# Chapter 1
# Introduction

## 1.1. Background

A real-time system is a system whose correctness depends not only on correct functionality but also on the timeliness of the generated results. Real-time applications span a wide range of domains including automotive and flight control systems, monitoring systems, multimedia systems, virtual reality, interactive gaming, robotics, telecommunications, etc. A late response in these systems might cause a wrong behaviour which could possibly even lead to a critical system failure. A typical example is an *anti-lock braking system* in a car [63, 125] that is required to release the brakes within (say) 60 milliseconds to prevent the wheel from locking.

In most real-time systems, the task / application scheduler is a major architectural component responsible for ensuring proper processing of all tasks having timeliness constraints on their execution response. In the presence of several concurrent activities running on a processor, the real-time scheduler has to ensure that each activity completes within its deadline.

Most traditional real-time execution models are based on Liu and Layland's periodic task model [80] or Mok's sporadic task model [91]. In periodic task systems, the task instances (or jobs) arrive regularly at periodic time intervals. Jobs of a sporadic task on the other hand have a lower bound on their inter-arrival times, but may otherwise arrive randomly (have no upper bound). Scheduling of task systems which are based on these traditional execution models have usually been carried out by priority-driven scheduling techniques like Rate Monotonic Analysis (RMA) and Earliest Deadline First (EDF) (in which the highest priority task present in the ready queue always gets hold of the resource).

However, there also exists a large class of applications which not only demands meeting deadlines, but also demands CPU reservation to ensure a

minimum guaranteed Quality of Service (QoS). These demands are often of the form: *reserve X units of time for application A out of every Y time units*. Ensuring the timing constraints of these systems is difficult using the traditional scheduling models like RMA and EDF. *Rate Based Resource Allocators* [2, 103, 111, 120] with their ability to provide such well-defined rate specifications form a more effective scheduling strategy in the design of many of these systems.

Rate based schedulers form a flexible and natural resource management strategy in many of today's embedded systems running complex real-time applications which require to operate under stringent performance and resource constraints. These complex real-time systems are often composed of independent, coexisting, possibly misbehaving (by attempting to use more CPU time than was allocated to it) real-time applications with different timeliness constraints. An interesting example is provided by many of today's hand-held devices simultaneously executing a mix of different kinds of applications such as real-time signal processing, continuous media (audio and video streams), interactive gaming, email, web browsing, etc. To achieve predictable behaviour and to provide a guarantee of application performance, efficient algorithms are required to manage the available resources, especially processor bandwidth, inter-processor communication bandwidth, cache memory, power, etc. With increased use of multiprocessor and multi-core systems in these embedded applications, overheads of scheduling these applications across the available computing resources have become a design concern of high criticality.

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)* [41, 94, 102], *server-based allocation* [4, 112], and *generalized Liu and Layland style allocation* [57, 103]. This research work is based in the area of fluid-flow allocation techniques.

The *Fluid-flow allocation* policies primarily aim to ensure that each task receives a precise share of the CPU at all times. A sub-class of the fluid-flow allocation techniques, *Proportional fair scheduling*, provides optimum fairness accuracy in a dynamic environment (where new tasks arrive and existing tasks depart at run time) and a fully utilizable processor bandwidth, but at the cost of high scheduling, migration and context switch overheads [7, 22, 23, 115]. We now present an overview of the proportional fair scheduling methodology.

Consider a set of tasks $\{T_1, T_2, ..., T_n\}$, with each task $T_i$ having a computation requirement of $e_i$ time units, required to be completed within a period of $p_i$ time units from the start of the task. Proportional fair schedulers define *weight*

of $T_i$ as $wt_i = \frac{e_i}{p_i}$ and prescribe that task allocations should be managed in such a way that each task is executed at a consistent rate proportional to its own weight. More formally, let the start time of a task $T_i$ be $s_i$. Then proportional fairness guarantees the following for every task $T_i$: *at the end of any time slot* $t$, $s_i \leq t \leq s_i + p_i$, *at least* $\frac{e_i}{p_i} * (t - s_i)$ *of the total execution requirement of* $e_i$ *must be completed*. Obviously, for such a criterion to be guaranteed in a system of $m$ processors, we must have $\sum_{i=1}^{n} e_i/p_i \leq m$. Also, since we usually consider discrete timelines, appropriate integral values must be considered while examining fairness.

Typically, proportional fair schedulers divide the tasks into equal sized subtasks. At every time slot, appropriate subtasks from the set of runnable tasks are scheduled to ensure fairness. Numerous proportional fair algorithms have appeared in literature. These include *PD* (1995) [23], $Pfair$ (1996) [22], $ERfair$ (2000) [7] and $PD^2$ (2004) [8] and their variants. Generally, these schedulers use priority queues to select the next subtask, leading to $O(lg\ n)$ overheads per processor per time slot. Being global in nature, they also suffer from unrestricted migration and context-switch related overheads. These overheads turn out to be reasonably expensive especially in real-time systems [51]. Several variants of proportional fair schedulers which sacrifice on the stringency of proportional fairness guarantees to control and reduce these overheads have been proposed [37, 51, 78].

This thesis primarily aims at the development of low overhead real-time multi-processor proportional fair schedulers which provide high throughput and fairness and have the ability to efficiently handle dynamic task sets. In view of the fact that proportional fair schedulers form an attractive scheduling alternative in today's emerging multiprocessor / multicore embedded systems which often work under high workloads and stringent resource constraints, this work also endeavors to develop proportional fair algorithms that can work effectively under practical situations like limited power, overloads, etc.

The rest of the chapter is organized as follows. Motivation and objectives of our work are specified in sections 1.2 and 1.3. We present a summary of the work done by providing a brief description of each algorithm developed by us in Section 1.4. The chapter ends by providing the organization of this thesis in Section 1.5.

## 1.2. Motivation

Proportional fair schedulers with their strong notions of fairness in resource allocation, ability to provide guaranteed rate-based execution progress for all tasks, optimal resource utilizability and fault resilience, potentially form a very attractive scheduling mechanism. However, in spite of all these desirable features, actual implementations of Pfair and ERfair schedulers are limited mainly due to relatively high scheduling complexity, inter-processor task migration and context-switch related overheads. Along with predictable performance guarantees, emerging real-time systems and applications are confronted with situations which also demand improved throughput, lower energy overheads and ability to handle transient overloads.

Reducing the number of task migrations and context switches has been found to be one of the most useful methods to gain system throughput [51]. In addition, fewer migrations also help improve energy efficiency. These observations have been the principal motivation towards our endeavor to develop algorithms for fast and flexible proportional fair schedulers with reduced migration / context-switch overhead.

Power reduction has now emerged as a first-class design criterion. We have observed that processor slack in underloaded proportional fair systems may be judiciously used to shutdown the processor at idle times, thereby reducing static energy consumption in the process. As static energy dissipation is becoming the major source of energy drain in today's processors [1], we have directed our efforts to design and develop proportional fair algorithms that attempt to maximize continuous processor idle periods and shutdown a processor during these intervals.

With exponential increase in the number of real-time applications and their complexity, both server grade and embedded systems are often subject to sudden surges in workload leading to transient overloads. Detecting such situations a priori and avoiding them, maintaining an acceptable quality of service to all clients until such surge of computing requirement subsides, is of utmost importance in many cases. Hence, we have endeavoured to suggest variations of proportional fair algorithms that effectively handle such overloads.

1. **Unrestricted migration and context-switch related overheads:** In order to meet the objectives of optimal fairness and bandwidth utilization in multiprocessing environments, proportional fair schedulers gener-

ally resort to a global scheduling policy where a single run-queue allocates appropriate client tasks to all the processors at each time-slot. This often leads to unrestricted task migrations and context switches. Inter-processor task migrations primarily effect two types of overheads, namely, cache-miss related overheads and communication related overheads. Cache-miss related overheads refer to the delay suffered by resumed threads of execution due to compulsory and conflict misses while populating the caches with their evicted working sets after a migration takes place. Communication 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, the more loosely-coupled a system, the higher will be the communication overhead. Context switches primarily effect cache-miss related overheads only. Recently, there has been a few attempts [37, 69, 78, 128] to control these overheads by trading off on fairness while still preserving optimal schedulability. However, not much effort has gone towards the development of algorithms for restricting overheads while simultaneously guarantying optimality in terms of both fairness and schedulability.

2. **Power Awareness:** While increasing throughput through the reduction of scheduling, migration and preemption overheads is extremely important in real-time systems where time is at a premium, power-aware scheduling techniques have also gained importance in recent times, especially with the advent of mobile and hand-held embedded systems which operate with limited energy budget [17, 53, 108, 110]. The scheduling requirements in these systems is generally characterized by a trade-off between a need to meet the performance constraints while simultaneously trying to minimize the overhead of power consumption. There has been considerable research in this area in recent years and various low-power scheduling algorithms have been developed. However, little work has been done in the area of power-aware proportional fair scheduling even though these schedulers with their ability to provide hard timeliness and QoS guarantees and optimal schedulability to a diverse variety of real-time applications on multi-processors promise to be an attractive scheduling alternative on these systems. Hence, research on energy-efficient proportional fair algorithms appears to be an area of significant importance.

3. **Overload Handling:** A further observation is that, with more and more diverse and complex real-time applications being executed on today's embedded systems, these systems are expected to be subjected to much higher workloads resulting in increased chances of transient overload situations. In such situations, although highly desirable, it may sometimes become impossible to meet all task deadlines. It may be more desirable at this stage to complete some portions of every task rather than giving up completely the execution of some tasks. The imprecise computation model [79] allows for this trade-off of the quality of computations in favour of meeting the deadline constraints. In this model, a task is logically decomposed into two sub-parts, mandatory and optional. The mandatory sub-part of each task is required to be completed by its deadline, while we have a choice whether or not to execute the optional part depending on the criticality of a given task or any other reward function. However, the generic proportional fair scheduling framework provides no resource allocation mechanisms for such imprecise computation models or means for preferential execution of an application with respect to others. Proportional fair schedulers attempt to provide graceful and fair performance degradation to all tasks in times of overload, and herein lies their primary drawback - fair performance degradation for all tasks often cause a majority or all of the tasks to miss their deadlines in transient overload situations. We have identified this deficiency and the necessity for algorithms which fare better under transient overloads.

## 1.3. Objectives

The aim of this work has been to investigate into some of the theoretical and practical aspects of proportional fair schedulers keeping the problems discussed in the previous section in view. In particular, the objectives may be summarized as follows:

1. Design of techniques that provide optimal algorithms (in terms of both schedulability and fairness) but with low migration and context switch related overheads.

2. Development of low overhead scheduling strategies with bounded migration and unfairness properties. Providing fairness accuracy and migration overheads as controllable design parameters which may be set according to the fairness and speed requirements of the system.

3. Devising mechanisms which can utilize task execution slacks in underloaded systems to shutdown the processor at idle times, thereby minimizing power consumption while still preserving optimal schedulability and fairness properties.

4. Incorporating techniques for the a priori detection and avoidance of transient overloads in proportional fair resource allocators.

## 1.4. Contributions of this Work

As part of the research work, we have developed the following five real-time, proportional fair schedulers:

1. **Sticky-ERfair: A Task-Processor Affinity Aware Scheduler:** This is a fully global optimal algorithm with reduced migrations and context switches.

2. **Partition Oriented ERfair Scheduler (*POES*):** The second algorithm is also optimal with reduced migrations but follows a semi-partitioned approach switching towards higher global-ism with increasing load and vice-versa.

3. **Partition Oriented Frame Based Fair Scheduler (*POFBFS*):** Our third algorithm is a frame-based work-conserving proportional fair scheduler that allows much higher reduction in the number of migrations although allowing slight deviations from perfect fairness at high workloads.

4. **ERfair Scheduler with Shutdown on Multiprocessors (*ESSM*):** This algorithm is an ERfair scheduler which attempts to locally maximize the total length of processor shutdown intervals over the schedule length and thereby reduces static energy dissipation in the system.

5. **Safe-ERfair: ERfair with a priori Overload Detection and Avoidance:** The last algorithm developed by us is a fair scheduler with the ability to detect and avoid possible transient overloads in imprecise computational workloads.

We now provide an overview of each of the above algorithms.

### 1.4.1. Sticky-ERfair: A Task-Processor Affinity Aware Scheduler

Sticky-ERfair is a global algorithm which modifies Basic-ERfair to minimize the number of inter-processor task migrations and preemptions while maintaining the same order of its scheduling complexity. To achieve this, the algorithm primarily employs two mechanisms: (I) keeping track of the processor where a task last executed, and (II) utilizing slack resulting from higher execution rates of tasks in ERfair systems which are not fully loaded. (When a system has spare capacity, tasks executing in ERfair fashion utilize this spare bandwidth and proceed at higher execution rates proportional to their task weights.)

For any given processor $V_i$ and a set of tasks $\tau = \{T_1, T_2, ..., T_\sigma\}$ which executed on $V_i$ the last time it was allocated a processor, Sticky-ERfair attempts to execute the most recently executed task $T_\rho$ from the set $\tau$ such that such an execution does not generate any possibility of future ERfairness violations. In addition to saving cache-misses by reducing preemptions, this methodology also allows re-use of any traces of non-dirty data of previously executed tasks that still remain in cache. This is because, the more recent the last execution of a task on a given processor, higher becomes the probability that some of its cache contents still remain valid.

Experimental results reveal that Sticky-ERfair can achieve upto 40 times reduction both in the number of migrations and preemptions suffered with respect to the original ERfair scheduler [7] (henceforth, we call this scheduler *Basic-ERfair*) for a set of 25 to 100 tasks running on 2 to 10 processors, while simultaneously guaranteeing ERfairness at each time slot. Theoretical analysis with equal priority tasks using Sticky-ERfair shows that unlike Basic-ERfair, the number of migrations is independent of the number of tasks for Sticky-ERfair.

### 1.4.2. Partition-Oriented ERfair Scheduler (POES)

POES is an ERfair scheduler that includes a low-overhead partitioning / merging mechanism which ensures 100% schedulability (as in global scheduling) while simultaneously leveraging the benefits of lower scheduling and migration overheads of partition-based scheduling.

The principal objective of the POES scheduler is to maximize scheduling locality while not jeopardizing the optimal schedulability and fairness properties of global ERfair. Given a set of $n$ periodic tasks $\{T_1, T_2, ..., T_n\}$ to be scheduled in a system of $m$ processors $\{V_1, V_2, ..., V_m\}$, POES maintains the task and pro-

cessor sets partitioned into $k$ disjoint groups ($\{\tau 1, \tau 2, ..., \tau k\}$ and $\{\varrho 1, \varrho 2, ..., \varrho k\}$ respectively) at any instant during the schedule length. The tasks in group $\tau i$ are allowed to execute and migrate only within its corresponding processor group $\varrho i$. If $\{T_{\tau i_1}, T_{\tau i_2}, ..., T_{\tau i_{|\tau i|}}\}$ denote the tasks in group $\tau i$ and $\{V_{\varrho i_1}, V_{\varrho i_2}, ..., V_{\varrho i_{|\varrho i|}}\}$ denote the processors in group $\varrho i$, then such a partition is feasible, only if,

$$\forall_{i=1}^{k} \sum_{j=1}^{|\tau i|} wt_{\tau i_j} \leq |\varrho i| \tag{1.1}$$

That is, the sum of weights of the tasks in each group must be less than or equal to the total capacity of the processors in its corresponding processor group.

At lower workloads, as long as the schedulability of the task set is not compromised, a complete partition is feasible and the number of groups $k$ is equal to the number of processors $m$. If a new task arrives which cannot be feasibly accommodated into any unique task group, two or more task groups and their corresponding processor groups are *merged* so that the newly formed group will be able to feasibly accommodate the new task. Similarly, when an existing task departs from any given task-group and the corresponding processor-group contains more than one processor, POES tries to *split* this group into two or more groups such that the feasibility condition in Equation 1.1 remains valid after such a split. Therefore, instead of shifting from a completely partitioned system to a fully global one when a certain hard workload threshold is surpassed, POES adopts a more continuous design allowing global scheduling within an overall partitioned approach.

Experimental results reveal that POES incurs almost no migrations at low workloads and achieves upto 32 times reduction in the number of migrations suffered with respect to Basic-ERfair on a set of 2 to 16 processors even when the average task workload in the system is as high as 85%.

### 1.4.3. Partition Oriented Frame Based Fair Scheduler (POF-BFS)

The work presented in this subsection is motivated by the fact that there exists a large class of systems which need to support a dynamic mix of coexisting, independent applications which are strictly not hard real-time but has firm or soft real-time requirements [82, 83, 104]. Missed deadlines are undesirable but not catastrophic. Thus, in these systems, while maintaining fairness is important, a slight deviation from perfect fairness can be tolerated if it sub-

stantially reduces migration overheads. This area of our research introduces the notion of frames (where a *frame* denotes a time interval) in multiprocessor proportional fair scheduling.

POFBFS works as follows: we define a frame/time-window of a certain specific size and allocate shares (of time slots) to each task in proportion to their weights $\frac{e_i}{p_i}$ within the frame. Then a two phased mechanism is used to partition the task set into the $m$ available processors. The first phase allocates tasks to individual processors using the worst-fit decreasing (WFD) [88, 96] heuristic such that the sum of task shares in each processor is less than the frame size $G$ and simultaneously builds a sorted list $\Lambda_{MGR}$ of tasks that cannot receive its full share into any single processor. The second phase allocates the tasks in list $\Lambda_{MGR}$ by partitioning the share of each task into more than one processor. Having assigned the tasks into the available processors, the task shares are executed using an ERfair scheduler in each processor. After execution completes inside a frame, each task is put in an appropriate future frame such that ERfairness [7] of the system remains preserved at frame boundaries.

Theoretical analysis with POFBFS shows that the algorithm guarantees at most $(m - 1)$ task migrations per frame while experimental results reveal that when compared to Basic-ERfair, it achieves 3 to 100 times reduction in the number of migrations (for a set of 25 to 100 tasks running on 2 to 8 processors). This reduction is however achieved at the cost of slight deviations from perfect fairness under high workloads.

### 1.4.4. ERfair Scheduling with Processor Shutdown (ESSM)

As leakage power becomes a critical concern especially in embedded systems that are powered by limited energy sources, reduction of system energy consumption through processor shutdown is becoming a policy of increasing importance. Realizing this need, we have designed and developed the *ESSM (ERfair Scheduler with Shutdown on Multiprocessors)* algorithm which attempts to locally maximize the total length of shut-down intervals while simultaneously reducing the number of such intervals (because with each shutdown / wakeup is associated an extra energy consumption overhead). We now present an overview of the basic working principle of this algorithm.

ERfair being a work conserving algorithm (that is, the system can never remain idle when there are ready tasks available for execution in the run-queue), all tasks $T_i$ running in an underloaded ERfair system always execute at rates $ef\_wt_i$ (effective weight of task $T_i$) which is higher than the required

execution rates $wt_i$ (original weight of $T_i$). This higher execution rate is effected by the fact that, due to the nature of the ERfair scheduler, the spare capacity gets naturally shared equally among client tasks. This results in the addition of a part of the spare system capacity to the original weight of each task, ultimately causing faster execution progress.

Faster execution progress in turn results in the accumulation of *slack* by each process. The slack of a task $T_i$ at a given time $t$ denotes the maximum interval for which execution of $T_i$ may be suspended starting from $t$ without the possibility of ERfairness violation. From the discussion on proportional fair scheduling in Section 1.1, it may be intuitively understood that a task's slack will be 0 when it just arrives. As it keeps executing at rate $ef\_wt_i$, its slack $slack_i(t)$ also keeps on increasing linearly with time till the time where it completes execution of its current instance. This is the point where the slack of the task becomes maximal. Thereafter, its slack decreases linearly until its deadline / period is reached where the slack becomes 0 again and the next instance / job of the task arrives.

A processor may sleep for a maximum duration of time $t_s$ at time $t$, provided there exists a subset of tasks $\tau = \{T_{\tau 1}, T_{\tau 2}, ..., T_{\tau k}\}$, such that the sum of the weights of this subset ($\sum_{i=1}^{k} wt_{\tau i}$) is atleast enough to shutdown a single unit capacity processor and the minimum of the slacks accumulated by these tasks is equal to $t_s$ ($\mathbf{min}_{i=1}^{k} slack_{\tau i}(t) = t_s$). Given this feasibility condition for processor shutdown, the *ESSM* algorithm uses a procrastination scheme to search for time points within the schedule where shutdown intervals are locally maximal and shuts down one or more processors at those time points.

Evaluation results reveal that ESSM achieves good shutdown efficiency and provides 2 to 15 times higher effective shutdown lengths as compared to the Basic ERfair scheduler on systems consisting of 1 to 8 processors.

### 1.4.5. Safe-ERfair: ERfair with a priori Overload Detection and Avoidance

Overload handling in dynamic hard real-time systems is of utmost importance since meeting deadlines in these systems is not an option but a necessity [121]. Many of these systems include critical tasks such as telepresence, real-time voice communication, interactive gaming, etc., where possible overloads must be detected and avoided *a priori* (when a task arrives) through efficient on-line admission control because a particular quality-of-service (QoS), once offered to these applications, cannot be altered in the middle of task execution.

In this area of research, we have developed the *Safe-ERfair* algorithm for *a priori* detection and avoidance of transient overloads in imprecise computational workloads. The imprecise computational tasks that we have considered consist of two parts - a mandatory part and an optional part. A task is said to execute in maximal mode when it executes both its mandatory and optional parts. When the task executes only it's mandatory part, it executes in minimal mode. Executing a task in maximal mode (that is, with the optional part) leads to a reward, where the value of the reward is dependent on a task's criticality. The objective is to maximize processor utilization by maximizing the reward for executing optional parts while not sacrificing the ERfairness timing constraints of the system and ensuring that all tasks are able to execute atleast their mandatory parts. Experimental results show that for all the test-cases considered, Safe-ERfair performs better than the simplistic $O(n^2)$ algorithm ESOM.

## 1.5. Organization of the Thesis

The thesis is organized into eight chapters. A summary of the contents of the chapters is as follows:

**Chapter 2** This chapter gives a survey of real-time and rate based resource allocation, emphasizes its need and area of applicability. Different proportional share and proportional fair algorithms have been surveyed here. Finally, the chapter presents an overview of a few emerging scheduling considerations like energy awareness, overload handling, fault-tolerance, etc.

**Chapter 3** This chapter discusses the *Sticky-ERfair* scheduler, a global algorithm which attempts to restrict the migration and preemption related overheads of the *Basic-ERfair* scheduler. Theoretical analysis and experimental results for the Sticky-ERfair algorithm have been presented and discussed.

**Chapter 4** In this chapter, we present the *Partition-Oriented ERfair Scheduler (POES)*, another low-overhead ERfair scheduler that employs an online partitioning / merging mechanism that retains the optimal schedulability of a fully global scheduler by merging processor groups as resources become critical while using partitioning for fast scheduling at other times.

**Chapter 5** This chapter deals with the *Partition-Oriented Frame-Based Fair Scheduler (POFBFS)*, a frame-based proportional fair scheduler which parti-

tions task sets into individual processors at the beginning of each frame (where *frame* denotes a time interval) and globally resynchronizes at the end of the frame. We show (through theoretical analysis and experiments) that POFBFS achieves achieves high reductions in task migrations albeit at the cost of occasional slight deviations from perfect fairness.

**Chapter 6** Here, we present a novel ERfair scheduling algorithm called *ERfair Scheduling with Shutdown on Multiprocessors (ESSM)*, that attempts to reduce system wide energy consumption by locally maximizing the processor shut-down intervals while not sacrificing the ERfairness timing constraints of the system.

**Chapter 7** In this chapter, we discuss *Safe-ERfair*, a fair algorithm with built-in *a priori* overload handling mechanisms that detects and avoids possible transient overloads in future using a given task arrival statistics.

**Chapter 8** Finally, we conclude in this chapter. We summarize with a comparative study of the developed algorithms and also discuss the work in progress, possible extensions and future work that can be done in this area.