conducted assuming only 12 available discrete frequency levels (refer Table 3.2.4.1), this additional overhead in terms of calculated frequency very rarely translates into an increase in the discrete frequency level of a core. Hence, the cost of context switching did not have any significant effect on power consumption in our experiments.

TH-2256_136101018

3.3 DPFair Scheduling with Slowdown and Suspension (DPFair-SS)

With exponential growth in chip transistor densities over technology generations, static energy dissipation due to leakage drain from transistors has steeply increased over the years [71]. Today, static energy dissipation has already become the major source of power wastage within a chip. Static energy wastage is mainly controlled using DPM, where a processor is put into inactive low power suspension/sleep states by procrastinating task executions while simultaneously guaranteeing their timely completion. While the strategy stated earlier named *EAFBFS* tries to minimize dynamic energy consumption in a system by scaling down the frequencies of one or more core, the strategy discussed below named *DPFair-SS* applies both DVFS and DPM in a coordinated manner over a DPFair [78] allocation scheduler to minimize both static and dynamic energy dissipation. DPFair is an important optimal proportional fair scheduler which ensures lower design costs through full resource utilization while guaranteeing the satisfaction of all deadlines, even for dynamically arriving task sets. While there exists a DVFS based state-of-the-art energy-aware DPFair scheduler [57], our proposed scheduling technique, DPFair with slowdown and suspension (DPFair-SS), exhibits appreciable performance gains over the existing strategy in situations when the system experiences low workloads over significantly long durations.

In the next section, we explain the power specifications needed to describe our work. Section 3.3.2 presents the proposed algorithm along with it's pseudo-code and illustrative examples followed by experimental results in Section 3.3.5.

3.3.1 Power Specifications

We have adopted the analytical core energy model presented in [69]. Let total power consumption of a CMOS-based core be denoted by P during active operation and P_{sleep} when the core is suspended/sleeping. The active power consumption P further consists of dynamic power consumption (P_d), static power consumption (P_s) and an inherent

TH-2256_136101018

power cost to keep the processor on (P_{on}).

$$P = P_d + P_s + P_{on} \tag{3.21}$$

The dynamic power consumption (P_d) of CMOS circuits is given by:

$$P_d = C_{eff} \times V_{dd}^2 \times f \tag{3.22}$$

where, V_{dd} is the supply voltage, C_{eff} is the average switched capacitance per cycle, and f is the clock frequency. Here, V_{dd} may be considered to be roughly proportional to f. The major components of static current in a standard inverter are due to reverse bias junction current [131] and subthreshold conduction [131]. Hence, the static power consumption, P_s, is given by:

$$P_s = (V_{dd} \times I_{subn} + |V_{bs}| \times I_j)L_g \tag{3.23}$$

where, V_{bs} is the the body bias voltage, I_{subn} is the subthreshold leakage current, I_j reverse bias junction current in the NMOS device and L_g is the number of devices in the circuit. The technology constants for the 70nm technology have been taken from [89,131]. According to this energy model, we can calculate the frequency f, dynamic power P_d and static power P_s for different voltage levels, as shown in Table 3.3. The value of V_{bs} in our experiments was taken to be -0.7. It may be observed from the voltage frequency relationships described in Table 3.3 that the minimum normalized frequency $f_1 = 0.19$ is obtained when $V_{dd} = 0.55V$ and $V_{dd} = 1.0V$ corresponding to the maximum normalized frequency $f_{max} = 1$. Following [131], values of P_{on} and P_{sleep} has been assumed to be $276mW$ and $80\mu W$ in our experiments.

Critical Frequency: As defined in Equation 3.22, the dynamic power consumption of a core (P_d) is a cubically increasing function of the operating frequency. However, since lowering the operating frequency of a core stretches the execution times of tasks scheduled on that core, frequency reduction results in higher static energy consumption. Due to this effect, beyond a certain degree of reduction in frequency, static power wastage overhauls the gain obtained through dynamic voltage/frequency scaling. Thus, there

TH-2256_136101018

V_{dd} (V)	f (GHz)	Normalized Freq.	P_d (mW)	P_s (mW)
0.55	0.58	0.19	75	171
0.60	0.79	0.26	122	204
0.65	1.01	0.33	185	245
0.70	1.26	0.41	267	289
0.75	1.53	0.49	370	339
0.80	1.81	0.58	499	396
0.85	2.10	0.68	655	461
0.90	2.42	0.78	843	535
0.95	2.74	0.88	1066	619
1.00	3.10	1.00	1327	714

Table 3.3: *Dynamic and Static power consumption for 70nm processor*

exists a critical frequency ($fr_{critical}$) beyond which further frequency reduction actually leads to increased net energy (summation of dynamic and static energy) consumption. Following [69], which considers a 70nm technology based core design, we have assumed the critical frequency to be 1.26 GHz ($V_{dd} = 0.7\ V$). Hence, in our system which assumes a maximum frequency of 3.1 GHz (refer Table 3.3), the normalized frequency becomes $fr_{critical} = 1.26/3.1 = 0.406$.

3.3.2 DPFair-SS Scheduling Strategy

The *DPFair-SS* scheduler is based on basic DPFair and hence, the schedule progresses in a time-slice by time-slice fashion, where time-slices are demarcated by the period/deadlines of all tasks at a given time. At the beginning of the next time-slice say TS_k, the share shr_i (with respect to the maximum frequency $f_{max} = 1$) of each task T_i is determined as:

$$shr_i = wt_i \times |TS_k| \tag{3.24}$$

where, $|TS_k|$ denotes the length of TS_k and wt_i denotes weight of the task i.e. $wt_i = e_i/p_i$. It may be noted, that by such an allocation and execution of shares within TS_k, *DPFair-SS* maintains accurate proportional fairness at time-slice boundaries.

Next, *DPFair-SS* calls the function *Slowdown-Suspend-Schedule (SSS)* to determine appropriate core frequencies, suspension strategy and schedule of tasks in TS_k. Algorithm 5 presents the pseudo-code of *DPFair-SS*.

TH-2256_136101018

ALGORITHM 5: DPFair-SS Algorithm

Input: Task set T, Platform V, Frequency Set F
Output: Schedule

```
1 k ← 0; time ← 0
2 while true do
3     Determine length |TS_k| of the next time-slice TS_k based on the nearest
        deadline among all executing tasks
4     for each task T_i do
5         Determine share shr_i of T_i
6     Call function SSS for TS_k
7     k ← k + 1; time ← time + |TS_k|
```

3.3.2.1 The Slowdown-Suspend-Schedule Function (SSS)

At the start of TS_k, *SSS* determines the minimum frequency fr_1 required to execute the given workload in TS_k in the following way:

$$fr_1 = \frac{\sum_{i=1}^{n} shr_i}{|TS_k| \times m} \tag{3.25}$$

As the system under consideration contains a discrete set of alternative operating frequencies, the exact frequency fr_1 may not be available. In such a situation, we choose the next higher available frequency denoted by $\lceil fr_1 \rceil$ as the operating frequency.

Now, the frequency that is sufficient to complete execution of the share of any given task (say T_i), when executed for the entire duration of a time-slice on a dedicated core, is given by:

$$fr_{T_i} = shr_i / |TS_k| \tag{3.26}$$

If $fr_{T_i} \geq \lceil fr_1 \rceil$, a dedicated core running at $\lceil fr_{T_i} \rceil$ must be allocated to T_i for the time-slice duration $|TS_k|$. Let TD be the set of tasks for which $fr_{T_i} > \lceil fr_1 \rceil$ and let shr_{TD} be the summation of task shares in TD. When all the tasks in TD are allocated to dedicated cores and the rest of the tasks are allotted the remaining $(m - |TD|)$ cores, the minimum frequency at which these $(m - |TD|)$ cores must execute is given by:

$$fr_g = \frac{\sum_{i=1}^{|T|} shr_i - shr_{TD}}{(m - |TD|) \times |TS_k|} \tag{3.27}$$

TH-2256_136101018

ALGORITHM 6: FUNCTION SLOWDOWN-SUSPEND-SCHEDULE

Input: Task set T with their share for TS_k
Output: Schedule

Determine operating frequency fr_1
Allot a dedicated core to each task T_i having: $fr_{T_i} > \lceil fr_1 \rceil$. The core's operating frequency becomes: $\lceil fr_{T_i} \rceil$
Determine minimum operating frequency (fr_g) for the rest $(n - |TD|)$ tasks on the remaining $(m - |TD|)$ cores
if $fr_g > fr_{critical}$ **then**
    Modify shares of each task T_i: $shr_i = shr_i / fr_g$
    Assign task shares to the $(m - |TD|)$ cores in any order
    Determine the amount of time t_1 $(t1 \leq |TS_k|)$ for which the $(m - |TD|)$ core should operate at frequency $\lfloor fr_g \rfloor$
else
    Calculate m', the exact system capacity that is necessary to execute the $(n - |TD|)$ tasks at frequency $\lceil fr_{critical} \rceil$
    $\{\lfloor m' \rfloor$ cores must execute for entire time-slice duration$\}$
    $\{(m - |TD| - \lceil m' \rceil)$ cores may sleep for entire time-slice duration$\}$
    Determine t_2, the number of time slots for which the remaining core should execute
    Modify shares of each task T_i: $shr_i = shr_i / \lceil fr_{critical} \rceil$ and assign to the cores

As $fr_{critical}$ denotes the frequency of execution which maximizes net energy savings, all cores need to execute for the entire time-slice duration only if, $fr_{critical} \leq fr_g$. In case $fr_{critical} \leq fr_g$, the $(m - |TD|)$ cores execute at frequency $\lfloor fr_g \rfloor$ for a duration say, t_1 and at frequency $\lceil fr_g \rceil$ for the remaining duration of the time-slice $(|TS_k| - t_1)$, where $\lfloor fr_g \rfloor \times t_1 + (|TS_k| - t_1) \times \lceil fr_g \rceil = fr_g \times |TS_k|$. Hence,

$$t_1 = \frac{|TS_k| \times (\lceil fr_g \rceil - fr_g)}{\lceil fr_g \rceil - \lfloor fr_g \rfloor} \tag{3.28}$$

In case $fr_{critical} > fr_g$, the exact system capacity that is sufficient to execute the task set *(T-TD)* is given by:

$$m' = (m - |TD|) \times (fr_g / \lceil fr_{critical} \rceil) \tag{3.29}$$

Now, $\lfloor m' \rfloor$ cores must execute for the entire time-slice duration $|TS_k|$ and one more core should execute for say, t_2 time slots at frequency $\lceil fr_{critical} \rceil$, where t_2 is given by:

$$t_2 = (m' - \lfloor m' \rfloor) \times |TS_k| \tag{3.30}$$

TH-2256_136101018

This partially filled core can be suspended for the remaining time ($|TS_k| - t_2$). The remaining ($m - |TD| - \lceil m' \rceil$) cores can sleep for the whole time-slice duration $|TS_k|$.

The pseudo-code of the function *Slowdown-Suspend-Schedule (SSS)* presented in Algorithm 6, depicts the overall energy-aware intra-time-slice scheduling strategy of *DPFair-SS*. We now present an example to illustrate the working principle of function *SSS*.

3.3.3 Analysis of the algorithm

In this section, we theoretically analyze the DPFair-SS algorithm.

Theorem: *Algorithm DPFair-SS is optimal in it's resource utilization.*

Proof: DPFair [78] is an important optimal proportional fair scheduler which ensures lower design costs through full resource utilization while guaranteeing the satisfaction of all deadlines, even for dynamically arriving task sets. Initially, DPFair-SS divides time into slices demarcated by the periods/deadlines of all tasks in the system. At the beginning of each time-slice, execution shares of each task is computed using Equation 3.24. Then, the algorithm performs task partitioning using DPFair. When there is slack available within a time-slice, DPFair-SS incorporates a DVFS-cum-DPM based mechanism to reduce energy consumption. On the other hand, when the system is fully loaded, DPFair-SS works exactly same as DPFair. At end of a time-slice, the progress made by each task using DPFair-SS is same as of that of DPFair. Hence, our algorithm is optimal in resource utilization.

Analysis of Time Complexity:

We know that DPFair-SS makes use of the algorithms *DPFair-SS* and *SSS* for calculation of frequency for the cores, task assignment and task scheduling. Let R be the total number of time-slices. The detailed analysis of the algorithms are as follows:

- *DPFair-SS*: Initially, DPFair-SS algorithm starts by computing the size of ensuing time-slice. In order to do so, it has to find the nearest deadline among all the

TH-2256_136101018

deadlines of the tasks which takes O(n) time. The computation of shares of tasks also takes O(n) time. Next, *SSS* function is called for frequency calculation, task assignment and generation of task schedule.

- *Slowdown-Suspend-Schedule Function (SSS)*: The algorithm starts by calculating the optimum frequency (fr_1) required to run the workload in the current time-slice. Since, DPFair-SS algorithm will have computed shares for all the tasks before calling SSS, this step will take O(1) time. Next, the tasks requiring frequency greater than fr_1 are allocated to dedicated cores which takes O(n) time. Then, the global optimum frequency (fr_g) required to execute rest of the tasks on remaining available cores is calculated which takes O(1) time. If the value of fr_g is greater than $fr_{critical}$, modification of shares of tasks take O(1) time, task to core assignment takes O(n) time and calculation of t_1 takes O(1) time. In cases where fr_g is less than $fr_{critical}$, calculation of m' and t_2 takes O(1) time each and calculation of modified shares and task-to-core assignment takes O(n) time each. Hence, the overall time complexity of the function SSS becomes O(n).

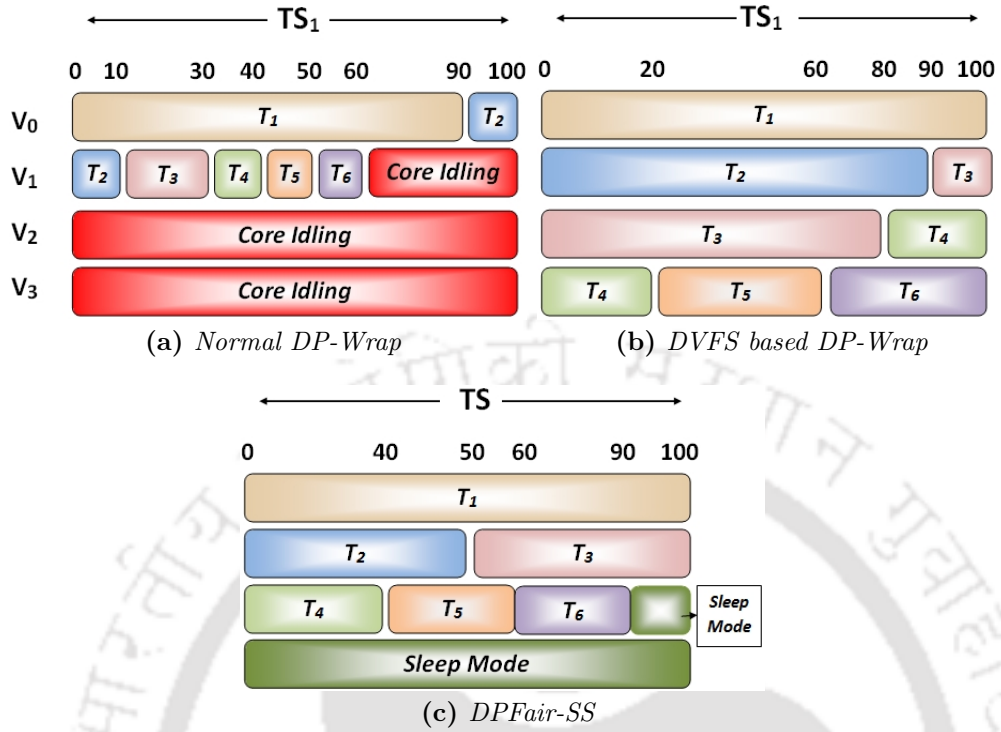
- The overall time complexity of the algorithm becomes O($R \times (O(n+n)) =$ O(Rn) per time-slice.

3.3.4 An Illustrative Example

Consider a list of 6 tasks T_1, T_2, T_3, T_4, T_5 and T_6 having execution requirements 90, 20, 20, 10, 10 and 10 respectively, to be scheduled within a time-slice of size 100 time-slots, in a system of $m = 4$ cores.

Normal DP-Wrap: The task allocation using energy oblivious DP-Wrap has been shown in Fig. 3.6a. We can observe that V_0 runs at f_{max} for 100 time-slots, V_1 runs at f_{max} for 60 slots and idles for the remaining time in *power on* mode while V_2 and V_3 idles in *power on* mode for the entire time-slice. The energy consumption in a system using this strategy (from Equation 3.21) can be calculated as: $E_{sys} = (P_d + P_s + P_{on}) \times 160 + (P_{on} \times 240) =$

TH-2256_136101018



(a) *Normal DP-Wrap* **(b)** *DVFS based DP-Wrap*

(c) *DPFair-SS*

Figure 3.6: *Task Allocation for Example*

$(1327 + 714 + 276) \times 160 + (276 \times 240) = 43.7W$.

DVFS based DP-Wrap: Using this strategy, the optimum frequency required to execute the workload is found out using Equation 3.25, $fr_1 = (90 + 20 + 20 + 10 + 10 + 10)/(4 \times 100) = 0.34$. But if the system executes at fr_1, then T_1 won't be able to complete it's required share in the time-slice. Hence, T_1 is allotted to V_0 with operating frequency of 0.9. For rest of tasks, required operating frequency is recalculated using Equation 3.27, which comes out to be 0.26. The modified shares for all the tasks are recalculated with respect to operating frequency of corresponding cores as: $shr_i = shr_i/fr_g$. The tasks are allocated using DP-Wrap scheduling as shown in Fig. 3.6b. Therefore, the energy consumption in the system will be: $E_{sys} = (1066 + 619 + 276) \times 100 \times 1 + (122 + 204 + 276) \times 100 \times 3 = 37.67W$. Although, the energy consumption is reduced significantly in comparison to the energy

TH-2256_136101018

oblivious DP-Wrap scheduling strategy, but dynamic energy savings due to frequency reduction is overhauled by static energy wastage as we see next.

DPFair-SS: Similar to *DVFS based DP-Wrap*, T_1 is allocated to V_0 and frequency fr_g for the rest of the cores becomes 0.26 (using equation 3.27). For the system considered by us, $fr_{critical} = 0.406$ and so, $fr_g < fr_{critical}$. The value of m' becomes 1.9 using equation 3.29. Therefore, one core must execute for the entire time-slice at frequency $\lceil fr_{critical} \rceil$. One core may sleep for the entire time-slice duration. The remaining core will execute for $t_2 = 90$ time slots (refer equation 3.30) at frequency $\lceil fr_{critical} \rceil$. This core can sleep for the remaining 10 time slots in the time-slice. The tasks are then assigned to the cores using *DP-Wrap* as shown in Fig. 3.6c. An estimate of the energy consumed may be found as: $E_{sys} = (1066+619+276)\times100\times1+(267+289+276)\times190+P_{sleep}\times10+(P_{sleep}\times100) = 35.41W$.

3.3.5 Experimental Set Up and Results

In order to evaluate the performance of our algorithm, we have considered task sets consisting of 64 tasks to be scheduled on 8 cores. The execution times of individual tasks present in the set have been generated randomly from a normal distribution with standard deviation $\sigma_e = 20$ and mean $\mu_e = 100$. The task weights have also been generated from normal distributions with standard deviation $\sigma_{wt} = 0.1$ and mean $\mu_{wt} = 0.3$. The summation of the randomly generated task weights may not exactly be equal to the desired *Utilization Factor* (U). Therefore, the individual weights are scaled so that the summation becomes equal to U. The execution time for each simulation was 100000 time slots. For each set of input parameters, we ran the simulation on 50 different test cases. The average of these 50 test cases has been considered as the final result. We compared performance of our proposed *DPFair-SS* algorithm against *DVFS based DP-Wrap* [57]. We now provide an analysis of the results obtained.

The system under consideration is subjected to two distinct average workload values over the simulation duration. The average *Utilization Factor* of the system is U_{low}

TH-2256_136101018

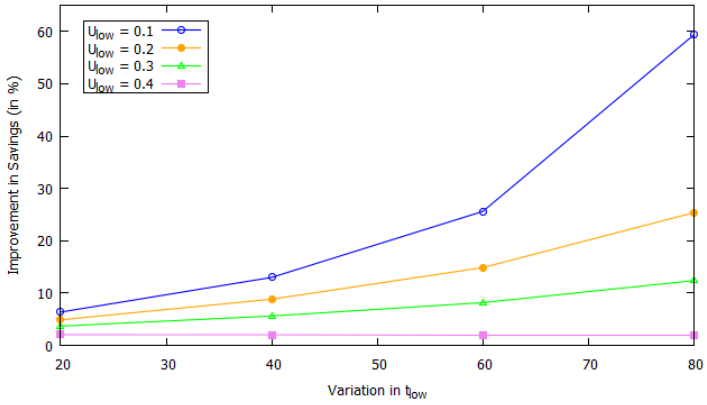


Figure 3.7: *% Improvement in Energy Savings for* DPFair-SS *over* DVFS based DP-Wrap *($U = 0.7$, $m = 8$ and $n = 64$)*

for t_{low}% of the total simulation duration, while for the other $(100 - t_{low})$% of the simulation duration, the average *Utilization Factor* is fixed at 0.7. We have considered four distinct values (0.4, 0.3, 0.2 and 0.1) for U_{low} while t_{low} has been varied between 20% and 80%. Fig. 3.7 depicts results for the percentage improvement in energy savings achieved by *DPFair-SS* over *DVFS based DP-Wrap*. From the figure, it may be seen that for any given value of t_{low} energy savings increase as U_{low} decreases. Further, for a particular value of U_{low}, energy savings improve as the percentage of time under the lower workload, increases. Overall, the gains in energy savings obtained by *DPFair-SS* may be observed to be quite significant. This is especially because it is quite common for many systems to experience brief periods with heavily CPU intensive workloads along with significantly longer periods when computation demands are quite low. In particular, for $t_{low} = 80\%$, improvements in energy savings may be observed to be 59.36%, 25.37%, 12.38% and 1.71% for U_{low} values 0.1, 0.2, 0.3 and 0.4, respectively. The normalized energy consumption values obtained by varying U_{low} and t_{low} has been presented in Table 3.4. Here, normalized energy consumption is determined by the ratio of the energy consumption value obtained for *DPFair-SS / DVFS based DP-Wrap*, with respect to the energy consumed when all cores run at the highest frequency f_{max} for the entire simulation duration.

TH-2256_136101018

Time Spent under $fr_{critical}$: t_{low} (in %)	20	40	60	80
DVFS based DP-Wrap ($U_{low} = 0.1$)	0.54	0.44	0.33	0.23
DPFair-SS ($U_{low} = 0.1$)	0.51	0.39	0.26	0.14
DVFS based DP-Wrap ($U_{low} = 0.2$)	0.56	0.47	0.39	0.30
DPFair-SS ($U_{low} = 0.2$)	0.53	0.44	0.34	0.24
DVFS based DP-Wrap ($U_{low} = 0.3$)	0.57	0.50	0.43	0.35
DPFair-SS ($U_{low} = 0.3$)	0.55	0.47	0.39	0.30
DVFS based DP-Wrap ($U_{low} = 0.4$)	0.59	0.53	0.47	0.41
DPFair-SS ($U_{low} = 0.4$)	0.58	0.52	0.46	0.40

Table 3.4: *Normalized Energy Consumption for various values of t_{low} and U_{low} ($U = 0.7$, $m = 8$ and $n = 64$)*

3.4 Summary

In this chapter, we presented *EAFBFS* and *DPFair-SS*, two low-overhead energy-aware strategies for scheduling periodic real-time tasks on multi-cores in a proportional fair manner. Both the algorithms are not only able to appreciably reduce energy consumption in the system but also provide efficient resource utilization. While EAFBFS uses the DVFS mechanism to reduce dynamic energy consumption in the system, DPFair-SS combines the benefits of both DVFS and DPM to minimize system-wide energy consumption. Experimental results prove the efficacy of the proposed strategies. In the next chapter, we present two low-overhead semi-partitioned heuristic strategies namely *COST* and *HETERO-SCHED*, for scheduling periodic tasks on heterogeneous multi-core platforms.

TH-2256_136101018

TH-2256_136101018

Chapter 4

Scheduling on heterogeneous multi-core systems

Over the years, the industry is witnessing a significant shift in the nature of processing platforms in real-time embedded systems. Homogeneous single cores have slowly given way to multi-core platforms in order to cater to higher computation demands while adhering to restrictions on temperature/energy dissipation. In addition, the need to satisfy stringent performance requirements, often along with additional constraints on size, weight, power, etc., has ushered in the era of heterogeneous processing platforms in today's complex embedded control systems. For example, a modern System-on-Chip platform can contain multi-core CPUs with specialized graphics processing cores, digital signal processing cores, floating-point units, customizable FPGAs, ASIP, ASIC, etc. Platforms with such varying types of computing elements are called *heterogeneous* (or unrelated) processing platforms. *On a heterogeneous platform, the same piece of code may require different amounts of time to execute on different processing cores.* For example, a task responsible for rendering images may take far less time to execute on a graphics processor compared to a general-purpose CPU, while number-crunching routines would execute more efficiently on CPUs. Given a heterogeneous multi-core processing platform and a set of real-time applications modeled as periodic tasks, successfully satisfying all timing related specifications is ultimately a *scheduling* problem.

Development of efficient resource allocation strategies for real-time tasks on heterogeneous platforms has traditionally proved to be a challenging as well as a computa-

TH-2256_136101018

tionally expensive problem. Optimal schedulers are typically applied on static problems but when the input parameters vary over time for the problem, the dimension of the constraints increase. The solution of such problems should be able to work with incomplete information and deliver efficient resource utilization. In [25, 26], different ILP representations for the problem of partitioning constrained-deadline task systems on heterogeneous multiprocessor platforms have been proposed. All of these have been represented through a significant number of variables and constraints. As the number of processors and tasks grows in the system, scheduling of the task sets using such models becomes very expensive. Moreover, strategies which can efficiently schedule real-time task sets on generic heterogeneous platforms having an arbitrary number of processor types, are rare. Hence, to efficiently schedule task sets online on heterogeneous platforms, researchers have started focussing on heuristic algorithms. In this chapter, we present two efficient heuristic scheduling strategies for such platforms namely *COST* and *HETERO-SCHED*. Initially, both of the algorithms employ *deadline partitioning* [78] to divide time into slices. Next, they apply different heuristic strategies to allocate tasks onto available cores. While *COST* has a restriction on the number of migrations for an individual task within a time-slice, *HETERO-SCHED* allows unrestricted inter-core task migrations. A few important terminologies used in the later sections of the chapter have been listed in Table 4.1. Next, we present the system model used in our work and present our problem statement with help of an example.

4.1 Specifications

We consider a real-time system consisting of a set of n independent periodic tasks ($T = \{T_1, T_2, ..., T_n\}$) to be preemptively executed on m heterogeneous multi-cores ($V = \{V_1, V_2, ..., V_m\}$). Each task T_i is characterized by a two tuple $T_i\langle\langle u_{i,1}, u_{i,2}, \ldots, u_{i,m}\rangle, p_i\rangle$, where $u_{i,j}(\in \mathbb{R}^+)$ is the *utilization* of T_i on V_j (denotes the fraction of the computing capacity of the V_j^{th} core that is needed to completely execute task T_i) and p_i ($\in \mathbb{N}$) is the fixed period as well as deadline of T_i. Next, we present our problem statement with help of an example.

TH-2256_136101018

Symbol	Description
T	Task set $\{T_1, T_2, \ldots, T_n\}$
V	Heterogeneous multi-core platform $\{V_1, V_2, \ldots, V_m\}$
T_i	i^{th} task. $T_i\langle\langle u_{i,1}, u_{i,2}, \ldots, u_{i,m}\rangle, p_i\rangle$
$u_{i,j}$	Utilization of task T_i on core V_j
p_i	Period of task T_i
U	*Utilization matrix*, $U_{[n\times m]}$.
H	*Hyper-period*. It is defined as the least common multiple of all task periods, $H = lcm(p_1, p_2, \ldots, p_n)$
DP	*Deadline Partitioning*. It divides H into slices demarcated by deadlines of tasks
TS_k	k^{th} time-slice in the interval $[0, H]$
CL	Set of clusters $\{CL_1, CL_2, \ldots, CL_{\lceil m/2 \rceil}\}$
CL_i	i^{th} cluster. $CL_i = \langle CL_i^V, CL_i^T\rangle$ where CL_i^V and CL_i^t are list of cores and list of tasks of CL_i
$TA_{[1\times n]}$	Task-cluster allocation matrix
$CA_{[1\times m]}$	Core-cluster allocation matrix
AM_k	*Allocation Matrix* of size $[n \times m]$ for time-slice TS_k. Each entry $AM_k[i][j]$ represents fraction of V_j's capacity allocated for T_i
MC_k	*Migration Count* vector of size $[1 \times n]$ for time-slice TS_k. Entry $MC_k[i]$ represents the migration count of T_i and set to 0
SM_k	*Schedule Matrix* of size $[n \times m]$ at time-slice TS_k. Each entry $SM_k[i][j]$ represents a pointer to a list consisting of 2-tuple $\langle start\_time(T_{i,j}),\ end\_time(T_{i,j})\rangle$
$sh_{i,j,k}$	The share required by T_i on V_j during k^{th} time-slice TS_k; $sh_{i,j,k} = \lceil u_{i,j} \times \lvert TS_k \rvert \rceil$ and $sh_{i,j,k} \in \mathbb{N}$

Table 4.1: *Important Terminologies*

4.2 Motivational Example

Let us consider a system consisting of a set of six tasks ($T = \{T_1, T_2, ..., T_6\}$) to be preemptively scheduled on a heterogeneous platform consisting of four cores ($V = \{V_1, V_2, V_3, V_4\}$). The *periods* (as well as *deadline*) of the tasks as follows: $p_1 = p_2 = 20$, $p_3 = p_4 = 10$ and $p_5 = p_6 = 40$. The *Utilization Matrix* $U_{[6\times 4]}$ for the taskset is shown in Table 4.2. The scheduling strategy in the system should be able to allot and schedule tasks on the available cores in such a way that all tasks finish their execution requirements within their respective periods. As we can observe from Table 4.2, the tasks have different execution requirements on the cores of the system and hence, design of such scheduling strategy for such systems is very challenging problem. In the next section, we present the working principles of *COST* and *HETERO-SCHED* with their respective

TH-2256_136101018

experimental analysis.

$U_{[6\times4]}$	V_1	V_2	V_3	V_4
T_1	0.3	0.5	0.7	0.6
T_2	0.4	0.6	0.8	0.8
T_3	1.5	0.5	0.6	1.2
T_4	0.5	1.3	0.7	0.9
T_5	1.4	1.6	0.5	0.7
T_6	1.2	1.1	0.6	0.6

Table 4.2: *Example: Utilization Matrix* $U_{[6\times4]}$

4.3 Cluster-Oriented Scheduling Technique (COST)

In this section, we first present a low-overhead semi-partitioned algorithm, *Cluster-Oriented Scheduling Technique* (COST) (Algorithm 7), for the scheduling of a set of real-time periodic tasks on a heterogeneous multi-core system. Next, we illustrate the working of our proposed algorithm using an example. Subsequently, we analyze the time complexity of our proposed algorithm.

4.3.1 COST: A Cluster-Oriented Scheduling Technique

The proposed algorithm, *COST*, starts with computation of the hyperperiod H of a given task set and then, it divides the hyperperiod interval $[0, H]$ into *time-slices* demarcated by the periods/deadlines of all tasks in the system. Next, it forms a set of clusters CL, with each cluster $CL_i\langle CL_i^V, CL_i^T\rangle$ consisting of at most 2 cores along with a disjoint subset of tasks associated with it. Use of such a mechanism, restricts the number of migrations within each cluster to at most one in a given time-slice. It can be represented in the following way:

$$\bigcup_{i=1}^{\lceil m/2\rceil} CL_i^T = T, \quad \bigcap_{i=1}^{\lceil m/2\rceil} CL_i^T = \emptyset$$

$$\bigcup_{i=1}^{\lceil m/2\rceil} CL_i^V = V, \quad \bigcap_{i=1}^{\lceil m/2\rceil} CL_i^V = \emptyset$$

TH-2256_136101018

Here, CL_i^V and CL_i^T denotes list of cores and tasks associated with cluster CL_i. During cluster formation, it also partitions and assigns tasks to the clusters. Next, given a *task-to-cluster* mapping, it allots each task onto cores of a cluster using *Hetero-Split* [45] strategy, with possibly at most one task requiring migration between the cores of a cluster. Subsequently, COST schedules the tasks onto their allocated cores using *Hetero-Wrap* [45] strategy, which results in computation of global schedule template S, which can be used to generate schedules for time-slices of any arbitrary length. The length of a time-slice TS_k in terms of time slots is denoted by $|TS_k|$. Next, we discuss the COST algorithm in detail.

Detailed Algorithm: *Cluster-Oriented Scheduling Technique* (COST) (Algorithm 7), first obtains all the time-slices within the entire hyper-period (Line 3-5) corresponding to the task set according to the *Deadline Partitioning* (DP) scheme. Then, it invokes the *FORM-CLUSTERS* algorithm (Algorithm 8), to form clusters as well as assign the given task set onto the clusters (Line 6-8).

FORM-CLUSTERS algorithm first partitions m cores into multiple clusters, with each cluster having a maximum of two cores. The clustering is done by first choosing a task-core pair (say, T_i and V_j) having the lowest utilization factor $u_{i,j}$ from the Utilization Matrix $U_{[n\times m]}$ in the time-slice. If the corresponding core V_j is already part of some cluster, then *FORM-CLUSTERS* assigns task T_i to that cluster, provided the remaining capacity of that cluster is more than the average utilization of T_i on the cores of the cluster (Line 5 - 10). If V_j has not been considered yet for any cluster, then a new cluster is formed with the two cores having lowest utilization factors for T_i (Line 12 - 14). The process continues until all the tasks have been assigned to some cluster (Line 4 - 19).

Subsequent to the formation of the clusters, Algorithm 7 uses *Hetero-Split* [45] to find the per-core task workload assignment, for every cluster (Line 9 - 11). *Hetero-Split* guarantees to find a feasible workload assignment between two cores, as long as there exists a solution. Then, within each cluster, *COST* employs *Hetero-Wrap* [45] to compute a global schedule template S based on the allocation provided by *Hetero-Split* [45],

TH-2256_136101018

for each cluster CL_i in CL. Next, COST computes the local schedules for each core at every time-slice by appropriately scaling the global schedule of the cores with respect to the size of the time-slices. It may be noted that the global schedule table/template S may contain idle time slots and subsequently, the local schedule tables derived using S may also contain idle slots in it. To effectively utilize computing resources, we try to allocate the unused time slots proportionally among non-migrating tasks within a core based on their utilizations (Line 14). In the next subsection, we explain the working of *COST* algorithm, using an illustrative example.

ALGORITHM 7: COST

Input: Task Set T, Multi-core Platform V
Output: A set of schedule tables ($\forall V_j \in V$ in $[0, H]$)

1 $k \leftarrow 0$; $time \leftarrow 0$
2 Compute hyperperiod $H = lcm(p_1, p_2, \ldots, p_n)$
3 Using *Deadline Partitioning*, compute time-slices in $[0, H]$
4 Let $TS = \{TS_1, TS_2, \ldots\}$ be the set of time-slices
5 Let $TA_{[1\times n]}$ be task to cluster and $CA_{[1\times m]}$ be core to cluster allocation matrices
6 Let $CL = \{CL_1, CL_2, \ldots\}$ be the set of clusters, where $CL_i = \langle CL_i^V, CL_i^T \rangle$ and initialize it to $\emptyset$
7 CL = FORM-CLUSTERS (T, V, $TA_{[1\times n]}$, $CA_{[1\times m]}$)
8 **for** *each cluster in* $CL_i \in CL$ **do**
9 Construct a global schedule template S, by partitioning and allocating tasks to cores using *Hetero-Split* and *Hetero-Wrap* algorithms, respectively
10 **for** ***each*** *time-slice* $TS_k \in TS$ **do**
11 Prepare a local schedule table from S after scaling with respect to $|TS_k|$
12 Allocate the unused time slots proportionally among non-migrating tasks within a core based on their utilizations

4.3.2 An Illustrative Example

Let us try to solve the problem which was presented in Section 4.2. According to *COST* (Algorithm 7), we compute the hyper-period, $H = lcm(20, 20, 10, 10, 40, 40) = 40$. Using *deadline partitioning*, let us find the time-slices in the interval $[0, 40)$. It may be observed from Figure 4.1b that there are four time-slices, i.e., $TS = \{TS_1, TS_2, TS_3, TS4\}$, where $TS_1 = [0, 10)$, $TS_2 = [10, 20)$, $TS_3 = [20, 30)$ and $TS_4 = [30, 40)$.

Algorithm 7 (COST) invokes Algorithm 8 (FORM-CLUSTERS) to form clusters of cores, based on the utilization of the given task set. Algorithm 8 first selects the lowest

TH-2256_136101018

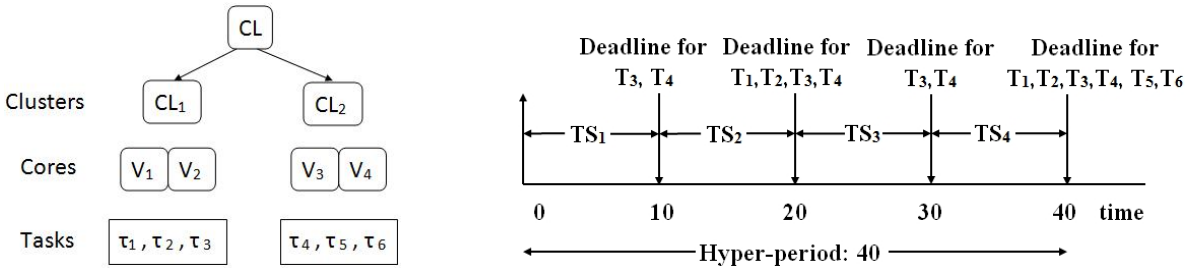
ALGORITHM 8: FORM-CLUSTERS

Input: T, V, $TA_{[1\times n]}$, $CA_{[1\times m]}$
Output: Set of clusters CL

```
1  Set cluster count c = 0
2  while all tasks are not allocated to some cluster do
3      Let u_{i,core_1} be the lowest utilization factor in U_{[n×m]}, where T_i is not part of
         any existing cluster
4      if core_1 is not part of any cluster then
5          Let u_{i,core_2} be the next lowest utilization factor for T_i such that core_2 is
             also not part of any cluster
6          Form a new cluster CL_c with ⟨core_1, core_2⟩ and add the task T_i to it
7          Update task and core cluster matrices: TA[i] ← c, CA[core_1] ← c,
             CA[core_2] ← c
8          Increment the cluster count: c ← c + 1
9      else
10         Let CL_j be the cluster in which core_1 is present
11         if computation demand of the task T_i ≤ unused capacity of the cluster CL_j
             then
12             Assign T_i to cluster CL_j: TA[i] ← j
13         else
14             Declare that T_i cannot be assigned onto core_1
15 return CL
```

utilization $u_{1,1}$ from U and checks whether the core V_1 is part of any existing cluster. Since, core V_1 is not part of any cluster, Algorithm 8 attempts to find the next lowest utilization factor for T_1, i.e., $u_{1,2}$. Now, Algorithm 8 forms a new cluster CL_1 with cores $\langle V_1, V_2\rangle$. Also, it allocates T_1 to the cluster CL_1 and the remaining/unused capacity of CL_1 is updated as: (*Capacity of Cluster - Average Utilization of the task on the cores of cluster*) $= 2.0 - (0.3 + 0.5)/2 = 1.6$. Next, the lowest utilization factor $u_{2,1}$ is selected from U. Here, V_1 belongs to CL_1. Also, the remaining capacity of CL_1 (1.6) is greater than the computation demand of T_2 ($(0.4 + 0.6)/2 = 0.5$) and hence, T_2 is allocated to CL_1. Next, $u_{3,2}$ is selected and it is also assigned to the cluster CL_1 again. Then, $u_{4,1}$ is selected. Here, the task T_4 cannot be assigned to CL_1, as the remaining unused capacity of CL_1 ($= 0.1$) is not sufficient enough to meet the computation demand of the task T_4 ($(0.5 + 1.3)/2 = 0.9$). Next, $u_{5,3}$ is selected from U and a new cluster CL_2 is formed. Subsequently, T_5 is assigned to this cluster and the remaining capacity of CL_2 is adjusted accordingly. Finally, T_6 and T_4 are also assigned to the cluster CL_2. The

TH-2256_136101018

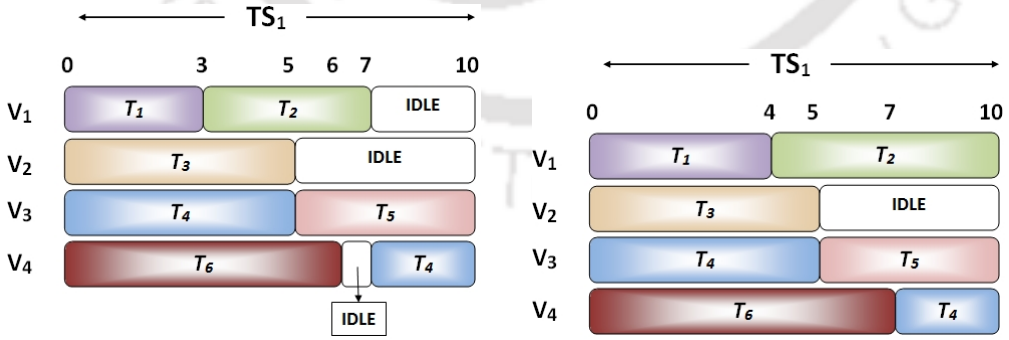


(a) *Cluster Formation and Task Assignment* **(b)** *Time-Slices*

Figure 4.1: *Deadline Partitioning & Cluster Formation for Example*

resulting set of clusters are depicted in Figure 4.1a.

Next, for each cluster, COST applies HETERO-FAIR (i.e., HETERO-SPLIT followed by HETERO-WRAP) to allocate and schedule tasks among the cores within the cluster. The schedule obtained using HETERO-FAIR is shown in Figure 4.2a. It may be observed that the obtained schedule contains idle time slots in both the clusters. According to our proposed algorithm COST, these idle slots are proportionally allocated among tasks within a core based on their utilizations. In core V_1 of cluster CL_1, three units of idle slots have been shared between T_1 (1 time-slot) and T_2 (2 time-slots). In case of core V_2, the task T_3 has already completed its execution requirement of 5 and hence, the idle slots in V_2 are not utilized. In core V_4, there are two tasks T_6 and T_4. According to COST, one unit of idle slot has been allocated to the task T_6. The final schedule is shown in Figure 4.2b. In a similar manner, the schedule table for the remaining time-slices TS_2, TS_3 and TS_4 can be obtained.



(a) *Initial Schedule* **(b)** *Final Schedule*

Figure 4.2: *Task Schedule for Example*

TH-2256_136101018

4.3.3 Analysis of the Algorithm

Now, we analyze the space and time complexities of the *COST* algorithm. Let R be the total number of time-slices in the time interval $[0, H]$.

Space Complexity: The state space complexity of the *COST* algorithm, may be analyzed as follows:

- The size of the *task to cluster matrix* $TA_{[1\times n]}$ for time-slice TS_k is $O(n)$. Since, the same matrix is used across R time-slices over the hyper-period H, the total amount of space required for this matrix is $O(n)$.
- The size of the *core to cluster matrix* $CA_{[1\times m]}$ for time-slice TS_k is $O(m)$. Since, the same matrix is used across R time-slices over the hyper-period H, the total amount of space required for this matrix is $O(m)$.
- There are total m cores and n tasks in the cluster set CL. Hence, the total amount of space required for this matrix is $O(m + n)$.
- Within each cluster, the algorithm HETERO-FAIR [45] requires a single list of tasks for task partitioning and task scheduling. Since CL may have $m/2$ clusters, space requirement for HETERO-FAIR is $O(n \times m/2)$. The prepared schedule is scaled for every time-slice based on the size of time-slice. Hence, same lists may be used across R time-slices.

Therefore, the space complexity of *COST* is $O(m \times n)$ $(= O(n) + O(m) + O(m + n) + O(n \times m/2))$.

Time Complexity: The algorithm *COST* performs cluster formation (using function FORM-CLUSTERS()), task assignment (using function FORM-CLUSTERS()), task partitioning (using function HETERO-SPLIT()) and task scheduling (using function HETERO-WRAP()). A detailed time complexity analysis of the algorithm is as follows:

TH-2256_136101018

- Given the task instances to be scheduled within the hyper-period, the main function COST() (Algorithm 7) determines all the deadlines within the hyper-period. For this, we require to sort the task instances based on their deadlines. This step has a time complexity of $O(n\ lgn)$. Next, a sorted list based on the utilization factors of each task on all processors in non-decreasing order is constructed. Given n tasks and m cores, creation of this list takes $O(nm\ lg\ nm)$ time.

- The function FORM-CLUSTERS() iterates over the list created in previous step until all tasks are assigned onto some cluster. At each node, the corresponding task can either be successfully allotted to the specified core in the node or we move to the next node in the list. Since, there are nm nodes in the list, the time complexity of FORM-CLUSTERS() is $O(mn)$.

- The HETERO-SPLIT algorithm [45] assigns the task workload onto cores of a cluster. For this purpose, it needs to create a sorted list based on the ratio of utilization factors of the tasks on the cores of the cluster. We can insert tasks into this list while performing the task-to-core assignment in the function FORM-CLUSTERS(). Hence, this step will take $O(n)$ time. The HETERO-WRAP algorithm [45] schedules the tasks onto available cores based on the list created using HETERO-SPLIT. This step has a time complexity of $O(n)$. COST invokes the algorithms HETERO-SPLIT [45] and HETERO-WRAP [45] for each cluster, to find the global schedule in every time-slice TS_k.

- The initial construction of time-slices based on deadlines of tasks ($O(n\ lgn)$), creation of sorted list based on utilization factors of tasks ($O(nm\ lgnm)$) and creation of clusters ($O(mn)$) are performed only once at the start and hence, they can be ignored. If there are R time-slices in the interval $[0, H]$, the overall time complexity of $COST$ is: $O(Rmn) + O(Rmn) = O(Rmn)$ per hyper-period.

TH-2256_136101018

4.3.4 Experimental Set Up and Results

We have implemented the *COST* algorithm and compared the same against *DOHSA* [96], an existing scheduler for periodic tasks executing on a k-type (where $k \geq 1$) heterogeneous multi-core platform. Similar to *COST*, *DOHSA* is also a *Deadline Partitioning* based algorithm. Within every time-slice, *DOHSA* first determines the shares of tasks on different cores based on their utilization factors. Next, it tries to schedule each task fully on their preferred cores based on a heuristic. Finally, it schedules all the *migrating tasks* (which were not scheduled fully on a single core) on multiple cores such that no task executes concurrently on multiple cores. We discuss about our experimental framework in the following section.

4.3.4.1 Experimental Set Up

The evaluation of the proposed algorithm is based on extensive simulation based experiments. The size of the task sets have been varied from $n = 10$ to $n = 40$. Each task set consists of randomly generated hypothetical periodic tasks whose execution requirements are generated from a Normal Distribution having standard deviation $\sigma_e = 10$ and mean $\mu_e = 50$. Entries in the utilization matrix ($U_{[n \times m]}$) are also obtained from a Normal Distribution with standard deviation of $\sigma_u = 0.2$ and mean $\mu_u = 0.4$. To evaluate the performance of *COST*, we introduce a parameter called *Utilization Factor (UF)*, which is defined as the ratio between the *summation over the average utilizations of tasks* and the *number of available cores*. That is, $UF = \frac{\sum_{i=i}^{n} avg_{j=1}^{m}(u_{i,j})}{m}$

Results are generated for various distinct utilization factors. It may be noted that the randomly generated utilization values may not lead to desired utilization factors. In order to create a task set with a specified UF, the obtained utilization values have been appropriately scaled. Given the number of tasks n and the number of cores m in a heterogeneous platform and the required utilization factor UF, we needed to generate utilization factors for each task on all available cores. We applied a two step process to do so. In the first step, we randomly generated utilization factors for the tasks on a hypothetical core. If the ratio of summation of these utilization factors for the tasks

TH-2256_136101018

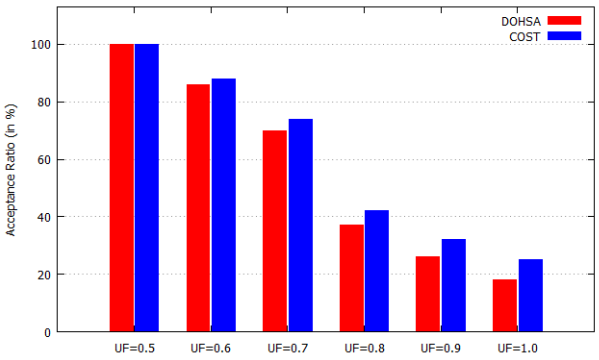
to the number of cores is higher or lower than the required utilization factor UF, we multiply each of these utilization factors by a fraction such that the ratio becomes equal to UF. Let the utilization for a task T_i on the hypothetical core is u_i^h. In the second step, we want to ensure that the average utilization of T_i over m cores is equal to u_i^h, i.e. $u_i^h = avg_{j=1}^{m}(u_{i,j}$. In order to do so, we randomly generate utilization factors from a normal distribution having standard deviation $\sigma_{u_i} = 0.1 \times u_i^h$ and mean $\mu_{u_i} = u_i^h$ for T_i on m cores. If the average of these generated utilization values is higher or lower than u_i^h, we again multiply each of these randomly generated utilization factors for T_i such that $avg_{j=1}^{m}(u_{i,j})$ becomes equal to u_i^h. We ran our simulation for a total execution time of 100000 time slots on systems having 2 to 8 processing cores. For each set of input parameters, we ran the simulation on 50 different test cases. For any given data set and algorithm being evaluated, the *acceptance ratio*, which is measured as the number of task sets accepted by the algorithm with respect to the total number of task sets submitted, has been determined.

4.3.4.2 Experimental Results

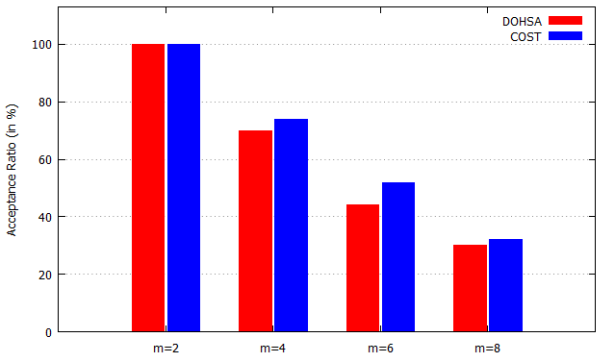
The proposed scheduler has been evaluated through extensive simulation based experiments. We have varied the the values for the following parameters in our experiments: utilization factor (Experiment 1), number of processing cores (Experiment 2) and number of tasks (Experiment 3). And then we have compared and analyzed our results with that of the algorithm *DOHSA*. Detailed analysis of the experimental results are discussed below.

Experiment 1 ***(Effect of Utilization Factor)***: In our experiments, we have varied the utilization factor between 0.5 and 1.0. The system under consideration has $n = 20$ tasks with 4 cores. We may observe from Figure 4.3a that acceptance ratio decreases for both the algorithms with increase in utilization factor of the system. Since the number of tasks remain constant against variations in utilization factor, the weights of individual tasks increase. So when workload on heterogeneous cores is increased, the probability of having migrating tasks within the time-slices also increases. For *DOHSA*, there is a

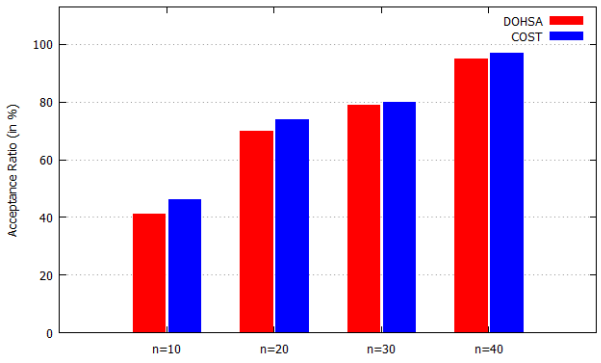
TH-2256_136101018



(a) *Effect of varying UF* (m = *4 and* n = *20)*



(b) *Effect of varying m* (UF = *0.7 and* n = *20)*



(c) *Effect of varying n* (UF = *0.7 and* m = *4)*

Figure 4.3: *COST: Experimental Results*

restriction of allowing a task to migrate only once within a given time-slice. And for *COST*, the increase in utilization factor leads to higher probability of having tasks that require inter-cluster migration. As inter-cluster migration is not allowed, hence increase in the utilization factor leads to reduction in acceptance ratio. We may observe from Figure 4.3a that the acceptance ratio reduces from 100% to 18% and 100% to 25% for *DOHSA* and *COST* respectively. Thus, *COST* fares comparatively better than *DOHSA*.

Experiment 2 ***(Effect of the number of cores)***: We evaluated both the algorithms on systems having 2 to 8 cores, 20 tasks and an utilization factor of 0.7. As we may observe from Figure 4.3b, the acceptance ratio of both *DOHSA* and *COST* decreases with increase in the number of cores, when the number of tasks remain constant. Al-

TH-2256_136101018

though, *COST* is an optimal scheduler for platforms with two types of heterogeneous cores (as it uses HETERO-FAIR [45] for scheduling within each cluster) but an increase in the number of cores results in increased number of clusters, which in turn leads to higher requirement of inter-cluster migration. Hence, the acceptance ratio decreases with the increase in number of cores for *COST*. For *DOHSA*, an increase in individual task weights due to increase in number of cores results in a higher probability of having migrating tasks in a time-slice. Also, since there is a restriction on number of migrations, so the acceptance ratios decrease. In particular, the acceptance ratio decreases from 100% to 30% and 100% to 32% for the algorithms *DOHSA* and *COST*, respectively.

Experiment 3 ***(Effect of the number of tasks)***: In our experiments, we have varied the number of tasks (n) between 10 and 40. The system under consideration has utilization factor of 0.7 with 4 cores. We may observe from Figure 4.3c that increase in the number of tasks also results in increased acceptance ratio for both the algorithms. As the utilization factor and number of cores remain constant, an increase in the number of tasks leads to reduced individual task weights. Hence, it results in increased acceptance ratio of the algorithms. However, as seen from the Figure 4.3c, *COST* delivers slightly better performance than *DOHSA* even for this case.

TH-2256_136101018

4.4 HETERO-SCHED: A Low-overhead Heterogeneous Multi-core Scheduler for Real-time Periodic Tasks

In this section, we present a low-overhead algorithm, HETERO-SCHED, for the scheduling of real-time periodic tasks on a heterogeneous multi-core system. It takes the task set T and the multi-core platform V as inputs and computes the schedule for each task T_i on each of the processing cores selected for its execution, over the hyperperiod interval $[0, H]$.

4.4.1 HETERO-SCHED Algorithm

Initially, HETERO-SCHED (Algorithm 9) determines all the time-slices within hyperperiod H, according to the *deadline partitioning* scheme (Lines 3 and 4). Let $TS = \{TS_1, TS_2, \dots\}$ be the set of time-slices (Line 5). Next, for each time-slice $TS_k \in TS$, *HETERO-SCHED* computes the schedule matrix SM_k (Lines 6 to 11). For this purpose, it internally uses COMPUTE-ALLOCATION (Algorithm 10) and COMPUTE-SCHEDULE (Algorithm 14).

ALGORITHM 9: HETERO-SCHED

Input: Task set T, Platform V
Output: A set of schedule tables ($\forall V_j \in V$ in $[0, H]$)

1 Compute hyperperiod $H = lcm(p_1, p_2, \dots, p_n)$
2 Using DP, compute time-slices in $[0, H]$
3 Let TS be the set of time-slices $TS = \{TS_1, TS_2, \dots\}$
4 **for *each*** *time-slice* $TS_k \in TS$ **do**
5 Let AM_k be the allocation matrix at TS_k and initialize all its $[n \times m]$ entries to 0
6 COMPUTE-ALLOCATION (T, V, TS_k, AM_k)
7 Let SM_k be the set of schedule matrix at TS_k and initialize all its $[n \times m]$ entries to $\emptyset$
8 COMPUTE-SCHEDULE (AM_k, SM_k)

4.4.1.1 COMPUTE-ALLOCATION

It takes the task set T, processing platform V, time-slice TS_k as inputs and computes the allocation matrix AM_k, for TS_k. It first creates two empty lists L_1 and L_2. Then, it

TH-2256_136101018

invokes COMPUTE-SHARES-REQUIRED (Algorithm 11) to compute the share $sh_{i,j,k}$ required by each task $T_i \in T$ at TS_k on the V_j^{th} processing core, as follows:

$$sh_{i,j,k} = \lceil u_{i,j} \times |TS_k| \rceil \tag{4.1}$$

The computed share $sh_{i,j,k}$ along with the task (T_i) and the processor (V_j) id's have been used to form the tuple $\langle i, j, sh_{i,j,k} \rangle$, which is then inserted into the list L_1 (Line 6 of Algorithm 11). After computing the shares required by each task over all the m processing cores, Algorithm 11 sorts the list L_1 in *non-decreasing* order of shares and returns it back to COMPUTE-ALLOCATION algorithm.

ALGORITHM 10: COMPUTE-ALLOCATION

Input: T, V, TS_k, AM_k
Output: Allocation matrix AM_k, Feasibility flag F

1 Initialize the lists $L_1 = \emptyset$, $L_2 = \emptyset$
2 COMPUTE-SHARES-REQUIRED (T, TS_k, L_1)
3 ALLOCATE-NON-MIGRATE (L_1, L_2, AM_k, V)
4 ALLOCATE-MIGRATE (L_2, AM_k, V)
5 **return** AM_k

ALGORITHM 11: COMPUTE-SHARES-REQUIRED

Input: T, TS_k, L_1
Output: The sorted list L_1 consisting of shares required

1 {Compute the shares required by each task at TS_k }
2 **for** $i = 1$ *to* n **do**
3 **for** $j = 1$ *to* m **do**
4 $sh_{i,j,k} = \lceil u_{i,j} \times |TS_k| \rceil$
5 $L_1 = L_1 \cup \{\langle i, j, sh_{i,j,k} \rangle\}$
6 Sort the list L_1 in *non-decreasing* order of $sh_{i,j,k}$
7 **return** L_1

After computing required shares, Algorithm 10 invokes ASSIGN-NON-MIGRATE (Algorithm 12) to compute the schedule for the set of tasks which can be fully assigned on to any one of the cores for its complete share of $sh_{i,j,k}$.

TH-2256_136101018

4.4.1.2 ASSIGN-NON-MIGRATE

It extracts-out the first element (say, $\langle i, j, sh_{i,j,k} \rangle$) from the list L_1. Then, it checks whether it is possible to schedule task T_i on V_j for $sh_{i,j,k}$ time units. If possible, it allocates T_i on V_j by setting the allocation matrix $AM_k[i][j] = sh_{i,j,k}$ and subsequently, sets the migration count of T_i to 0. Since T_i has been allocated completely, Algorithm 12 deletes all the entries corresponding to T_i from both L_1 and L_2.

On the other hand, if the allocation requirement $sh_{i,j,k}$ of task T_i cannot be fulfilled on core V_j, then Algorithm 12 inserts the element $\langle i, j, sh_{i,j,k} \rangle$ in to the list L_2. Once all elements in L_1 has been verified, Algorithm 12 returns the schedule for non-migrating tasks and list L_2 containing the migrating tasks. Then, Algorithm 10 invokes ALLOCATE-MIGRATE (Algorithm 13) to compute the schedule for each migrating task in L_2.

ALGORITHM 12: ALLOCATE-NON-MIGRATE

Input: L_1, L_2, AM_k, V
Output: AM_k (Allocation matrix with non-migrating tasks), L_2 (Sorted list of migrating tasks)

```
1 while L1 is not empty do
2     Extract-out the first element ⟨i, j, sh_{i,j,k}⟩ from L1
3     if T_i can be allocated fully on V_j for sh_{i,j,k} then
4         Update AM_k[i][j] = sh_{i,j,k}
5         Set migration count of T_i to 0
6         Delete all entries of T_i from L1 and L2
7     else
8         Tentatively insert ⟨i, j, sh_{i,j,k}⟩ at the end of L2
9 return AM_k, L2
```

4.4.1.3 ASSIGN-MIGRATE

It starts the allocation process by selecting the task that is part of the first element $\langle i, j, sh_{i,j,k} \rangle$ in L_2 (Line 4). Initially, ALLOCATE-MIGRATE moves all elements related to T_i from the list L_2 to L_3 (Line 6). Then, it iterates over L_3 until the task T_i is completely allocated with its computation demand on the processing platform (Lines 8 to 27). Suppose, the allocation of T_i is infeasible, then the entire set T is declared to be

TH-2256_136101018

infeasible (Lines 28 to 30).

(Lines 8 to 27): ALLOCATE-MIGRATE extracts-out the first element (say, $\langle i, j, sh_{i,j,k} \rangle$) from L_3 and computes the unused capacity uc_j of V_j (Lines 9 and 10). If uc_j is non-zero, then ALLOCATE-MIGRATE checks the migration count of T_i to compute the unallocated share of T_i with respect to core V_j, i.e., uc_j (Line 12 to 15). While utilizing the unused capacity of V_j, there are two possibilities:

- $us_i > uc_j$: This implies that the unallocated share of T_i is greater than the unused capacity of core V_j. Hence, the task T_i is partially allocated on V_j ($AM_k[i][j] = uc_j$) and the normalized unallocated share of T_i is updated as: $us_i = (us_i - uc_j)/u_{i,j}$. Finally, the migration count is incremented by one ($MC_k[i] = MC_k[i] + 1$).

- $us_i \leq uc_j$: This implies that the unused capacity of core V_j is sufficient enough to meet the unallocated demand of T_i. Hence, the task T_i is allocated on V_j. Since, T_i's allocation is completed, us_i is reset to 0 and all entries of T_i are deleted from L_3.

4.4.1.4 COMPUTE-SCHEDULE

If task allocation is successful, then HETERO-SCHED invokes COMPUTE-SCHEDULE. It may be noted that the objective of this scheduling phase is to assign start and finish times of all tasks on their allocated processing cores such that, (i) all tasks are scheduled according to their allocation, and (ii) the same task is never simultaneously executed on more than one processing core.

In order to achieve this objective, we use the following guidelines (also known as HETERO-SCHED guidelines) during the scheduling of tasks:
(i) Schedule tasks on unused time slots starting from left to right over the given time-slice,
(ii) Always schedule the task with the highest number of migrations,
(iii) Schedule a migrating task in stair-case fashion such that its executions are not overlapped, where starting time of a migrating task on a core is assigned after incrementing

TH-2256_136101018

ALGORITHM 13: ALLOCATE-MIGRATE

Input: L_2, AM_k, V
Output: AM_k (Allocation matrix with all tasks)

```
while L_2 is not empty do
    Let ⟨i, j, sh_{i,j,k}⟩ be the first element in L_2
    Create and Initialize a list L_3 to ∅
    Extract all entries of T_i from L_2 and move to L_3
    Let us_i be the normalized unallocated share of T_i
    while L_3 is not empty do
        Extract-out the first element ⟨i, j, sh_{i,j,k}⟩ from L_3
        Compute unused capacity of V_j in TS_k, i.e., uc_j
        if uc_j ≠ 0 then
            if migration count MC_k[i] = 0 then
                Unallocated share of T_i: us_i = sh_{i,j,k}
            else
                Unallocated share of T_i: us_i = us_i × u_{i,j}
            if us_i > uc_j then
                Allocate T_i on V_j: AM_k[i][j] = uc_j
                Update the normalized unallocated share of T_i:
                  us_i = (us_i − uc_i)/u_{i,j}
                Migration count: MC_k[i] = MC_k[i] + 1
            else
                AM_k[i][j] = uc_j − ⌈us_i⌉
                Reset us_i to 0
                Delete all entries of T_i from L_3
    if us_i ≠ 0 then
        Declare that the allocation of task set T on multi-core platform V is
          infeasible
return AM_k
```

it's ending time on previous core by 1. The starting and ending times of execution for each task in the times-slice G_k is stored in the Schedule Matrix SM_k,

(iv) Break the execution of a non-migrating task into multiple chunks when it cannot be scheduled non-preemptively on the same allocated core. COMPUTE-SCHEDULE (Algorithm 14) uses the above guidelines to guarantee the feasible scheduling of tasks on a given heterogeneous platform.

TH-2256_136101018

ALGORITHM 14: COMPUTE-SCHEDULE

Input: AM_k, SM_k
Output: Schedule matrix SM_k at time-slice TS_k

```
1 while all tasks in SM_k are not scheduled do
2 |   From AM_k, select T_i with the highest number of migrations (handle tie-brakes
  |     arbitrarily)
3 |   for each V_j (j = 1 to m) and AM_k[i][j] ≠ 0 do
4 |   |   Schedule T_i's allocated share on V_j according to HETERO-SCHED
  |   |     guidelines
5 return SM_k
```

4.4.2 Analysis of the Algorithm

Now, we analyze the space and time complexities of the *HETERO-SCHED* algorithm. Let R be the total number of time-slices in the time interval $[0, H]$.

Space Complexity: The state space complexity of the *HETERO-SCHED* algorithm, may be analyzed as follows:

- The size of the *allocation matrix* $AM_{k[n\times m]}$ for time-slice TS_k is $O(mn)$. Since, the same matrix can be reused across R time-slices over the hyper-period H, the total amount of space required for the allocation matrix is $O(mn)$.
- The size of the *schedule matrix* $SM_{k[n\times m]}$ for time-slice TS_k is $O(mn)$ and it requires a dedicated matrix for each time-slice. Hence, the total size of the *schedule matrix* is $O(Rmn)$ for R time-slices over the hyper-period H.
- The size of the *migration count vector* $MC_{k[1\times n]}$ for time-slice TS_k is $O(n)$ and it can be reused across R time-slices. Hence, the size of the *migration count vector* is $O(n)$.
- There are three different lists L_1, L_2 and L_3. The size of L_1 and L_2 is $O(mn)$. The size of L_3 is $O(m)$. Since, these lists are reused for every time-slice, there is no change in their sizes over the hyper-period.

Therefore, the space complexity of *HETERO-SCHED* is $O(Rmn)$ $(= O(mn)+O(Rmn)+$

TH-2256_136101018

$O(n) + O(mn) + O(m)$).

Time Complexity: We know that HETERO-SCHED internally makes use of two algorithms namely, *COMPUTE-ALLOCATION* and *COMPUTE-SCHEDULE*. It may be noted that Algorithm 10 (COMPUTE-ALLOCATION) invokes Algorithms 11, 12 and 13.

- The time complexity of Algorithm 11 (COMPUTE-SHARES-REQUIRED) is $O(mn)$ since it computes the shares of all n tasks over all m cores.
- The time complexity of Algorithm 12 (ASSIGN-NON-MIGRATE) is $O(mn)$ since the list L_1 is guaranteed to become empty after $O(mn)$ iterations. Similarly, Algorithm 13 (ASSIGN-MIGRATE) iterates $O(mn)$ times. Hence, the complexity of Algorithm 10 is $O(mn)$ $(= O(mn) + O(mn) + O(mn))$.
- The time complexity of Algorithm 14 (COMPUTE-SCHEDULE) is $O(mnlg\ n)$ since the construction of the schedule matrix requires $O(nlg\ n)$ iterations over each of the m processing cores.
- Algorithm 9 (HETERO-SCHED) starts with *deadline-partitioning* whose time complexity is $O(nlg\ n)$. As stated earlier, Algorithm 9 invokes both Algorithm 10 and Algorithm 14, for R times. Hence, the overall time complexity of *HETERO-SCHED* is $O(Rmnlg\ n)$ $(= O(nlg\ n) + O(Rmn) + O(Rmnlg\ n))$.

Migration Overheads: In our approach, when a task cannot be fully allocated to any single core, it is carefully allocated and scheduled across multiple cores in a time-slice such that it's execution at a core does not overlap with any other core in the system. From the partitioning and scheduling mechanism discussed above, we may observe that a particular core in the system may hold a set of fixed tasks along with either: i. no other migrating tasks, ii. multiple terminating migrating tasks, iii. one non-terminating migrating task, and iv. one non-terminating migrating task and multiple terminating migrating tasks. Cores having fixed tasks and terminating migrating tasks does not

TH-2256_136101018

cause any migration in a time-slice. Whenever a non-terminating migrating task is scheduled on a particular core, that core will not have any remaining capacity to allot any other task. Hence, number of migrations at a core with a non-terminating migrating task is bounded to only 1. There can be at most $m-1$ cores with non-terminating migrating tasks in any time-slice. Therefore, number of migrations in a time-slice using our strategy is bounded by $m-1$.

4.4.3 An Illustrative Example

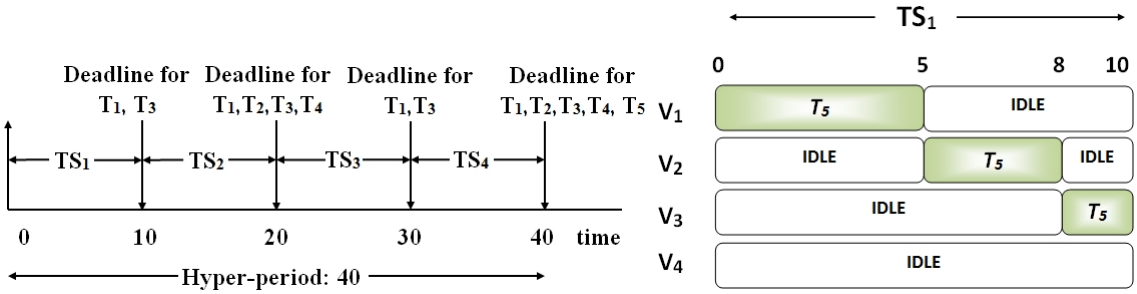
Let us consider a real-time system consisting of a set of five tasks ($T = \{T_1, T_2, ..., T_5\}$) to be preemptively scheduled on a heterogeneous multi-core platform consisting of three processing cores ($V = \{V_1, V_2, V_3, V_4\}$). The *Utilization Matrix* $U_{[5\times4]}$ is as follows:

$U_{[5\times4]}$	V_1	V_2	V_3	V_4
T_1	0.5	1.3	0.8	1.2
T_2	0.8	0.7	1.0	1.1
T_3	1.1	0.8	0.7	0.9
T_4	1.2	0.9	0.8	0.7
T_5	0.9	1.1	1.1	1.4

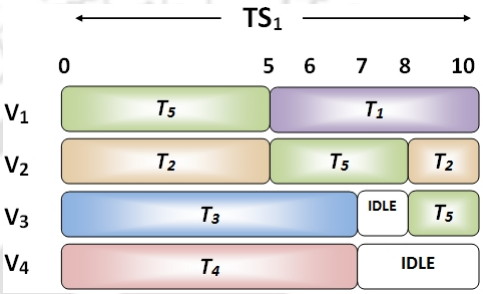
The *period* (as well as *deadline*) of each task is as follows: $p_1 = 10$, $p_2 = 20$, $p_3 = 10$, $p_4 = 20$ and $p_5 = 40$. According to *HETERO-SCHED* (Algorithm 9), we compute first the hyper-period, $H = lcm(10, 20, 10, 20, 40) = 40$. Using *deadline partitioning*, let us find the time-slices in the interval $[0, 40)$. It may be observed from Figure 4.4a that there are four time-slices, i.e., $TS = \{TS_1, TS_2, TS_3, TS_4\}$, where $TS_1 = [0, 10)$, $TS_2 = [10, 20)$, $TS_3 = [20, 30)$ and $TS_4 = [30, 40)$.

Computation of Shares Required: Algorithm 9 invokes Algorithm 10 to compute the allocation matrix AM_k for each time-slice TS_k. First, Algorithm 10 creates the two lists L_1, L_2 and initializes them to $\emptyset$. Then, Algorithm 10 invokes Algorithm 11 to compute the shares required by each task $T_i \in T$ at the time-slice TS_1 whose interval is $[0, 10)$. The computed shares are as follows:

TH-2256_136101018



(a) *Deadline Partitioning* (b) *Schedule for migrating tasks*



(c) *Final schedule for all tasks*

Figure 4.4: *Example*

$sh_{[5\times4]}$	$sh_{i,1,1}$	$sh_{i,2,1}$	$sh_{i,3,1}$	$sh_{i,4,1}$
T_1	5	13	8	12
T_2	8	7	10	11
T_3	11	8	7	9
T_4	12	9	8	7
T_5	9	11	11	14

Now, Algorithm 11 sorts the computed shares in *non-decreasing* order, i.e., The sorted list L_1 at time-slice TS_1 is as follows ($\langle i, j, sh_{i,j,1}\rangle$): $\{\langle 1,1,5\rangle, \langle 2,2,7\rangle, \langle 3,3,7\rangle, \langle 4,4,7\rangle, \langle 1,3,8\rangle, \langle 2,1,8\rangle, \langle 3,2,8\rangle, \langle 4,3,8\rangle, \langle 3,4,9\rangle, \langle 4,2,9\rangle, \langle 5,1,9\rangle, \langle 2,3,10\rangle, \langle 2,4,11\rangle, \langle 3,1,11\rangle, \langle 5,2,11\rangle\}, \langle 5,3,11\rangle, \langle 1,4,12\rangle, \langle 4,1,12\rangle, \langle 1,2,13\rangle, \langle 5,4,14\rangle\}$. This list L_1 is returned to Algorithm 10. Then, Algorithm 10 invokes Algorithm 12 to compute the allocation matrix AM_1 for non-migrating tasks.

Allocation of Non-migrating Tasks: Algorithm 12 extracts the first element from the list L_1, i.e., $\langle 1,1,5\rangle$ (task T_1 requires a share of 5 units on processor V_1). The task T_1

TH-2256_136101018

can be fully allocated on processing core V_1 and hence, the allocation matrix $AM_1[1][1]$ is updated as 5 and the migration count for T_1 is set to 0, i.e., $MC_1[1] = 0$. Since, task T_1 has been completely allocated on the processing core V_1, Algorithm 12 deletes all entries of T_1 from list L_1, i.e., $\langle 1, 3, 8\rangle$, $\langle 1, 4, 12\rangle$ and $\langle 1, 2, 13\rangle$ have been removed from L_1. Now, Algorithm 12 repeats the above process by extracting the first element from L_1, i.e., $\langle 2, 2, 7\rangle$. Since, processing core V_2 can fully accommodate task T_2, it has been allocated to V_2 with $AM_1[2][2]$ being updated to 7 and the migration count for T_1 is set to 0, i.e., $MC_1[2] = 0$. Similarly, the task T_3 is allocated (according to $\langle 3, 3, 7\rangle$) on V_3 and T_4 is allocated (according to $\langle 4, 4, 7\rangle$) on V_4.

Now, $\langle 5, 1, 9\rangle$ becomes the first element of list L_1. According to this, if task T_5 is allocated on processing core V_1, it requires a share of 9 units. However, processing core V_1 does not have the capacity to accommodate T_1 and hence, the element $\langle 5, 1, 9\rangle$ has been inserted into L_2. Similarly, the other three elements corresponding to task T_5 has been moved to L_2, i.e., $\langle 5, 2, 11\rangle$, $\langle 5, 3, 11\rangle$ and $\langle 5, 4, 14\rangle$. Since, the list L_1 becomes empty, the execution moves from Algorithm 12 to Algorithm 10. The allocation matrix $AM_{1[5\times4]}$ for non-migrating tasks at time-slice TS_1, is as follows:

$AM_{1[5\times4]}$	V_1	V_2	V_3	V_4
T_1	5	0	0	0
T_2	0	7	0	0
T_3	0	0	7	0
T_4	0	0	0	7
T_5	0	0	0	0

Allocation of Migrating Tasks: It may be observed that task T_5 is unallocated and Algorithm 10 invokes Algorithm 13 to schedule T_5. It first moves all entries corresponding to T_5 from list L_2 to L_3. Then, Algorithm 13 extracts the first element from L_3, i.e., $\langle 5, 1, 9\rangle$, and computes the spare capacity of V_1. Since, there is a residual spare capacity of 5 on processing core V_1, the task T_5 has been partially allocated on V_1, $AM_1[5][1] = 5$. The unallocated share of T_5 is $9-5 = 4$. This has been normalized as follows: $us_5 = 4/0.9 = 4.4$ (approx). The normalized value of the unallocated share for task T_5 (us_5) and the migration count ($MC_1[5]$) have been updated as 4.4 and 1, respectively.

TH-2256_136101018

Now, Algorithm 13 extracts the next element $\langle 5, 2, 11\rangle$ from L_3 and checks the spare capacity of processing core V_2. Since, the spare capacity of V_2 (i.e. 3), is not sufficient enough to accommodate the unallocated share of T_5 (i.e., $4.4 * 1.1 = 4.8$), it has been partially allocated on V_2. That is, $AM_1[5][2] = 3$. The unallocated share of T_5 is $4.8-3 = 1.8$. This has been normalized as follows: $us_5 = 1.8/1.1 = 1.6$ (approx). The normalized value of the unallocated share for task T_5 (us_5) and the migration count ($MC_1[5]$) have been updated as 1.6 and 2, respectively.

Next, Algorithm 13 extracts the element $\langle 5, 3, 11\rangle$ from L_3 and checks the spare capacity of processing core V_3. Since, the spare capacity of V_3 (i.e. 3), is sufficient enough to accommodate the unallocated share of T_5 (i.e., $1.6 * 1.1 = 1.76$), it has been allocated on V_3. That is, $AM_1[5][2] = 2$. The final allocation matrix $AM_{1[5\times4]}$ (including migrating tasks) for time-slice TS_1, is as follows:

$AM_{1[5\times4]}$	V_1	V_2	V_3	V_4
T_1	5	0	0	0
T_2	0	7	0	0
T_3	0	0	7	0
T_4	0	0	0	7
T_5	5	3	2	0

Scheduling of Migrating Tasks: Since the task allocation is feasible, Algorithm 9 invokes Algorithm 14 to construct a schedule. It may be noted that the migration count of T_5 is the highest among all tasks and hence, T_5 is scheduled first according to HETERO-SCHED guidelines. The resulting schedule consisting of T_5 is depicted in Figure 4.4b.

Scheduling of Non-migrating Tasks: It may be observed that all the remaining unscheduled tasks have the same migration count. Since ties are broken arbitrarily, let us consider the task T_1. By following the allocation matrix and HETERO-SCHED guidelines, T_1 is scheduled on V_1 from time slot 5 to 10. Next, T_2 is scheduled on V_2. It may be observed that the execution of T_2 is broken into two pieces since T_5 is already scheduled on V_2. Similarly, the remaining tasks are scheduled and the final schedule for time-slice TS_1 is depicted in Figure 4.4c.

TH-2256_136101018

It may be noted that the size of all time-slices in TS is same and hence, the schedule constructed for the first time-slice TS_1 can be utilized for all the remaining time-slices by only updating the time-slice boundaries.

4.4.4 Experimental Set Up and Results

We have implemented the *HETERO-SCHED* algorithm and compared the same against *SA-M*, a variation of the algorithm *SA* [108], which is a partition-oriented task assignment algorithm for a *two-type* platform, i.e. a heterogeneous multiprocessor platform comprising two different types of processing cores.

4.4.4.1 Experimental Set Up

The evaluation of the proposed algorithm is based on extensive simulation based experiments. The size of the task sets have been varied from $n = 10$ to $n = 40$. Each task set consists of randomly generated hypothetical periodic tasks whose execution requirements are generated from a normal distribution having standard deviation $\sigma_e = 20$ and mean $\mu_e = 100$. Entries in the utilization matrix ($U_{[n \times m]}$) are also obtained from a normal distribution with standard deviation of $\sigma_u = 0.15$ and mean $\mu_u = 0.3$. It helped us to test the robustness of our algorithms by scheduling periodic tasks with a wide range of period values. To evaluate the performance of *HETERO-SCHED*, we introduce a parameter called *Utilization Factor (UF)*, which is defined as the ratio between the *summation over the average utilizations of tasks* and the *number of available cores*. That is, $UF = \frac{\sum_{i=i}^{n} avg_{j=1}^{m}(u_{i,j})}{m}$. Results are generated for various distinct utilization factors. It may be noted that the randomly generated utilization values may not lead to desired utilization factors. In order to create a task set with a specific UF, the obtained utilization values have been appropriately scaled using the same process as illustrated in Section 4.3.4.1. We ran our simulation for a total execution time of 100000 time-slots on systems having 2 to 8 processing cores. For each set of input parameters, we ran the simulation on 50 different test cases. For any given data set and algorithm being evaluated, the *acceptance ratio*, which is measured as the number of task sets accepted by the algorithm with respect to the total number of task sets submitted, has been determined.

TH-2256_136101018

Comparison with state-of-the-art scheme: The framework presented in this work has been compared with a modified version of a state-of-the-art scheme called *Sort and Assign (SA)* [108]. *SA* is an optimal scheduling algorithm for heterogeneous multi-core platforms consisting of two types (type-1 and type-2) of processing cores. The algorithm *SA* works in two phases. In the *first phase*, the task set is partitioned into four subsets namely, $H1$, $H2$, $H12$ and L. $H1$ and $H2$ consists of tasks which must be executed only on type-1 and type-2 processing cores, respectively, L contains tasks that can be scheduled on both types of processing cores, while the remaining unschedulable tasks are put in $H12$.

In the *second phase*, tasks are assigned to their respective processing cores. In particular, the tasks in the sets $H1$ and $H2$ are assigned onto type-1 and type-2 processing cores, respectively. The tasks in the set L are inserted in a list in non-increasing order of their preferences to be assigned to type-1 processing cores. This list is then traversed left to right and the tasks are assigned sequentially to type-1 processing cores until there is no capacity left on type-1 processing cores to assign a task integrally. Subsequently, the list is traversed from right to left and the tasks are sequentially assigned to type-2 processing cores. Finally, at most one task (say, T_i) may still remain unallocated in the list. This task T_i is then partitioned with fractional allocations being assigned onto both type-1 and type-2 processing cores. In this way, *SA* is able to schedule tasks in two-type processing cores platform.

Algorithm SA-M: It may be noted that the algorithm *SA* has been designed for scheduling of tasks in systems containing two types of processing cores only. Hence, we propose an modified version of *SA* called *SA-M*, so that it may be applied to generic heterogeneous systems containing k ($k \geq 2$) types of processing cores. Initially, *SA-M* partitions m processing cores into $m/2$ clusters, with each cluster having a maximum of two processing cores. The clustering is done by first choosing a task-processing core pair (say, T_i and V_j) having the lowest utilization (highest affinity) from the utilization matrix U. If the corresponding processing core V_j already belongs to some cluster, then the *SA-M* allots the task T_i to that cluster, provided the remaining capacity of that

TH-2256_136101018

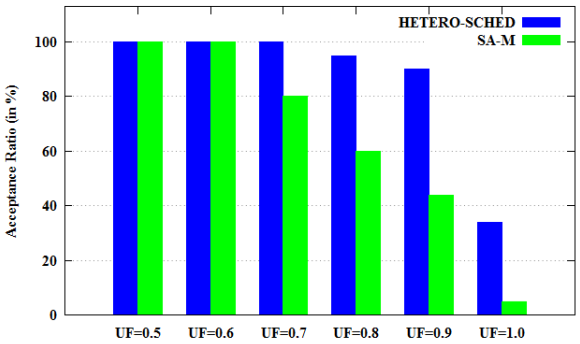
cluster is more than the average utilization of T_i in the processing cores of the cluster. If V_j has not been considered yet for any cluster, then a new cluster is formed with the two processing cores having lowest utilization for T_i. The process continues till all the tasks have been assigned to some cluster. If any processing core has not been considered yet due to low utilization factor of the task set, they are randomly partitioned to some of the new clusters. After formation of the $m/2$ clusters, tasks are partitioned in their respective clusters using the original *SA* algorithm.

4.4.4.2 Experimental Results

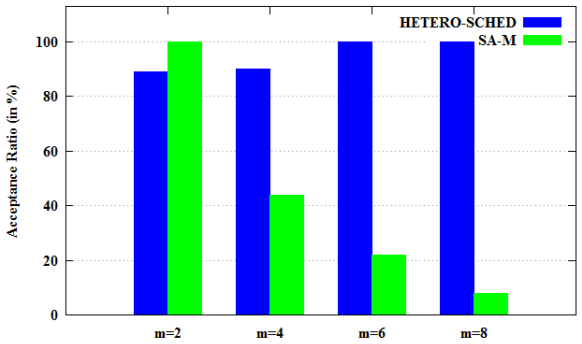
The proposed scheduler synthesis scheme has been evaluated through extensive simulation based experiments. We have varied the the values for the following parameters in our experiments: utilization factor (Experiment 1), number of processing cores (Experiment 2) and number of tasks (Experiment 3). And then we have compared and analyzed our results with that of the algorithm *SA-M*. A detailed analysis of the experimental results are discussed below.

Experiment 1 ***(Effect of Utilization Factor)***: In our experiments, we have varied the utilization factor between 0.5 and 1.0. The system under consideration has $n = 30$ tasks with 4 processing cores. We may observe from Figure 4.5a that the acceptance ratio decreases for both the algorithms with increase in utilization factor of the system. Since the number of tasks remain constant against variations in utilization factor, the weights of individual tasks increase. Hence, the given heterogeneous processing cores are assigned more workload, which in turn increases the probability of having migrating tasks within time-slices. Also for *HETERO-SCHED*, there may be overlapped execution for the migrating tasks at multiple processing cores because of variations in rates of execution across processing cores. In such a scenario, we reject the task set. For *SA-M*, the increase in utilization factor leads to higher probability of having tasks that require inter-cluster migration. As inter-cluster migration is not allowed, hence increase in the utilization factor leads to reduction in acceptance ratio. We may observe from Figure 4.5a that the acceptance ratio reduces from 100% to 34% and 100% to 5% for

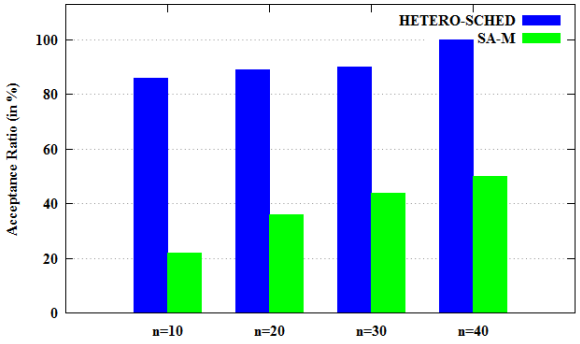
TH-2256_136101018



(a) *Varying UF (*n = *30 and* m = *4)*



(b) *Varying m (*n = *30 and* UF = *0.9)*



(c) *Varying n (*m = *4 and* UF = *0.9)*

Figure 4.5: *Experimental Results*

HETERO-SCHED and *SA-M* respectively.

Experiment 2 ***(Effect of the number of processing cores)***: We evaluated both the algorithms on systems having 2 to 8 processing cores with utilization factor of 0.9 and $n = 30$. As we may observe from Figure 4.5b, the acceptance ratio of the *HETERO-SCHED* algorithm increases with increase in the number of processing cores, when the number of tasks remain constant. This is due to the fact that at lower number of processing cores and a high utilization factor in the system, the scope to schedule a migrating task without having any overlap across the processing cores is less for the *HETERO-SCHED* algorithm. As the number of processing cores increase, the algorithm is able to schedule the task sets in a better way due to its fully migrative nature. On the other hand, *SA-M* is an optimal scheduler for platforms with two types of processing cores. An increase in

TH-2256_136101018

number of processing cores results in increased number of clusters, which in turn leads to higher requirement of inter-cluster migration. Hence, the acceptance ratio decreases with the increase in number of processing cores for *SA-M*, while it increases for the algorithm *HETERO-SCHED*. In particular, the acceptance ratio decreases from 100% to 8% and increases from 89% to 100% for the algorithms *SA-M* and *HETERO-SCHED*, respectively.

Experiment 3 ***(Effect of the number of tasks)***: In this experiment, we have varied the number of tasks (n) between 10 and 40. The system under consideration has utilization factor of 0.9 with 4 processing cores. We may observe from Figure 4.5c that increase in the number of tasks also results in increased acceptance ratio for both the algorithms. As utilization factor and number of processing cores remain constant, an increase in the number of tasks leads to reduced individual task weights. Hence, it results in increased acceptance ratio. In particular, acceptance ratio was found to be 86%, 89%, 90% and 100% for *HETERO-SCHED* and 22%, 36%, 42% and 50% for *SA-M*.

4.5 Summary

In this chapter, we presented *COST* and *HETERO-SCHED*, two low-overhead semi-partitioned heuristic strategies for scheduling periodic real-time tasks on heterogeneous multi-core platforms. Both the algorithms are able to efficiently schedule task sets as well as provide high resource utilizations. While *COST* has a restriction on the number of migrations for an individual task within a time-slice, *HETERO-SCHED* allows unrestricted inter-core task migrations. In the next chapter, we present a low-overhead heuristic strategy called, *HEALERS*, for energy-aware scheduling of a set of periodic tasks on a heterogeneous multi-core platform.

TH-2256_136101018

Chapter 5

Energy-Aware scheduling on heterogeneous multi-core systems

Nowadays, many of the embedded systems like PDAs, laptops, mobiles, etc. depend upon battery as their principal source of energy. Hence, these devices are not only judged by their real-time functional performance but also their efficiencies in terms of energy management. Devising energy-efficient scheduling strategies for real-time periodic tasks on heterogeneous platforms is a challenging as well as a computationally demanding problem. In recent years, researchers have explored energy-efficient scheduling techniques for multi-cores [24, 97, 98]. However, only a few works have been targeted towards heterogeneous platforms. As stated in previous chapter, HETERO-SCHED is an efficient scheduler for heterogeneous platforms but it is energy oblivious and hence, not suited for systems running on batteries as their primary source of energy. To overcome this issue, in this chapter we present HEALERS, which is an extension of HETERO-SCHED but uses a DVFS-cum-DPM based energy-efficient strategy to schedule periodic real-time tasks on heterogeneous platforms. HEALERS is composed of two major components: i) *COMPUTE-SCHEDULE* and, ii) *COMPUTE-EA-SCHEDULE*. These two components work in unison to not only deliver appreciable energy savings but also very high resource utilizations.

In the next section, we explain the specifications used in our work. Then, we present our proposed algorithm in Section 5.3 followed by an illustrative example to explain the algorithm in Section 5.5. We also present the Space and Time Complexity Analysis of

TH-2256_136101018

Table 5.1: *Important Terminologies*

Symbol	Description
t	Measurement of time in terms of discrete *time slots*, where a *time slot* denotes the smallest interval/unit of time with which the scheduler operates.
T	Task set $\{T_1, T, \ldots, T_n\}$ with n tasks.
T_i	i^{th} task. $T_i\langle\langle u_{i,1}, u_{i,2}, \ldots, u_{i,m}\rangle, p_i\rangle$.
$u_{i,j}$	Utilization of task T_i on core V_j.
p_i	Period of task T_i.
V	Heterogeneous multi-core platform $\{V_1, V_2, \ldots, V_m\}$ with m processing cores.
V_k	k^{th} processing core.
U	*Utilization matrix*, $U_{[n\times m]}$. Each element of the matrix is $u_{i,j}$, where $i \in [1, n]$, $j \in [1, m]$.
DP	*Deadline Partitioning*. It divides the time interval into slices demarcated by deadlines of all tasks.
TS_k	k^{th} *time-slice*.
$sh_{i,j,k}$	The share (in terms of time slots) required by T_i on V_j during the k^{th} time-slice TS_k. $sh_{i,j,k} = \lceil u_{i,j} \times \lvert TS_k \rvert \rceil$ and $sh_{i,j,k} \in \mathbb{N}$.
L_1, L_2, L_3	*Lists consisting of shares.*
$us_{i,j}$	*Unallocated share of* T_i *on* V_j.
uc_j	*Unused capacity of* V_j.
SM_k	*Schedule Matrix* of size $[n \times m]$ at time-slice TS_k. Each entry $SM_k[i][j]$ represents a pointer to a list consisting of 2-tuple $\langle start\_time(T_i),\ end\_time(T_i)\rangle$ on V_j.
f_{cr}	Normalized critical frequency.
f_{max}	Maximum value of normalized operating frequency.
f_{opt}	Optimum operating frequency selected for a core in a time-slice.

the proposed algorithm in Section 5.4. The experimental set up and the results have been discussed in Section 5.6. Finally, in Section 5.7, we conclude our work.

5.1 Specifications

5.1.1 System Model

The system under consideration consists of a set of n periodic tasks $T = \{T_1, T_2, ..., T_n\}$ to be scheduled on a set of m heterogeneous multi-cores $V = \{V_1, V_2, ..., V_m\}$ which can operate on a discrete normalized set of frequencies $F = \{f_1, f_2, ..., f_{max}\}$, such that, f_{max} represents the normalized frequency of 1 and all other frequencies lie between $(\frac{f_1}{f_{max}})$ and 1. Each instance of a periodic task T_i has an *execution requirement* of $e_{i,j}$ (when

TH-2256_136101018

executing at f_{max}), *period* p_i, and *utilization* $u_{i,j} = (\frac{e_{i,j}}{p_i})$ on core V_j. We assume task deadlines to be *implicit*, that is, same as its period p_i.

5.1.2 Power Model

Let P denote the total power consumption of a CMOS-based core during active operation while P_{sleep} denote its dissipated power when suspended. The total power P has three major components namely, dynamic power consumption (P_d), static power consumption (P_s) and an inherent power cost to keep the processor on (P_{on}) [97].

$$P = P_d + P_s + P_{on} \tag{5.1}$$

The dynamic power (P_d) can be further expressed as:

$$P_d = C_{eff} \times V_{dd}^2 \times f \tag{5.2}$$

where, C_{eff} is the average switched capacitance per cycle, V_{dd} represents the supply voltage, and f is the clock frequency. Here, V_{dd} may be considered to be roughly proportional to f. The static current has two major components in a standard inverter - reverse bias junction current [131] and sub-threshold conduction [131]. Hence, the static power dissipation, P_s, is given by:

$$P_s = (V_{dd} \times I_{subn} + |V_{bs}| \times I_j)L_g \tag{5.3}$$

where, I_{subn} is the sub-threshold leakage current, the body bias voltage is denoted by V_{bs}, reverse bias junction current in the NMOS device is denoted by I_j, and L_g represents the number of devices in the circuit. Experiments have been carried-out assuming 70nm technology and associated constants for this technology have been taken from [89, 131]. For the above energy model, Table 5.2 shows frequencies with their corresponding dynamic power P_d and static power P_s for different voltage levels. The value of V_{bs} in our experiments was taken to be -0.7. It can be observed from the Table 5.2 that the minimum normalized frequency $f_1 = 0.12$ is obtained when $V_{dd} = 0.5V$ while the maximum normalized frequency $f_{max} = 1$ when $V_{dd} = 1.0V$. From [131], values of P_{on} and P_{sleep}

TH-2256_136101018

V_{dd} (V)	f (GHz)	Normalized Freq.	P_d (mW)	P_s (mW)
0.50	0.39	0.12	42	142
0.55	0.58	0.19	75	171
0.60	0.79	0.26	122	204
0.65	1.01	0.33	185	245
0.70	1.26	0.41	267	289
0.75	1.53	0.49	370	339
0.80	1.81	0.58	499	396
0.85	2.10	0.68	655	461
0.90	2.42	0.78	843	535
0.95	2.74	0.88	1066	619
1.00	3.10	1.00	1327	714

Table 5.2: *Dynamic and Static power consumption for 70nm processor*

have been considered to be $276mW$ and $80\mu W$ in our experiments.

Critical Frequency: It may observed from Equation(5.2) that although the dynamic power consumption of a core (P_d) has a cubic relationship with the operating frequency, lowering f increases task execution times and may lead to higher overall energy consumption when execution times become significantly high for small values of f. Due to this effect, beyond a certain degree of reduction in frequency, static power wastage overhauls the gain obtained through dynamic voltage/frequency scaling. Thus, there exists a critical frequency (f_{cr}) beyond which further frequency reduction actually leads to increased net energy (summation of dynamic and static energy) consumption. Assuming a $70nm$ technology similar to [69], the critical frequency becomes 1.26 GHz ($V_{dd} = 0.7\ V$). Correspondingly, for a system with maximum frequency of 3.1 GHz (as used in this work; refer Table 5.2), the normalized frequency becomes $f_{cr} = 1.26/3.1 = 0.406$. Next, we present our problem statement with help of an example.

5.2 Motivational Example

Let us consider a system consisting of a set of seven real-time periodic tasks, $T = \{T_1, T_2, ..., T_7\}$, to be scheduled on a heterogeneous multi-core platform consisting of four cores, $V = \{V_1, \ldots, V_4\}$. The *Utilization Matrix* $U_{[7\times4]}$ is:

TH-2256_136101018

$U_{[7\times4]}$	V_1	V_2	V_3	V_4
T_1	0.5	1.3	0.8	1.2
T_2	0.8	0.7	1.0	1.1
T_3	1.1	0.8	0.4	0.9
T_4	0.9	0.9	0.8	0.1
T_5	0.9	1.1	1.1	1.4
T_6	0.9	0.6	0.1	0.8
T_7	1.2	0.4	0.7	0.2

The *period* (as well as *deadline*) of each task is: $p_1 = 10$, $p_2 = 20$, $p_3 = 10$, $p_4 = 20$, $p_5 = 40$, $p_6 = 40$ and $p_7 = 40$. The scheduling strategy in such a system should not only be able to allot and schedule tasks on the available cores but it has to do it in such a way that energy consumption is minimized, which makes it a challenging problem. In the following section, we present the working principle of *HEALERS* algorithm which is able to propose solution for such problems.

5.3 HEALERS Algorithm

The proposed scheduling strategy HEALERS is a three-level hierarchical resource allocation mechanism. At the first level, it applies deadline partitioning (similar to *DP-Fair* [58]) to *compute a set of time-slices*, where a single time-slice is the interval between two consecutive deadlines corresponding to the set of ready tasks. Next, within each time-slice, HEALERS *schedules tasks onto the processing cores*, such that each task receives its appropriate execution share (when all processing cores are operating at f_{max}). Finally, it tries to *reconfigure the operating frequencies in each core and put them in suspension mode (if possible)* such that the overall energy consumption of the platform is minimized while ensuring that this reconfiguration does not cause overloads in the system.

By following the above steps, we present our proposed scheduling strategy, HEALERS (Algorithm 15), for the energy-aware scheduling of real-time tasks on heterogeneous multi-cores. It takes the task set T, the heterogeneous platform V, and the frequency set F as inputs and computes the schedule for each ready task T_i, over the time-slice. HEALERS first determines the set of ready tasks, and computes a time-

TH-2256_136101018

slice according to the *deadline partitioning* (DP) [58] scheme (Lines 1 and 2). Within the time-slice, HEALERS computes a schedule matrix SM_k. For this purpose, it internally uses COMPUTE-SCHEDULE (Algorithm 16), and COMPUTE-EA-SCHEDULE (Algorithm 19).

ALGORITHM 15: HEALERS

Input: Task set T, Platform V, Frequency Set F
Output: Schedule Matrix SM_k at time-slice TS_k
1 Let $\{T_1, T_2, \ldots, T_n\}$ be set of tasks ready for execution
2 Using *deadline-partitioning*, compute time-slice TS_k
3 Let SM_k be the *Schedule Matrix* at TS_k and initialize all its $[n \times m]$ entries to $\emptyset$
4 COMPUTE-SCHEDULE (T, V, TS_k, SM_k)
5 **if** *task set T is feasible on platform V* **then**
6 ⌊ COMPUTE-EA-SCHEDULE (TS_k, SM_k, F)

5.3.1 COMPUTE-SCHEDULE

COMPUTE-SCHEDULE attempts to allocate each task $T_i \in T$ onto the heterogeneous platform V such that their execution requirements within the given time-slice TS_k are satisfied. It first computes the shares required by each task in TS_k, and inserts them into the list L_1 (Lines 2 to 5). As the tasks may possibly have distinct share values corresponding to each one of the m heterogeneous processing cores, there are m entries for each task in L_1. Then, it sorts L_1 in *non-decreasing* order of shares. Subsequently, it invokes SCHEDULE-NON-MIGRATE (Algorithm 17) to assign tasks whose shares can be fully allocated on any one of the available processing cores and then adds the rest of the tasks in list L_2. It may be noted that L_2 contains tasks whose execution demand $sh_{i,j,k}$ in TS_k cannot be entirely fulfilled by the remaining capacities of any individual processing core. Therefore, these tasks must be allocated upon multiple cores each of which together satisfy its total execution requirement. Hence, tasks in L_2 form the set of migrating tasks in TS_k. COMPUTE-SCHEDULE invokes SCHEDULE-MIGRATE (Algorithm 18) in order to allocate each migrating task T_i on more than one processing core. Each such core satisfies a fraction of T_i's total execution demand.

TH-2256_136101018

ALGORITHM 16: COMPUTE-SCHEDULE

Input: T, V, TS_k, SM_k
Output: Schedule Matrix SM_k

```
1 Initialize the lists L1 = ∅, L2 = ∅ {COMPUTE-SHARES-REQUIRED by tasks at TSk}
2 for i = 1 to n do
3     for j = 1 to m do
4         sh_{i,j,k} = ⌈u_{i,j} × |TSk|⌉
5         L1 = L1 ∪ {⟨i, j, sh_{i,j,k}⟩}
6 Sort L1 in non-decreasing order of sh_{i,j,k}
7 SCHEDULE-NON-MIGRATE (L1, L2, SMk, V)
8 SCHEDULE-MIGRATE (L2, SMk, V)
9 return SMk
```

ALGORITHM 17: SCHEDULE-NON-MIGRATE

Input: L_1, L_2, SM_k, V
Output: SM_k (Schedule matrix with non-migrating tasks), L_2 (Sorted list of migrating tasks)

```
1 while L1 is not empty do
2     Extract the first element ⟨i, j, sh_{i,j,k}⟩ from L1
3     if τi can be allocated fully on Vj for sh_{i,j,k} then
4         Assign start and end times of τi on Vj, i.e., SMk[i][j] = ⟨start_time(Ti),
            end_time(Ti)⟩
5         Decrement the capacity of Vk by sh_{i,j,k}
6         Delete all entries of Ti from L1 and L2
7     else
8         Tentatively insert ⟨i, j, sh_{i,j,k}⟩ at end of L2
9 return SMk, L2
```


TH-2256_136101018

5.3.1.1 SCHEDULE-NON-MIGRATE

SCHEDULE-NON-MIGRATE attempts to assign start and end times to tasks which can be fully allocated onto any one of the m cores in the platform. In order to conduct such assignment, this function sequentially considers the shares $sh_{i,j,k}$ in L_1 and attempts to allocate $sh_{i,j,k}$ entirely on core V_k. If such an allocation is successful, task T_i becomes a non-migrating/fixed task in TS_k and the remaining capacity of V_j is decremented by $sh_{i,j,k}$. On the other hand, if allocation of $sh_{i,j,k}$ on V_j is unsuccessful, it is removed from L_1 and inserted in L_2. Once a task T_i is scheduled, all its entries are deleted from both the lists L_1 and L_2 (Lines 3 to 8).

5.3.1.2 SCHEDULE-MIGRATE

This attempts to schedule tasks which require more than one processing core for their execution. First, it creates a list L_3 from L_2 to keep track of the unallocated share of T_i (Lines 2 to 4). Next, it extracts the first element (say, $\langle i, j, sh_{i,j,k} \rangle$) from L_3 and also computes the unused capacity uc_j of V_j (Line 7). If uc_j is non-zero, then SCHEDULE-MIGRATE computes the unallocated share of T_i with respect to V_j, i.e., $us_{i,j}$ (Lines 9 to 13). While utilizing the unused capacity of V_j, there are two possibilities (Lines 14 to 20):

- $us_{i,j} > uc_j$: This implies that the unallocated share of T_i is greater than the unused capacity of core V_j. Hence, T_i is partially scheduled on V_j and the unallocated share of T_i is updated.
- $us_{i,j} \leq uc_j$: This implies that the unused capacity of core V_j is sufficient enough to meet the unallocated demand of T_i. Hence, task T_i is allocated on V_j. Since, T_i's allocation is completed, $us_{i,j}$ is reset to 0 and all entries of T_i are deleted from L_3.

While scheduling a migrating task T_i, HEALERS attempts to ensure that T_i does not execute on multiple processing cores simultaneously (Lines 22 to 23). In order to schedule a migrating task T_i, HEALERS uses the following heuristic mechanism. First,

TH-2256_136101018

non-terminating part of a migrating task is always scheduled at start of the time-slice on its available favoured core. Thereafter, all other parts of it's execution are scheduled on cores in staircase fashion, where the starting time of the task on the second and subsequent cores is assigned by incrementing it's end time on the previous core, by 1. The starting and ending times of execution for each task in time-slice TS_k is stored in the Schedule Matrix SM_k. In the absence of sufficient spare capacity to schedule a migrating task on multiple cores, HEALERS declares the scheduling of T on V to be infeasible. Otherwise, it returns the schedule matrix SM_k, consisting of schedules for all ready tasks in the current time-slice TS_k.

ALGORITHM 18: SCHEDULE-MIGRATE

Input: L_2, SM_k, V
Output: SM_k (*Schedule Matrix* for all tasks)

```
1  while L_2 is not empty do
2      Create a list L_3 and initialize it to ∅
3      Extract all entries of T_i from L_2 and move to L_3
4      Let us_{i,j} be the unallocated share of T_i on V_j
5      while L_3 is not empty do
6          Extract-out the first entry ⟨i, j, sh_{i,j,k}⟩ from L_3
7          Compute unused capacity of V_j: uc_j
8          if uc_j ≠ 0 then
9              /* Compute unallocated share of T_i on V_j: us_{i,j} */
10             if T_i is not yet allocated partially on any core then
11                 unallocated share of T_i w.r.t V_j: us_{i,j} = sh_{i,j,k}
12             else
13                 Let V_l be the last core in which T_i was partially scheduled
14                 unallocated share of T_i w.r.t V_j: us_{i,j} = us_{i,l} × u_{i,j}/u_{i,l}
15             if us_{i,j} > uc_j then
16                 Update SM_k[i][j] to schedule T_i on V_j for the duration uc_j
17                 Update the unallocated share of T_i w.r.t V_j: us_{i,j} = us_{i,j} − uc_j
18             else
19                 Update SM_k[i][j] to schedule T_i on V_j for the duration uc_j − ⌈us_{i,j}⌉
20                 Reset us_{i,j} to 0
21                 Delete all entries of T_i from L_3
22     if (us_{i,j} ≠ 0) then
23         Declare the scheduling of T on V is infeasible
24 return SM_k
```


TH-2256_136101018

ALGORITHM 19: COMPUTE-EA-SCHEDULE

Input: TS_k, SM_k, Frequency Set F
Output: Schedule Matrix SM_k at time-slice TS_k

1 **for** *each processor* $V_j \in V$ **do**
2 Let $|T^j|$ denotes the set of tasks scheduled on V_j in TS_k Compute unused capacity of V_j: uc_j
3 **if** $uc_j \neq 0$ **then**
4 Compute the frequency f_{opt} for V_j using Eqn. (5.4)
5 **if** $f_{opt} < f_{cr}$ **then**
6 Modify shares of each non-migrating task T_i on V_j: $sh_{i,j,k} = sh_{i,j,k}/f_{cr}$
7 Set $f_{opt} \leftarrow f_{cr}$
8 Calculate time duration t_{sm} for which V_j can be suspended using Eqn. (5.6)
9 **else**
10 Modify shares of each non-migrating task T_i on V_j: $sh_{i,j,k} = sh_{i,j,k}/f_{opt}$
11 Determine the amount of time t_1 ($t_1 \leq |TS_k|$) for which V_j should operate at frequency $\lfloor f_{opt} \rfloor$
12 Update SM_k for all tasks scheduled on V_j
13 **return** SM_k

5.3.2 COMPUTE-EA-SCHEDULE

If task allocation is successful, then HEALERS invokes COMPUTE-EA-SCHEDULE (Algorithm 19). It may be noted that the objective of this phase is to re-assign start and finish times for all tasks on their already allocated processing cores such that operating frequencies of the cores get reduced from their maximum values f_{max}. This allows energy consumption in the system to be minimized. COMPUTE-EA-SCHEDULE first finds the spare capacity uc_j of each processing core V_j. Then, it uses uc_j to reconfigure the operating frequency of V_j from f_{max} to f_{opt} (Lines 1 to 5) in the following way:

$$f_{opt} = \frac{\Sigma_{T_i \epsilon |NON-MGR|} sh_{i,j,k}}{|TS_k| - \Sigma_{T_i \epsilon |MGR|} sh_{i,j,k}} \tag{5.4}$$

where, $|NON-MGR|$ and $|MGR|$ are sets of non-migrating and migrating tasks scheduled on V_j in time-slice TS_k. As discussed earlier, reduction in frequency to f_{opt} causes the execution time of task to be increased. In case $f_{opt} > f_{cr}$, the core execute at frequency $\lfloor f_{opt} \rfloor$ for a duration say, t_1 and at frequency $\lceil f_{opt} \rceil$ for the remaining duration of the time-slice ($|TS_k| - t_1$), where $\lfloor f_{opt} \rfloor \times t_1 + (|TS_k| - t_1) \times \lceil f_{opt} \rceil = f_{opt} \times |TS_k|$.

TH-2256_136101018

Hence,

$$t_1 = \frac{|TS_k| \times (\lceil f_{opt} \rceil - f_{opt})}{\lceil f_{opt} \rceil - \lfloor f_{opt} \rfloor} \tag{5.5}$$

If the computed value of f_{opt} is less than f_{cr}, f_{opt} is set as f_{cr}. The time period for which V_j is kept in suspension mode can be determined as:

$$t_{sm} = |TS_k| - \Sigma_{T_i \epsilon |T^j|} sh_{i,j,k} \tag{5.6}$$

where, $|T^j|$ is the set of tasks scheduled on V_j in TS_k. Finally, it returns SM_k which contains the energy-aware schedule for the current time-slice.

5.4 Analysis of the Algorithm

Now, we analyze the complexity of the *HEALERS* algorithm.

Space Complexity: The state space complexity of the *HEALERS* algorithm, may be analyzed as follows:

- HEALERS requires a dedicated schedule matrix to store the task-to-core assignment for each time-slice. The size of the *schedule matrix* $SM_{k[n \times m]}$ for time-slice TS_k is $O(mn)$. The schedule matrix is reused for every time-slice.
- The algorithm uses three different lists within each time-slice: L_1, L_2 and L_3. L_1 contains shares for each task at every core and L_2 contains shares for migrating tasks. Hence, the space complexity of both the lists are $O(mn)$. These lists are reused for every time-slice. For each migrating task, the list L_3 keeps track of its unallocated share for a time-slice. Hence, size of the list L_3 can be $O(m)$ for each task. L_3 is reused for every migrating task.

Therefore, the space complexity of *HEALERS* is $O(mn)$ $(= O(mn) + O(mn) + O(m))$.

Time Complexity: We know that HEALERS internally makes use of two algorithms namely, *COMPUTE-SCHEDULE* and *COMPUTE-EA-SCHEDULE*. It may be noted that Algorithm 16 (COMPUTE-SCHEDULE) invokes Algorithms 17 and 18.

TH-2256_136101018

- Initially, algorithm *COMPUTE-SCHEDULE* determines shares of all tasks at every core for a time-slice, which takes $O(mn)$ time. The time complexity of Algorithm 17 (ASSIGN-NON-MIGRATE) is $O(mn)$, since the list L_1 is guaranteed to become empty after $O(mn)$ iterations. Similarly, Algorithm 18 (ASSIGN-MIGRATE) may also iterate $O(mn)$ times. Hence, the complexity of Algorithm 16 is $O(mn)$ $(= O(mn) + O(mn)+)(mn))$.

- Algorithm 19 determines the appropriate operating frequency for each core based on their workloads. Hence, time complexity of *COMPUTE-EA-SCHEDULE* is $O(m)$.

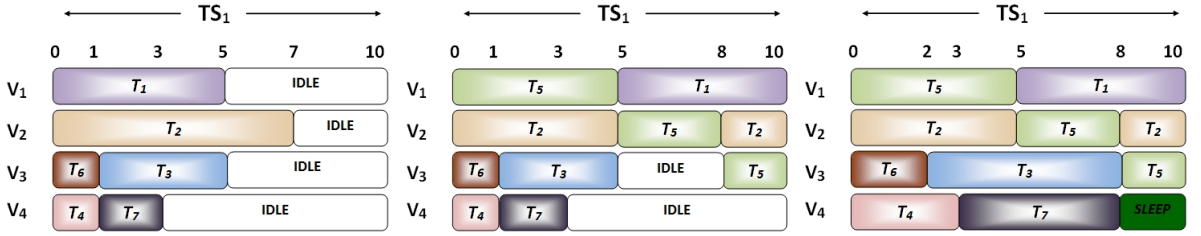
- Algorithm 15 (HEALERS) starts with *deadline-partitioning* whose time complexity is $O(nlg\ n)$. As stated earlier, Algorithm 15 invokes Algorithm 16 and Algorithm 19, for each time-slice. Hence, the overall time complexity of *HEALERS* is $O(n(m + lg\ n))$ $(= O(nlg\ n) + O(mn) + O(m)))$.

Dynamic Task Arrivals/Departures: A periodic task can arrive or depart at any time in a dynamic system. When a new task (lets say T_i) arrives in a system, our algorithm first checks the period of the new task. If there is sufficient time to complete execution of the new task starting from the end of current time-slice, the scheduling of the new task is then procrastinated till the start of next time-slice. Otherwise, the system suspends it's execution from the current time instant, recomputes new time-slices by including consideration of the period of T_i, and then prepares a new schedule for the tasks. When a task departs from the system, it's entry is deleted from the utilization matrix and it is not considered for scheduling from the next time-slices.

5.5 An Illustrative Example

Let us try to solve the problem which was presented in Section 5.2. According to *HEALERS* (Algorithm 15), we first use *deadline partitioning* to compute the current time-slice. Since, all tasks are ready for execution, the first time-slice $TS_1 = [0, 10)$.

TH-2256_136101018



(a) *Schedule for non-migrating tasks)* **(b)** *Schedule for all tasks* **(c)** *Final energy-aware schedule for all tasks*

Figure 5.1: *An example to illustrate our proposed algorithm HEALERS*

In order to compute the schedule matrix SM_1 at TS_1, HEALERS calls Algorithm 16. First, it creates the lists L_1 and L_2 and initializes them to $\emptyset$. Then, it **computes the required shares** for each task $T_i \in T$ at TS_1. The computed shares are as follows:

$sh_{[7\times4]}$	$sh_{i,1,1}$	$sh_{i,2,1}$	$sh_{i,3,1}$	$sh_{i,4,1}$
T_1	5	13	8	12
T_2	8	7	10	11
T_3	11	8	4	9
T_4	9	9	8	1
T_5	9	11	11	14
T_6	9	6	1	8
T_7	12	4	7	2

Now, Algorithm 16 sorts L_1 based on computed shares in *non-decreasing* order, i.e., The sorted list L_1 at time-slice TS_1 is ($\langle i, j, sh_{i,j,1}\rangle$): $\langle 4,4,1\rangle$, $\{\langle 6,3,1\rangle$, $\langle 7,4,2\rangle$, $\langle 3,3,4\rangle$, $\langle 7,2,4\rangle$, $\langle 1,1,5\rangle$, $\langle 6,2,6\rangle$, $\langle 2,2,7\rangle$, $\langle 7,3,7\rangle$, $\langle 1,3,8\rangle$, $\langle 2,1,8\rangle$, $\langle 3,2,8\rangle$, $\langle 4,3,8\rangle$, $\langle 6,4,8\rangle$, $\langle 3,4,9\rangle$, $\langle 4,1,9\rangle$, $\langle 4,2,9\rangle$, $\langle 5,1,9\rangle$, $\langle 6,1,9\rangle$, $\langle 2,3,10\rangle\}$, $\langle 2,4,11\rangle$, $\langle 3,1,11\rangle$, $\langle 5,2,11\rangle$, $\langle 5,3,11\rangle$, $\langle 1,4,12\rangle$, $\langle 7,1,12\rangle$, $\langle 1,2,13\rangle\}$. Then, Algorithm 16 computes the schedule for non-migrating tasks.

Scheduling Non-migrating Tasks: Algorithm 17 extracts the first element from the list L_1, i.e., $\langle 4,4,1\rangle$ (task T_4 requires a share of 1 unit on V_4). The task T_4 can be fully allocated on V_4 and hence, the schedule matrix $SM_1[4][4]$ is updated as $\langle 0,1\rangle$. Since, T_4 has been completely scheduled on V_4, all entries of T_4, i.e. $\langle 4,3,8\rangle$, $\langle 4,1,9\rangle$ and $\langle 4,2,9\rangle$ have been removed from L_1. Now, the above process is repeated by extracting the first

TH-2256_136101018

element from L_1, i.e., $\langle 6, 3, 1\rangle$. Since, V_3 can fully accommodate T_6, it has been scheduled on V_3 with $SM_1[6][3] = \langle 0, 1\rangle$. Similarly, T_7 is scheduled (according to $\langle 7, 4, 2\rangle$) on V_4, T_3 is scheduled (according to $\langle 3, 3, 4\rangle$) on V_3, T_1 is scheduled (according to $\langle 1, 1, 5\rangle$) on V_1 and T_2 is scheduled (according to $\langle 2, 2, 7\rangle$) on V_2. Now, $\langle 5, 1, 9\rangle$ becomes the first element of the list L_1. According to this, if T_5 is allocated on V_1, it requires a share of 9 units. However, V_1 does not have the capacity to accommodate T_1 and hence, the element $\langle 5, 1, 9\rangle$ has been inserted into L_2. Similarly, the other three elements corresponding to the task T_5 has been moved to L_2, i.e., $\langle 5, 2, 11\rangle$, $\langle 5, 3, 11\rangle$ and $\langle 5, 4, 14\rangle$. Since, the list L_1 becomes empty, the execution moves from Algorithm 17 to Algorithm 18. The schedule matrix $SM_{1[7\times4]}$ for non-migrating tasks at TS_1, is as follows:

$SM_{1[7\times4]}$	V_1	V_2	V_3	V_4
T_1	$\langle 0, 5\rangle$	0	0	0
T_2	0	$\langle 0, 7\rangle$	0	0
T_3	0	0	$\langle 0, 4\rangle$	0
T_4	0	0	0	$\langle 0, 1\rangle$
T_5	0	0	0	0
T_6	0	0	$\langle 0, 1\rangle$	0
T_7	0	0	0	$\langle 1, 3\rangle$

Scheduling of Migrating Tasks: It may be observed that T_5 could not be allocated in the earlier phase and therefore it must be partitioned into multiple chunks and scheduled on more than one core using the SCHEDULE-MIGRATE algorithm (Algorithm 18). It first moves all entries corresponding to T_5 from list L_2 to L_3. Then, it extracts the first element from L_3, i.e. $\langle 5, 1, 9\rangle$, and computes the unused capacity of V_1. Since, there is a residual spare capacity uc_1 of 5 on V_1, T_5 has been partially allocated on V_1, $SM_1[5][1] = \langle 5, 10\rangle$. The unallocated share of T_5 with respect to V_1 $us_{5,1}$ becomes $9 - 5 = 4$.

Now, Algorithm 18 extracts the next element $\langle 5, 2, 11\rangle$ from L_3 and computes unallocated shares of T_5 with respect to V_2 (i.e. $us_{5,2} = 4 \times 1.1/0.9 = 4.8$ (approx)). Then, it checks the spare capacity of V_2. Since, the spare capacity of V_2 (i.e. $uc_2 = 3$), is not sufficient enough to accommodate the unallocated share of τ_5 (i.e. 4.8), it has been partially allocated on V_2. That is, $SM_1[5][2] = \langle 0, 3\rangle$. The unallocated share of T_5 become

TH-2256_136101018

$us_{5,2} = 4.8 - 3 = 1.8$. It may be noted that T_2 has already been scheduled at V_2 from 0 to 7. To avoid the overlap with T_5, the schedule of T_2 is updated as: $SM_1[2][2] = \langle 3, 10\rangle$.

Next, Algorithm 18 extracts the element $\langle 5, 3, 11\rangle$ from L_3 and checks the spare capacity of V_3. Since, the spare capacity of V_3 (i.e. $uc_3 = 5$) is sufficient enough to accommodate the unallocated share of T_5 (i.e. $us_{5,3} = 1.8 \times 1.1/1.1 = 1.8$), it has been allocated on V_3. That is, $SM_1[5][3] = \langle 3, 5\rangle$. The tasks that are already scheduled on V_3 are then adjusted to avoid overlaps. The Gantt chart representation of the schedule matrix $SM_{1[7\times4]}$ (including migrating tasks) for time-slice TS_1, is depicted in Figure 5.1b.

Energy-aware scheduling: As we can observe from the schedule matrix SM_1 (in Figure 5.1a), V_1 and V_2 do not have any spare capacity in the current time-slice. Hence, COMPUTE-EA-SCHEDULE allows V_1 and V_2 to run at their highest available frequency and we are not able to reduce any power consumption at these cores. Hence, total normalized power consumption on V_1 and V_2 (refer Table 5.2): $(P_d + P_s + P_{on}) \times 20$ $= (1327 + 714 + 276) \times 20 = 46.34W$. On the other hand, V_3 has a spare capacity of 3 time slots. This can be utilized to lower the operating frequency of V_3. However, the execution window of T_5 will overlap with its own execution on other cores, if the operating frequency is decreased during its execution. Hence, the spare capacity of 3 units is shared only among the non-migrating tasks T_6 and T_3 which together require 5 time units at frequency f_{max}. The required frequency (refer Table 5.2) to execute the tasks T_6 and T_3 is $f_{opt} = \lceil(1 + 4)/8\rceil = 0.68$. For simplicity of explanation, we are using only $\lceil f_{opt}\rceil$ in this example but the algorithm uses combination of two frequencies $\lceil f_{opt}\rceil$ and $\lfloor f_{opt}\rfloor$ practically. The modified execution shares of T_6 and T_3 becomes 2 $(= 2/0.68)$ and 6 $(= 4/0.68)$, respectively. Therefore, normalized power consumption on V_3: $= (655 + 461 + 276) \times 8 + (1327 + 714 + 276) \times 2 = 15.77W$. In core V_4, the required frequency to execute the tasks T_4 and T_7 is $f_{opt} = (1 + 2)/10 = 0.33$. Since, $f_{opt} < f_{cr}$ (0.41 in our system) , we set $f_{opt} = 0.41$. The modified execution shares of T_4 and T_7 becomes 3 $(= 1/0.41)$ and 5 $(= 2/0.41)$, respectively. V_4 is kept in suspension mode for the remaining 2 slots in the time-slice. Hence, normalized power consumption

TH-2256_136101018

on V_4: $= (267 + 289 + 276) \times 8 + (80\mu W) \times 2 = 6.65W$. The power consumption in the system using energy oblivious strategy would have been $(1327 + 714 + 276) \times 40 = 92.68W$. Therefore, the overall percentage of fractional power saved in a system is: P $= (92.68 - (46.34 + 15.77 + 6.65))/92.68 \times 100 = 25.80\%$. The final schedule for the given task set for TS_1 is shown in Figure 5.1c. Similarly, the schedule for subsequent time-slices can be computed.

5.6 Experimental Set Up and Results

We have implemented the *HEALERS* algorithm and compared the same against two algorithms: i) *MaxMin-M*, a variation of the *MaxMin* [16] algorithm, which is a DVFS-cum-DPM based task partitioning scheme, and ii) HEART [95], a DVFS based heuristic scheduler. Both of these energy-efficient algorithms are targeted towards heterogeneous multi-core platforms. Before presenting the detailed experimental set up and results, we now provide overviews of MaxMin and HEART.

Overview of MaxMin [16]: MaxMin is a heuristic energy-aware task allocation strategy targeted towards heterogeneous processing platforms. It uses a parameter called *Energy Density* ED_{ij}, which is defined as the dynamic energy consumption rate of the i^{th} task at the highest operating frequency f_{max}, of processing core j. The algorithm MaxMin works in four phases. In the first phase, it finds the *Maximum Energy Density* ED_i^{max} $(=max_{j=1}^{m}\{ED_{ij}\})$ and the *Minimum Energy Density* ED_i^{min} $(=min_{j=1}^{m}\{ED_{ij}\})$ for each task T_i, over all processing cores and calculates the difference ED_i^{diff} $(= ED_i^{max} - ED_i^{min})$. Each task T_i is then inserted into a list L_{ED}, which is sorted in non-increasing order of ED_i^{diff}. In the second phase, each task in L_{ED} (starting with the first) is allocated to its most preferred core such that it's entire execution demand can be satisfied on that core. In the third phase, the algorithm finds a suitable operating frequency for each core based on workloads assigned in the previous phase. In the last phase, MaxMin further refines the allocation to apply DPM technique and achieve better energy savings.

TH-2256_136101018

Modified MaxMin (MaxMin-M) Algorithm: The basic MaxMin algorithm does not allow a single task to be allocated to multiple cores, where each core satisfies a distinct fraction of its total execution demand. The literature on bin packing strategies [22, 88] show that such a mechanism which does not allow inter-processor task migrations may lead to very poor resource utilizations. Hence, we have used a modified version of MaxMin called *MaxMin-M*, which embeds the strategy over a deadline partitioning framework. The MaxMin-M algorithm divides time into slices demarcated by the arrivals/departures of all tasks in the system (also known as *Deadline Partitioning* (DP)). At the beginning of each time-slice, MaxMin-M determines the proportional execution share for each task within the time-slice. Inside each time-slice, the basic MaxMin algorithm is applied for energy-aware task allocation on heterogeneous cores. The system re-synchronizes globally at the end of every time-slice. Such a strategy, enables MaxMin-M to allow migration at the end of every time-slice boundary and hence deliver significantly better resource utilizations compared to basic MaxMin.

Overview of HEART [95]: HEART is a heuristic deadline partitioning based strategy for scheduling periodic tasks on DVFS enabled heterogeneous multi-core platforms. Initially, the algorithm generates schedule for all tasks based on their execution demands on available processing cores. Next, it applies DVFS on all processing cores to minimize energy consumption of the system. Their energy-aware strategy is different from HEALERS in following two ways. Firstly, since the exact required operating frequency f_{opt} for a core may not be available in the frequency set, HEART chooses the nearest available frequency $\lceil f_{opt} \rceil$ which is sufficient to meet a core's workload. This leads to wastage of energy in the system. On the other hand, HEALERS uses a combination of two different frequencies ($\lfloor f_{opt} \rfloor$ and $\lceil f_{opt} \rceil$) within each time-slice, such that the total workload executed on individual cores matches more closely with the workload when executed throughout f_{opt} . Secondly, HEART only considers dynamic power consumption in the system. On the other hand, HEALERS uses a DVFS-cum-DPM strategy to min-

TH-2256_136101018

imize the system-wide energy consumption. This DPM strategy becomes useful when there are long periods of time with relatively low CPU utilization. The experimentation framework used in our work is discussed next.

5.6.1 Experimental Set Up

All our simulations have been run for a total execution time of 100000 time-slots on systems having 4 processing cores. For each set of input parameters, we ran the simulation on 50 different test cases. A parameter called *Utilization Factor (UF)* is used in order to obtain a measure of resource utilization corresponding to a given task set. UF is defined as $\frac{\sum_{i=i}^{n} avg_{j=1}^{m}(u_{i,j})}{m}$, where $\sum_{i=i}^{n} avg_{j=1}^{m}(u_{i,j})$ is the summation of the average utilization of the tasks over the m processing cores. For creating task sets with a specific UF, the randomly generated utilization values have been scaled appropriately. Results are generated for various distinct utilization factors. The following three metrics have been used to compare the performance of our proposed algorithm against MaxMin-M. The first metric, *ARat*, defines *acceptance ratio* as the number of task-sets successfully scheduled by the system against the total number of task-sets submitted to it. The second metric is *NPow*, which represents the *normalized power consumption* in the system and the last metric is *CSC*, which represents the average context switch cost per core per time slot (in μs). Two sets of experiments have been carried out to validate and compare our proposed algorithm with MaxMin-M. In the first set, we have used benchmark programs to test the algorithms in real-life situations, whereas in the second set, extensive simulation based experiments have been carried out using synthetic task sets to validate the efficacy of the algorithms over varied scenarios that may be encountered.

TH-2256_136101018

Application	Execution Time (in ms)	Application	Execution Time (in ms)
body	3120	stream	11820
can	12300	swap	34500
fluid	960	x264	60
freq	1440	bsort	9
duff	73	edn	62
lms	146	ndes	8340
qurt	130	select	135

Table 5.3: *Execution Requirements of programs for Parsec [129] and Mälardalen benchmarks [62]*

Framework for Benchmarks: We selected a set of 14 tasks from Parsec [129] and Mälardalen [62] benchmarks to evaluate the performance of our algorithm. A detailed discussion on the procedure for measuring execution requirements of each program is presented in [36] and the results have been listed in Table 5.3. These values have been obtained with the help of Gem5 [33] for an Intel Xeon processor, 65 nm CMOS technology, operating at 3.1 GHz. For creating task sets with a specific UF, periods of these tasks have been generated appropriately.

Framework for Synthetic Tasks: The performance of HEALERS has been evaluated and compared through a set of carefully chosen simulation based experiments. Each task set considered is composed of randomly generated hypothetical periodic tasks, whose execution requirements are generated from a Normal Distribution having standard deviation $\sigma_e = 10$ and mean $\mu_e = 50$. The size of the task sets have been varied from n = 10 to n = 40. Entries in the utilization matrix ($U_{[n \times m]}$) are also obtained from a Normal Distribution with standard deviation of $\sigma_u = 0.2$ and mean $\mu_u = 0.4$.

5.6.2 Experimental Results

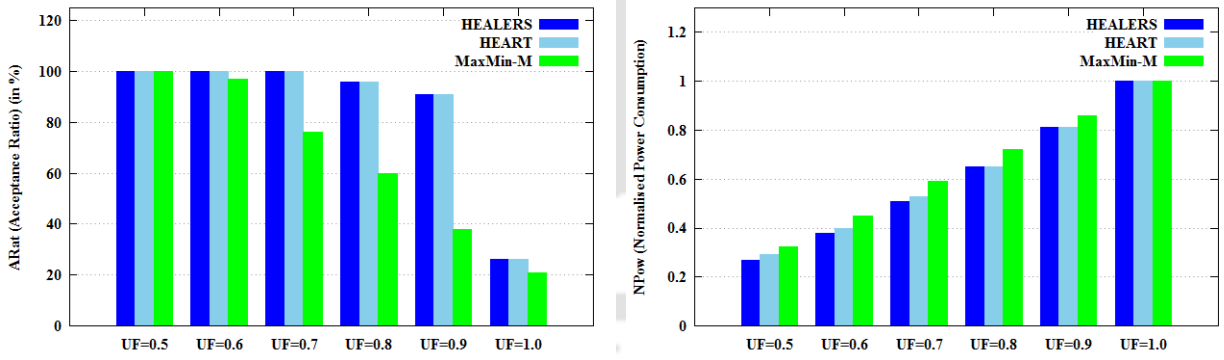
5.6.2.1 Benchmark Program Results

To compare and analyze our results with that of MaxMin-M and HEART, we have measured *ARat* and *NPow* for different values of utilization factors (UF). Detailed analysis

TH-2256_136101018

of our experimental results is presented below.

Experiment 1- *Effect on Acceptance Ratio (ARat)*: In this experiment, we have varied UF between 0.5 and 1.0. We may observe from Figure 5.2a that at lower UF upto about 0.6, MaxMin-M, HEART and HEALERS exhibit similar performance efficiencies. However, HEALERS and HEART progressively outperform MaxMin-M as UF increases beyond 0.6. This phenomenon may be attributed to the non-migrative execution policy of MaxMin-M within time-slices. At lower UF values (below 0.6), the probability that MaxMin-M can successfully allocate entire execution shares of every task onto a single processing core, is very high. However, at higher UF values, the possibility of allocating every task onto cores without causing any migration becomes lower. Therefore, MaxMin-M shows poorer performance compared to HEALERS and HEART. On the other hand, both HEART and HEALERS use similar task allocation strategies. Hence, they may be seen to achieve similar results for this experiment. In particular, ARat reduces from 100% to 21%, 100% to 26% and 100% to 26% for MaxMin-M, HEART and HEALERS, respectively.



(a) *Effect on Acceptance Ratio* ARat **(b)** *Effect on Normalized Power Consumption* NPow

Figure 5.2: *Result Comparison for Benchmark programs with varying UF (*m *= 4 and* n *= 30)*

Experiment 2- *Effect on Normalized Power Consumption (NPow)*: As the value of ARat for MaxMin-M is significantly lower than that of HEART and HEALERS,

TH-2256_136101018

only those task-sets which have been successfully completed by all the algorithms have been considered in this experiment. We may observe from Figure 5.2b, that NPow is directly proportional to the UF of the system. This is because residual capacity in the system decreases with an increase in UF of the task-set, thereby reducing the scope for lowering core operating frequencies. This leads to higher power consumption with an increase in UF. As discussed earlier, MaxMin-M sequentially allocates tasks to their most favoured processing cores in the order of their ED_{diff} values. In comparison, HEART always runs the processing cores on next available frequency in the frequency set. Finally, HEALERS tries to directly allocate tasks to their most preferred processing cores in the order of increasing execution requirements and run individual cores on a combination of two frequencies ($\lfloor f_{opt} \rfloor$ and $\lceil f_{opt} \rceil$) within a time-slice. This allows HEALERS to perform slightly better than MaxMin-M and HEART in most scenarios where the system has some spare capacity, so that most tasks get assigned to cores where their energy efficiencies are comparatively high. In particular, improvements in energy savings for HEALERS over MaxMin-M and HEART varies from 5.81% to 16.92% and 3.77% to 6.89% with variation in UF values from 1.0 to 0.5, respectively.

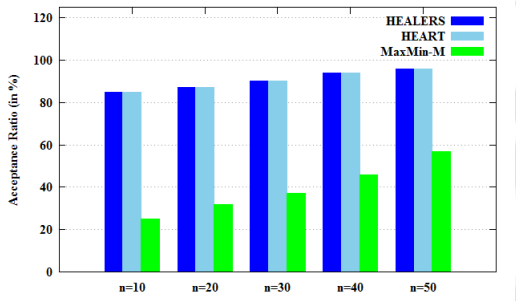
5.6.2.2 Synthetic Task Set Results

We have measured and compared the *ARat*, *NPow* and *CSC*values of HEALERS, HEART and MaxMin-M for different values of the number of tasks (n) and Utilization Factors (UF) with constant number of processing cores (m) of 4. Detailed analysis of our experimental results is discussed below.

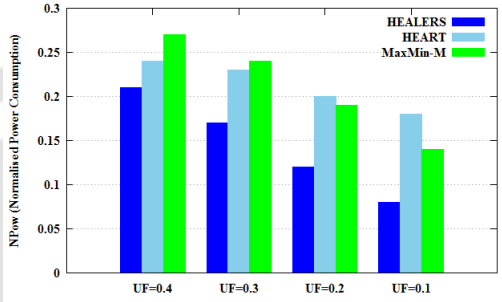
Experiment 3- *Effect on Acceptance Ratio (ARat)*: In this experiment, we have varied the number of tasks (n) between 10 and 50, when UF is kept constant at 0.9. We may observe from Figure 5.3a that there is a significant difference in the *ARat* values of HEALERS, HEART and MaxMin-M at $UF = 0.9$. Also, we can observe that increase in the number of tasks results in increased acceptance ratio for both the algorithms. As utilization factor and number of processing cores remain constant, an increase in the number of tasks leads to reduced individual task weights, which leads to less number of

TH-2256_136101018

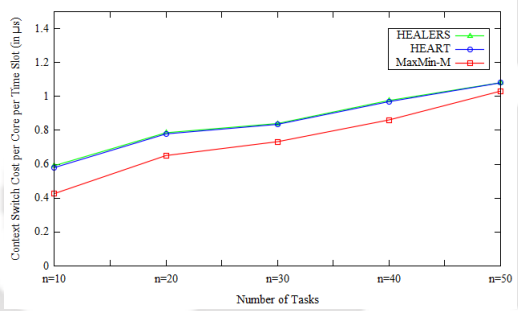
migrating tasks in the system. Hence, both algorithms are able to perform better with increase in tasks. As stated earlier, HEALERS and HEART uses similar task-allocation strategy. Hence, they have similar *ARat* values for this experiment. In particular, acceptance ratios ARat were found to be 85%, 87%, 90%, 94% and 96% for *HEALERS* (and *HEART*) and 25%, 32%, 37%, 46% and 57% for *MaxMin-M*.



(a) *Effect on Acceptance Ratio* ARat *(m=4 and $UF = 0.9$)*



(b) *Effect on Normalized Power Consumption* NPow *(n=40 and m=4)*



(c) *Effect on Cost of Context Switch*CSC *(m=4 and $UF = 0.3$)*

Figure 5.3: *Result Comparison for Synthetic tasks*

Experiment 4- ***Effect of the critical frequency***: In this experiment, we have varied the UF values between 0.4 and 0.1, when the number of tasks n is kept constant at 30. We may observe from Figure 5.3b that the *NPow* values decreases for all the algorithms with reduction in UF values. As stated earlier (in Experiment 2), the algorithms are able to achieve higher savings at lower UF values because of the availability of more residual capacities. Further, we may observe that improvements in energy savings for HEALERS

TH-2256_136101018

over HEART increases as UF is reduced from 0.4 to 0.1. This is due to the fact that HEART only employs DVFS to reduce energy consumption and it is oblivious of the static power dissipated in the system. In the absence of DPM, as the required operating frequency becomes lower compared to the critical frequency (which is 0.41 in our case), the relative static power wastage in the system gets higher. Hence, HEALERS is able to perform better when the optimal operating frequency is lower than critical frequency. On the other hand, MaxMin-M also deploys a DVFS-cum-DPM based heuristic strategy and hence, its is able to perform better than HEART for small values of UF. However, HEALERS outperforms even MaxMin-M with it's finely tuned heuristic approach. In particular, improvements in energy savings for HEALERS over MaxMin-M and HEART varies from 22.22% to 42.85% and 12.5% to 55.55% with variation in UF values from 0.4 to 0.1, respectively.

Experiment 5- *Cost of Context Switch (CSC)*: In our experiments, we have assumed the delay corresponding to a single context switch to be 5.24 μs [29], which is the actual average context switch overhead on a 24-core Intel Xeon L7455 system under typical workloads [29]. We computed the total number of context switches in the system over the entire simulation duration. Then, we obtained the total delay due to context switches by multiplying the delay caused by a single context switch (5.24μs [29]) with total number of context switches. Next, we calculated the average context switch overhead (in μs) per core per time slot for the HEALERS, HEART and MaxMin-M algorithms. As observed from Figure 5.3c, HEALERS and HEART incur similar context switches but MaxMin-M incurs slightly fewer context switches compared to other algorithms. This is because MaxMin-M uses a non-migrative scheduling strategy within time-slices, whereas HEALERS and HEART are fully-migrative strategies. Higher $ARat$ values as achieved by HEALERS and HEART is essentially effected by appropriately switching task executions on cores (using preemptions and migrations) so that tasks can be scheduled in an efficient manner in the system. To include the cost of context switches into the overall power consumption, the operating frequencies of the individual

TH-2256_136101018

cores were adjusted accordingly. For example from Figure 5.3c, we observe that for 30 tasks, HEALERS suffers a context switch overhead of $\sim 0.869\mu s$ while Max-Min-M suffers a context switch overhead of only $\sim 0.752\mu s$. So, HEALERS incurs an extra overhead of $0.117\mu s$ per core per task per time slot with respect to MaxMin-M. Given a time slot size of $1ms$, HEALERS will therefore be able to complete as much work in $8548ms$ as MaxMin-M will complete in $(1ms/0.117\mu s =)$ $8547ms$. To overcome this additional context switching overhead, HEALERS must execute at 8548/8547 times the operating frequency of MaxMin-M. As all our experiments have been conducted assuming only 11 available discrete frequency levels (refer Table 5.2), this additional overhead in terms of calculated frequency very rarely translates into an increase in the discrete frequency level of a core. Hence, the cost of context switching did not have any significant effect on power consumption in our experiments.

5.7 Summary

In this chapter, we have proposed a low-overhead heuristic strategy called, *HEALERS*, for the energy-aware scheduling of a set of periodic tasks on a heterogeneous multi-core platform. The proposed algorithm works in three phases to reduce energy consumption in the system. In the first phase, it computes the fragments of the execution requirements of all tasks on different processing cores of the heterogeneous platform. In the second phase, it generates a schedule of each task on one or more processing cores, such that the total execution demand of all tasks is satisfied. Finally, HEALERS applies DVFS and DPM on all processing cores to minimize the energy consumed in the system. Experimental studies show that our proposed scheduling scheme is able to significantly improve acceptance ratios for task sets and energy savings of the platform, compared to the state-of-the-art. In the next chapter, we present *TARTS*, a low overhead temperature-aware semi-partitioned algorithm for scheduling periodic tasks in a proportional fair manner.

TH-2256_136101018

Chapter 6

Temperature-Aware resource allocation strategy for real-time systems

The advent of deep sub-micron VLSI technologies have resulted in extremely dense multi-million gate chips, where power dissipation and thermal hotspots have become very serious design concerns [117]. Uncontrolled rise in temperature not only increases cooling costs, but may also reduce a system's efficiency and life span. A study in [80] has shown that the lifespan of a chip can be reduced by upto 50% with a temperature increase of $10 - 15°C$ beyond normal operating temperature. Therefore, modern multi-core processors typically come with a stipulated temperature threshold which must be adhered to for safe and efficient operation of the system. Dynamic Thermal Management (DTM) mechanisms which enforce performance throttling in order to mitigate temperature hot-spots, are triggered whenever operating temperature in the system crosses the stipulated temperature threshold. Activation of DTM involves steps such as powering-down processors, clock gating, dynamic supply voltage and frequency scaling, etc. These steps introduce unpredictability in a system's timing behavior. An important objective of real-time scheduler designs implemented on these platforms is therefore, to ensure DTM-free operation over the entire schedule length. This can be achieved by always ensuring the operating temperature of the processors to be within a specified temperature threshold [99]. *Given, a set of real-time tasks along with their temperature characteristics and a multi-core platform, successfully satisfying execution and deadline requirements of all task instances while guaranteeing that all cores operate within a specified temperature*

TH-2256_136101018

threshold, is ultimately a scheduling problem.

Our work: Using DPFair as the underlying scheduling approach, this work proposes a two-level temperature-aware semi-partitioned scheduling strategy for multi-core systems. At the first level, time-slices are determined based on the deadlines of tasks and task execution proceeds time-slice by time-slice, in a proportional fair manner. At any time-slice boundary, shares of all tasks to be executed in the next time-slice, are determined. The second level performs intra-time-slice schedule generation. Each slice is further divided into a sequence of possibly unequal time-intervals called *frames*, and a schedule is generated for each frame in a sequential manner. Given the temperatures of the cores at a frame boundary (say at time t_1) and the set of m task-to-core assignments (with known temperature characteristics), size of the next frame is given by the maximum context-switch free time-interval which do not lead to the possibility of temperature threshold violation for any core. If the frame size is positive, the selected task-to-core assignments are used to construct the schedule for the frame. If the obtained frame size is zero, a different task-to-core assignment must be used for the frame. Experimental results show that TARTS is able to perform appreciably under various realistic scenarios while incurring slightly higher context-switch/migration overheads compared to the existing temperature oblivious version of DPFair [78].

The rest of the chapter is organized as follows. In the next section, we explain the system and power models used for our work, in Section 6.1. Section 6.3 presents the proposed algorithm along with it's pseudo-code followed by an illustrative example in Section 6.4. The experimental framework used in our work along with the results are presented in Section 6.6. Finally, we conclude our work in Section 6.7.

6.1 System and Power Model

6.1.1 System Model

The system under consideration consists of a set of n periodic tasks $T = \{T_1, T_2, ..., T_n\}$ to be scheduled on a set of m homogeneous multi-cores $V = \{V_1, V_2, ..., V_m\}$. Each instance of T_i has an execution requirement e_i to be completed within period p_i and also

TH-2256_136101018

a temperature characteristic defined by $\Gamma_{ss}(i)$. Time is measured on a discrete scale in terms of an integral number of time-slots, where each time-slot is assumed to be $1ms$ in this work. The weight of any task T_i is defined as $wt_i = e_i/p_i$. At any given instant, re_i and rp_i denote the remaining execution requirement and remaining period of the current instance of T_i.

6.1.2 Thermal Model

The rate of change of temperature of a processor is modeled through the following equation [64]:

$$\frac{d\Gamma(t)}{dt} = \frac{P(t)}{C} - \frac{(\Gamma(t) - \Gamma_{amb})}{RC}, \tag{6.1}$$

where, $\Gamma(t)$ and Γ_{amb} are the processor's temperature and ambient temperature, respectively. $P(t)$ denotes the power consumption at time t, while R and C denote the processor's thermal resistance and thermal capacitance, respectively. Scaling $\Gamma(t)$ such that Γ_{amb} is zero (i.e., by replacing $\Gamma(t)$ - Γ_{amb} with $\Gamma(t)$), Equation 6.1 gets transformed as:

$$\frac{d\Gamma(t)}{dt} = aP(t) - b\Gamma(t), \tag{6.2}$$

where, $a = 1/C$ and $b = 1/RC$.

For a processor with discrete voltage/frequency levels, the overall power consumption, including dynamic as well as leakage power, when operating at the k^{th} voltage level, can be modeled as [64]:

$$P(k) = (C_0(k) \times v_k + C_1(k).\Gamma(t).v_k) + C_{ef} \times v_k^3 \tag{6.3}$$

where C_{ef} is the effective switching capacitance and $C_0(k)$ and $C_1(k)$ are curve fitting constants which are dependent on voltage level v_k. Accordingly, the rate of temperature variation can be formulated from Equation 6.2 and Equation 6.3 as:

$$\frac{d\Gamma(t)}{dt} = a[C_0(k) \times v_k + C_{ef}.v_k^3] + a.C_1(k).\Gamma(t).v_k - b.\Gamma(t)$$

$$\Rightarrow \frac{d\Gamma(t)}{dt} = A(k) - B(k)\Gamma(t), \tag{6.4}$$

TH-2256_136101018

Symbol	Description
n	Number of tasks
T_i	i^{th} task
e_i	Execution requirement of i^{th} task
p_i	Period of i^{th} task
wt_i	Weight of i^{th} task
rp_i	Remaining period of T_i at any moment
$time$	Current time instant
TS_k	k^{th} time-slice
shr_i	Share of T_i in a time-slice
shr_i^{rem}	Remaining execution share of T_i in a time-slice
m	Number of cores in the system
V	Set of cores
V_k	k^{th} core
Γ_i^{ss}	Steady state temperature of T_i
Γ_l	System level temperature constraint or threshold limit
shr_i^{rem}	Remaining shares of T_i in TS_k (Eqn. 6.11)
t_{ij}^l	Time duration by which core V_j executing task T_i reaches the threshold temperature Γ_l (Eqn. 6.9)
$\triangle_j$	Maximum time duration for which continuous execution of T_i is possible on V_j (Eqn. 6.10)
sp	Remaining sparable time-slots in time-slice TS_k (Eqn. 6.8)
g	Size of the current frame (Eqn. 6.12)
M	A task-core mapping array; $M[j] \leftarrow i$, if task T_i is assigned to core V_j
TL	Sorted list of active task instances in system based on e_i/p_i
FL	List of tasks who have completed their execution requirements in TS_k
U	Utilization Factor of a taskset
$ATemp$	Average steady state temperature of a taskset

Table 6.1: *Important Terminologies*

where $A(k) = a.(C_0(k) \times v_k + C_{ef} \times v_k^3)$ and $B(k) = b - a.C_1(k).v_k$.

For an interval $[t_0, t_e]$ in which T_i is executing on V_j, if the processor temperature is Γ_0 at time t_0, the temperature Γ_e at the end of the interval at time t_e is given by [85]:

$$\Gamma_e = \frac{A(k)}{B(k)} + (\Gamma_0 - \frac{A(k)}{B(k)})e^{-B(k)(t_e - t_0)}$$

$$\Rightarrow \Gamma_e = \Gamma_i^{ss}(k) + (\Gamma_0 - \Gamma_i^{ss}(k))e^{-B(k)(t_e - t_0)} \tag{6.5}$$

where Γ_i^{ss} is the steady state temperature of task T_i and is dependent on the overall power consumption of T_i. The steady state temperature of a task represents the core temperature that is reached when the task executes continuously on the core for a

TH-2256_136101018

sufficiently long time, possibly over multiple instances.

6.2 Motivational Example

Consider a list of 5 tasks having specifications as shown in Table 6.2, to be scheduled in a system of $m = 2$ cores, both of which have an initial temperature of $59°C$ with ambient temperature being $25°C$. Let the system level temperature threshold (Γ_l) be $60°C$. For the simplicity of explanation, we have chosen the relative deadlines to be equal to 45 time-slots for all tasks. The scheduling strategy in such a system should not only be able to allot and schedule tasks on the available cores but it has to do it in such a way that temperatures of all cores remains below a stipulated threshold, which makes it a challenging problem. In the following section, we present the working principle of *TARTS* algorithm which is able to propose solution for such problems.

Table 6.2: *Task Specifications for Example*

T_i	e_i	p_i	$\Gamma_i^{ss}(f_{max})$
0	10	45	60
1	12	45	40
2	19	45	80
3	25	45	75
4	2	45	40

6.3 The TARTS Algorithm

The proposed scheduling algorithm *TARTS* (*Temperature-Aware Real-Time Semi-partitioned Scheduler*) is a two-level hierarchical fair resource-allocation strategy targeted towards homogeneous real-time multi-core systems. These two levels together attempt to schedule a given set of tasks in such a way that resource utilization is maximized while ensuring deadlines and a stipulated system-wide temperature threshold bound (Γ_l).

At the first level, time is partitioned into slices (k^{th} time-slice is denoted by TS_k) demarcated by the deadlines of all tasks in the system. At the $(k-1)^{th}$ time-slice boundary, length $|TS_k|$ of the ensuing time-slice TS_k is obtained by the earliest deadline

TH-2256_136101018

among all active tasks. Then, TARTS computes the execution shares/workloads for all tasks within TS_k. At the second level, time-slice TS_k is further divided into a sequence of possibly unequal time intervals called *frames*. As has been elaborated in the algorithm discussed below, the frames, whose sizes are dynamically determined, denote locally maximal context-switch free time-intervals. Context-switches are only allowed at the boundaries of the frames. The exact schedule and task-to-core assignments within each frame is determined next.

Algorithm TARTS is composed of the following three functions: TARTS() (Algorithm 20), Task_Schedule()(Algorithm 21) and Find_Mapping() (Algorithm 22). The main function TARTS(), which conducts the overall scheduling, calls function Task_Schedule() at each time-slice boundary to divide the ensuing time-slice into a non-uniform group of time-slots called *frames*, based on the task's workload demands and instantaneous core temperatures. Within each frame, the function Task_Schedule() determines a tentative task-to-core mapping and ensures that the tasks are scheduled in *highest urgency first* order while ensuring that temperature constraints are not violated. At last, it repeatedly calls function Find_Mapping() for each frame, to prepare the final schedule for the time-slice. We now discuss the TARTS algorithm in detail.

6.3.1 Function TARTS()

It starts by constructing constructing TL, the list of all the ready tasks in the system. At the start of the k^{th} time-slice TS_k, TARTS determines the size $|TS_k|$ of the ensuing time-slice. This is accomplished by finding the earliest among the remaining periods of all task instances. That is,

$$|TS_k| = min\{rp_1, rp_2, ..., rp_n\} \tag{6.6}$$

Next, the shares (shr_i) of each task T_i to be executed in time-slice TS_k is determined

TH-2256_136101018

ALGORITHM 20: Function TARTS()

Input: Taskset T, Core Set V
Output: The complete schedule for taskset T

```
1  Let sp denote the number of spare slots in |TS_k|
2  k ← 0; time ← 0
3  while true do
4      Construct TL, the list of ready tasks
5      Determine length |TS_k| of next time-slice TS_k (Eqn. 6.6)
6      for each task T_i at the beginning of TS_k do
7          Determine shr_i (Eqn. 6.7)
8      sp ← m × |TS_k| − Σ_{i=1}^{n} shr_i
9      Sort the list TL in non-increasing order of their share values
10     flag = function Task_Schedule( TL, V, sp, TS_k)
11     if flag = false then
12         Remove instance of the hottest task from TL
13         Go to Step 3
14     Execute tasks in TS_k according to the schedule generated by Task_Schedule()
15     (Algorithm 21) k ← k + 1; time ← time + |TS_k|
```

as:

$$shr_i = \frac{e_i \times |TS_k|}{p_i} \tag{6.7}$$

Thereafter, the total number of sparable time-slots in TS_k at the beginning of the time-slice is determined as follows:

$$sp = m \times |TS_k| - \sum_{i=1}^{n} shr_i \tag{6.8}$$

Then the task list TL is sorted in non-increasing order of share values of the tasks. Finally TARTS() calls function *Task_Schedule()* (Algorithm 21) to determine the intra time-slice schedule of tasks. If function *Task_Schedule()* is unable to find a feasible schedule for the tasks in current time-slice, the hottest task instance from the list TL is removed and it again tries to a find a schedule for the remaining tasks. The pseudo-code for function *TARTS()* is presented in Algorithm 20.

TH-2256_136101018

6.3.2 Function Task_Schedule()

It divides length of the time-slice into a group of non-preemptive execution windows called *frames* and generates the schedule in a frame-by-frame manner. This schedule also determines the exact frame size, which is obtained as follows. At the beginning of every frame, *Task_Schedule()* first conducts a tentative mapping of the m most urgent tasks onto available cores, using the *hottest task on coolest core* heuristic. For this purpose, the m most urgent tasks in list TL are extracted and arranged in non-increasing order of their steady state temperatures. Correspondingly, the cores are arranged in non-decreasing order of their chip temperatures. Then, the elements at the head of the sorted task and core lists are successively extracted and mapped to obtain the task-to-core assignment array $M[m]$. Element $M[j] \leftarrow i$, if task T_i is assigned to core V_j. Subsequently, function *Find_Mapping()* is called to validate the obtained tentative mapping M and determine the final feasible task-to-core assignments for the frame. If function *Find_Mapping()* is able to find a feasible task-to-core mapping, then based on the remaining execution requirements of the tasks in a time-slice, the tasks are either added to the list of finished tasks FL, or to the list TL for scheduling in subsequent frames. The pseudo-code for function *Task_Schedule* is presented in Algorithm 21.

6.3.3 Function Find_Mapping()

This function attempts to map tasks onto available cores such that the stipulated temperature threshold is not breached on any core, over the length of a frame. In order to achieve this, for each mapping $M[j] \leftarrow i$ of array M, the function determines the time duration $t^r_{M[j],j}$ (refer Equation 6.9) by which core V_j (executing task $M[j]$ having steady state temperature $\Gamma^{ss}_{M[j]}$) reaches the threshold temperature Γ_{lim} starting with initial temperature Γ_j. Equation 6.9 can be easily derived from Equation 6.5 by replacing Γ_e with Γ_l and Γ_0 with Γ_j:

$$t^r_{M[j],j} = \frac{-1}{B}.ln(\frac{\Gamma_{lim} - \Gamma^{ss}_{M[j]}}{\Gamma_j - \Gamma^{ss}_{M[j]}}) \tag{6.9}$$

TH-2256_136101018

ALGORITHM 21: Function Task_Schedule()

Input: TL, V, sp, TS_k
Output: Task schedule for TS_k

```
1  Let, (i) FL be the list of tasks who have completed their execution requirements
     in time-slice TS_k, (ii) M[m] be a task-core mapping array; M[j] denotes the task
     assigned to the j^th core, and (iii) time captures the progression of intra-slice time
2  Set time ← 0
3  while time < |TS_k| do
4      Set M[j] ← φ; ∀j, 1 ≤ j ≤ m
5      Set current frame size, g ← |TS_k| − time
6      Initialize the list: FL ← φ
7      Extract the first m tasks from TL and populate array M using the hottest task
         on coolest core heuristic
8      Call M = Find_Mapping(TL,FL,M, sp, TS_k)
9      if M ≠ φ then
10         for j ← 1 to m do
11             shr_M[j] ← shr_M[j] − g
12             if shr_M[j] = 0 then
13                 Insert T_M[j] into FL
14             else
15                 Insert T_M[j] into the sorted list TL
16         time ← time + g
17     else
18         {Schedule to be backtracked to the beginning of time-slice}
19         Add all elements of the list FL to TL
20         return false
21 return true
```

It may be noted that remaining share $shr^{rem}_{M[j]}$ of the task $M[j]$ may not always be large enough to reach the threshold temperature Γ_{lim}. Hence, the maximum duration $\triangle_j$, for which continuous execution of $M[j]$ is possible on V_j, is obtained as:

$$\triangle_j = min\{shr^{rem}_{M[j]}, \lfloor t^r_{M[j],j} \rfloor\} \tag{6.10}$$

Values of $\lfloor t^r_{M[j],j} \rfloor$ less than 1 indicates that the temperature of V_j exceeds the stipulated threshold even if $M[j]$ is scheduled for the minimum allowable time quantum (1 time-slot). Hence, $M[j]$ cannot be currently allocated to V_j. More importantly, the *hottest task on coolest core* policy signifies that $M[j]$ cannot be currently allocated to any of the available cores, even for a single time-slot. In order to handle this situation,

TH-2256_136101018

ALGORITHM 22: Function Find_Mapping()

Input: TL, FL, M, sp, TS_k
Output: Final mapping for TS_k

Let, i. UL denote list of unscheduled tasks, ii. shr_i^{rem} denote remaining execution shares of T_i, and iii. $\forall j$ $(1 \leq j \leq m)$: M[j] denote the current tentative task-to-core mapping
Set restart flag $fl \leftarrow 0$
for $j \leftarrow$ *1 to m* **do**
    Find $\triangle_j$ (Equation 6.10)
    if $\triangle_j > 0$ **then**
        Lock the mapping $M[j]$ for V_j
        $g \leftarrow \min\{g, \triangle_j\}$
    else if $shr_{M[j]}^{rem} < (|TS_k| - time - 1)$ **then**
        Add $T_{M[j]}$ to the unscheduled task list UL
        Discard the current mapping for V_j; $M[j] \leftarrow \phi$
        Set $g \leftarrow 1$
        Extract coolest task T_k from TL
        Find $\triangle_j$ (Equation 6.10) using T_k
        if $\triangle_j > 0$ **then**
            Update mapping for V_j; $M[j] \leftarrow k$
            break
        else
            Add T_k to UL
        if $sp > 0 \bigwedge M[j] = \phi$ **then**
            Keep V_j idle for the current time-slot
            $sp \leftarrow sp - 1$
        else
            {V_j's temperature is not cool enough to allow execution of any task, even for one slot and there is no remaining capacity to allow V_j to be kept idle for current slot}
            Set $M \leftarrow \phi$; exit
    else
        {$T_{M[j]}$ cannot be scheduled or delayed}
        Set $M \leftarrow \phi$; exit

Find_Mapping() attempts to construct a temporally and thermally feasible schedule by delaying the allocation of $M[j]$ by at least one time-slot. However, $M[j]$ can only be delayed if the remaining shares $shr_{M[j]}^{rem}$ of $M[j]$ is less than the remaining time in the

TH-2256_136101018

current time-slice TS_k:

$$shr^{rem}_{M[j]} < (|TS_k| - time - 1) \tag{6.11}$$

If the condition in Equation 6.11 is true, then the allocation of $M[j]$ is cancelled for the current time-slot and an attempt to progress with the generation of the partial schedule (by one time-slot) is made in two alternative ways: **i. Substitute-Allocation:** Allocating V_j to the highest priority task T_k in the ready queue (T_k has not been allocated in the current frame) for which the condition: $t^r_{kj} > 1$, holds. That is, task T_k does not induce a breach of the temperature threshold on V_j when executed for one time-slot. Next, $M[j] \leftarrow k$ is set in the mapping array M. **ii. Spare-Slot-Allocation:** If no task in the ready queue satisfies the above condition, core V_j is left idle (thereby, allowing it to cool down) for the current time-slot, i.e. $M[j] \leftarrow \phi$ is set in the mapping array M. In both the cases, $\triangle_j$ is set to 1, so that $M[j]$ may again get a chance to be scheduled in the next time-slot and doesn't miss it's deadline. However, this is possible only if there remains sparable time-slots in time-slice TS_k. The value of sp is decremented by one every time a spare slot is allocated during schedule generation. After successfully performing task-to-core assignments for the ensuing frame, size of the frame is determined in the following way:

$$g = min^m_{j=1}\{\triangle_j\} \tag{6.12}$$

It may be observed that frame size g has an effect on: (i) the degree of control that TARTS can impose on core temperature, and, (ii) context-switch overheads. Lower values of g compared to that obtained in Equation 6.12 results in more frequent applications of the *hottest task on coolest core* heuristic by TARTS, which thereby allows finer control over core temperatures. However, an adverse side-effect of this approach is the increase in induced preemption overheads. Therefore, based on core temperatures at a frame boundary and task-to-core assignments (with known temperature characteristics), TARTS dynamically selects current frame size ($g \geq 1$) as the maximum context-switch free time-interval that do not lead to the possibility of the safe temperature threshold

TH-2256_136101018

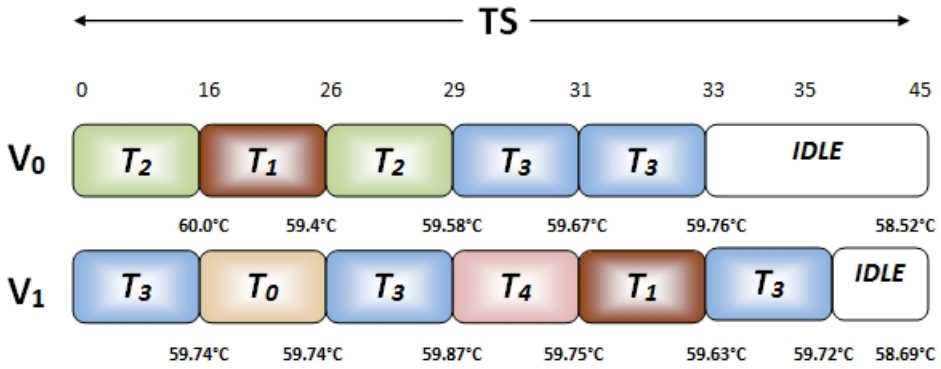


Figure 6.1: *Task Allotment for Example*

violation for any core. With dynamic selection of frame sizes, TARTS is able to achieve a performance almost similar to TARTS with fixed frame size ($g = 1$), as shown in the experimental results. The pseudo-code for the function *Find_Mapping()* has been shown in Algorithm 22.

6.4 An Illustrative Example

Let us try to solve the problem which was presented in Section 6.2. In TARTS, the hottest task is always tried to be allotted first to the coolest available cores. At start of the time-slice, T_3 and T_2 are extracted from the priority to find value of g (frame size). Since, T_3 is able to execute for 19 time-slots (refer Equation 6.9) and T_2 for 16 time-slots (refer Equation 6.9), without breaching the temperature threshold, the value g is chosen as 16 time-slots (refer Equation 6.10) for the first frame. Hence, T_2 is assigned to V_0 and T_3 to V_1. In the second frame, we may observe that T_0 is able to finish it's entire execution share of 10 time-slots on V_1 without breaching the temperature threshold, while T_1 can execute for 12 time-slots (refer Equation 6.9) on V_0. Hence, g is chosen as 10 time-slots (refer Equation 6.10) and T_0 and T_1 are allotted to core V_1 and V_0, respectively. In the third frame, T_3 and T_2 are the two highest priority tasks and shares of T_2 gets over after executing for 3 time-slots. Hence g is chosen as 3 and T_2 and T_3 are assigned to cores V_0 and V_1, respectively. In this manner, the tasks are extracted

TH-2256_136101018

from heap based on their priority and then assigned to cores based on *Hottest Task on Coolest Core* heuristic. The task scheduling for this time-slice gets over after 7 frames. The entire task allocation schedule along with the core temperatures for different frames have been shown in Fig.6.1.

6.5 Analysis of the Algorithm

Dynamic Task Arrivals/Departures:
A periodic task can arrive or depart at any time in a dynamic system. When a new task (lets say T_i) arrives in a system, our algorithm first checks the period of the new task. If there is sufficient time to complete execution of the new task starting from the end of current time-slice, the scheduling of the new task is then procrastinated till the start of next time-slice. Otherwise, the system suspends it's execution from the current time instant, recomputes new time-slices by including consideration of the period of T_i, and then prepares a new schedule for the tasks. When a task departs from the system, it's entry is deleted from the list of ready tasks TL and it is not considered for scheduling from the next time-slices.

Analysis of Time Complexity:
As stated earlier, Algorithm TARTS is composed of the following three functions: TARTS() (Algorithm 20), Task_Schedule() (Algorithm 21) and Find_Mapping() (Algorithm 22). Let us analyze the complexity of each step of the TARTS scheduling strategy.

- Function *Find_Mapping()* validates task-to-core mappings for all cores. Initially, it calculates $\triangle_j$ (using Equation (6.10) and locks all successful mappings. Both of these operations take constant amount of time. If the selected task $T_{M[j]}$ cannot be allocated to the mapped core, it is added to the list UL. Next, allocation of the coolest task in the ready queue (which can be determined in O($lg\ n$) time) is attempted on core V_j. Such an allocation attempt is made on all m cores in the worst case. If the average context-switch free duration is assumed to be O($lg\ n$),

TH-2256_136101018

then the complexity of *Find_Mapping()* per time-slot becomes O(m).

- Task_Schedule() generates a frame by frame schedule for the current time-slice. For each frame, while extracting the m most urgent tasks from the heap takes O($mlg\ n$) time, preparing a tentative mapping M, using *hottest task on coolest core* heuristic, takes O($mlg\ m$) time. As before considering the average frame size of O($lg\ n$), the overall complexity due to these operations become O($mlg\ n$) (considering $m << n$). Then, the function *Find_Mapping()* is called for each frame, which has a time complexity of O(m) per time-slot. At the end of every frame, unfinished tasks which were mapped to cores, are inserted back into the list TL, which consumes O($mlg\ n$) time. Therefore, like the *Find_Mapping()* function, the complexity of *Task_Schedule()* is also O(m) per time-slot.

- In function TARTS, the initial calculation for the time-slice size $|G|$ (using Equation 6.6) and finding of shares for all tasks (using Equation 6.7) for the time-slice takes O(n) time. Then,the sorted list TL of ready tasks is formed and this takes O($nlg\ n$) time. However, this is done only once during entire schedule length as part of the initialization process. Thereafter, TL is maintained in a sorted manner as discussed in the previous step. TARTS calls Task_Schedule() at each time-slice boundary and this takes O(m) time as seen in the last step. Hence, the time complexity of the function TARTS() can be considered as O(m) per time-slice.

6.6 Experimental Set Up and Results

We have evaluated the performance of *TARTS* against a baseline greedy approach called *Temperature-Aware Multi-core Task Scheduler (TA-MTS)*, developed by us. We now present a brief overview of the TA-MTS algorithm before discussing our experimental framework and results.

Temperature-Aware Multi-core Task Scheduler (TA-MTS): A common temperature-aware scheduling strategy as employed in [132,135,138], etc., involves bi-partitioning

TH-2256_136101018

of the taskset into classes referred to as *hot* and *cool*, based on steady state temperature characteristics of the tasks. The *hot* and *cool* tasks are then carefully interleaved in a schedule to meet temperature specifications related to a given system.

Based on this heuristic, we have developed a temperature-aware semi-partitioned strategy for multi-core real-time systems called *Temperature-Aware Multi-core Task Scheduler (TA-MTS)*. Like TARTS, TA-MTS divides time into slices demarcated by the arrivals/departures of all tasks in the system (also known as Deadline Partitioning (DP)). At the beginning of each time-slice, TA-MTS determines proportional execution shares for all tasks within the time-slice. Then, it classifies each task either as hot or cool based on it's steady state temperature. A task T_i is classified as a hot (cool) task if it's steady state temperature ($\Gamma_{ss}(i)$) is greater than the average steady state temperature of the taskset. After that, TA-MTS adds all hot tasks into a queue Q_{hot}, which is kept sorted in non-increasing order of steady state temperatures of the tasks. Correspondingly, all cool tasks are added to a sorted queue Q_{cool} in non-decreasing order of their steady state temperatures. In the next step, TA-MTS dequeues one task each from Q_{hot} and Q_{cool}, forms a pair between these two tasks and adds them into a target queue (Q_{tgt}). If any of the two queues become empty, all tasks from the other queue are added into Q_{tgt}. In this way, a temperature-aware execution sequence of tasks is prepared by TA-MTS. At last, it uses McNaughtons's wrap around rule [90], to assign the prepared task sequence across available cores in the system.

McNaughton's wrap around rule: First sequence the tasks in any arbitrary order, obtaining a sequence of length $(e_1 + e_2 + ... + e_n)$ time units. Next, break the time sequence at the points, $j = 1, 2, ..., m - 1$. In a time-slice (say TS_k), each of the core has a capacity equal to size of the time-slice $|TS_k|$. Schedule all tasks in the interval $[(j-1) \times |TS_k| : j \times |TS_k|]$ on core V_j, $j = 1, 2, ..., m$. Finally, any task that is preempted and processed on two different cores V_j and V_{j+1} will have to be processed first on V_{j+1}, then preempted, and finished on V_j.

The system re-synchronises globally at the end of every time-slice. This strategy enables TA-MTS to offer a bounded number of migrations and deliver efficient resource utilizations. We now demonstrate the working of TA-MTS through an illustrative ex-

TH-2256_136101018

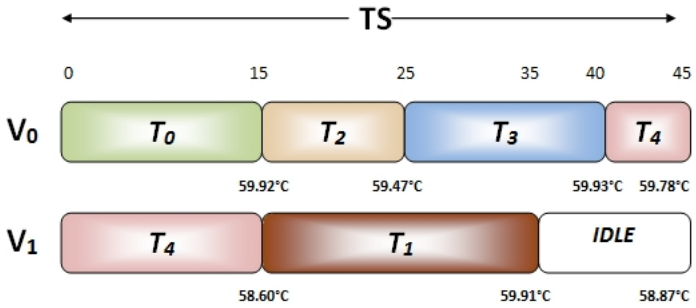


Figure 6.2: *Task Schedule prepared by TA-MTS for Example 2*

ample.

Table 6.3: *Task Specifications for Example 2*

T_i	e_i	p_i	Γ_i^{ss}
0	15	45	80
1	20	45	65
2	10	45	45
3	15	45	70
4	20	45	50

Example 2: Consider a list of 5 tasks having specifications as shown in Table 6.3, to be scheduled in a system of $m = 2$ cores, both of which have an initial temperature of $59°C$ with ambient temperature being $25°C$. Let the system level temperature threshold (Γ_l) be $60°C$. For the simplicity of explanation, we have chosen the relative deadlines to be equal to 45 time-slots for all tasks.

As all deadlines are same (=45), TA-MTS determines length of the first time-slice to be $|TS_1| = 45$ time-slots. Next, it determines the shares of all tasks within the time-slice (this is same as their execution requirements for the example). Then, all the tasks are classified as either hot or cool based on average steady state temperature of the taskset which is found to be $62°C$. So, T_0, T_1, T_3 and T_2, T_4 gets classified as hot and cool tasks, respectively. Next, TA-MTS constructs sorted queues Q_{hot} ($= T_0 -> T_3 -> T_1$; hotter to cooler) and Q_{cool} ($= T_2 -> T_4$; cooler to hotter) of the hot and cool tasks. Correspondingly, Q_{tgt} is constructed as $T_0 -> T_2 -> T_3 -> T_4 -> T_1$. The final

TH-2256_136101018

schedule of tasks along with core temperatures has been shown in Figure 6.2.

6.6.1 Experimental Set Up

All our simulations have been run for a total execution time of 100000 time slots with tasksets having pre-defined *utilization factors*, where *utilization factor (U)* is defined as the ratio of the summation of task weights (wt_i) and the number of available cores (m). For creating tasksets with a specific U, the randomly generated utilization values have been scaled appropriately. Results are generated for various distinct utilization factors. For each set of input parameters, we ran the simulation on 50 different test cases. The average of these 50 test cases has been considered as the final result. Two sets of experiments have been carried out to validate and compare our proposed algorithm with TA-MTS. In the first set, extensive simulation based experiments have been carried out using synthetic tasksets to validate the efficacy of the algorithms over varied scenarios that may be encountered, whereas in the second set, we have used benchmark programs to test the algorithms in real-life situations.

Framework for Synthetic Tasks: The experimentation framework used in this work considers randomly generated tasksets whose sizes vary from $n = 20$ to $n = 100$. For a given utilization factor *(U)*, task weights wt_i have been generated from a normal distribution with a distinct mean $\mu_{wt} = 0.3$ and standard deviation $\sigma_{wt} = 0.2$. The summation of the randomly generated task weights may not exactly be equal to U. Therefore, the individual weights are scaled such that their summation becomes equal to U. The task execution times have been generated from another normal distribution, having $\mu_e = 20$ and $\sigma_e = 100$. Each task T_i is also characterised by a *steady state temperature* (Γ_i^{ss}). Steady state temperatures for the tasks are generated randomly from a uniform distribution within the range $40°C$ and $120°C$. We have used a parameter called *Average Steady State Temperature (ATemp)* in order to obtain a measure of hotness corresponding to a given taskset. *Atemp* is defined as $\Sigma_{i=1}^{n}\Gamma_i^{ss}/n$. For creating tasksets with a specific ATemp, the randomly generated steady state temperature values have been scaled appropriately. Results are generated for various distinct Atemp values. We

TH-2256_136101018

ran our simulation on systems having 2 to 8 cores. In our scheduling framework, the maximum value of the allowed temperature (Γ_l) in any core at any time instant has been varied from $60°C$ to $100°C$.

Table 6.4: *Task Specifications for Benchmark Programs*

Program	Execution Requirement (in ms)	Steady State Temperature (in $°C$)
Body	3824	85
Canneal	1007	80
Dedup	6455	91
Fluid	4090	81
Freq	11082	84
Stream	6156	68
Swap	4535	76
x264	1203	85

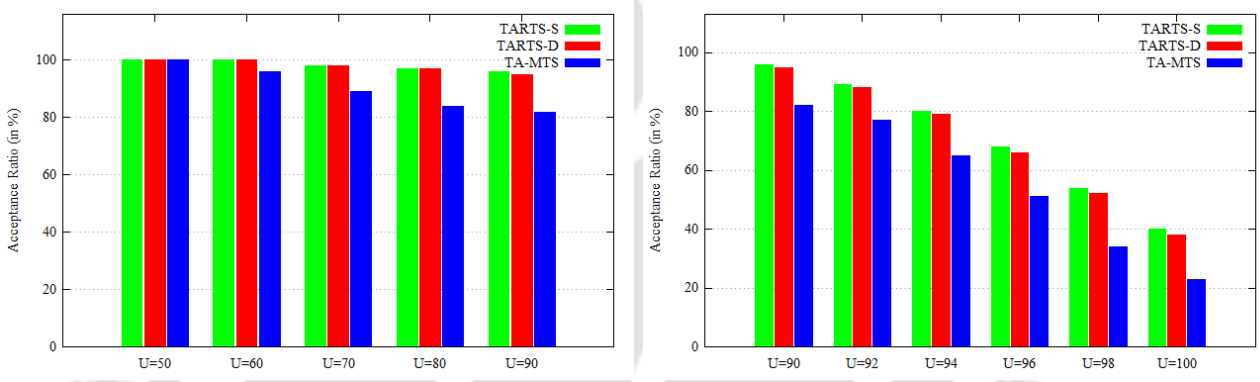
Framework for Benchmarks: We selected a set of 8 tasks from the Parsec [129] benchmark to evaluate the performance of our algorithm. A detailed discussion on the procedure for measuring execution requirements of each program along with their peak temperatures is presented in [36] and the results have been listed in Table 6.4. These values have been obtained with the help of Gem5 [33] for an Intel Xeon processor, 65 nm CMOS technology, operating at 3.1 GHz. For creating tasksets with a specific U, periods of these tasks have been generated appropriately. We have measured and compared performance of the algorithms by varying following input parameters: i. Utilization Factor (U), ii. Number of tasks (n), iii. Number of cores (m), iv. Average Steady State Temperature ($ATemp$), and v. Temperature threshold (Γ_l). The detailed analysis of our experiments has been presented below.

6.6.2 Experimental Results

We have measured the performance of *TARTS-D* (with dynamic frame size adaptation), *TARTS-S* (with static frame size $g = 1$) and *TA-MTS* against different values of input parameters. It may be noted that TARTS-S having $g = 1$ provides the possibility of

TH-2256_136101018

context-switches at every time-slot and potentially allows finer control over core temperatures. However, as the experimental results (presented below) reveal, with it's core temperature-aware dynamic frame size adaptation, TARTS-D is able to deliver similar performance as TARTS-S (in terms of achieved resource utilization under temperature constraints), while incurring significantly lower context-switch related overheads. Following two metrics have been used to compare the performance of the proposed algorithm TARTS-D, against TARTS-S and TA-MTS: i. *Acceptance Ratio* ($ARat$) and ii. Average Context Switch Cost (in μs) per Core per Time-Slot CSC. These metrics have been defined in the previous chapter.



(a) *U = 50% to 90%* **(b)** *U = 90% to 100%*

Figure 6.3: *Effect of Utilization Factor on Synthetic Tasksets (*n *= 80,* m *= 4,* ATemp *=* $80°C$ *and* $\Gamma_l = 80°C$*)*

Experiment 1- ***Effect of utilization factor***: In this experiment, we have varied the utilization factor (U) between 0.5 and 1.0. The system under consideration has $\Gamma_l = 80°C$, average steady state temperature of the tasksets $ATemp = 80°C$, $n = 80$ tasks with 4 cores. We may observe from Figure 6.3 that ARat values decrease for both the algorithms with increase in utilization factor of the system. As the utilization factor rises in a system, the cores have to handle higher workloads and hence, the scope of keeping cores idle, reduces. With static frame size of $g = 1$, TARTS-S is able to maintain a finer control over temperatures of the cores, and hence, its's acceptance ratios reduce from 100% to 96% (Figure 6.3a) with the variation of utilization factors from 50% to 90%.

TH-2256_136101018

We may also observe from Figure 6.3a that the ARat values reduce from 100% to 95% for TARTS-D. At lower utilization values, the number of sparable slots in a time-slice are higher. Hence, *TARTS-D* is able to use it's procrastination strategy of *Spare-Slot-Allocation* efficiently and subsequently balance temperatures of the cores. For *TA-MTS*, acceptance ratios reduces at a much higher rate (100% to 82%) than TARTS. This is due to the fact that in *TA-MTS*, the tasks are allotted to cores using McNaughtons's wrap around rule [90] and intra-slice temperature of the cores are not checked minutely, as in *TARTS*. From Figure 6.3b, we may observe that acceptance ratios reduces from 96% to 40%, 95% to 39% and 82% to 22% with utilization variation from 90% to 100%. It may be noted, even though TARTS-D selects the frame size dynamically but it is able to perform almost at par with TARTS-S having a static frame size of $g = 1$.

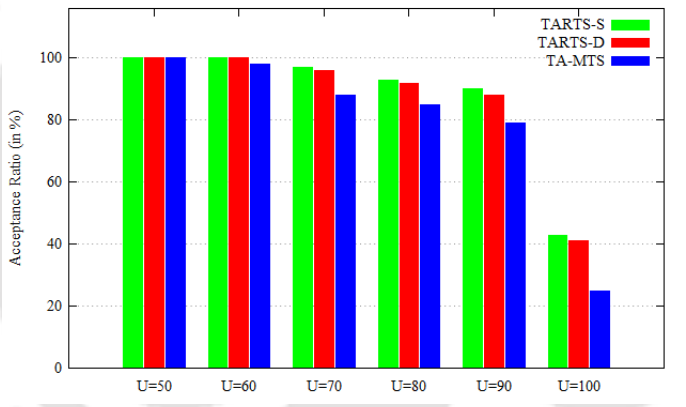


Figure 6.4: *Effect of Utilization Factor on Benchmark Programs(*m $= 4$, ATemp $= 81.25°C$ *and* $\Gamma_l = 80°C$*)*

We also performed this experiment with benchmark programs presented in Table IV. The system under consideration has $\Gamma_l = 80°C$ with 4 cores. We may observe from Figure 6.4 that results are similar to that of the synthetic tasksets. It may be noted that the *ATemp* value for the taskset is $81.25°C$, which is more than the stipulated temperature threshold $\Gamma_l = 80°C$. Therefore, rate of decrease in ARat values increases for all algorithms with increase in utilization factor and it is slightly higher than the synthetic tasksets. This is due to the fact that the benchmark programs represent only realistic

TH-2256_136101018

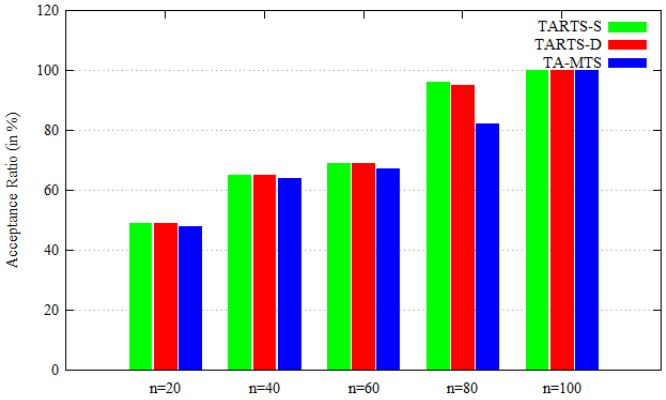


Figure 6.5: *Effect of the number of tasks (*U $= 0.9$, m $= 4$, ATemp $= 80°C$ *and* $\Gamma_l = 80°C$*)*

scenarios, whereas we are able to perform more rigorous experiments using skewed input parameters from synthetic tasksets. Further, we may observe from Figure 6.4, that with variation in utilization factor from 50% to 100%, ARat values reduces from 100% to 43%, 100% to 41% and 100% to 25% for *TARTS-S*, *TARTS-D* and *TA-MTS* respectively.

Experiment 2- *Effect of the number of tasks*: In our experiments, we have varied the number of tasks (n) between 20 and 100. The system under consideration has an utilization factor of 0.9, $\Gamma_l = 80°C$, $ATemp = 80°C$ with 4 cores. We may observe from Figure 6.5 that increase in the number of tasks also results in increased ARat values for both the algorithms. As utilization factor and number of cores remain constant, an increase in the number of tasks leads to reduced individual task weights. In *TARTS*, this may result in shorter frame sizes, which indirectly leads to better temperature control. For *TA-MTS*, this results in the execution of hot and cold tasks for shorter time duration, leading to more stable core temperatures. Improved temperature control in turn results in increased ARat value. In particular, ARat values were found to be 65%, 69%, 96% and 100% for the *TARTS* algorithm with $g = 1$, 65%, 69%, 95% and 100% for the *TARTS* algorithm and 65%, 67%, 82% and 100% for the *TA-MTS* algorithm. Also, we may observe from Figure 6.5 that frame size does not have any significant impact on the ARat value of the TARTS algorithm.

TH-2256_136101018

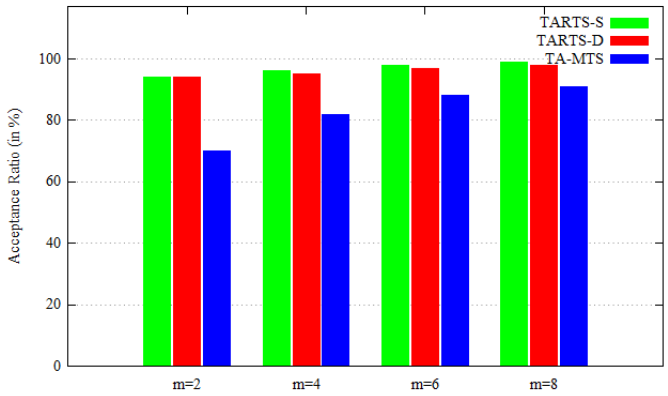


Figure 6.6: *Effect of the number of cores (*U = *0.9,* n = *80),* ATemp = $80°C$ *and* Γ_l = $80°C$*)*

Experiment 3- ***Effect of the number of Cores***: We evaluated both the algorithms on systems having 2 to 8 cores with utilization factor of 0.9, $\Gamma_l = 80°C$, $n = 80$ tasks and $ATemp = 80°C$. As we may observe from Figure 6.6, the ARat values of both the algorithms increase with increase in the number of cores, when the number of tasks remains fixed. This is due to the fact that at lower number of cores, there are less options to schedule a task in the system. We may also observe that the rates of increase in ARat values for *TA-MTS* is less than in *TARTS*. This is due to the fact that *TA-MTS* can only take a context-switch (and hence execute task with a different Γ^{ss}) after finishing execution share of the current task, while *TARTS* is able to maintain better control over core temperatures due to the use of frames. In particular, ARat values increase from 94% to 100%, 94% to 98% and 70% to 91% for the algorithms - *TARTS-S*, *TARTS-D* and *TA-MTS* respectively.

Experiment 4- ***Effect of Average Steady State Temperature (ATemp)***: In this experiment, we have varied the average steady state temperature (ATemp) of the tasksets between $60°C$ and $100°C$. The systems under consideration have fixed utilization factor of 0.9, number of tasks $(n) = 80$, $\Gamma_l = 80°C$ with 4 cores. We may observe from Figure 6.7 that increase in ATemp values of the tasksets, results in decreased ARat values for both the algorithms. Further, it may be seen that TARTS is able to sched-

TH-2256_136101018

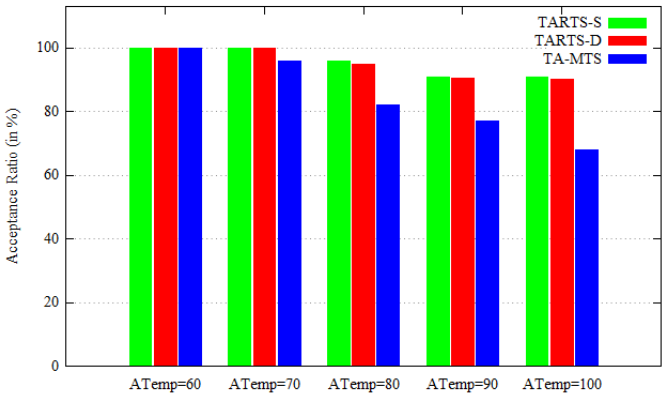


Figure 6.7: *Effect of ATemp (*U *= 0.9,* n *= 80,* m *= 4) and* $\Gamma_l = 80°C$*)*

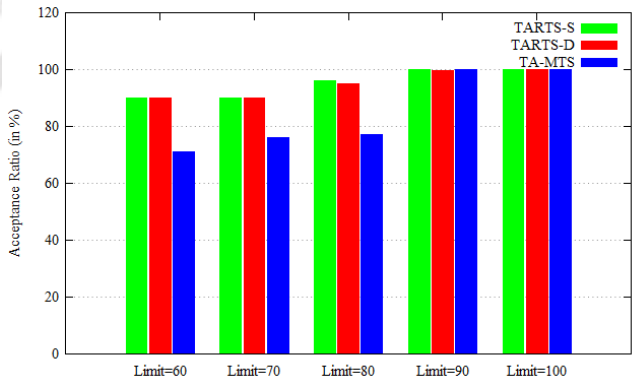


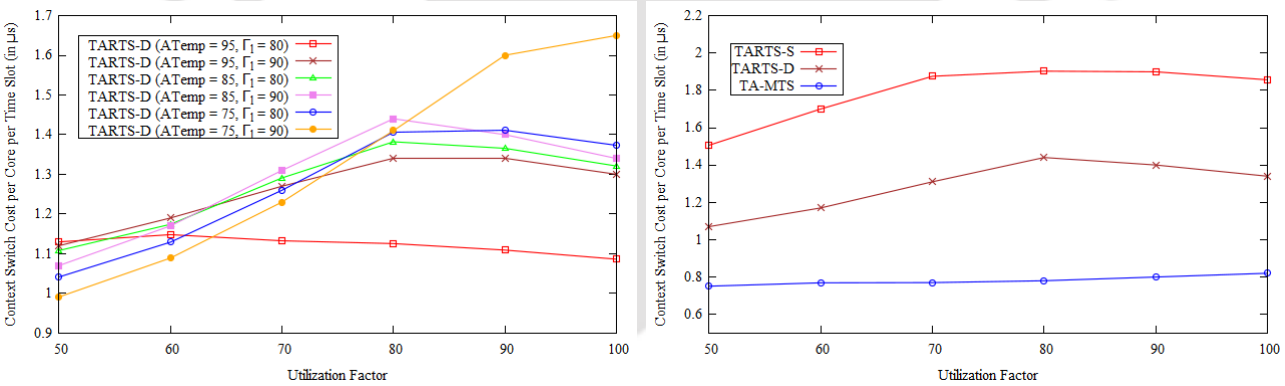
Figure 6.8: *Effect of Temperature Threshold (*U *= 0.9,* n *= 80,* m *= 4) and* ATemp $= 80°C$*)*

ule more than 90% of the total task instances when the ATemp value is less than or equal to the system-wide temperature threshold ($\Gamma_l = 80°C$). As the values of ATemp increase, temperature of the cores rise more rapidly and hence, the number of rejected task instances becomes higher, resulting in decreased ARat values. However, TARTS is able to deal with this scenario in a better way due to it's procrastination strategies of *Substitute-Allocation* and *Spare-Slot-Allocation*. In particular, ARat values were found to be 100%, 100%, 96%, 91%, 91% for TARTS-S, 100%, 100%, 95%, 90.5%, 90% for TARTS-D and 100%, 96%, 82%, 77% and 68% for TA-MTS.

Experiment 5- *Effect of Temperature threshold*: We have varied the temperature threshold (Γ_l) of the system between $60°C$ and $100°C$. The system under consideration

TH-2256_136101018

has utilization factor of 0.9, number of tasks $(n) = 80$, average steady state temperature (ATemp) for the tasksets is $80°C$ with 4 cores. We may observe from Figure 6.8 that increase in the temperature threshold results in increased ARat values for both the algorithms. As the system-wide temperature threshold (Γ_l) is enhanced, the algorithms get more liberty towards task selection, as the number of eligible tasks increases. So, ARat values increase for both the algorithms. It may be noted that both the algorithms are able to achieve 100% ARat value, when Γ_l is greater than the value of ATemp ($= 80°C$). This is due to the fact that the cores never breach the temperature threshold of Γ_l with their steady-state hotness ($ATemp$) being lower than Γ_l. However, with more finely tuned frame based temperature-aware scheduling, TARTS is able to achieve better results even at lower temperature thresholds. In particular, acceptance ratios were found to be 90%, 90%, 96%, 100%, 100% for TARTS-S, 90%, 90%, 95%, 99.5% and 100% for TARTS-D and 71%, 76%, 79%, 100%, 100% for TA-MTS.



(a) *Context-Switch/Migration overheads for TARTS-D* **(b)** *Comparison of Context-Switch/Migration for all algorithms (ATemp* $= 85°C$ *and* $\Gamma_l = 90°C$*)*

Figure 6.9: *Context-Switch/Migration overheads*

Experiment 6- ***Effect of Context-Switch/Migration overheads***: We may observe from Figure 6.9a that for TARTS-D, context-switch/migration overheads become high with the increase in utilization factor, under most of the scenarios. This is due to the fact that higher workloads lead to a lower number of spare slots in a time-slice and

TH-2256_136101018

longer execution times for tasks. Hence, TARTS-D has to frequently perform context-switches in order to maintain temperature stability of the cores. Further, we may see that when *Atemp* is greater than Γ_l, there is a slight drop in context-switch/migration overheads. This phenomenon may be attributed to the fact that at higher workloads combined with high ATemp values, TARTS-D starts rejecting the hottest task instances from the system. Hence, the number of tasks starts decreasing and this indirectly lead to less context-switches/migrations. This phenomenon becomes more pronounced when ATemp is significantly high than Γ_l. Thus, when *ATemp* and Γ_l values are set at $95°C$ and $80°C$, TARTS-D starts rejecting tasks even at lower utilizations factors. Correspondingly, the plot shows a dip in context-switch overheads.

It may be observed from Figure 6.9b that TARTS incurs higher context-switch/migration overheads as compared to TA-MTS. This is due to the fact that TA-MTS is oblivious about instantaneous core temperatures and applies the DPFair-like McNaughtons's wrap around rule [90] to schedule the tasks within a time-slice. This leads to fewer context-switch/migration overheads as compared to TARTS, although at the expense of significantly lower achieved acceptance ratios (ARat). On the other hand, TARTS is able to maintain better temperature control with the help of frames. Further, as may be seen from Figure 6.9b, TARTS-S incurs even higher context-switches with respect to TARTS-D. This is due to the fact that TART-S applies *hottest task on coolest core* heuristic at each time-slot and therefore, context-switches are far higher than TARTS-D, even though control over core temperatures is slightly better. As the experimental results show, TARTS-D is able to achieve almost same performance as TARTS-S in terms of ARat values.

6.7 Summary

In this chapter, we presented TARTS, a low over-head temperature-aware semi-partitioned algorithm for scheduling periodic tasks in a proportional fair manner. The proposed strategy is able to deliver high resource utilizations in real-time multi-core systems. It uses a two-level hierarchical scheduling strategy where the outer level determines

TH-2256_136101018

time-slices based on the deadlines of tasks and task execution proceeds time-slice by time-slice, in a proportional fair manner. Shares of all tasks to be executed in the next time-slice are determined at boundary of every time-slice. The second level performs intra-time-slice schedule generation by further dividing each slice into a sequence of possibly unequal time-intervals called *frames*, and a schedule is generated for each frame in a sequential manner such that it does not lead to temperature threshold violation for any core at any instant of time. The experimental results show that TARTS is able to perform appreciably under various realistic scenarios while incurring slightly higher context-switch/migration overheads compared to the baseline greedy approach called, TA-MTS. In the next chapter, we conclude our thesis work. We discuss the work in progress, possible extensions and future work that can be done in this area.

TH-2256_136101018

Chapter 7

Conclusions and Future Perspectives

In this chapter, we summarize the contributions of this dissertation and outline some possible directions for future work.

7.1 Summarization

In spite of progressively increasing computing capabilities of hardware platforms, effective allocation of computing resources to diverse competing applications is set to remain a daunting problem in real-time embedded systems. This is because, these systems impose stringent timing, resource, performance and safety related constraints which must be accurately captured in the design process in order to ensure proper functioning of the system. In this context, *scheduling* acts as a vital design component which determines an appropriate co-execution order for the application tasks such that the desired performance objectives may be achieved while satisfying all constraints. Efficient usage and management of energy and temperature in resource-constrained devices like mobiles, laptops, PDAs, etc., while satisfying all temporal and resource constraints, has become a design parameter of paramount importance. Therefore, this thesis has delved towards the design of various energy and temperature aware scheduling strategies for such real-time multi-core platforms. Here, we enumerate a few challenging constraints which must be considered while designing schedulers for real-time systems:

- The first challenge is related to the handling of *timing constraints* associated with the various types of computation activities (i.e., tasks) that occur in such systems.

TH-2256_136101018

The timing constraints of tasks are captured by their execution requirements, deadlines and arrival patterns.

- The second challenge is consideration of *resource constraints* imposed by the underlying hardware platform on which the system is implemented. Over the years, the nature of execution platforms is witnessing a shift from single core to homogeneous and heterogeneous multi-cores, where several applications share the same platform [7].

- Handling of *safety and performance related constraints* such as peak power minimization, makespan minimization, temperature control, etc. is the third important challenge real-time system designers are confronted with.

A lot of research has been conducted towards their power management at various levels of abstraction, starting from hardware and firmware to architectural, system and even application levels. Even as the hardware becomes more and more energy-efficient, the onus of extracting maximum efficiency out of the hardware very often falls on the software. At the operating system level, two primary mechanisms are generally used to reduce energy consumption: (1) Dynamic Voltage and Frequency Scaling (DVFS) [18, 69, 115] and, (2) Dynamic Power Management (DPM) [13, 21, 69]. DVFS involves dynamic adaptation of a processor's operating voltage/frequency according to instantaneous requirements of the workload being handled at a given time. As energy dissipated per cycle in CMOS circuits scale quadratically to the supply voltage, this strategy is able to provide large energy savings in DVFS enabled processors. DPM, on the other hand, involves suspending parts of a system when processors are idling (due to low workloads), because energy consumed during the suspension period is negligible. An interplay between these two mechanisms can be utilized to minimize the overall energy consumption of multi-core platforms.

In Chapter 3, our first contributory chapter, we have presented both DVFS as well as DVFS-cum-DPM based scheduling strategies of real-time periodic tasks on homogeneous multi-core platforms. Using DPFair [58], as the underlying scheduling strategy,

TH-2256_136101018

we have proposed *EAFBFS* and *DPFair-SS*, two low-overhead energy-aware strategies for scheduling periodic real-time tasks on multi-cores in a proportional fair manner. While EAFBFS uses DVFS mechanism to reduce dynamic power consumption in the system, DPFair-SS combines the benefits of both DVFS and DPM to minimize system-wide power consumption. Experimental results show that both the algorithms are not only able to provide efficient resource utilization but reduce energy consumptions in the system as well.

Over the years, the industry is witnessing a significant shift in the nature of processing platforms in real-time embedded systems. Homogeneous single cores have slowly given way to multi-core platforms in order to cater to higher computation demands while adhering to restrictions on temperature/energy dissipation. Hence, we present two efficient and low-overhead semi-partitioned heuristic scheduling strategies for such platforms namely *COST* and *HETERO-SCHED* in Chapter 4. While *COST* has a restriction on number of migrations for a task within a time-slice, the proposed strategy in *HETERO-SCHED* is fully-migrative in nature. The experimental result shows the efficacy of the proposed strategies.

In recent years, researchers have explored energy-efficient scheduling techniques for multi-cores [24, 97, 98]. However, only a few works have been targeted towards heterogeneous platforms. Hence, we present a DVFS-cum-DPM based fully-migrative energy efficient scheduler called *HEALERS*, for periodic real-time tasks to be executed on heterogeneous platforms in Chapter 5. Experimental results show that our scheme is not only able to achieve appreciable energy savings with respect to state-of-the-art [16] but also enables significant improvement in resource utilization.

Apart from minimizing energy dissipation, modern real-time systems implemented on modern multi-core chips with high gate densities, must adhere to strict thermal constraints, in order to control functional unreliability due to temperature hot-spots [99]. Hence, in Chapter 6, we propose a two-level temperature-aware semi-partitioned scheduling strategy for multi-core systems called *TARTS*. At the first level, time-slices are determined based on the deadlines of tasks and task execution proceeds time-slice by

TH-2256_136101018

time-slice, in a proportional fair manner. At any time-slice boundary, shares of all tasks to be executed in the next time-slice, are determined. The second level performs intra-time-slice schedule generation. Each slice is further divided into a sequence of possibly unequal time-intervals called *frames*, and a schedule is generated for each frame in a sequential manner. Given the temperatures of the cores at a frame boundary (say at time t_1) and the set of m task-to-core assignments (with known temperature characteristics), size of the next frame is given by the maximum context-switch free time-interval which do not lead to the possibility of temperature threshold violation for any core. If the frame size is positive, the selected task-to-core assignments are used to construct the schedule for the frame. If the obtained frame size is zero, a different task-to-core assignment must be used for the frame. Experimental results show that TARTS is able to perform appreciably under various realistic scenarios while incurring slightly higher context-switch/migration overheads compared to the existing temperature oblivious version of DPFair [58].

In summary, the work conducted as part of this thesis deals with the development of various scheduling methodologies for different types of real-time applications on homogeneous as well as heterogeneous multi-cores. Techniques necessary to imbibe power awareness, temperature minimization, etc. within the designed scheduling strategies, is a major focus of the proposed works.

7.2 Future Works

The work presented in this thesis leaves several open directions and there is ample scope for future research in this area. In this section, we present three such future perspectives.

- **Temperature-Aware scheduler for heterogeneous platforms:** On heterogeneous platforms, the same piece of code may have different characteristics like execution time requirements, power consumption, thermal features, etc. on different cores. Hence, scheduling on such platforms becomes more challenging. We would like to devise a heuristic temperature-aware scheduler for such platforms.

TH-2256_136101018

- **Cache-Aware Scheduling for Energy Management:** The applications in real-time systems may have different cache requirements. Hence, the full last-level cache may not be needed at all times while co-scheduling of such applications on multi-core platforms. It may be possible to prudently schedule such a choice of co-running threads which allow large parts of the cache to be deactivated to low power sleep states over large time intervals. We purview that the mechanism may have the potential to deliver high energy savings in many systems.

- **Scheduling for distributed processing platforms:** In the current research work, we have assumed a tightly-coupled interconnection of individual processing elements. Hence, inter-processor communication delays have been ignored. However, in a loosely-coupled distributed processing platform, such an assumption does not hold [126]. In addition, depending on the type and structure of the interconnection network between processing elements, the inter-processor message transmission time may significantly vary. To match the requirements of the distributed systems mentioned above, the research presented here must be suitably adapted.

TH-2256_136101018

TH-2256_136101018

Disseminations out of this Work

Journal Papers

1. Sanjay Moulik, Arnab Sarkar and Hemangee K. Kapoor. "Energy aware frame based fair scheduling. "[Sustainable Computing: Informatics and Systems (2018) 2210-5379] Volume 18, Pages 66 77, 2018.

2. Sanjay Moulik, Rajesh Devaraj and Arnab Sarkar. "HEALERS: A Heterogeneous Energy-Aware Low-overhead Real-time Scheduler. "[IET Computers & Digital Techniques (2019) 1751-8601].

3. Sanjay Moulik, Arnab Sarkar and Hemangee K. Kapoor. "TARTS: A Temperature-Aware Real-Time Semi-partitioned Scheduler."[Journal of System Architecture] (Under review).

Conference Papers

1. Sanjay Moulik, Rajesh Devaraj, Arnab Sarkar and Arijit Shaw. "A Deadline-Partition Oriented Heterogeneous Multi-Core Scheduler for Periodic Tasks." International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). Pages 204-210, 2017.

2. Sanjay Moulik, Arnab Sarkar and Hemangee K. Kapoor. "DPFair Scheduling with Slowdown and Suspension." International Conference on VLSI Design (VLSID). Pages 43-48, 2018.

TH-2256_136101018

3. Sanjay Moulik, Rajesh Devaraj and Arnab Sarkar. "HETERO-SCHED: A Low-Overhead Heterogeneous Multi-core Scheduler for Real-Time Periodic Tasks." International Conference on High Performance Computing and Communications (HPCC). Pages 659-666, 2018.

4. Sanjay Moulik, Rajesh Devaraj and Arnab Sarkar. "COST: A Cluster-Oriented Scheduling Technique for Heterogeneous Multi-cores." IEEE International Conference on Systems, Man, and Cybernetics (SMC). Pages 1951-1957, 2018.

5. Sanjay Moulik, Rajesh Devaraj and Arnab Sarkar. "HEART: A Heterogeneous Energy-Aware Real-Time scheduler." International Conference on VLSI Design (VLSID), Pages 476-481, 2018.

TH-2256_136101018

References

[1] "Intel Xeon Processor E5 v4 Product Family Datasheet, volume one: Electrical: [online] https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v4-datasheet-vol-1.pdf, June 2016." [Pg.7]

[2] R. G. A. Parekh, "A generalized processor sharing approach to flow-control in integrated services networks: The single-node case," *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pp. 344–357, 1993. [Pg.27]

[3] L. Abeni, "Server mechanisms for multimedia applications," Tech. Rep. [Pg.26]

[4] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *19th IEEE Real-Time Systems Symposium*, 1998, pp. 4–13. [Online]. Available: citeseer.ist.psu.edu/abeni98integrating.html [Pg.25], [Pg.26]

[5] A. Alsughayyir and T. Erlebach, "Energy aware scheduling of hpc tasks in decentralised cloud systems," in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Feb 2016, pp. 617–621. [Pg.36]

[6] P. Altenbernd, "Deadline-monotonic software scheduling for the co-synthesis of parallel hard real-time systems," in *EDTC '95: 1995 European conference on Design and Test*. Washington, DC, USA: IEEE Computer Society, 1995, p. 190. [Pg.22]

TH-2256_136101018

[7] A. Aminifar, P. Eles, Z. Peng, A. Cervin, and K.-E. Årzén, "Control-Quality Driven Design of Embedded Control Systems with Stability Guarantees," *IEEE Design & Test*, 2017. [Pg.164]

[8] J. Anderson and A. Srinivasan, "Mixed pfair/erfair scheduling of asynchronous periodic tasks," *Journal of Computer and System Sciences*, vol. 68, no. 1, pp. 157–204, Feb 2004. [Pg.30], [Pg.31]

[9] J. Anderson and A. Srinivasan, "Early-release fair scheduling," in *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, 2000, pp. 35–43. [Pg.30], [Pg.32], [Pg.34], [Pg.45]

[10] B. Andersson and J. Jonsson, "The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%," in *15th Euromicro Conference on Real-Time Systems*, Jul 2003, pp. 33–40. [Pg.22]

[11] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2006, pp. 322–334. [Pg.22]

[12] S. Angelopoulos, C. Dürr, S. Kamali, M. Renault, and A. Rosén, "Online bin packing with advice of small size," in *Algorithms and Data Structures*, F. Dehne, J.-R. Sack, and U. Stege, Eds. Cham: Springer International Publishing, 2015, pp. 40–53. [Pg.24]

[13] J. Augustine, S. Irani, and C. Swamy, "Optimal Power-Down Strategies," *SIAM J. Comput.*, vol. 37, no. 5, pp. 1499–1516, Jan. 2008. [Pg.3], [Pg.43], [Pg.164]

[14] M. A. Awan and S. M. Petters, "Enhanced race-to-halt: A leakage-aware energy management approach for dynamic priority systems," in *2011 23rd Euromicro Conference on Real-Time Systems*, July 2011, pp. 92–101. [Pg.10], [Pg.37]

[15] M. A. Awan and S. M. Petters, "Energy-aware partitioning of tasks onto a heterogeneous multi-core platform," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013, pp. 205–214. [Pg.38]

TH-2256_136101018

[16] M. A. Awan, P. M. Yomsi, G. Nelissen, and S. M. Petters, "Energy-aware task mapping onto heterogeneous platforms using DVFS and sleep states," *Real-Time Systems*, vol. 52, no. 4, pp. 450–485, Jul 2016. [Pg.12], [Pg.39], [Pg.128], [Pg.165]

[17] M. A. Awan, P. M. Yomsi, G. Nelissen, and S. M. Petters, "Energy-aware task mapping onto heterogeneous platforms using dvfs and sleep states," *Real-Time Systems*, vol. 52, no. 4, pp. 450–485, Jul 2016. [Online]. Available: https://doi.org/10.1007/s11241-015-9236-x [Pg.39]

[18] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez, "Determining optimal processor speeds for periodic real-time tasks with different power characteristics," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, ser. ECRTS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 225–232. [Pg.3], [Pg.43], [Pg.45], [Pg.164]

[19] P. Azad and N. J. Navimipour, "An energy-aware task scheduling in the cloud computing using a hybrid cultural and ant colony optimization algorithm," *International Journal of Cloud Applications and Computing*, vol. 7, no. 4, 2017. [Pg.36]

[20] E. Bampis, D. Letsios, G. Lucarelli, E. Markakis, and I. Milis, "On multiprocessor temperature-aware scheduling problems," *Journal of Scheduling*, vol. 16, no. 5, pp. 529–538, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10951-013-0319-z [Pg.40]

[21] P. Baptiste, M. Chrobak, and C. Dürr, "Polynomial-time algorithms for minimum energy scheduling," *ACM Trans. Algorithms*, vol. 8, no. 3, pp. 26:1–26:29, July 2012. [Pg.3], [Pg.43], [Pg.164]

[22] S. Baruah and J. Carpenter, "Multiprocessor fixed-priority scheduling with restricted interprocessor migrations," in *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, July 2003, pp. 195–202. [Pg.129]

[23] S. Baruah, J. Gehrke, and C. Plaxton, "Fast scheduling of periodic tasks on multiple resources," in *9th International Parallel Processing Symposium*, Apr 1995, pp. 280–288. [Pg.30], [Pg.31], [Pg.32]

TH-2256_136101018

[24] S. Baruah, M. Bertogna, and G. Buttazzo, "Multiprocessor scheduling for real-time systems," 2015. [Pg.4], [Pg.113], [Pg.165]

[25] S. Baruah, V. Bonifaci, R. Bruni, and A. Marchetti-Spaccamela, "Ilp models for the allocation of recurrent workloads upon heterogeneous multiprocessors," *Journal of Scheduling*, p. 195209, 2019. [Pg.84]

[26] S. K. Baruah *et al.*, "Ilp-based approaches to partitioning recurrent workloads upon heterogeneous multiprocessors," in *28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2016, pp. 215–225. [Pg.84]

[27] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996. [Online]. Available: http://dx.doi.org/10.1007/BF01940883 [Pg.30], [Pg.31]

[28] S. Baruah, J. Gehrke, C. Plaxton, I. Stoica, H. Abdel-Wahab, and K. Jeffay, "Fair on-line scheduling of a dynamic set of tasks on a single resource," *Information Processing Letters*, vol. 64, no. 1, pp. 43–51, 1997. [Online]. Available: citeseer.ist.psu.edu/baruah96fair.html [Pg.25], [Pg.30]

[29] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," in *Operating Systems Platforms for Embedded Real-Time Applications (OSPERT). 6th International Workshop on*, 2010. [Pg.69], [Pg.135]

[30] E. Berg and E. Hagersten, "Statcache: a probabilistic approach to efficient and accurate data locality analysis," in *ISPASS '04: 2004 IEEE International Symposium on Performance Analysis of Systems and Software*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 20–27. [Pg.33]

[31] M. Bhatti, M. Farooq, C. Belleudy, and M. Auguin, "Controlling energy profile of rt multiprocessor systems by anticipating workload at runtime," in *SYMPosium en Architectures nouvelles de machines*, 2009. [Pg.37]

[32] M. Bhatti, C. Belleudy, and M. Auguin, "An inter-task real time DVFS scheme for multiprocessor embedded systems," in *Design and Architectures for Signal and*

TH-2256_136101018

Image Processing (DASIP), 2010 Conference on, Oct 2010, pp. 136–143. [Pg.35], [Pg.38]

[33] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011. [Pg.131], [Pg.154]

[34] M. Birks and S. P. Fung, "Temperature aware online scheduling for throughput maximisation: The effect of the cooling factor," *Sustainable Computing: Informatics and Systems*, vol. 4, no. 3, pp. 151 – 159, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2210537914000419 [Pg.40]

[35] G. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24. [Pg.xxvii], [Pg.1], [Pg.5], [Pg.17]

[36] S. Bygde, A. Ermedahl, and B. Lisper, "An efficient algorithm for parametric wcet calculation," in *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2009, pp. 13–21. [Pg.131], [Pg.154]

[37] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms," 2004. [Online]. Available: citeseer.ist.psu.edu/601206.html [Pg.22]

[38] B. Chattopadhyay and S. Baruah, "A Lookup-Table Driven Approach to Partitioned Scheduling," in *Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 257–265. [Pg.2]

[39] G. Chen, K. Huang, and A. Knoll, "Abstract: Energy optimization for real-time multiprocessor system-on-chip with optimal dvfs and dpm combination," in *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*, Oct 2013, pp. 40–40. [Pg.37], [Pg.38]

TH-2256_136101018

[40] G. Chen, K. Huang, and A. Knoll, "Energy optimization for real-time multiprocessor system-on-chip with optimal DVFS and DPM combination," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 3s, pp. 111:1–111:21, Mar. 2014. [Pg.37]

[41] J.-J. Chen, M.-J. Kao, D. Lee, I. Rutter, and D. Wagner, "Online dynamic power management with hard real-time guarantees," *Theor. Comput. Sci.*, vol. 595, no. C, pp. 46–64, Aug. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2015.06.014 [Pg.37]

[42] J.-J. Chen and T.-W. Kuo, "Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor," in *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, ser. LCTES '06. New York, NY, USA: ACM, 2006, pp. 153–162. [Online]. Available: http://doi.acm.org/10.1145/1134650.1134673 [Pg.37]

[43] M. Chetto, "Optimal scheduling for real-time jobs in energy harvesting computing systems," *IEEE Trans. Emerging Topics Comput.*, vol. 2, no. 2, pp. 122–133, 2014. [Online]. Available: https://doi.org/10.1109/TETC.2013.2296537 [Pg.6]

[44] H. Cho, B. Ravindran, and E. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *IEEE Real-Time Systems Symposium (RTSS)*, 2006, pp. 101–110. [Pg.34]

[45] H. S. Chwa, J. Seo, J. Lee, and I. Shin, "Optimal real-time scheduling on two-type heterogeneous multicore platforms," in *Real-Time Systems Symposium*, pp. 119–129, 2015. [Pg.4], [Pg.87], [Pg.91], [Pg.92], [Pg.96]

[46] R. Conway, W. Maxwell, and L. Miller, *Theory of Scheduling*. 31 East 2nd Street, Mineola, N.Y. 11501: Dover Publications, Inc., 2003. [Pg.20]

[47] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM computing surveys*, vol. 43, no. 4, p. 35, 2011. [Pg.1]

[48] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *ACM SIGCOMM '89, Austin, TX*, Sep 1989, pp. 1–12. [Pg.26], [Pg.28]

TH-2256_136101018

[49] Z. Deng and J. Liu, "Scheduling real-time applications in open environment," in *18th IEEE Real-Time Systems Symposium*, Dec 1997, pp. 308–319. [Pg.26]

[50] Z. Deng, J. Liu, and J. Sun, "A scheme for scheduling hard real-time applications in open system environment," in *9th Euromicro Workshop on Real-Time Systems*, 1997, pp. 191–199. [Pg.26]

[51] A. Ejlali, B. M. Al-Hashimi, and P. Eles, "A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2009, pp. 193–202. [Pg.38]

[52] C. Fidge, "Real-time scheduling theory, Tech. Rep. 02-19, Apr 2002. [Pg.20]

[53] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, "Scheduling resources in multi-user, heterogeneous, computing environments with smartnet," in *Proceedings Seventh Heterogeneous Computing Workshop (HCW'98)*, March 1998, pp. 184–199. [Pg.39]

[54] K. Funaoka, S. Kato, and N. Yamasaki, "Energy-efficient optimal real-time scheduling on multiprocessors," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, May 2008, pp. 23–30. [Pg.35]

[55] K. Funaoka, S. Kato, and N. Yamasaki, "Work-conserving optimal real-time scheduling on multiprocessors," in *2008 Euromicro Conference on Real-Time Systems*, July 2008, pp. 13–22. [Pg.34]

[56] K. Funaoka, A. Takeda, S. Kato, and N. Yamasaki, "Dynamic voltage and frequency scaling for optimal real-time scheduling on multiprocessors," in *Industrial Embedded Systems, 2008. SIES 2008. International Symposium on*, June 2008, pp. 27–33. [Pg.36]

[57] S. Funk, V. Berten, C. Ho, and J. Goossens, "A global optimal scheduling algorithm for multiprocessor low-power platforms," in *Proceedings of the 20th International Conference on Real-Time and Network Systems*, ser. RTNS '12. New

TH-2256_136101018

York, NY, USA: ACM, 2012, pp. 71–80. [Pg.9], [Pg.10], [Pg.47], [Pg.61], [Pg.67], [Pg.71], [Pg.79]

[58] S. Funk *et al.*, "Dp-fair: a unifying theory for optimal hard real-time multiprocessor scheduling," *Real-Time Systems*, vol. 47, no. 5, p. 389, 2011. [Pg.117], [Pg.118], [Pg.164], [Pg.166]

[59] F. S. Ghahfarokhi and A. Ejlali, "Schedule swapping: A technique for temperature management of distributed embedded systems," in *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, Dec 2010, pp. 1–6. [Pg.40]

[60] Y. Guo, D. Zhu, and H. Aydin, "Generalized standby-sparing techniques for energy-efficient fault tolerance in multiprocessor real-time systems," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*. IEEE, 2013, pp. 62–71. [Pg.38]

[61] A. Gupta, A. Tucker, and S. Urushibara, "The impact of operating system scheduling policies and synchronization methods of the performance of parallel application," in *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1991, pp. 120–132. [Pg.33]

[62] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," B. Lisper, Ed. Brussels, Belgium: OCG, July 2010, pp. 137–147. [Pg.xxiii], [Pg.131]

[63] M. A. Haque, H. Aydin, and D. Zhu, "Energy-aware standby-sparing technique for periodic real-time applications," in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*. IEEE, 2011, pp. 190–197. [Pg.38]

[64] H. Huang, V. Chaturvedi, G. Quan, J. Fan, and M. Qiu, "Throughput maximization for periodic real-time systems under the maximal temperature constraint," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2s, pp. 70:1–70:22, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2544375.2544390 [Pg.40], [Pg.139]

[65] K. Jeffay and D. Bennett, "A rate-based execution abstraction for multimedia computing," in *Network and Operating System Support for Digital Audio and Video*,

TH-2256_136101018

1995, pp. 64–75. [Online]. Available: citeseer.ist.psu.edu/jeffay95ratebased.html [Pg.25]

[66] K. Jeffay and S. Goddard, "A theory of rate-based execution," in *20th IEEE Real-Time Systems Symposium*, 1999, pp. 304–314. [Pg.25], [Pg.26], [Pg.27]

[67] K. Jeffay and S. Goddard, "Rate-based resource allocation models for embedded systems," *Lecture Notes in Computer Science*, vol. 2211, pp. 204–, 2001. [Pg.25], [Pg.27]

[68] K. Jeffay and G. Lamastra, "A comparative study of the realization of rate-based computing services in general purpose operating systems," in *7th IEEE International Conference on Real-Time Computing Systems and Applications*, Dec 2000, pp. 81–90. [Online]. Available: citeseer.ist.psu.edu/jeffay00comparative.html [Pg.25]

[69] R. Jejurikar, C. Pereira, and R. Gupta, "Leakage aware dynamic voltage scaling for real-time embedded systems," in *Design Automation Conference, 2004. Proceedings. 41st*, July 2004, pp. 275–280. [Pg.3], [Pg.10], [Pg.37], [Pg.43], [Pg.71], [Pg.73], [Pg.116], [Pg.164]

[70] R. Jejurikar and R. Gupta, "Dynamic slack reclamation with procrastination scheduling in real-time embedded systems," in *Proceedings of the 42Nd Annual Design Automation Conference*, ser. DAC '05. New York, NY, USA: ACM, 2005, pp. 111–116. [Online]. Available: http://doi.acm.org/10.1145/1065579.1065612 [Pg.37]

[71] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68–75, Dec 2003. [Pg.3], [Pg.36], [Pg.71]

[72] S. Kriaa, L. Pietre-Cambacedes, M. Bouissou, and Y. Halgand, "A survey of approaches combining safety and security for industrial control systems," *Reliability Engineering & System Safety*, vol. 139, pp. 156–178, 2015. [Pg.6]

[73] C. Krishna, "Fault-tolerant scheduling in homogeneous real-time systems," *ACM Computing Surveys*, vol. 46, no. 4, p. 48, 2014. [Pg.6]

TH-2256_136101018

[74] K. Kwon, S. Chae, and K. G. Woo, "An application-level energy-efficient scheduling for dynamic voltage and frequency scaling," in *2013 IEEE International Conference on Consumer Electronics (ICCE)*, Jan 2013, pp. 3–6. [Pg.36]

[75] J. Lee, B. Yun, and K. G. Shin, "Reducing peak power consumption in multi-core systems without violating real-time constraints," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 4, pp. 1024–1033, 2014. [Pg.7]

[76] V. Legout, M. Jan, and L. Pautet, "Scheduling algorithms to reduce the static energy consumption of real-time systems," *Real-Time Systems*, vol. 51, no. 2, pp. 153–191, Mar 2015. [Online]. Available: https://doi.org/10.1007/s11241-014-9207-7 [Pg.37]

[77] J. Leung, "A new algorithm for scheduling periodic, real-time tasks," *Algorithmica*, vol. 4, no. 1, pp. 209–219, 1989. [Pg.25]

[78] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "DP-FAIR: A simple model for understanding optimal multiprocessor scheduling," in *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, July 2010, pp. 3–13. [Pg.2], [Pg.11], [Pg.34], [Pg.47], [Pg.71], [Pg.76], [Pg.84], [Pg.138]

[79] K. Li, X. Tang, and Q. Yin, "Energy aware scheduling algorithm for task execution cycles with normal distribution on heterogeneous computing systems," in *2012 41st International Conference on Parallel Processing*, 2012, pp. 40–47. [Pg.39]

[80] R. C. C. Lian Tuu Yeh, *Thermal Management of Microelectronic Equipment: Heat Transfer Theory, Analysis Methods and Design Practices*. ACME Press, 2002. [Pg.3], [Pg.12], [Pg.137]

[81] W. Liao, L. He, and K. M. Lepak, "Temperature and supply voltage aware performance and power modeling at microarchitecture level," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 7, pp. 1042–1053, July 2005. [Pg.41]

[82] C.-C. Lin, Y.-C. Syu, C.-J. Chang, J.-J. Wu, P. Liu, P.-W. Cheng, and W.-T. Hsu, "Energy-efficient task scheduling for multi-core platforms with per-core dvfs," *J. Parallel Distrib. Comput.*, vol. 86, no. C, Aug. 2015. [Pg.35]

TH-2256_136101018

[83] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973. [Pg.21], [Pg.22], [Pg.24], [Pg.25], [Pg.26]

[84] J. W. S. Liu, *Real-Time Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000. [Pg.21], [Pg.22], [Pg.24], [Pg.25], [Pg.26]

[85] S. Liu, M. Qiu, W. Gao, X.-j. Tang, and B. Guo, "Hybrid of job sequencing and dvfs for peak temperature reduction with nondeterministic applications," in *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, ser. CIT '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1780–1787. [Online]. Available: http://dx.doi.org/10.1109/CIT.2010.309 [Pg.140]

[86] S. Liu, Q. Wu, and Q. Qiu, "An adaptive scheduling and voltage/frequency selection algorithm for real-time energy harvesting systems," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09. New York, NY, USA: ACM, 2009, pp. 782–787. [Pg.35]

[87] P. Lun, "Report on simple one-dimensional on-line bin packing algorithms," http://www.tcs.fudan.edu.cn/rudolf/Courses/Online/Online03/po_report.pdf. [Pg.23]

[88] J. Malkevitch, "Bin packing and machine scheduling," *Feature Column from the AMS: Monthly Essays on Mathematical Topics*, Jun 2004. [Online]. Available: http://www.ams.org/featurecolumn/archive/packings2.html [Pg.22], [Pg.23], [Pg.129]

[89] S. M. Martin, K. Flautner, T. Mudge, and D. Blaauw, "Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads," in *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '02. New York, NY, USA: ACM, 2002, pp. 721–725. [Online]. Available: http://doi.acm.org/10.1145/774572.774678 [Pg.72], [Pg.115]

[90] R. McNaughton, "Scheduling with deadlines and loss functions," *Manage. Sci.*, vol. 6, no. 1, pp. 1–12, Oct. 1959. [Online]. Available: http://dx.doi.org/10.1287/mnsc.6.1.1 [Pg.151], [Pg.156], [Pg.161]

TH-2256_136101018

[91] V. Moghaddas, M. Fazeli, and A. Patooghy, "Reliability-oriented scheduling for static-priority real-time tasks in standby-sparing systems," *Microprocessors and Microsystems*, vol. 45, pp. 208 – 215, 2016. [Pg.10], [Pg.38]

[92] M. Mohaqeqi, M. Kargahi, and A. Movaghar, "Analytical leakage/temperature-aware power modeling and optimization for a variable speed real-time system," in *Proceedings of the 20th International Conference on Real-Time and Network Systems*, ser. RTNS '12. New York, NY, USA: ACM, 2012, pp. 81–89. [Online]. Available: http://doi.acm.org/10.1145/2392987.2392997 [Pg.41]

[93] A. K. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," Cambridge, MA, USA, Tech. Rep., 1983. [Pg.25]

[94] G. A. Moreno and D. de Niz, "An optimal real-time voltage and frequency scaling for uniform multiprocessors," in *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2012, pp. 21–30. [Pg.36]

[95] S. Moulik, R. Devaraj, and A. Sarkar, "Heart: A heterogeneous energy-aware real-time scheduler," in *2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)*, Jan 2019, pp. 476–481. [Pg.128], [Pg.129]

[96] S. Moulik, R. Devaraj, A. Sarkar, and A. Shaw, "A deadline-partition oriented heterogeneous multi-core scheduler for periodic tasks," in *2017 18th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, Dec 2017, pp. 204–210. [Pg.11], [Pg.93]

[97] S. Moulik, A. Sarkar, and H. K. Kapoor, "Dpfair scheduling with slowdown and suspension," in *2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID)*, Jan 2018, pp. 43–48. [Pg.2], [Pg.113], [Pg.115], [Pg.165]

[98] S. Moulik, A. Sarkar, and H. K. Kapoor, "Energy aware frame based fair scheduling," *Sustainable Computing: Informatics and Systems*, vol. 18, pp. 66 – 77, 2018. [Pg.2], [Pg.113], [Pg.165]

TH-2256_136101018

[99] W. Munawar, H. Khdr, S. Pagani, M. Shafique, J. Chen, and J. Henkel, "Peak power management for scheduling real-time tasks on heterogeneous many-core systems," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Dec 2014, pp. 200–209. [Pg.4], [Pg.7], [Pg.137], [Pg.165]

[100] H. Negi, T.Mitra, and A. Roychoudhury, "Accurate estimation of cache-related preemption delay," in *CODES+ISSS '03: 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2003, pp. 201–206. [Pg.33]

[101] G. Nelissen, "Efficient optimal multiprocessor scheduling algorithms for real-time systems," Ph.D. dissertation, Université libre de Bruxelles, 2012. [Pg.15]

[102] J. Nieh, C. Vaill, and H. Zhong, "Virtual-time round-robin: An o(1) proportional share scheduler," in *USENIX Annual Technical Conference*, Jun 2001, pp. 245–259. [Pg.26], [Pg.29], [Pg.34]

[103] J. Nieh, C. Vaill, and H. Zhong, "Group ratio round-robin: An O(1) proportional share scheduler," in *General Track: 2004 USENIX Annual Technical Conference*, Jun 2004, pp. 245–259. [Pg.30]

[104] A. S. Pillai and T. Isha, "Energy efficient task allocation and scheduling in distributed homogeneous multiprocessor systems," *WSEAS Transactions on Computers*, vol. 13, 2014. [Pg.36]

[105] Qualcomm, "Snapdragon S4, S3, S2 and S1," https://www.qualcomm.com/snapdragon/processors/s4-s1. [Pg.63]

[106] S. Ramabhadran and J. Pasquale, "Stratified round robin: A low complexity packet scheduler with bandwidth fairness and bounded delay," in *ACM SIGCOMM*, 2003, pp. 239–249. [Pg.30]

[107] P. Ramanathan and M. Hamdaoui, "A dynamic priority assignment technique for streams with (m, k)-firm deadlines," *IEEE Transaction on Computers*, vol. 44, no. 12, pp. 1443–1451, 1995. [Pg.26]

TH-2256_136101018

[108] G. Raravi, B. Andersson, V. Nélis, and K. Bletsas, "Task assignment algorithms for two-type heterogeneous multiprocessors," *Real-Time Systems*, vol. 50, no. 1, pp. 87–141, Jan 2014. [Pg.6], [Pg.10], [Pg.11], [Pg.108], [Pg.109]

[109] J. Regehr, M. Jones, and J. Stankovic, "Operating system support for multimedia: The programming model matters, Tech. Rep. Microsoft Research (MSR-TR-2000-89), Sep 2000. [Pg.25]

[110] A. Sarkar, P. P. Chakrabarti, and R. Kumar, "Frame-based proportional round-robin," *IEEE Transactions on Computers*, vol. 55, no. 9, pp. 1121–1129, Sep. 2006. [Pg.2]

[111] A. Sarkar, P. Chakrabarti, and R. Kumar, "Frame-based proportional round-robin," *IEEE Transactions on Computers*, vol. 55, no. 9, pp. 1121–1129, 2006. [Pg.34]

[112] A. Sarkar, A. Shanker, S. Ghose, and P. P. Chakrabarti, "A low-overhead partition-oriented erfair scheduler for hard real-time embedded systems," *IEEE Embedded Systems Letters*, vol. 3, no. 1, pp. 5–8, March 2011. [Pg.2]

[113] A. Sarkar, P. P. Chakrabarti, and S. Ghose, "Partition oriented frame based fair scheduler," *J. Parallel Distrib. Comput.*, vol. 70, no. 7, pp. 707–718, July 2010. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2010.03.005 [Pg.2], [Pg.7]

[114] M. Shafique, S. Garg, J. Henkel, and D. Marculescu, "The eda challenges in the dark silicon era: Temperature, reliability, and variability perspectives," in *Annual Design Automation Conference*. ACM, 2014, pp. 1–6. [Pg.7]

[115] Y. Shin, K. Choi, and T. Sakurai, "Power optimization of real-time embedded systems on variable speed processors," in *Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '00. Piscataway, NJ, USA: IEEE Press, 2000, pp. 365–368. [Pg.3], [Pg.43], [Pg.164]

[116] M. Shojafar, N. Cordeschi, D. Amendola, and E. Baccarelli, "Energy-saving adaptive computing and traffic engineering for real-time-service data centers," in *2015 IEEE International Conference on Communication Workshop (ICCW)*, June 2015, pp. 1800–1806. [Pg.36]

TH-2256_136101018

[117] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," *SIGARCH Comput. Archit. News*, vol. 31, no. 2, pp. 2–13, May 2003. [Online]. Available: http://doi.acm.org/10.1145/871656.859620 [Pg.3], [Pg.12], [Pg.137]

[118] M. Spuri and G. Buttazzo, "Efficient aperiodic service under earliest deadline scheduling," in *15th IEEE Real-Time Systems Symposium*, Dec 1994, pp. 2–11. [Pg.26]

[119] M. Spuri, G. Buttazzo, and F. Sensini, "Robust aperiodic scheduling under dynamic priority systems," in *16th IEEE Real-Time Systems Symposium*, 1995, pp. 210–221. [Online]. Available: citeseer.ist.psu.edu/spuri95robust.html [Pg.26]

[120] M. S. Squillante and E. D. Lazowska, "Using processor-cache affinity information in shared-memory multiprocessor scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 131 – 143, 1993. [Pg.33]

[121] M. Squillante and R. Nelson, "Analysis of task migration in shared-memory multiprocessor scheduling," in *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1991, pp. 143–155. [Pg.33]

[122] A. Srinivasan, P. Holman, and J. Anderson, "The case for fair multiprocessor scheduling," in *11th International Workshop on Parallel and Distributed Real-time Systems, Nice, France*, Apr 2003. [Pg.22], [Pg.23], [Pg.25], [Pg.30]

[123] J. Stankovic, "Continuous and multimedia os support in real-time control applications," in *HOTOS '95: Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*. Washington, DC, USA: IEEE Computer Society, 1995, p. 8. [Pg.25]

[124] I. Stoica, H. Abdel-Wahab, and K. Jeffay, "On the duality between resource reservation and proportional share resource allocation, Tech. Rep. TR_96_19, 1996. [Online]. Available: citeseer.ist.psu.edu/stoica97duality.html [Pg.25], [Pg.26]

[125] I. Stoica, H. Abdel-Wahab, K.Jeffay, S.Baruah, J. Gehrke, and C.Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *17th IEEE Real-Time Systems Symposium*, Dec 1996, pp. 288–299. [Pg.26], [Pg.28]

TH-2256_136101018

[126] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002. [Pg.167]

[127] J. Torrellas, A. Tucker, and A. Gupta, "Benefits of cache-affinity scheduling in shared-memory multiprocessors: a summary," *SIGMETRICS Perform. Eval. Rev.*, vol. 21, no. 1, pp. 272–274, 1993. [Pg.33]

[128] S. Tosun, "Energy- and reliability-aware task scheduling onto heterogeneous mpsoc architectures," *The Journal of Supercomputing*, vol. 62, no. 1, pp. 265–289, Oct 2012. [Pg.38]

[129] P. University, "Princeton application repository for shared-memory computers (PARSEC)," *url: http://parsec.cs.princeton.edu.* [Pg.xxiii], [Pg.131], [Pg.154]

[130] C. A. Waldspurger, "Lottery and stride scheduling: Flexible proportional-share resource management, Tech. Rep. MIT/LCS/TR-667, 1995. [Pg.29]

[131] W. Wang and P. Mishra, "Leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in real-time systems," in *2010 23rd International Conference on VLSI Design*, Jan 2010, pp. 357–362. [Pg.72], [Pg.115]

[132] Z. Wang, S. Ranka, and P. Mishra, "Efficient task partitioning and scheduling for thermal management in multicore processors," in *International Symposium on Quality Electronic Design*, 2015. [Pg.39], [Pg.150]

[133] Y. wen Zhang, "Energy-aware mixed partitioning scheduling in standby-sparing systems," *Computer Standards and Interfaces*, vol. 61, pp. 129 – 136, 2019. [Pg.38]

[134] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, and et al., "The worst-case execution-time problemoverview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, May 2008. [Online]. Available: https://doi.org/10.1145/1347375.1347389 [Pg.5]

[135] J. Yang, X. Zhou, M. Chrobak, Y. Zhang, and L. Jin, "Dynamic thermal management through task scheduling," in *ISPASS 2008 - IEEE International Sympo-*

TH-2256_136101018

sium on Performance Analysis of Systems and software, April 2008, pp. 191–201. [Pg.150]

[136] L. Yuan, S. Leventhal, and G. Qu, "Temperature-aware leakage minimization technique for real-time systems," in *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '06. New York, NY, USA: ACM, 2006, pp. 761–764. [Online]. Available: http://doi.acm.org/10.1145/1233501.1233658 [Pg.41]

[137] J. Zhou, T. Wei, M. Chen, J. Yan, X. S. Hu, and Y. Ma, "Thermal-aware task scheduling for energy minimization in heterogeneous real-time mpsoc systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 8, pp. 1269–1282, Aug 2016. [Pg.41]

[138] J. Zhou, K. Cao, P. Cong, T. Wei, M. Chen, G. Zhang, J. Yan, and Y. Ma, "Reliability and temperature constrained task scheduling for makespan minimization on heterogeneous multi-core platforms," *Journal of Systems and Software*, vol. 133, pp. 1–16, 2017. [Pg.40], [Pg.150]

[139] J. Zhou, J. Yan, J. Chen, and T. Wei, "Peak temperature minimization via task allocation and splitting for heterogeneous mpsoc real-time systems," *Journal of Signal Processing Systems*, vol. 84, no. 1, pp. 111–121, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11265-015-0994-4 [Pg.40]

[140] D. Zhu, D. Mosse, and R. Melhem, "Multiple-resource periodic scheduling problem: how much fairness is necessary?" in *IEEE Real-Time Systems Symposium*, Dec 2003, pp. 142–151. [Pg.34]

TH-2256_136101018