Institute of Software Technology

Department of Programming Languages and Compilers

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis Nr. 3578

Analysis and Simulation of Scheduling Techniques for Real-Time Embedded Multi-core Architectures

Sanjib Das

Course of Study: INFOTECH

Examiner: Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder

Supervisor: Dipl.-Inf. Mikhail Prokharau

Commenced: October 31, 2013

Completed: June 27, 2014

CR-Classification: C.1.4, D.4.7, D.4.8, I.6.6, I.6.7

Abstract

In this modern era of technological progress, multi-core processors have brought significant and consequential improvements in the available processing potential to the world of real-time embedded systems. These improvements impose a rapid increment of software complexity as well as processing demand placed on the underlying hardware. As a consequence, the need for efficient yet predictable multi-core scheduling techniques is on the rise.

As part of this thesis, in-depth research of currently available multi-core scheduling techniques, belonging to both partitioned and global approaches, is done in the context of real-time embedded systems. The emphasis is on the degree of their usability on hard real-time systems, focusing on the scheduling techniques offering better processor affinity and the lower number of context switching. Also, an extensive research of currently available real-time test-beds as well as real-time operating systems is performed.

Finally, a subset of the analyzed multi-core scheduling techniques comprising PSN-EDF, GSN-EDF, PD^2 and PD^{2*} is simulated on the real-time test-bed LITMUS^{RT}.

Acknowledgments

First and foremost, I offer my sincerest obligation to honorable Prof.Dr.rer.nat./Harvard Univ. Erhard Plödereder for giving the opportunity to write my master's thesis at Department of Programming Languages and Compilers of the Institute of Software Technology at the University of Stuttgart.

Moreover, I would like to show my humble gratitude to my supervisor Dipl.-Inf. Mikhail Prokharau who has supported me throughout my thesis with his knowledge, cordial supervision, valuable pieces of advice and patience while at the same time giving me the room to work in my own way.

I am profoundly grateful to my parents Jatindra Nath Das and Shikha Das for their unconditional support and constant encouragement throughout my life.

In addition, it is my pleasure to express gratefulness to all the people who contributed, in whatever manner, to the success of this work.

Contents

List of Figures

List of Tables

Abbreviations

CFS	Completely Fair Scheduler
DM	Deadline Monotonic Scheduling
EDF-BF	Earliest Deadline First - Best Fit
EDF-FF	Earliest Deadline First - First Fit
FPU	Floating-Point Unit
G-EDF	Global Earliest Deadline First
GSN-EDF	Global Suspendable Non-Preemptive EDF
GUA	Global Utility Accrual
IPC	Interposes communication
NMIs	Non Maskable Interrupts
NUMA	Non-uniform memory access
P-EDF	Partitioned Earliest Deadline First
Pfair	Proportionate Fairness
POSIX	Portable Operating System Interface
PSN-EDF	Partitioned EDF with synchronization support
RMGT	Rate Monotonic Scheduler for General Task
RMS	Rate Monotonic Scheduler
RMST	Rate Monotonic Scheduler for Short Task
RTOS	Real Time Operating System
TSC	Time Stamp Counter
VFS	Virtual File System
WCET	Worst Case Execution Time

1. Introduction

1.1 Motivation

These days embedded systems are sewn into our day-to-day life in various forms of visible and invisible manner via many different application areas which include consumer electronics, medical imaging, telecommunications, automotive electronics, avionics, space systems, etc. For instance, the progress in use of multi-core platforms in embedded systems has already reached our hands as a form of mobile phones and related devices with small form factor.

The main purpose of a real-time system is to produce the required result within strict time constraints including computational correctness. In other words, in the physical world the purpose is to construct a physical effect within a chosen time-frame [Mohammadi and Akl, 2005]. There are a number of perspectives to classify real-time systems. Depending on the system characteristics, a real-time system can be categorized as hard real-time or soft real-time by considering factors inside the system and factors outside the system [Juvva, 1998].

As many embedded systems are used in safety-critical applications, their correct functionality in the whole system is imperative to avoid severe consequences. It is estimated that 99% of produced microprocessors are integrated into embedded systems [Burns and Wellings, 2001]. Furthermore, as a result of this abrupt technological progress, a significant increment in software complexity and processing demands of real-time systems is seen [Davis and Burns, 2011]. To cope with these processing demands, silicon vendors are concentrating on using multi-core platforms for high-end real-time applications instead of incrementing processor clock speeds in uni-core platforms. By the same token, scheduling research of multi-core architectures offers a broad spectrum of significant opportunities for real-time system producers. [Davis and Burns, 2011] .

Research of uni-core and multi-core real-time scheduling both originated back in late 1960s and early 1970s, consequential advances were made in 1980s and 1990s [Davis and Burns, 2011]. Still, there is sufficient scope for research, although uni-core real-time scheduling is considered reasonably mature to be in industrial practice [Burns and Wellings, 2001]. On the other hand, many of well researched multi-core scheduling techniques are not mature enough to either be applicable or optimal as much as currently available uni-core real-time scheduling techniques.

For this reason, reliable simulation platforms are required to augment the research of scheduling techniques for real-time embedded multi-core architectures, which is also coupled with analytical results that expect guaranteed real-time administration over the system by the most effective use of the available processing capability through employing efficient scheduling policies placed on the underlying hardware.

1.2 Objective

The purpose of this work is to give an overview of currently available real-time embedded multi-core scheduling techniques along with simulation test-beds while giving detailed comparison of their advantages and limitations. Consequently simulate analyzed scheduling techniques on suitable test-bed. The objectives are as categorized as follows:

1. Analysis of scheduling techniques for real-time embedded multi-core architectures
    - In-depth analysis of currently available literature on multi-core scheduling in real-time contexts.

- Comparison of multi-core scheduling techniques, emphasizing the degree of their usability on hard real-time embedded systems.
- Comparison of multi-core scheduling techniques by focusing on techniques offering better processor affinity and the lower number of context switching. In-depth analysis, considering above constraints, of the following scheduling policies:
  - Partitioned approach (includes Partitioned-EDF)
    - Tasksets with implicit deadlines including partitioned RMST, partitioned RMGT, EDF-FF, EDF-BF
    - Tasksets with constrained and arbitrary deadlines including EDF-FFID
  - Global approach with
    - Fixed-job priority including global EDF-US[ς],global EDF(κ)
    - Fixed-task priority including global RM, global RM-US[ς], global DM-DS, global FP
    - Dynamic priority including Pfair, PF, ERfair, PD^2, PD^{2*}, BF, BF^2, LLREF, EDZL

2. Analysis of currently available simulation test-beds for scheduling techniques for real-time embedded multi-core architectures.
   - Analysis and comparison of existing real-time test-beds to evaluate scheduling techniques including the following ones:
     - RTLinux,
     - S.Ha.R.K,
     - MaRTE,
     - RTAI,
     - Xenomai,
     - XtratuM/PaRTiKle,
     - ChronOS,
     - LITMUS^{RT}.
   - Selection of suitable real-time test-bed for simulation of scheduling algorithms analyzed in the previous phase.
3. Simulation of analyzed scheduling techniques focusing on scheduling algorithm performance as well as simulator performance, considering the baseline platforms below:
   - Virtual machine (QEMU emulator): GenuineIntel $x86\_64$, CPU(s):16, CPU MHz:2260.996, Hypervisor vendor:KVM,
   - Physical machine: GenuineIntel $x86\_64$,CPU(s):4, CPU MHz:933.000.

1.3 Organization

The rest of this thesis is composed as follows: Chapters 2 and 3 provide the definitions and terminology required to establish a common notation. Chapter 4 contains a brief classification of available real-time scheduling algorithms. Chapter 5 contains analytical points of view and detailed classification of currently available scheduling techniques for multi-core architectures along with scheduling techniques for real-time embedded multi-core architecture followed by an overview of currently existing simulation test-beds in Chapter 6. The architecture of simulation platforms, the simulation strategies and the simulation results are described in chapter 7. The thesis is concluded by a summary of this whole work and discussion of future work in chapter 8.

2. Definitions

Over the past decades, several scheduling algorithms for real-time systems have been proposed. They evolved through research aimed at the improvement of the predictability of real-time systems. In order to describe the consequences of this research in the next chapters, we describe some basic concepts in the current chapter.

We take the first step with the most fundamental definition of a real-time system and embedded system followed by multi-core architectures. Also, a very important software entity of the operating system, the *process* is defined. Finally, *resources*, *scheduling policy* and *scheduler* come into the focus. In this work, the keywords *task* and *process* are used as synonyms.

2.1 A Real-Time System

The concept of *time* is the principal characteristic which distinguishes real-time computing from other types and comes in the form of the computation time. Where by the word *time* not only the logical result of the system, but also at which point of time the outcome is formed, are described as prerequisites of the system's correctness. Furthermore, by the word *real* an obvious occurrence of an external event as a reaction of the system is indicated during the system's evolution time. Where the system time and the time in a controlled environment are measured using the same time scale [Buttazzo, 2004]. Considering the *deadline*, which is the maximum execution time of a real-time task, real-time systems can be categorized as *hard real-time* and *soft real-time*[Buttazzo, 2004].

2.2 An Embedded System

An embedded system is an information processing system that is encapsulated in a fixed context, built inside a larger system for the purposes of regulating and controlling the system with a predefined functionality [Marwedel, 2006]. Examples can be drawn with information processing systems embedded into enclosing products such as avionics, automobile, and communication equipment. In most of the cases, these systems come with a large number of common real-time constraints, as well as required dependability and efficiency characteristics.

2.3 Multi-Core Systems

A single computing component consisting of more than one autonomous processing unit is called a multi-core processor, where multiprocessing is implemented in a single physical package. Currently, the term *core* is much more preferable in research and production practice than the term *processor.*

Taking the scheduling criterion into account, multi-core systems are described as follows [Davis and Burns, 2011]:

1. *Heterogeneous*: Where the processors are different, on that account task execution rate is dependent on the task and the processor.As a matter of fact, execution of all tasks will not be held on available all processors.
2. *Homogenous*: Here all the processing cores are identical; henceforth execution rate for all tasks is equivalent on each of them.
3. *Uniform*: Execution rate relies only on processor's speed for a task. Hence a processor of speed 2 will double the execution rate of all tasks with speed of 1.

2.4 Task Models

Several processes run on a real-time system with timing constraints. Each of them is known as task, which provides the functionality of the underlying real-time system. Number of invocation for a task can be finite or infinite. Every single invocation is referred as a job [Holman, 2004].

Generaly, a real-time task J_i is described by the parameters below:

- Arrival Time: Denoted as a_i, represents the point in time, when a task becomes ready for execution. It is also indicated by r_i.
- Computation Time: Denoted as C_i, uninterrupted execution time of the task.
- Deadline: Represented as d_i, is the time before which the task should finish execution to avoid damage to the system.
- Start Time: Represented as s_i, is the starting point of task execution.
- Finishing Time: Reffered as f_i, is the finishing point of task execution.

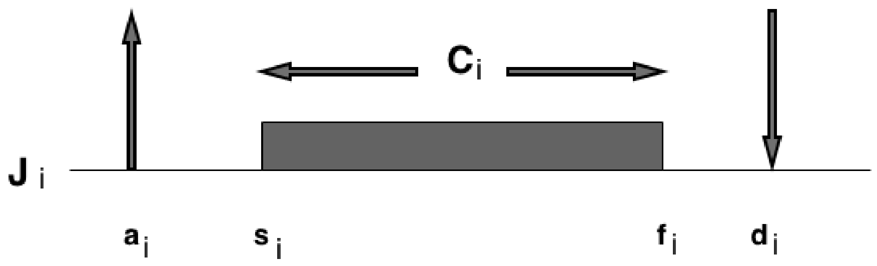


Figure 2.1: Real-Time Task Parameter Buttazzo [2004]

- Criticalness: Which relates the outcomes of a deadline miss.
- Value : The importance of a task with respect to others is represented by v_i.

Figure 2.1 contains an illustration of some of the task parameters.

Real-time applications are usually constructed based on multiple task sets with different criticality level. Though deadline misses are not expected in real-time tasks, soft real-time tasks could still work while missing some deadlines. On the other hand, hard real-time tasks will incur a severe penalty for missing any deadline. Another variant of task sets are named firm real-time tasks which gain reward based on their completion before the deadline.

Consider a task set $T = \tau_1, \tau_2, \tau_3 ....\tau_n$, where WCET of each task $\tau_i \epsilon T$ is C_i . A system will be considered real-time if there exist at least one task $\tau_i \epsilon T$ with the following properties [Mohammadi and Akl, 2005] :

1. *Hard real-time task*: Task τ_i should be complete it's execution by a given deadline D_i;i.e.,$C_i \leq D_i$, is a hard real-time task.
2. *Soft real-time task*: The task τ_i has to pay a penalty depending on how late it has completed its computation after a given deadline D_i. A penalty function $P(\tau_i)$ is defined for the task.

3. *Firm real-time tasks*: A task τ_i gains reward depending on how much earlier it finishes the computation before the given deadline D_i. A reward function is defined as $R(\tau_i)$

The deadline is one of the most roll playing parameter of real-time tasks, and for hard real-time task its importance is inevitable. For a task T_i, deadline D_i is the time when the job of the task musk accomplishes its execution.

Correlating with another parameter period or inter-arrival time, a deadline can be categorized as below:

- *Implicit Deadline*: A task T_i with period P_i and deadline D_i, is an implicit deadlined task if $D_i = P_i$.

- *Constrained deadline*: A task T_i with period P_i and deadline D_i, is an constrained deadlined task if $D_i \leq P_i$.

- *Arbitrary deadline*: A task T_i not constrained with deadline D_i, is an arbitrary deadlined task.

Considering the arrival behavior of tasks in a system, they can also be categorized into the following types:

1. *Periodic Task* : Periodic tasks are released or activated at fixed rates(periods). Usually, periodic tasks must execute once per period. For periodic tasks, the constraints are the period P. Periodic tasks can be categorized as *synchronous* and *asynchronous*.
   - *Synchronous*: When there is a specific point in time at which simultaneous activation or arrival of the tasks occurs;
   - *Asynchronous*: When task arrival times are not simultaneous and are separated by fixed offsets [Davis and Burns, 2011];

2. *Aperiodic Tasks*: Aperiodic tasks are activated in an irregular manner at a possibly unbounded and unknown rate. For aperiodic tasks the constraints are the deadline D.

3. *Sporadic Task* : Sporadic tasks are activated irregularly at a bounded rate. The minimum time interval between two successive activations is defined as the minimum inter-arrival period which characterizes the bounded rate. Usually D, the deadline is the time constraints for sporadic tasks.

Most of the scheduling research on multi-core real-time systems is focused on two types of task model:

1. Periodic task model.

2. Sporadic task model.

In both cases, tasks have an infinite sequence of jobs (invocations). In either model, intratask parallelism is not permitted. [Davis and Burns, 2011]

2.5 Resource

Considering a process, a *resource* is any software architecture that can be used by the process. Considering a core, a *resource* does not execute the instructions of the task. Nevertheless, in both cases a *resource* is used for advancement of task instruction's execution. Typical example of a resource is, a main memory area, or a set of variables, or data structure.

A resource assigned to a specific process is known as *private* and for two or more processes as a *shared* resource. Considering data consistency , a simultaneous access is not allowed by many shared resources and mutual exclusion is required among competing tasks. These types or resources are known as *exclusive resources*.

The section of the code executing under the mutual exclusion constraints is known as a *critical section*. A synchronization mechanism is provided by the operating system to guarantee the sequential access to exclusive resources, e.g., semaphores, which means, when number of tasks is two or more with resource constraints, they have to be synchronized, in case of share exclusive resources.

When a task is waiting to access an exclusive resource, it is defined as *blocked* for that specific resource. All the tasks which are blocked for the same resource are stored into a queue used with a semaphore. A running task enters into a waiting state after execution of a wait primitive on a locked semaphore, and waits in the same state for *signal* primitive execution by another task.

After leaving the waiting state, a task goes to the ready state rather than going to a running state to make CPU assignment to a higher-priority task possible by a scheduling algorithm. A state transition diagram depicted in Figure 2.2, represents the scenario described above.

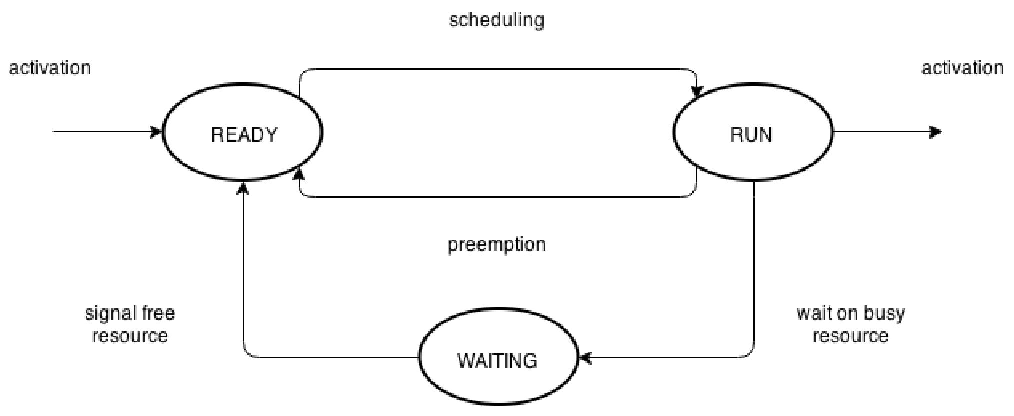


Figure 2.2: Task State Diagram Buttazzo [2004]

2.6 Scheduling Policy

Scheduling policy is the set of rules that are deployed to manage when and how to pick a new process to run. In the case of running a set of concurrent tasks on a single core, there is a possibility of CPU time overlapping. Using a scheduling policy tasks are allocated to the CPU core according to a predefined rule, e.g., priority of the task, or value of the task.

2.7 Schedulers

Scheduler is a functional entity of an operating system, where the scheduling policies are implemented. The main intention of the scheduler is to assign a CPU to a task by evaluating predefined scheduling algorithms. These specific operations of allocating CPU to a task are known as *dispatching*.

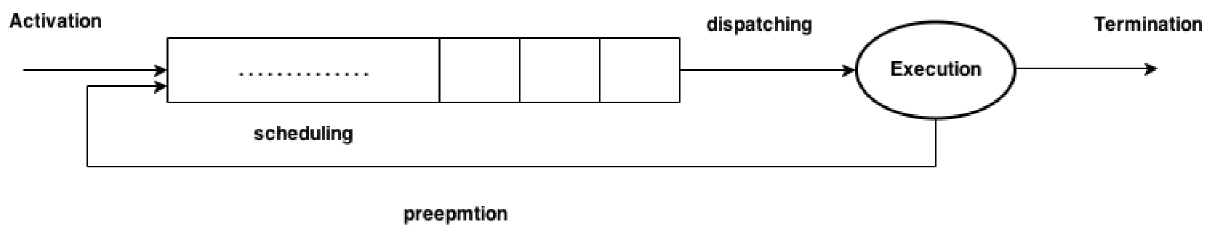


Figure 2.3: Task queue in Scheduler Buttazzo [2004]

In the Figure 2.3 a basic schematic structure of a scheduler is shown.

3. Terminology

3.1 Schedulability and Optimality of scheduling algorithm and Feasibility of tasksets

- *Feasibility*: Feasibility of a taskset with respect to a given system is defined by the existence of some scheduling algorithm which is able to schedule possible all combinations of jobs, originated by the taskset without any deadline miss.
- *Optimality*: Optimality of a scheduling algorithm with respect to the task model and to the system is defined by the ability to schedule all of the feasible tasksets satisfying the task model.
- *Schedulability*:For a assigned scheduling policy, if a task executes without missing deadline, then that task is referred to as schedulable under the assigned scheduling algorithm.

3.2 Processor Demand Bound Function

The term processor demand bound function, denoted by $h(t)$, is used extensively in multiprocessor scheduling. It is the maximum amount of task executions and completion that can be released in a time interval $[0,t)$ [Davis and Burns, 2011].

$$h(t) = \sum_{i=1}^{n} \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i. \tag{3.1}$$

Where the term processor load corresponds to the maximum of $h(t)$ separated by the portion of the time interval [Davis and Burns, 2011].

$$load(\tau) = \max_{\forall t} \left(\frac{h(t)}{t}\right) \tag{3.2}$$

A simple necessary condition for taskset feasibility can be found from the processor load Baruah and Fisher [2005]:

$$load(\tau) \leq m, \tag{3.3}$$

Where, the number of processors is m.

3.3 Utilization Bound

The utilization bound U_a of a real-time scheduling algorithm A is described as the smallest value of the entire utilization U of the task set τ which is only just schedulable according to the scheduling algorithm A, beyond which meeting deadlines is not guaranteed by all the jobs released by the tasks in τ [Davis and Burns, 2011].

3.4 Resource Augmentation or Speedup Factor

This is another way of performance comparison between any scheduling algorithm A and an optimal one. For A which is determined by the minimum factor by which the speed of all m processors might need to be raised to schedule all the feasible tasksets with the algorithm A

[Davis and Burns, 2011]. They also consider that using the scheduling algorithm A, the taskset τ is just schedulable on a system of m processors with individual speed $f(\tau)$. Then the speedup factor f_A is [Davis and Burns, 2011]:

$$f_A = \max_{\forall m, \forall \tau} (f(\tau)). \tag{3.4}$$

Therefore, $f_A \geq 1$ indicates more efficient algorithm and $f_A = 1$ an optimal algorithm.

4. Classification of Scheduling Algorithms for Multi-Core Systems

As we often picture a tree or graph by the word taxonomy, in Figure 4.1 we tried to summarize an overview of the categories of real-time scheduling techniques given in a technical report by [Mohammadi and Akl, 2005].

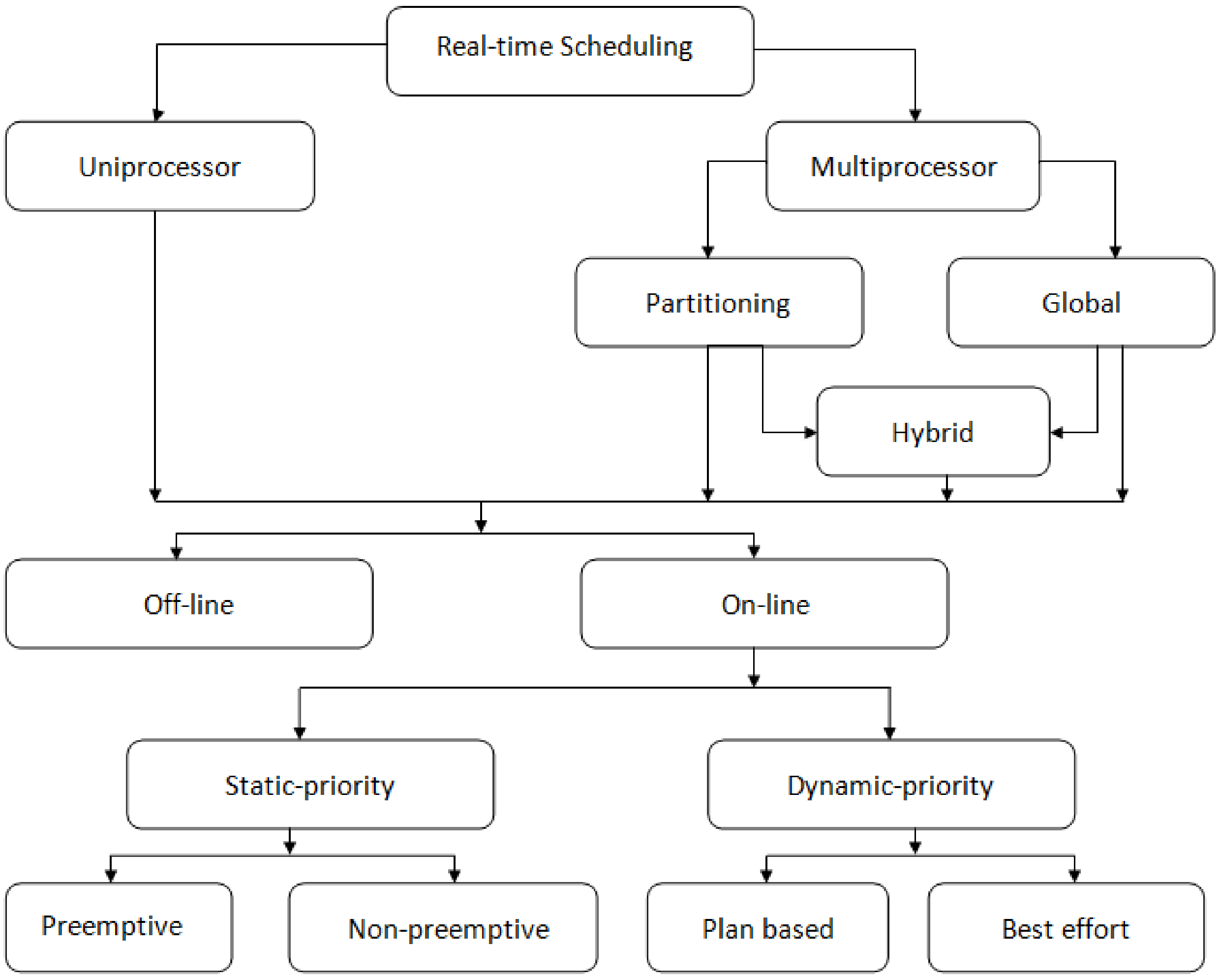


Figure 4.1: Categories of real-time scheduling algorithms [Mohammadi and Akl, 2005]

As whole Figure 4.1 also includes uni-processor along with multi-processor scheduling techniques, one can see the vastness of real-time scheduling research area. Our actual interest in this thesis is the branch dealing with multiprocessor real-time scheduling algorithms.

Real-time scheduling theorists have individualized at least three different types of multi-core architectures, described in section 2.3 dealing with the development of scheduling techniques. While deploying scheduling techniques along with the task allocation feasibility assessment, change of task priority is also considered.

Thus, multi-core scheduling tries to solve two problems [Davis and Burns, 2011]:

1. The *task allocation problem*: Solves on which processing core a task will be assigned and executed. Allocation problem is subdivided into the following categories:
   - *No migration*: Each of the tasks is assigned to on a fixed processor and no migration is allowed.
   - *Task level migration*: In this case, jobs of a task may take place on different processors; nevertheless, a single job only executes on a single one.
   - *Job level migration*: Here migration and execution of a single job is allowed on different processors, but, still, a job is not permitted to execute in parallel.
2. The *priority problem*: Solves the order of the job's execution with respect to other jobs.

   Priority problem is subdivided into the following categories:
   - *Fixed task priority*: A particular and fixed priority is applied to all of the jobs of each task.
   - *Fixed job priority*: In this case, the jobs may be applied with different priorities; however, each identical job has an identical static priority.
   - *Dynamic priority*: Here, it is possible for a job to have different priorities at different points of time.

Taking into consideration the permission to migrate, multi-core scheduling algorithms fall into two general categories:

- *Global Scheduling Algorithms*: In global scheduling algorithms, all the ready tasks are queued in one queue which is shared among all available processors, where the one single queue is referred to as global queue. In a system with m processors, at every time point m highest priority tasks from the global queue are selected for execution on the m processors employing preemption and migration if necessary. For instance, in the global version of EDF refereed to as G-EDF, the m active jobs with the earliest deadlines are executed on m processors of the underlying platform at any time t [Mohammadi and Akl, 2005][Ramamritham et al., 1990].

- *Partitioned Scheduling Algorithms*: In partitioned scheduling algorithms, all the tasks are grouped or partitioned as a set so that all the tasks in a set are assigned to the same processor. However, tasks in the partitioned set are not allowed to migrate to another processor that allows many uni-core scheduling algorithms to solve multi-core scheduling problems. For example, in partitioned version of EDF, the EDF algorithm is executed on each processor independently [Mohammadi and Akl, 2005][Ramamritham et al., 1990].

There is another category of multi-core scheduling algorithms, which is between partitioned and global scheduling policies, known as *Hybrid scheduling algorithms.* Among those the followings are worth mentioning:

- *Semi-partitioned scheduling algorithms*: In semi-partitioned scheduling algorithms the core idea is the improvement of processor utilization bound of partitioned scheduling algorithms by globally scheduling the tasks that cannot be assigned to only one processor due to the limitations of the bin packing heuristics.

- *Restricted migration scheduling algorithms*: In these types of algorithms, each job is assigned to only one processor while all the tasks can migrate between all the processors. Here, instead of task level partitioning the job level partitioning is applied.

- *Hierarchical scheduling algorithms*: In the hierarchical scheduling algorithms, depending on a particular algorithm, the tasks are partitioned into super tasks and component tasks. The super tasks are scheduled with multi-core scheduling algorithms while the component tasks of each server are scheduled using uni-core scheduling algorithms[Ramamritham et al., 1990].

- *Preemptive*: At any time, tasks are allowed for preemption by another task with higher priority.

- *Nonpreemptive*: When a task is already executing, can not be preempted by other task, even with higher priority one.

- *Cooperative*: Preemption is only possible within execution, at defined scheduling points.

5. Related Work on Real-time Scheduling Techniques

5.1 Partitioned Approach

In partition scheduling, the scheduler assigns tasks to available processors via partitioning. Thus, when a task is assigned to a processor, it is always scheduled particularly on that processor. As a result, the whole multi-core system turns into a set of uni-core system; where uni-core scheduling algorithm executes on each core, to execute assigned task to the processing core. For instance, a uni-core scheduling algorithm, earliest deadline first is used in multi-core scheduling algorithm partitioned EDF (P-EDF).

Advantages of partitioned scheduling compared to global scheduling are the following:

- In the case of overrun of a task's worst-execution time budget, it can only affect other tasks on the same processing core.
- No migration cost because of the execution of a task on a single processing core.
- In partitioned approach, a separate run-queue per processing core is used.

Main disadvantage of partitioned approach to multi-core scheduling is the analogous behavior of the allocation problem to the *bin packing problem*, which is recognized as NP-Hard [Garey and Johnson, 1979]. Besides, this approach is not *work-conserving*, i.e., a core can be in the idle state while there are still tasks to be scheduled on other cores that are possibly missing their deadlines.

From an implementation perspective, the influential advantage of partitioned approach on multi-core scheduling is: once the allocation of tasks to processing cores has been done, all the techniques of real-time scheduling and analysis for single-core systems can be applied [Davis and Burns, 2011]. Therefore, uni-core optimality results influence the research on partitioned multi-core scheduling.

Liu and Layland [1973] proved the optimality of RM priority assignment policy, for synchronous sporadic or periodic tasksets with implicit deadlines taking preemptive uni-core scheduling with fixed-task priorities under consideration. Under the same consideration Leung and Whitehead [1982] proved that DM priority assignment is optimal for tasksets consists of constrained deadlines. On the other hand, taking preemptive uni-core fixed-job priorities into view, Dertouzos [1974] showed that EDF (Earliest deadline first) is the optimal scheduling policy for sporadic tasksets where tasksets are independent of deadline constraints.

5.1.1 Tasksets consist of Implicit Deadlines

The relevant research during 1990s was directed to determining the utilization bound as a function of U_{max}, because of difficulties of allocating large utilization tasks by partitioned approach. Earlier and during that time, research on partitioned multi-core scheduling took place by Dhall and Liu [1978],Oh et al. [1993], Oh and Son [1995] and Burchard et al. [1995] using EDF or RM priority assignment on each processing core. They also combined the following bin packing heuristics:

- First Fit (FF),

- Best Fit (BF),
- Next Fit (NF),
- Worst Fit (WF)

The maximum worst-case utilization bound of any partitioned algorithm for tasksets having implicit deadlines is defined as [Andersson et al., 2001]:

$$U_{OPT} = (m+1)/2. \tag{5.1}$$

Utilization bounds for RMST algorithm were presented by Burchard et al. [1995]. This algorithm is applicable for tasks with utilization $< 1/2$ and tries to assign tasks on the same processor, for tasks having harmonics close to each other.

$$U_{RMST} = (m-2)(1-u_{max}) + 1 - \ln 2. \tag{5.2}$$

Utilization bounds for RMGT algorithm were also provided by Burchard et al. [1995]. This algorithm divides the tasks into two groups calculating whether their utilizations are above or below 1/3.

$$U_{RMGT} = \frac{1}{2}\left(m - \frac{5}{2}\ln 2 + \frac{1}{3}\right) \approx 0.5(m-1.42). \tag{5.3}$$

Utilization bounds of the RM-FFDU algorithm were presented by Oh and Bakker [1998]:

$$U_{RM-FFDU} = m(2^{1/2}-1) \approx 0.41m. \tag{5.4}$$

Utilization bounds of partitioned algorithms with fixed-task priority were also presented by Oh and Bakker [1998]:

$$U_{OPT(FTP)} < (m+1)/(1+2^{1/(m+1)}). \tag{5.5}$$

Andersson and Jonsson [2003] presented the utilization bound of the RBOUND-MP-NFR algorithm which is:

$$U_{RBOUND-MP-NFR} = m/2. \tag{5.6}$$

Any algorithm with reasonable allocation the lowest utilization and highest utilization bounds were described by Lopez et al. [2000], employing EDF:

$$L_{RA} = m - (m-1)u_{max}. \tag{5.7}$$

$$H_{RA} = \frac{(\lfloor 1/u_{max} \rfloor m + 1)}{(\lfloor 1/u_{max} \rfloor + 1)}. \tag{5.8}$$

(Assuming $n > m/(\lfloor 1/u_{max} \rfloor)$.)

Here, reasonable allocation represents the one that only fails to allocate a task when there is no processing core on which the task will fit. It is observable that $u_{max} = 1$, the highest limit given by 5.8, becomes the same as 5.1. Thus, they are also "optimal" in a limited sense, and utilization bounds of EDF-FF and EDF-BF are as large as any other optimal partitioning algorithm. Moreover, EDF-FF and RMST results reasonably high utilization bounds [Davis and Burns, 2011].

5.1.2 Tasksets consist of Constrained and Arbitrary Deadlines

Based on task ordering in increasing order EDF-FFID algorithm was developed by Baruah and Fisher [2005,2006a,2007]. Their schedulability was defined by a linear upper bound for processor demand bound function by conducting a sufficient test. They also presented that EDF-FFID is application for scheduling any sporadic taskset with constrained deadlines, that provides:

$$m \geq \left(\frac{2load(\tau) - \delta_{max}}{1 - \delta_{max}} \right). \tag{5.9}$$

And, for arbitrary deadlines:

$$m \geq \frac{load(\tau) - \delta_{max}}{1 - \delta_{max}} + \frac{u_{sum} - u_{max}}{1 - u_{max}}. \tag{5.10}$$

5.2 Global Approach

In this section, we will point out the main features and fundamental research outcomes in global multi-core scheduling techniques. As we have previously discussed, in multi-core global scheduling tasks are allowed to migrate from one core to another core if necessary.

Compared to partitioned multi-core scheduling, global scheduling has advantages stated below:

- Lower number of context switching or preemption, as the scheduler only has to preempt a task when there are no idle processors left [Andersson and Jonsson, 2000a].
- When the actual execution time of a task is less than its worst-case execution time, spare capacity is created which can be utilized by all other tasks.
- This scheduling algorithm is more suitable for open systems, where load balancing/ task allocation is not necessary with the change of taskset.

Disadvantages: The main disadvantage of global scheduling is that it uses a global single queue for ready tasks. As the queue length is long, queue access time gets longer accordingly.

In a seminal work by Dhall and Liu [1978], for global scheduling of periodic tasksets with implicit deadlines on m processors, they showed that the utilization bound is $1 + \epsilon$ for global EDF, where ϵ is considered arbitrarily small. As a result of this Dhall effect, throughout almost one decade in 1980s and 1990s, a general view of inferiority of global scheduling compared to partitioned scheduling was accepted, and thus, the majority of research was focused on partitioned approaches.

5.2.1 Global Scheduling with Fixed Job Priority

Tasksets consist of Implicit Deadlines: Utilization bounds for periodic tasksets were considered by Andersson et al. [2001]. They presented the maximum utilization bound for any global fixed job priority algorithm [Andersson et al., 2001], which is:

$$U_{OPT} = (m+1)/2 \tag{5.11}$$

In EDF-US[ς] algorithm proposed by Srinivasan and Baruah [2002] tasks with utilization greater than the threshold ς have the highest priority, where ties are broken arbitrarily. Resultant utilization bound is independent of u_{max}. With the threshold value $m(2m-1)$ gives the following result Davis and Burns [2011]:

$$U_{EDF-US[m/(2m-1)]} = m^2/(2m-1). \tag{5.12}$$

Another derivation of utilization bound for global EDF with periodic tasksets, was given by Goossens et al. [2003]:

$$U_{EDF} = m/(m-1)u_{max}. \tag{5.13}$$

Baker [2005] presented the following maximum possible utilization bound for global EDF algorithms with the threshold value to 1/2 in EDF-US[ς] :

$$U_{EDF-US[1/2]} = (m+1)/2. \tag{5.14}$$

A variant of EDF(κ) named EDF(κ_{min}) was proposed by Baker [2005] where κ_{min} is the minimum value of κ. Also the utilization bound of EDF(κ_{min}) was presented Baker [2005]:

$$U_{EDF[\kappa_{min}]} = (m+1)/2. \tag{5.15}$$

Tasksets consist of Constrained and Arbitrary Deadlines: The utilization bound given by Goossens et al. [2003] for sporadic tasksets with constrained deadlines was extended by Bertogna et al. [2005] and extended for the arbitrary deadline case by Baker and Baruah [2007] providing the sufficient schedulability test on the basis of task density stated below:

$$\delta_{sum} \leq m - (m-1)\delta_{max} \tag{5.16}$$

The utilization separation approach of EDF-US was adopted by Bertogna [2007] to develop EDF-US[ς] algorithm for sporadic tasksets with constrained and arbitrary deadlines. Here, a task with greater density than the threshold ς gets the highest priority. Schedulability according to EDF-DS[1/2] for sporadic tasksets was provided by Bertogna [2007] :

$$\delta_{sum} \leq (m+1)/2 \tag{5.17}$$

5.2.2 Global Fixed Task Priority Scheduling

Tasksets consist of Implicit Deadlines: As we have discussed before, Dhall and Liu [1978] considered global scheduling for periodic tasksets with implicit deadlines on m processing cores. They presented the utilization bound for global RM scheduling of $1+\epsilon$, where ϵ is arbitrarily small [Davis and Burns, 2011]. Which is known as the Dhall effect.

TkC priority assignment policy was developed by Andersson and Jonsson [2000a] to avoid the Dhall effect. Here, priority is assigned based on the period of a task T_i minus κ times its WECT C_i, given below [Davis and Burns, 2011]:

$$\kappa = \frac{m - 1 + \sqrt{5m^2 - 6m + 1}}{2m}. \tag{5.18}$$

where κ is computed based on the number of processing cores, which is a real value. Andersson and Jonsson [2000a] have done an empirical investigation to show the effectiveness of TkC as priority assignment policy for periodic tasksets consist of implicit deadlines.

For any periodic tasksets having implicit deadlines, Andersson et al. [2001] showed that those are employing global RM scheduling and also provided the utilization bound

$$u_{max} \leq m/(3m - 2) \; and \; u_{sum} \leq m^2/(3m - 1). \tag{5.19}$$

Baruah and Goossens [2003] also presented this result, but in a another form

$$u_{max} \leq 1/3 \; and \; u_{sum} \leq m/3. \tag{5.20}$$

RM-US[ς] was proposed by Andersson et al. [2001], where tasks with utilization greater than the threshold ς get the highest priority, and for the rest the priority is assigned in RM order. Utilization bound for RM-US[$m/(3m-2)$] was also shown by Andersson et al. [2001], which is

$$U_{RM-US[3/(3m-2)]} = m^2/(3m - 1). \tag{5.21}$$

For periodic tasksets with implicit deadlines, Andersson and Jonsson [2003] presented the maximum utilization bound and the WCET is

$$U_{OPT} \leq (\sqrt{2} - 1)m \approx 0.41m. \tag{5.22}$$

And the priorities are defined as a scale-invariant function of task periods [Davis and Burns, 2011]. This bound was tightened by Bertogna et al. [2006] for global RM scheduling to

$$u_{max} \leq \frac{m}{2}(1 - u_{max}) + u_{max}. \tag{5.23}$$

Tasksets with constrained and arbitrary deadlines: A response time upper bound R_k^{ub}, was presented by Andersson and Jonsson [2000b] for sporadic tasksets with constrained deadline scheduling using fixed priority. Though it was pessimistic, it was simple.

$$R_k^{ub} \leftarrow C_k + \frac{1}{5} \sum_{i<k} \left(\left\lceil \frac{R_k^{ub}}{T_i} \right\rceil C_i + C_i \right). \tag{5.24}$$

which effectively assumes that the time for executing carried-in and carried-out jobs in an interval is equal to the entire WCET of the task.

For global DM scheduling of sporadic tasksets with constrained deadlines the following density bound was proven by Bertogna et al. [2006]

$$\delta_{sum} \leq \frac{m}{2}(1 - \delta_{max}) + \delta_{max}. \tag{5.25}$$

Here, δ_{max} represents the maximum density of any task in the taskset and δ_{sum} represents taskset density (sum of task densities).

For DM-DS[1/3] the following sufficient test was proven by Bertogna et al. [2006] :

$$\delta_{sum} \leq \frac{m+1}{3}. \tag{5.26}$$

Assuming *intratask* parallelism, a sufficient test for global DM scheduling of sporadic tasksets with arbitrary deadlines was derived by Baruah and Fisher [2006b]. For each task τ_i this sufficient test is as follows:

$$load(\tau, k) \leq \frac{1}{1 + 2(max_{j \in hp(k)}(D_j/D_k))}(m - (m-1)(C_k/D_k)). \tag{5.27}$$

Here, $load(\tau, k)$ represents the processor load, which is higher than or equal to k because of all tasks' priority.

An alternative sufficient test was derived by Baruah [2007] for global DM scheduling of sporadic tasksets with constrained deadlines which was based on the similar approach as Baruah and Fisher [2006b]. For each task τ_i this sufficient test is as follows:

$$load(\tau, k) \leq \frac{1}{2}(m - (m-1)C_k/D_k - C_\Sigma(k)/D_k). \tag{5.28}$$

Here, $C_\Sigma(k)$ is the total of the m largest worst-case execution times of the tasks with the priority of k or higher.

5.2.3 Global Dynamic Priority Scheduling

There are a number of global dynamic priority scheduling algorithms known as optimal for periodic tasksets with implicit deadlines, such as Pfair and variants of Pfair PD, PD^2, ERfair, SA and LLREF. Though, Fisher [2007] proved the non-existence of optimal online algorithm for preemptive scheduling of sporadic tasksets on multiprocessor. Research shows that global dynamic priority algorithms monopolize over all the other classes of algorithms. Nonetheless, their practical implementation can be ambiguous due to significantly high overhead caused by frequent migration and preemption [Davis and Burns, 2011].

The Proportionate and Early-Release Fairness Algorithm:

The idea of fairness was first introduced by Baruah et al. [1996]. As the name implies, the fundamental idea is to distribute the total computational capacity of the platform among the tasks. Therefore, at any time point t, each task τ_i is executed on the processing platform for a time proportional to its utilization which goes in the direction of *fluid scheduling* defined below:

Theorem 5.2.1 (***Fluid Scheduling*** by Baruah et al. [1996]). *A schedule is said to be fluid if and only if at any time $t \geq 0$, the active job (if any) of every task τ_i arrived at time $a_i(t)$ has been executed for **exactly** $U_i \times (t - a_i(t))$ time units.*

As the systems in the real world are based on discrete time or are clock pulse based, the tasks are always measured in an integer number of system time units. Therefore, the task execution might deviate from fluid schedule throughout the system lifespan. The measurement of this deviation from the fluid schedule is denoted as *allocation error or lag* of a task, which is defined by Baruah et al. [1996] as follows:

Definition 1 (***Allocation Error (lag)*** by Baruah et al. [1996])**.** *The lag of a task τ_i at time t is the difference between the amount of work $exec_i(a_i(t), t)$ executed by the active job of τ_i until time t in the actual schedule, and the amount of work that it would have executed in the fluid schedule by the same instant t. That is,*

$$lag_i \overset{def}{=} U_i \times (t - a_i(t)) - exec_i(a_i(t), t)$$

with $a_i(t)$ being the arrival time of the active job of t.

In Fair scheduling, this lag is constrained to bound the deviation from fluid scheduling. In *Proportionate Fair* scheduler, this allocation error of those tasks is always bounded by smaller than one system time units [Baruah et al., 1996].

Definition 2 (***Proportionate Fair schedule*** by Baruah et al. [1996])**.** *A schedule is said to be proportionate fair (or PFair) if and only if,*

$$\forall \tau_i \in \tau, \forall t \geq 0 : |lag_i(t)| < 1$$

On the contrary, for *Early-Release Fair (ERFair)* scheduler, tasks are allowed to be ahead more than one time unit, but never be late by more than one time unit. Definition by Anderson and Srinivasan [2001] is the following:

Definition 3 (***Early-Release fair schedule*** by Anderson and Srinivasan [2001])**.** *A schedule is said to be Early-Release fair (or PFair) if and only if,*

$$\forall \tau_i \in \tau, \forall t \geq 0 : lag_i(t) < 1$$

In the rest of this report, the following terms will be used to determine the state of a task at time t.

- Task is *behind* at time t: if actual execution time for the task is less than in the corresponding fluid schedule until time t. Another representation is: $lag_i(t) > 0$.
- Task is *punctual* at time t: if actual execution time for the task is exactly the same as in the corresponding fluid schedule until time t. Another representation is: $lag_i(t) = 0$.
- Task is *ahead* at time t: if actual execution time for the task is more than in the corresponding fluid schedule until time t. Another representation is: $lag_i(t) < 0$.

The PF Scheduling algorithm:

The PF algorithm is the first optimal schedule generation algorithm presented by Baruah et al. [1996]. It is designed for periodic tasksets with implicit deadlines. In the early development stage, the scheduling decision taken by PF was based on *characteristic string.* Since then this procedure was refined by introducing the notion of *pseudo-deadline* by Anderson and Srinivasan [1999], we will continue with the refined version.

In proportional fair scheduling, a task τ_i is divided into an infinite sequence of *subtasks* which are basically the time slots. Each of the subtasks has an execution time of one unit and $j_{i,j}$ represents the j^{th} subtask of a task τ_i where $j < 1$ and a job $J_{i,q}$ consists of C_i consecutive subtasks $\tau_{i,j}$.

Each subtask $\tau_{i,j}$ of a job $J_{i,q}$ must execute in an associated *window* to keep the lag of a task τ_i smaller than 1 and greater than -1. The span of this window is from *pseudo-release* $pr(\tau_{i,j})$ to *pseudo-deadline* $pd(\tau_{i,j})$. For periodic tasks released at time 0 Anderson and Srinivasan [1999] defined $pr(\tau_{i,j})$ and $pd(\tau_{i,j})$ as follows:

$$pr(\tau_{i,j}) \overset{def}{=} \left\lfloor \frac{j-1}{U_i} \right\rfloor$$

and

$$pd(\tau_{i,j}) \overset{def}{=} \left\lceil \frac{j}{U_i} \right\rceil .$$

Srinivasan and Anderson [2002] presented a more simplified version of the definition of a pseudo-deadline of a subtask $\tau_{i,j}$ as follows:

$$pd(\tau_{i,j}) = pr(\tau_{i,j}) + \left\lceil \frac{j}{U_i} \right\rceil - \left\lfloor \frac{j-1}{U_i} \right\rfloor .$$

here the p^{th} subtask has to execute in a job $J_{i,q}$ given by following equation:

$$pd(\tau_{i,j}) \overset{def}{=} a_{i,q} + \left\lceil \frac{p}{U_i} \right\rceil . \tag{5.29}$$

where $a_{i,q}$ represents the arrival time of job $J_{i,q}$. As there are C_i consecutive subtasks in any job $J_{i,q}$, the equation states that $q = \left\lceil \frac{j}{C_i} \right\rceil$ and $p = j - (q-1) \times C_i$.

For each time t, PF determines which subtasks are eligible for scheduling. At any time t, a subtask $\tau_{i,j}$ of task τ is eligible under PF scheduling algorithm if it respects the following definition:

Theorem 5.2.2. *A subtask $\tau_{i,j}$ of task τ is eligible to be scheduled at time t if, the subtask $\tau_{i,j-1}$ has already been executed prior to t and $pr(\tau_{i,j}) \leq pd(\tau_{i,j})$, i.e., t lies within the execution window of $\tau_{i,j}$.*

An active subtask with earlier pseudo-deadlines gets the highest priority in PF algorithm. If any two subtasks have the same pseudo-deadline, an additional successor bit is used to handle this situation. A successor bit of a subtask $\tau_{i,j}$ is denoted by $b(\tau_{i,j})$, which equals to 1 if and only if $\tau_{i,j}$'s window overlaps $\tau_{i,j+1}$'s window, otherwise $b(\tau_{i,j})$ is zero. The following Equation was proven based on the definitions of pseudo-deadline and pseudo-release:

$$b(\tau_{i,j}) \overset{def}{=} \left\lceil \frac{j}{U_i} \right\rceil - \left\lfloor \frac{j}{U_i} \right\rfloor = \left\lceil \frac{p}{U_i} \right\rceil - \left\lfloor \frac{p}{U_i} \right\rfloor . \tag{5.30}$$

Prioritization Rule 5.2.1. *Hence, with PF, a subtask $\tau_{i,j}$ can achieve higher priority than a subtask $\tau_{k,l}$ (where $\tau_{i,j} \succ \tau_{k,l}$) iff:*

1. $pd(\tau_{i,j}) < pd(\tau_{k,l})$ *or*
2. $pd(\tau_{i,j}) = pd(\tau_{k,l}) \wedge b(\tau_{i,j}) > b(\tau_{k,l})$ *or*

3. $pd(\tau_{i,j}) = pd(\tau_{k,l}) \wedge b(\tau_{i,j}) = b(\tau_{k,l}) = 1 \wedge \tau_{i,j+1} \succ \tau_{k,l+1}$

Noticeable is that the third recursion of the prioritization rules 5.2.1 always ends. Also, it was proven by Anderson and Srinivasan [1999] that $b(\tau_{i,j}) = 0$ at least at the deadline of a job.

The PD^2 scheduling algorithm:

Due to the third recursive prioritization rule 5.2.1, the algorithm PF performs very poorly. This pitfall was detected by Baruah et al. [1995], and as the solution they proposed a new PFair scheduling algorithm PD by replacing the third tie breaking rule by additional three tie breaking rules.

Anderson and Srinivasan [1999] improved the PD scheduling algorithm. In their proposal, they proved the three additional tie breaking rules can be replaced by one value which is called *group deadline.*

An extension of Pfair and a variant of PD were extended by Anderson and Srinivasan [2000] and named EPDF (Earliest Pseudo Deadline First). They showed the optimality of this algorithm for sporadic tasksets with implicit deadlines assigned to two processors. But this algorithm is optimal only for two processors and not more than that.

Yet another variant of PFair named PD^2 was also proposed by Andersson et al. [2001], here efficiency of Pfair was improved by separating tasks into groups of *heavy*$(u_i < 0.5)$ and *light*.

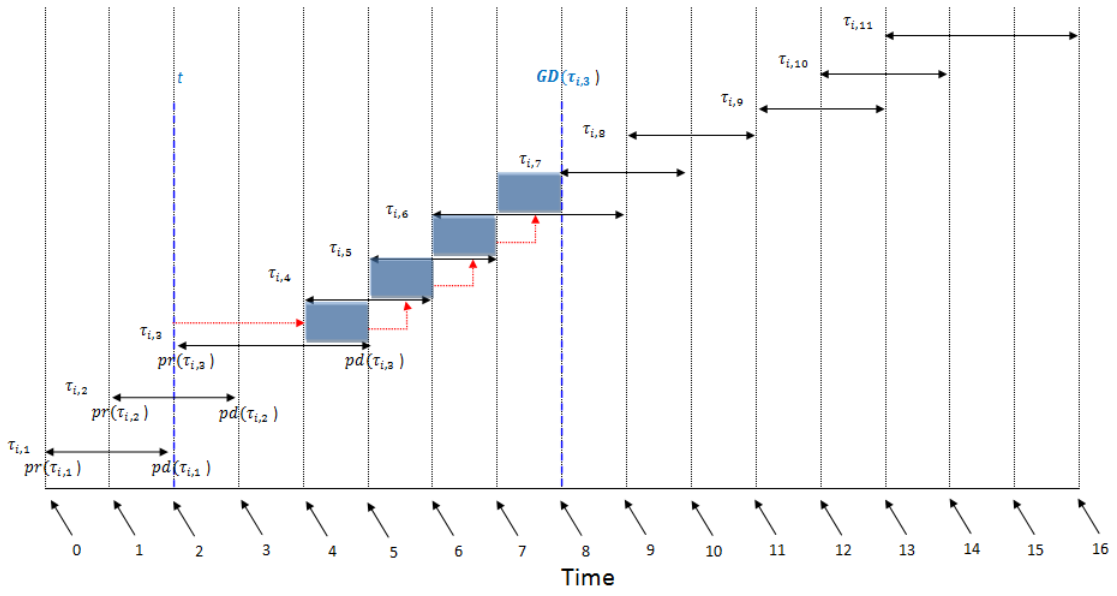


Figure 5.1: A periodic task τ_i containing 11 subtasks with $U_i = \frac{8}{11}$ representing the group deadline of subtask $\tau_{i,j}$ [Nelissen et al., 2014]

To make further details more understandable the definition of a group deadline is given below:

Definition 4 (***Group Deadline***)**.** *A group deadline of any subtask $\tau_{i,j}$ belonging to a task τ_i such that $U_i < 0.5$ (i.e., a light task), is $GD(\tau_{i,j}) \overset{def}{=}$).*
The group deadline $GD(\tau_{i,j})$ of a subtask $\tau_{i,j}$ belonging to a heavy task τ_i(i.e., $U_i \geq 0.5$), is the earliest time t, where $t \geq pd(\tau_{i,j})$, such that either $(t = pd(\tau_{i,k}) \wedge b(\tau_{i,k}) = 0)$ or $(t = pd(\tau_{i,k}) + 1 \wedge (pd(\tau_{i,k+1}) - pd(\tau_{i,k})) \geq 2)$ for some subtask $\tau_{i,k}$ of task τ_i such that $k \geq j$.

Therefore, in case of a heavy task, $GD(\tau_{i,j})$ is the earliest time instant greater than or equal to $pd(\tau_{i,j})$ that finishes a succession of pseudo-deadlines separated by only one time unit or were the task τ_i becomes punctual.

If a subtask $\tau_{i,j}$ is scheduled in the last slot of its window, all the next subtasks of the sequence are forced to be scheduled in the last slot of their window. For an instance, subtask $\tau_{i,3}$ to $\tau_{i,5}$ in Figure 5.1. If subtask $\tau_{i,3}$ is scheduled at the fourth time unit then subtask $\tau_{i,4}$ and $\tau_{i,5}$ have to be scheduled at the 5^{th} and 6^{th} time units respectively. By the definition 4, the group deadline of $\tau_{i,3}$ in Figure 5.1 is thereby $GD(\tau_{i,3}) = 8$.

Employing the group deadline concept the prioritization rules of PD^2 are as follows:

Prioritization Rule 5.2.2. *A subtask $\tau_{i,j}$ has a higher priority than a subtask $\tau_{k,l}$ under PD^2 (denoted $\tau_{i,j} \succ \tau_{k,l}$) iff:*

1. $pd(\tau_{i,j}) < pd(\tau_{k,l})$ *or*
2. $pd(\tau_{i,j}) = pd(\tau_{k,l}) \wedge b(\tau_{i,j}) > b(\tau_{k,l})$ *or*
3. $pd(\tau_{i,j}) = pd(\tau_{k,l}) \wedge b(\tau_{i,j}) = b(\tau_{k,l}) = 1 \wedge GD(\tau_{i,j}) > GD(\tau_{k,l})$

Again, if neither $\tau_{i,j} \succ \tau_{k,l}$ nor $\tau_{k,l} \succ \tau_{i,j}$ holds, then ties can be broken arbitrarily by the scheduler.

While implementing Pfair on symmetric multiprocessor, Holman and Anderson [2005] noticed that, the synchronized rescheduling of all processing cores each time quanta produced significant bus contention because of data reloading into cache. Addressing this problem, Holman and Anderson [2005] proposed another variant of Pfair with staggered time quanta, which reduces the bus contention and also the schedulability.

The ERfair scheduling algorithm:

In ERFair, a variant of Pfair scheduling class proposed by Anderson and Srinivasan [2000], the constraint that *lag* must be bigger than -1 was omitted. Also,

$$lag_i(t) < 1, \forall t \tag{5.31}$$

instead of

$$|lag_i(t)| < 1, \forall t.$$

Through this, ERFair allows quanta of a job to execute before the execution completion information is provided by the previous Pfair scheduling window. This turns ERFair into a work-conserving algorithm, while Pfair is not.

The BF scheduling algorithm:

A concept of *Boundary Fair* (BF) was introduced by Zhu et al. [2003]. Zhu et al. [2003] found that tasks with implicit deadlines only miss their deadlines at times that are their period boundaries. The Boundary Fair algorithm is analogous to Pfair except it only takes the scheduling decisions at the period boundaries. This concept makes BF less fair than Pfair.

If we denote the boundaries found in the scheudle by $B \overset{def}{=} b_0, b_1, b_2, ..$ with $b_k < b_{k+1}$ and $b_0 = 0$, then a boundary fair schedule is defined as follows:

Definition 5 (***B****oundary Fair Schedule* by Zhu et al. [2003]). *A schedule is said to be boundary fair if and only if, at any boundary $b_k \in B$, it holds for every $\tau_i \in \tau$ that $lag_i(b_k) < 1$.*

Here, b_k represents the k^{th} time-instance at which point the scheduler is invoked. And boundaries at k^{th} time-slice are b_k and b_{k+1} (also denoted by TS^k).

As a boundary fair scheduler is invoked at every boundary b_k and computes the scheduling decisions for the total time slice from b_k to the next boundary b_{k+1}, the invocation of boundary fair algorithm can be splited into the following steps:

1. Determining the next boundary b_{k+1}, and computation of *mandatory* execution time units and *optional* execution times. [Zhu et al., 2003] showed that the mandatory time unit denoted $mand_i(b_k, b_{k+1})$ can be computed using the following equation:

$$mand_i(b_k, b_{k+1}) \overset{def}{=} \max\{0, \lfloor lag_i(b_k + (b_{k+1} - b_k) \times U_i) \rfloor\} \tag{5.32}$$

2. If all the available time units have not been allotted in step 1, then distribute *remaining* time units as optional time units. For an m processor system, the total number of available time units in the time interval $[b_k, b_{k+1})$ is given by [Zhu et al., 2003]:

$$m \times (b_{k+1} - b_k).$$

   Thus, the number of remaining time units $RU(b_k, b_{k+1})$ is as follows :

$$RU(b_k, b_{k+1}) = m \times (b_{k+1} - b_k) - \sum_{\tau_i \in \tau} mand_i(b_k, b_{k+1}) \tag{5.33}$$

3. Generation of a schedule, which avoids intra-job parallelism within the interval $[b_k, b_{k+1})$, is based on the number of mandatory and optional time unit allotted to each task [Zhu et al., 2003].

It is worth noting that the algorithms PFair and ERFair are also boundary fair, but the opposite is not necessarily true.

In Figure 5.2 task τ_0 requires 4 preemptions using BFair instead of 11 using PFair. Which shows that it is possible to deduct the number of preemptions and migrations of corresponding tasks by only regrouping all the time units of the same task within a time slice TS^k,

Using empirical evaluation Zhu et al. [2003] showed that BF is also an optimal algorithm for periodic tasksets with implicit deadlines. And also the number of scheduling points is 25-50% of the number for a PD algorithm.

Another boundary fair scheduling algorithm PL was proposed by Kim and Cho [2011]. It was designed for a set of synchronous periodic tasks. PL stands for pseudo laxity. The PL algorithm uses prioritization rules of PD^2 to distribute the optimal time units and rather than utilizing McNaughton's wrap around algorithm, utilizes LLF (Least Laxity First) algorithm to generate the schedule within each time slices. The PL algorithm was also proven optimal for synchronous periodic tasks with implicit deadlines by Kim and Cho [2011].

The BF^2 scheduling algorithm:

The scheduling algorithm BF^2 is proposed by Nelissen et al. [2014]. It is based on BFair scheduling algorithm. But, there is a significant difference between BF algorithm proposed by Zhu et al.

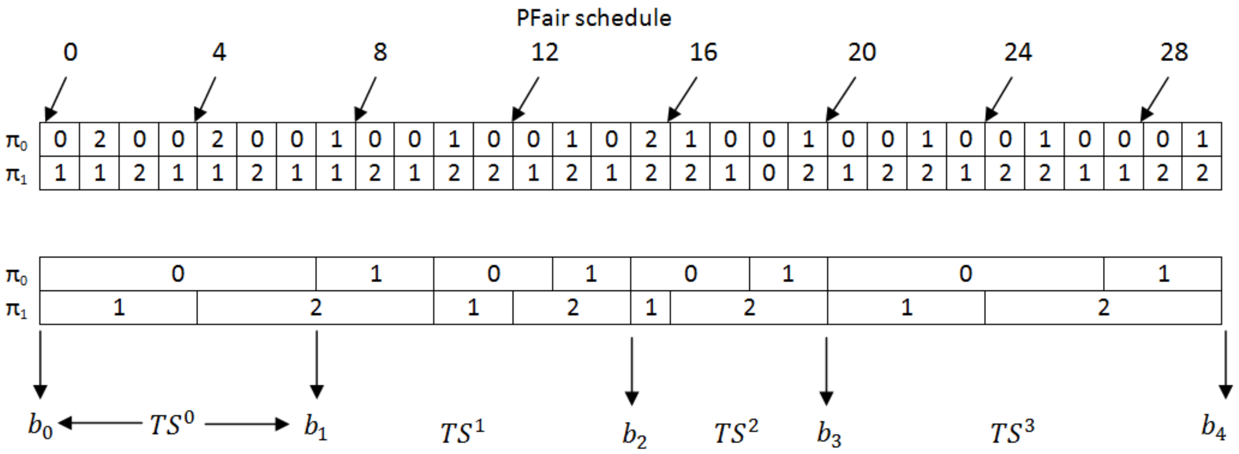

PFair schedule
0
4
8
12
16
20
24
28
π0 0 2 0 0 2 0 0 1 0 0 1 0 0 1 0 2 1 0 0 1 0 0 1 0 0 1 0 0 0 1
π1 1 1 2 1 1 2 1 1 2 1 2 2 1 2 1 2 2 1 0 2 1 2 2 1 2 2 1 1 2 2
π0 0 1 0 1 0 1 0 1
π1 1 2 1 2 1 2 1 2
b0
TS0
b1
TS1
b2
TS2
b3
TS3
b4

Figure 5.2: Pfair and BFair schedules of three tasks τ_0, τ_1 and τ_2 , where number of processors is $m = 2$. The periods are $T_0 = 15, T_1 = 10, T_2 = 30$, and the worst-case execution times are $C_0 = 10, C_1 = 7, C_2 = 19$, [Nelissen et al., 2014]

[2003] and BF^2. The algorithm BF is optimal only for periodic tasks while achieving full system utilization, but cannot handle sporadic tasks because of their inherent irregular and unpredictable job release pattern. By the same token, BF^2 is an optimal algorithm for sporadic tasks. Nelissen et al. [2014] also showed that their proposed algorithm BF^2 outperforms the state-of-the-art optimal scheduler PD^2 by benefiting from less scheduling overhead.

The PD^{2*} scheduling algorithm:

Previously we described that PD^2 defines the group deadline $GD(\tau_{i,j})$ as light tasks (i.e., tasks with $U_i < 0.5$) and heavy tasks (i.e., tasks with $U_i > 0.5$). In particular, the group deadline is defined in PD^{2*} not in the same way as it is in PD^2. In PD^{2*} for light tasks $GD(\tau_{i,j})$ is always 0 and for heavy tasks defined by the earliest time instant after or at the pseudo-deadline $pd(\tau_{i,j})$ following a succession of pseudo-deadlines separated by only one time unit [Nelissen et al., 2014]. Therefore, group deadlines for all the tasks are identical. This new algorithm PD^{2*} is proposed by Nelissen et al. [2014]. It is a slight variation of PD^2.

The *generalized group deadline* is denoted by $GD^*(\tau_{i,j})$ and defined as follows [Nelissen et al., 2014]: The generalized group deadline $GD^*(\tau_{i,j})$ of any subtask $\tau_{i,j}$ of a task τ_i, is the earliest time t, where $t \geq pd(\tau_{i,j})$, such that either $(t = pd(\tau_{i,k}) \wedge b_{\tau_{i,k}} = 0)$ or $(t = pd(\tau_{i,k}) + 1 \wedge (pd(\tau_{i,k+1}) - pd(\tau_{i,k})) \geq 2)$ for a subtask $\tau_{i,k}$ of τ_i such that $k \geq j$ [Nelissen et al., 2014].

And the prioritization rules are given by the same authors are as follows:

Prioritization Rule 5.2.3. *A subtask $\tau_{i,j}$ has a higher priority than a subtask $\tau_{k,l}$ under PD^{2*} iff:*

1. $pd(\tau_{i,j}) < pd(\tau_{k,l})$
2. $pd(\tau_{i,j}) = pd(\tau_{k,l}) \wedge b(\tau_{i,j}) > b(\tau_{k,l})$
3. $pd(\tau_{i,j}) = pd(\tau_{k,l}) \wedge b(\tau_{i,j}) = b(\tau_{k,l}) = 1 \wedge GD^*(\tau_{i,j}) > GD^*(\tau_{k,l})$

Though this proposed algorithm is a modified version of PD^2, the authors [Nelissen et al., 2014] proved the optimality of PD^{2*} for sporadic task sets as well as intra-sporadic and dynamic tasksets under a PFair or ERFair policy.

The LLREF scheduling algorithm:

LLREF is introduced by Cho et al. [2006]. It is based on fluid scheduling model employing L-T plane abstraction, it is also an optimal algorithm for periodic tasksets with implicit deadlines. In LLREF, scheduling timeline is separated into sections by normal scheduling events. At the beginning of each start section, selection of m task is done for execution based on largest local remaining execution time first (LLREF) [Davis and Burns, 2011]. The local remaining execution time decreases during task execution in the section.

The LLREF approach was extended by Funaoka et al. [2008] as a work-conserving algorithm by distributing the unused processing time among the tasks and combining the processing time after completion of a task earlier than expected. Funaoka et al. [2008] showed that with this approach the resultant preemption is significantly less than in the case of LLREF when taskset utilization is below 100%.

Another extension of LLREF was done by Funk [2010] and is known as LRE-TL. Funk [2010] noticed that a task selection is not necessary for the execution based on the largest local remaining execution time within each execution. Funk [2010] also represented that LRE-TL algorithm could be applicable to sporadic tasksets and also proved the optimality with utilization bound of 100% for sporadic tasksets with implicit deadlines [Davis and Burns, 2011].

The EDZL scheduling algorithm:

EDZL stands for Earliest Deadline until Zero Laxity. It was introduced by Lee [1994]. With this algorithm Lee [1994] showed that it dominates global EDF scheduling. Lee [1994] also presented that for two processing cores EDZL is suboptimal. Here the meaning of *suboptimal* is defined by the notion that EDZL can "schedule any feasible set of ready tasks" [Davis and Burns, 2011].

A variant of EDZL, introduced by Kato and Yamasaki [2008] is Earliest Deadline until Critical Laxity. This algorithm increases the job priority based on critical laxity at release time or completion time of a job. This technique reduces the minimum number of context switches to two per job which is achieved by slightly inferior schedulability compared to EDZL [Davis and Burns, 2011].

5.3 Summary

A performance measurement of partitioned, clustered, and global scheduling approaches was conducted by Brandenburg et al. [2008] using EDF and Pfair algorithms on LITMUS^{RT} test-bed on the Sun UltraSPRAC Niagra multi-core platform with 32 logical processors (four hardware threads running on each of eight CPUs). It was observed in their experiment that in the context of overhead, staggered Pfair performed much better that pure Pfair, in other words, pure Pfair performed very poorly. Another poor performance was found for Global EDF scheduling algorithm due to the overhead caused by manipulation of a lengthy global queue accessible to all processors. For hard real-time tasksets, Partitioned EDF works best except when the tasks had high individual utilization. In contrast to that, staggered Pfair was the best.

Another performance measurement experiment was conducted by Nelissen et al. [2014]. They explored fair scheduling approaches using pure Pfair, i.e., PD^2, partially Pfair BF^2 algorithms on Linux kernel version 2.6.34 on the multi-core platform of Lenovo ThinkStation having two Xeon E5405 Intel chips, where each chip has four cores of 2 GHz. Though, their machine was composed of eight identical cores, only six of them were used to run the real-time tasks and the rest of the CPUs were utilized for debugging and monitoring purposes. BF^2 was implemented using work conserving technique and PD^2 was implemented in its early-release version. It was the best performing implementation of PD^2 according to Anderson and Srinivasan [2000]. It is worth mentioning that the ERfair version of PD^2 is also work conserving. It was observed in their experiment that for BF^2 the number of preemptions, migrations, and scheduling points barely depends on the system time units when these values increase linearly with time resolution of the system running PD^2. Also confirming the same conclusion of Brandenburg et al. [2008], they showed that in their experiment BF^2 performed better than PD^2 in terms of overhead. In their report, they also mentioned that in terms of preemption and migration the results of PD^{2*} would be similar to PD^2. On the other hand, its scheduling overhead would be worse [Nelissen et al., 2014].

In this thesis work, performance measurement is done using PD^{2*} proposed by Nelissen et al. [2014] as a new slight variation of PD^2 already described in this chapter.

6. Related Work on Real-time Scheduling Test-beds

A drastic expansion of interest in real-time task scheduling for multi-core systems inevitably led to increased interest in multi-core architectures. In recent time, a significant amount of efforts has been devoted to this field by the academic community, focusing extensively on theoretical issues [Dellinger et al., 2012].

For analyzing temporal constraints of tasks in a real-time application/system a number of tools have been developed which include graphical editors and frameworks providing classical real-time scheduling/feasibility algorithms/tests, both in commercial and free distribution, e.g. STORM, Harmless (Hardware Architecture Modeling Language for Embedded Software Simulation) and YARTISS based on Java for comparing user-customized algorithms. The latter can also simulate tasksets considering energy consumption as a scheduling parameter in the same manner as Worst Case Execution Time (WCET). Another tool Cheddar is based on Ada, it includes a graphical editor made using GtkAda. Cheddar is compatible with Solaris, Linux, and win32 systems and GNAT/GtkAda supported platforms. It also supports AADL, STOOD, TOPCASED plug-ins.

Though research of algorithms is fundamental for the advances in multiprocessor real-time scheduling, in particular on platforms with a large number of cores, the effects of system overhead on scalability of existing schedulers are not explicit [Dellinger et al., 2012]. In other words, a simulation test-bed has to be real-time by itself, in the context of both logical and first of all physical functionality.

Also worth mentioning is the development of an optimal multiprocessor Real-Time Operating System (RTOS), while test-bed specific trade-offs of computational efficiency of the processing architecture should also be evaluated. That includes the architectures for general purpose microprocessors and powerful DSP kernels up to optimally tailored Application Specific Instruction set Processors (ASIPs).

In general, task scheduling is implemented in the operating system kernel. Therefore, a variety of kernel/kernel-level components have been developed as testing platforms of real-time scheduling on the Linux kernel as this approach has a few definite advantages such as the availability of libraries, software development tools and software packages for a wide variety of purposes.

For enabling implementation for small systems, POSIX1.3 is defined as RT-POSIX[Buttazzo, 2004] [iee, 2004][MARTE, 2011].

1. PSE51: Minimum Real-time System Profile
2. PSE52: Real-time Controller Profile
3. PSE53: Dedicated Real-time System Profile
4. PSE54: Multi-purpose Real-Time System Profile

The interest of using Linux as underlying operating system for embedded applications has also been actively pursued by the industry. This selection also provides the ability of hosting both real-time and best-effort tasks on the same CPU [Betz et al., 2009]. Moreover, some real-time extensions of Linux provided competitive results against commercial ones[Barbalace et al., 2008] [RTAI, 2014][Xenomai, 2014]. Therefore, we will avoid detailed discussions about commercial real-time systems with simulation features such as VxWORKS, OSE, ONX NEUTRIO.

6.1 Linux-kernel

Kernel is a program which constitutes the core of the operating system, acquiring the complete control over the operating system. A well known kernel for a Unix-like operating system is Linux kernel released under GNU GPL2 license. Linux kernels can be categorized considering their architecture.

- Monolithic kernel: A very large number of Linux variants use the monolithic kernel. Compared to others it is large, and each of the kernel layers is unified with the whole kernel program. In support of the current process, it runs in Kernel Mode [Bovet and Cesati, 2001]. In a monolithic kernel architecture, the entire operating system is in the kernel space and executing entirely in supervisor mode for all process scheduling subsystem memory management, IPC, virtual files, network. That directly opposes a microkernel architecture.

- Microkernel: Compared to the monolithic kernel, microkernel based operating systems require a tiny set of functions from the kernel which includes a few synchronization primitives, a scheduler, and an IPC mechanism [Bovet and Cesati, 2001]. A kernel module is such an object file which can be linked and unlinked to the kernel at runtime. To utilize the advantage of microkernels, Linux kernel offers modularized approach. Therefore, relevant data structures can be accessible by well defined interfaces [Bovet and Cesati, 2001].

- Hybrid kernel: A hybrid kernel architecture is built with flavour of both of microkernel and monolithic kernel architectures. The main goal behind this combined approach is to provide a kernel structure as microkernel in a monolithic kernel implementation methodology.

  Almost all the operating services as basic IPC, virtual memory, scheduling techniques including application IPC and device drivers are in kernel space, which avoids performance overhead for context switching and message passing between the kernel mode and user mode. From the architectural point of view, integration of application IPC and device drivers are different in hybrid kernel from the others.

6.2 Kernel Preemption

A kernel is considered to be a preemptive kernel if process switch is possible in Kernel Mode. In the case of non-preemptive process switching, the switch is called *planned process switch.* In contrast, in case of preemptive switching the switch is called *forced process switch.* The main feature of a preemptive kernel is the process switching in Kernel Mode. This can also be seen when in the middle of a kernel function a running process in Kernel Mode can be preempted and replaced by another one.[Bovet and Cesati, 2001]

Apart from supporting a variety of kernels, the SMP offers features which are significantly relevant for designing RTOS on multi-core architectures. From version 2.6 Linux kernel supports SMP with different memory models including NUMA.

Moreover, the first appearance of multiprocessor RTOS implementation in a hard real-time form for Linux is found during the 1990s and is also known as dual kernel [Lehrbaum and Rick, 2000][Bicer et al., 2006]. Generally multiprocessor RTOSs insert a thin layer between the hardware and general purpose operating system for taking over the interrupts and also have their own scheduler, where Linux kernel is running with low priority and only gets CPU as well as resources when there are no real-time tasks available. Unlike multithreaded process, with its own scheduler, in fact, a hardware abstraction layer is added between the hardware and the operating system. Most of RTOSs are already used and proved that they can satisfy hard real-time requirements in some industrial projects while reserving the Linux resource.

Apart from industrial purposes, Linux based multiprocessor RTOSs are also developed for research purposes. These are more focused on desired algorithm integration and simulation facilities, data acquisition tools along with visualization tools. Though most of them support hard and soft real-time features for algorithm specific simulation, the discontinuation of the development process and unstable releases may cause unexpected time delays when the development is stalled on inclusion of up-to-date features of the baseline platform.

In the following section, we outlined currently available Linux based multiprocessor RTOSs which are used as multiprocessor scheduling algorithm simulation test-beds as well as in industrial projects.

6.3 RTLinux

RTLinux (**R**eal**T**ime **A**pplication **I**nterface) is a hard realtime RTOS. RTLinux is developed as dual kernel operating system, and the architecture is transparent, moduler and extensible. Considered as the first generation of multi-core RTOS. RTLinux supports i386, PPC including ARM architecture [Yiqiao et al., 2008]. RTLinux is considered one of the hard real-time variants of Linux, employed manufacturing plants, inside control robots, data acquisition systems, and other time-sensitive machines and instruments [Yodaiken, 1999]. A schematic diagram of RTLimux architecture is drawn in Figure 6.1.

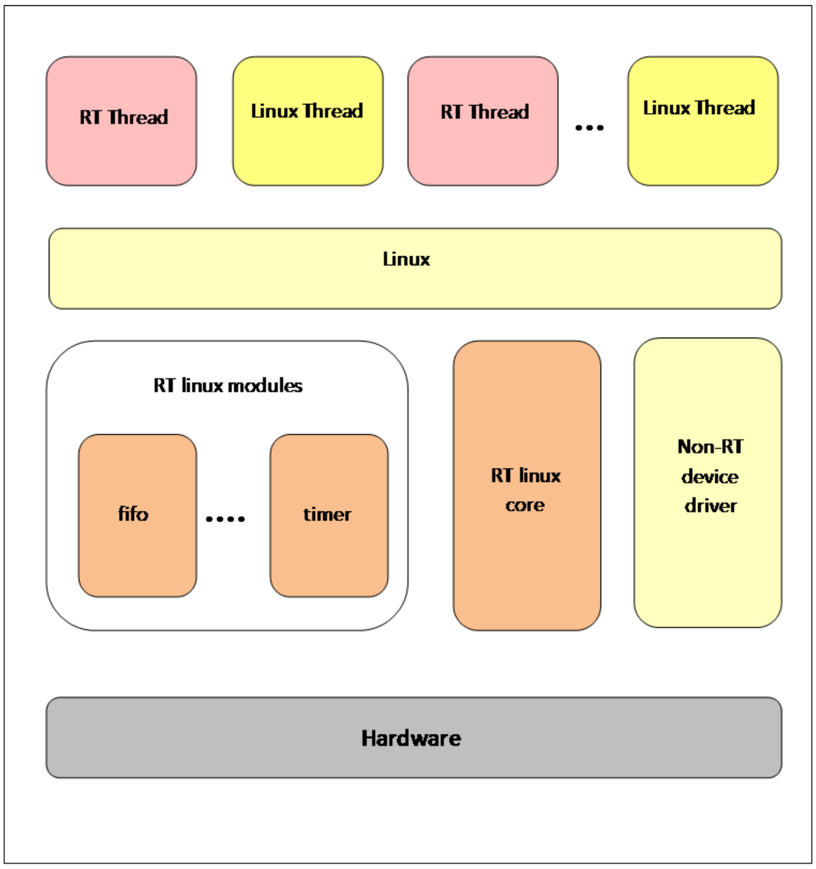


Figure 6.1: RTLinux Architecture [Yiqiao et al., 2008]

A clock structure is implemented as timer module for scheduler. Interrupts are divided into global and local interrupts. Where local interrupts come from the local processor and global interrupts come from shared devices [Kairui et al., 2006]. The integrated scheduler in RTLinux is purely priority driven. Prioritization is done by the Rate Monotonic algorithm.

Though RTLinux was first developed at FSMLabs, it was later on continued as a commercial product which was acquired by Wind River System in February 2007. The commercial support effectively ended through discontinuation of Wind Rivers Real-Time Core Product line. Being a transparent, modular, and extensible system are the key objectives of RTLinux. During the development phase of RTLinux, the maximization of the advantages of having Linux and its powerful capabilities available were considered foremost [RTLinux, 1999].

6.4 S.Ha.R.K

S.Ha.R.K. stands for **S**oft **Ha**rd **R**eal-time **K**ernel which is designed to support soft, hard, and also non real-time applications with interchangeable scheduling algorithms with a configurable kernel architecture [Gai et al., 2001] [Abeni and Buttazzo, 2000]. This soft, hard real-time kernel consists of two types of modules:

- First, Scheduling Modules which implement
  1. Scheduling Algorithms (POSIX, RM, EDF, EDFACT, RR, RRSOFT, RR2, SLSH and Hierarchical Scheduling) and
  2. Aperiodic Servers (PS, DS, SS, CBS, CASH and CBS_FT) [S.Ha.R.K, 2008],
- Second, Resource Modules and also Shared Resource Access protocols (NOP, NOPM, PI, PC, SRP, NPP and SEM) [S.Ha.R.K, 2008]

A Java tracer named JTracer is provided as graphical visualization tool to show the trace files generated by S.Ha.R.K [S.Ha.R.K, 2008].

6.5 MaRTE

A Minimal Real-Time Operating System for Embedded Applications, MaRTE follows POSIX.13 subsets. Which is based on the AdaCore GNU tool-chain, supports the development of Multi-Thread Real-Time applications and has Time measurements framework as utility. Also provides MAST: “Modeling and Analysis Suite for Real-Time Applications” [MARTE, 2011], with an automatic schedulability analysis can be incorporated in a UML design environment for designing real-time applications, while representing requirements and the real-time behavior along with the design structure [MARTE, 2011]. Another important feature, emulation support, is also provided as QEMU image and BOCHS image.

6.6 RTAI

RTAI stands for Real Time Application Interface and is focused on Real Time Distributed Control Systems. Developed by “Dipartimento di Ingegneria Aerospaziale del Politecnico di Milano” (DIAPM) in 1996 or 1997 [Cloutier et al., 2000] [Mantegazza et al., 2000]. A set of modified functions put into one module called Real-Time Hardware Abstraction Layer (RTHAL) is an interface to Linux and RTAI [Yiqiao et al., 2008]. It allows writing a program with strict timing constraints on Linux. In RTAI, a group of loadable modules, RTAI interfaces, LXRT, IPC is implemented. RTAI is mentioned as true community project where developers interact with each other a lot. Built upon a Linux kernel-patch, it also includes a broad variety of services [RTAI, 2014]. A schematic diagram of RTAI architecture is depicted in Figure 6.2.

Architectures supported by RTAI are:

- x86 with and without FPU and TSC
- x86_64
- PowerPC
- ARM architectures with ARM7: clps711x-family, CS89712, Cirrus Logic EP7xxx
- m68k : both NOMMU and MMU cpus

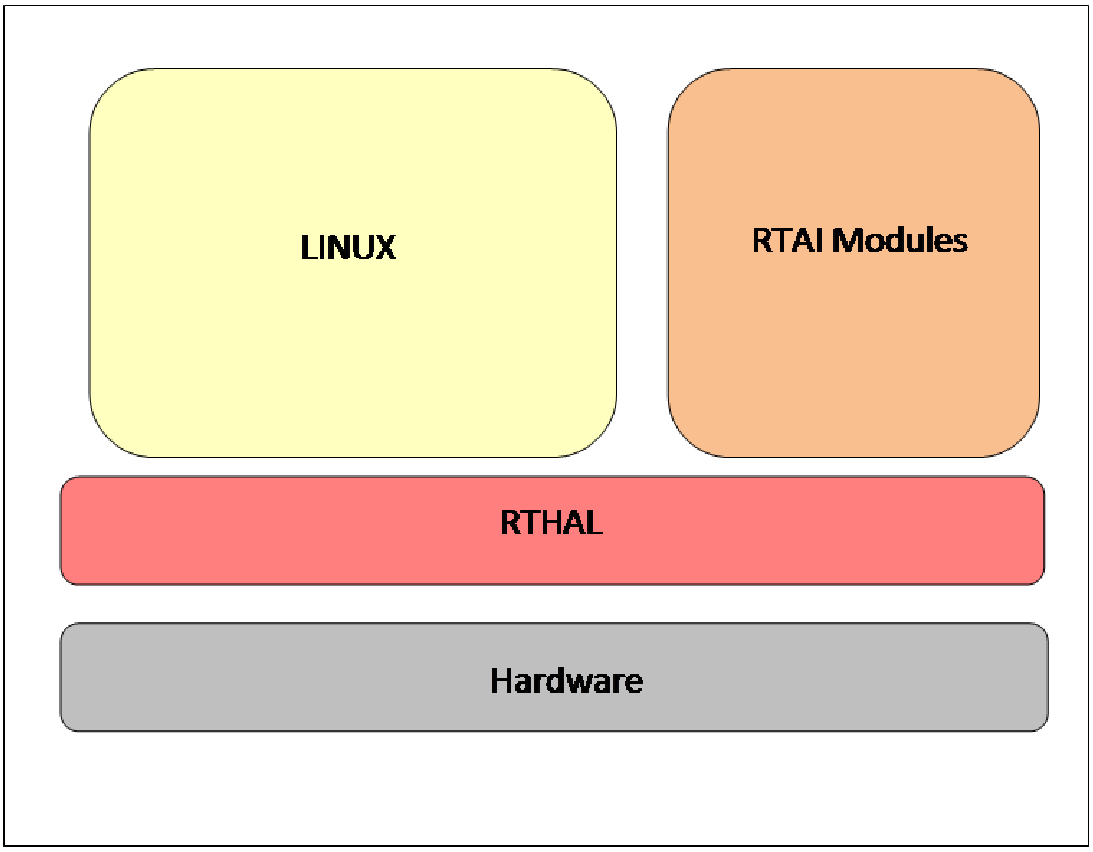


Figure 6.2: RTAI Architecture [Yiqiao et al., 2008]

It is important to mention that, for operational facility on various targets, RTAI distribution comes with a tool chain named RTAI-Lab that converts block diagrams into RTAI executables. Development of a block diagram can be done both on commercial and free tools like Matlab/Simulink/RTW and Scilab/Scicos respectively [RTAI, 2014].

6.7 Xenomai

Xenomai [Gerum, 2004] was developed with the aim of being a framework that supports traditional real-time operating system APIs and also provides portability of other industrial applications from other real-time operating systems to a Linux-based environment while guaranteeing the hard real-time properties. A schematic diagram of Xenomai architecture is depicted in Figure 6.3. In Xenomai, an abstract real-time nucleus is implemented to support the traditional real-time APIs. The pseudo APIs for other different modules are also implemented and named skins. The proposed skin concept is a set of APIs that are a simulation of a traditional real-time operating system. [Yiqiao et al., 2008].

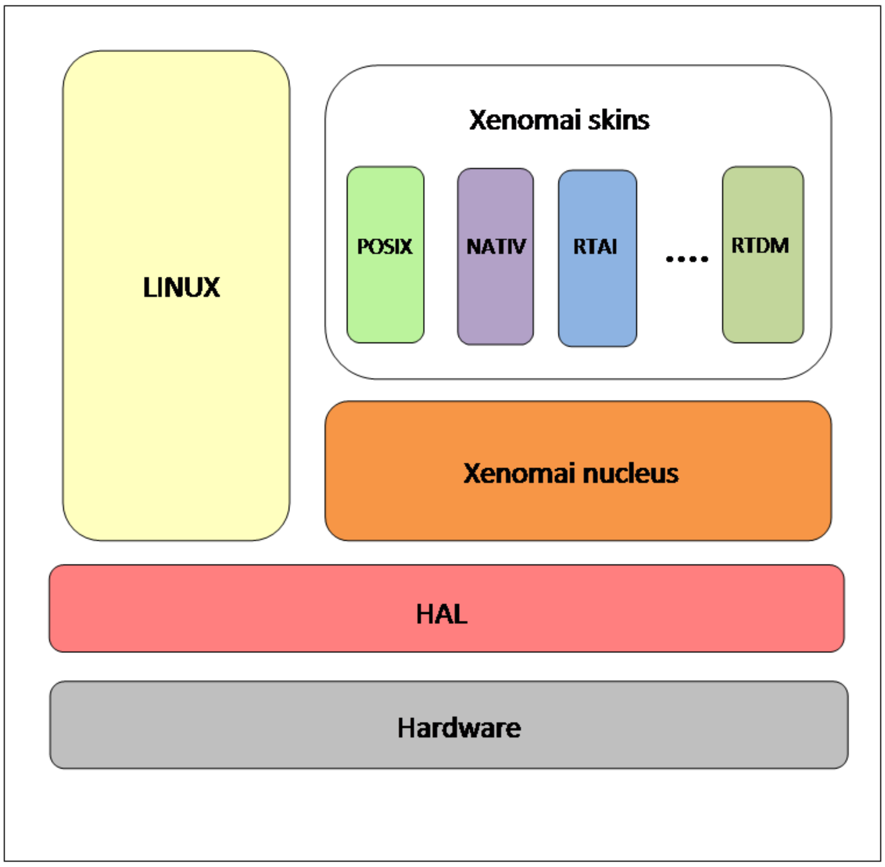


Figure 6.3: Xenomai Architecture [Yiqiao et al., 2008]

6.8 XtratuM/PaRTiKle

XtratuM is different from the others. XtratuM is a virtual machine or a Hypervisor for a real-time system developed focusing on better RTLinux visualization [Masmano et al., 2005]. One of the most significant characteristics of XtratuM is that it provides segmented memory space for the domain running upon it, which makes the system more robust and PaRTiKle (a real-time extension) runs on XtratuM (utilizing POSIX PSE51 interface) which is runnable as a stand-alone system too [S. et al., 2007]. A schematic diagram of XtratuM architecture is depicted in Figure 6.4

XtratuM supports the following features:

- Interrupts
- Timer
- Virtual Memory

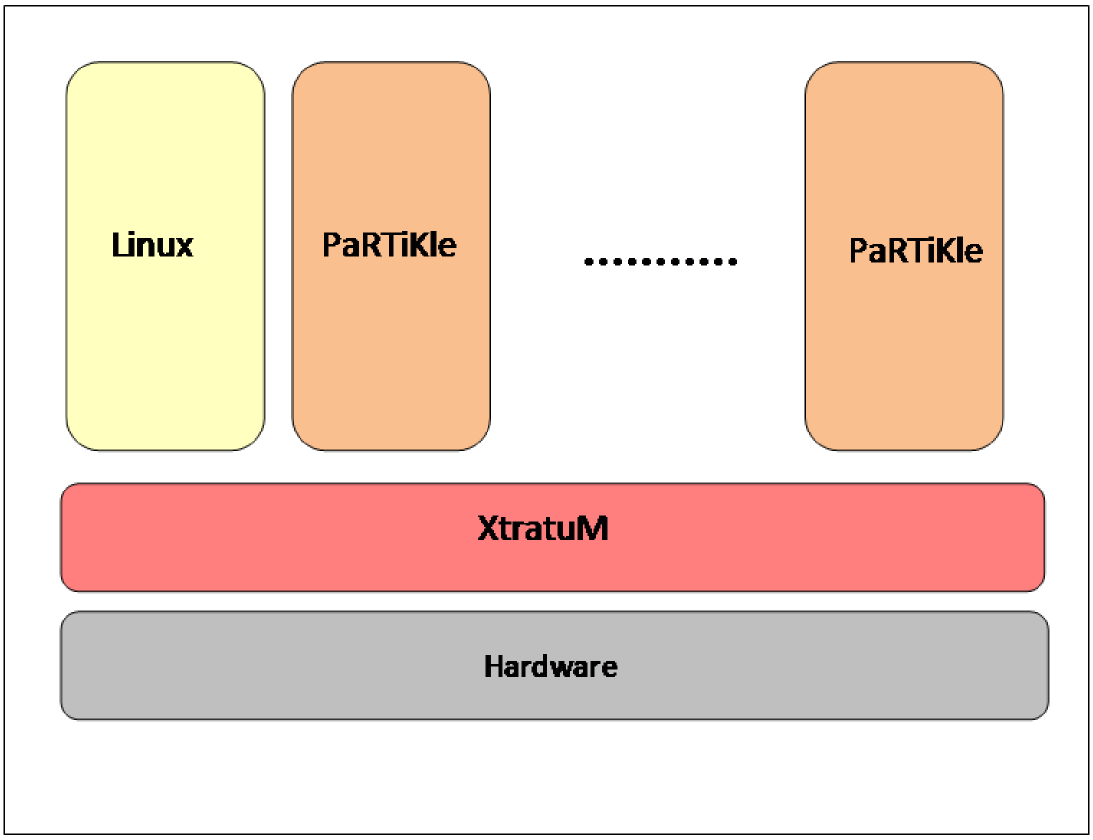


Figure 6.4: XtratuM/PaRTiKle Architecture [Yiqiao et al., 2008]

More specifically, PaRTiKle is an embedded RTOS, has its own kernel space and also user space that interact using system calls between each other [Yiqiao et al., 2008]. Moreover, PaRTiKle is featured with following functionality:

- Provides a simple RM scheduler
- Memory management for physical memory
- Management mechanism for Timer and clock with several different virtual timers
- A minimal C library as kernel library

6.9 ChronOS

ChronOS was created at Virginia Tech (by the Systems Software Research Group) aiming at providing a test-bed based on Linux for resource management research and real-time scheduling on multi-core platforms. The latest version of ChronOS is based on Linux kernel version 3.0.24 and the CONFIG_PREEMPT_RT patch. ChronOS Linux focuses on a best-effort real-time Linux kernel for chip multiprocessors, addressing three problem spaces:

1. OS support to obtain best-effort timing assurances
2. Real-time Linux kernel augmentation with the PREEMPT_RT patch
3. OS support for chip-multiprocessor-aware real-time scheduling

ChronOS architecture is presented in Figure 6.5.

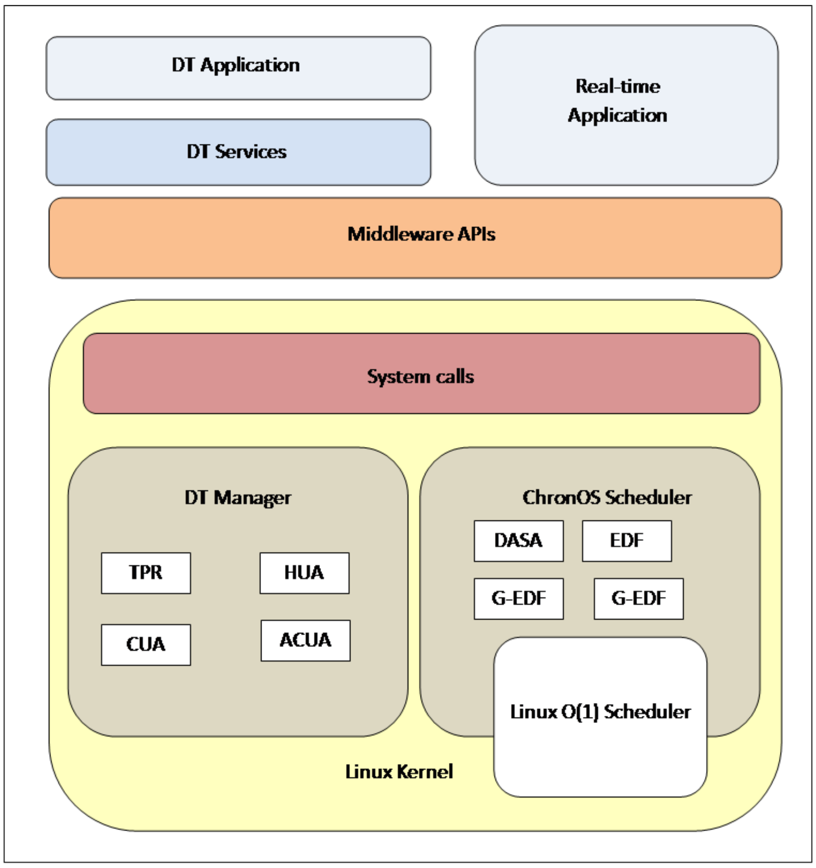


Figure 6.5: ChronOS Architecture [ChronOS, 2013]

Currently, a suite of schedulers has already been implemented in ChronOS that includes GEDF, G-NP-EDF, P-EDF, P-DASA, and GUA [Dellinger et al., 2011].

6.10 LITMUS^{RT}

LITMUS^{RT} stands for **LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in Real-Time systems. Constructed by UNC(The University of North Carolina at Chapel Hill) research group for comparison of various multi-core scheduling algorithms and approaches[Calandrino et al., 2006]. The latest release is built on current stable Linux version (3.10.5). Their first goal is to "provide a useful experimental platform for applied real-time systems research" [LITMUS^{RT}, 2013] and the second one is to serve as a "proof of concept" [LITMUS^{RT}, 2013]. Apart from these goals, a non-goal is also mentioned, which is not to provide reasonable stable and tested platform [LITMUS^{RT}, 2013]. It also provides prototyping support for multi-core real-time scheduling algorithms [LITMUS^{RT}, 2013]. A number of real-time algorithms have already been implemented, among them PSN-EDF, GSN-EDF, C-EDF, P-FP and PD^2 are worth mentioning.

Apart from them, there are also a number of open-source microkernel and kernel extensions are available, such as SCHED_DEADLINE, AQUOSA, Quest and ERIKA Enterprise which are also popular among researchers.

6.11 Summary

It is almost been more than one decade since the simulation platforms and RTOSs started to evolve using new features. The selection of multi-core scheduling algorithms depends on the physical architecture of the multi-core system and real-time constraints. The selection of a simulation test-bed plays a vital role in achieving a scalable result for scheduling algorithms by merging the gap between theoretical output and output from a physical system that has overheads. In such a case, measurement or acquisition of system data to analyze real-time constraints plays a key role. Taking that into consideration, researchers prefer microkernel based operating systems for their explicit message passing between kernel layers, though microkernel based operating systems are slower than the monolithic ones. As a consequence of choosing microkernels, the rest of the system becomes modularized. On the other hand, when the field of research is scheduling algorithms, scheduling techniques of the operating system and its operations of task processing and its operating architecture also require extensive study before selecting a RTOS as a simulation test-bed.

A categorization can simply be made of the current real-time test-beds only by considering which type of kernel or kernel extension is used. A number of simulation platforms use Linux/PREEMPT_RT Kernel, focused on implementing different types of scheduling algorithms.

Considering the programming language support for RTOS development, most RTOSs use C. Among them, PaRTiKle is worth mentioning, since it supports Ada, C, C++ and Java. Also, MaRTE uses AdaCore GNU tool-chain. In the context of simulation tools support, S.Ha.R.K. provides JTracer as a graphical visualization tool. RTAI is augmented by tools like Matlab/Simulink/RTW and Scilab/Scicos while MaRTE provides MAST.

Selection of a RTOS as a test-bed can also be done by comparing which types of multi-core scheduling algorithms are already implemented and which resource sharing protocols are being used.

ChronOS is created focusing on solving best-effort scheduling and scheduling on CMPs (Cheap Multiprocessors) using Linux/PREEMPT_RT Kernel while taking into account Alpha OS [Jensen and Northcutt, 1990] that is a non-Linux based operating system. Though RED-Linux [Wang and Lin, 1999]and LITMUS^{RT} [Brandenburg et al., 2007] are Linux based, none of them uses Linux/PREEMPT_RT Kernel [Dellinger et al., 2011]. As ChronOS is using a preemptive kernel, it has significantly lower worst-case latencies [Dellinger et al., 2012].

On the other hand, LITMUS^{RT} is known as a real-time test-bed for empirical evaluation of multi-core scheduling algorithms[Calandrino et al., 2006], also integrating Hard/Soft real-time jobs aiming to improve the response time of best effort jobs [Brandenburg and Anderson, 2007]. Apart from support of a number of multi-core scheduling algorithm, LITMUS^{RT} is the first to implement mixed-criticality scheduling algorithms on multi-core platforms [Herman et al., 2012]. In LITMUS^{RT} scheduling policies are implemented as scheduler plug-ins and an API providing system calls for interaction with then kernel. In Linux, scheduler task scheduling is done by traversing a list of scheduling classes. In decreasing order POSIX RT, CFS, and IDLE classes are available in stock Linux. To achieve a higher priority than the other classes, LITMUS^{RT} scheduling classes are inserted on top of the ones in stock Linux [Brandenburg et al., 2008].

In this thesis work, we chose LITMUS^{RT} as our simulation platform for the performance measurement of PD^{2*} scheduling algorithm. The logic behind this selection is that LITMUS^{RT} has already implemented the variants of scheduling algorithms like P-EDF, G-EDF and Pfair on which we are focusing in this thesis work. Also, it provides a number of modifiable open-source tools for schedulability analyses, taskset generation, experimental data acquisition, and even for data conversion. In the next chapter, all of these tools and their use are described in more detail.

7. Simulation

In the previous chapter we gave a brief idea about our selected multi-core scheduling algorithm simulation test-bed which is LITMUS^{RT} release version 2013.1, released in August 2013 version 3.10.5. To use LITMUS^{RT} the following open source tools are required for scheduling algorithm simulation and analysis:

- **litmus-rt**: LITMUS^{RT} is developed by modifying Linux kernel and creating user-space library. As development of LITMUS^{RT} is focused on real-time scheduling, most of the kernel modification is done in scheduler and timer-interrupt code. The kernel modification is divided into the following three components:

  1. Core infrastructure: consist of modification of scheduler, scheduler support structure and services, i.e. tracing.
  2. Scheduler plugins: here scheduling policies are implemented.
  3. System calls: To interact real-time tasks with the kernel through API.

  In stock Linux, three classes are available; they are POSIX, RT, CFS and IDLE, where scheduler selects a task to be scheduled using traversal over a list of scheduling classes and by choosing the task with highest-priority. In LITMUS^{RT}, a LITMUS^{RT} scheduling class is integrated as the very top of scheduling class to achieve higher priority for the jobs scheduled by LITMUS^{RT}.

- **liblitmus**: This is the user space library that provides the LITMUS^{RT} API. It provides the following tools:

  1. setsched: Selects active scheduler.
  2. showsched: Prints the name of the currently active scheduler.
  3. rt_launch: Launches the program as a real-time task provisioned with the given worst-case execution time and period.
  4. rtspin: A simple spin loop for emulating purely CPU-bound workloads.
  5. release_ts: Releases the task system.
  6. measure_syscall: Measures the cost of a system call.
  7. cycles: Displays cycles per time interval.
  8. setsched: Selects active scheduler.

- **feather-trace-tools**: There are three tracing mechanisms available in LITMUS^{RT}. They are implemented as three sets of device files:

  1. litmus_log : captures the text messages created with the TRACE() macro, exported as */dev/litmus/log* .
  2. ft_traceX : contains binary-encoded time stamps used for overhead tracing, exported as */dev/litmus/ft_trace0*, */dev/litmus/ft_trace1*, etc.
  3. sched_traceX: contains binary-encoded scheduling event information, e.g. whenever a task got scheduled, a job was released, a job completed, etc., exported as */dev/litmus/sched_trace0*, */dev/litmus/sched_trace1*, etc.

Automated process of enabling events and retrieving trace records is performed by *ftcat*, which is part of the *ft_tools* package. Events can be specified by their ID as follows [LITMUS^{RT}, 2013]:

1. SCHED_START, SCHED_END: To measure the time spent to make a scheduling decision.
2. CXS_START, CXS_END: To record the time spent to make a context switch .
3. SCHED2_START, SCHED2_END: To measure the time spent to perform post-context-switch cleanup and management activities.
4. TICK_START, TICK_END: To measure the overhead by Linux's periodic system tick.
5. RELEASE_START, RELEASE_END: To measure the time spent to enqueue a newly-released job in a ready queue .
6. RELEASE_LATENCY: To measure the difference between when a timer should have fired, and when it actually did fire, unlike the others this ID is measured in nanoseconds.

- **experiment-scripts**: There are Python scripts which provide a simpler way for creating, running, parsing, and plotting experiments:
  - gen_exps.py: To create sets of experiments.
  - run_exps.py: To run and trace experiments.
  - parse_exps.py: To parse LITMUS^{RT} trace data.
  - plot_exps.py: To plot directories of csv data generated by *parse_exps.py*.
- **schedcat**: Schedcat stands for Schedcat: the **sched**ulability test **c**ollection **a**nd **t**oolkit, developed by [Brandenburg, 2011]. It is a Python/C++ library useful for schedulability experiments which contains a number of schedulability tests for real-time schedulers including partitioned and global EDF, Pfair, etc.. More significant provided features are partitioning heuristics and blocking term analysis for real-time locking protocols. In addition to several spinlock types, i.e., the OMLP family, the FMLP and FMLP+, the MPCP, and the DPCP. Also provided are overhead accounting methods Schedcat [2011].

7.1 Baseline Platform

We have conducted our experiment in implementing LITMUS^{RT} on both a physical machine and QEMU emulator with the same tasksets and scheduling algorithms. We used QEMU emulator which is also a virtualizer. When running a target architecture that is the same as the host architecture, QEMU can make use of KVM (Kernel Virtual Machine). In our case, the architectures are *qemu-system-x86_64* and *x86_64*. These are the main reasons behind choosing QEMU as emulator. Also, an externally built Linux kernel can be loaded over an existing one which helps developers to save a huge amount of time by avoiding multiple compilation procedures. Also advantageous from the development perspective is that it supports GDB (NU Project Debugger).

Our desired simulation test-bed on QEMU emulator was initiated by the following command to achieve the following multiprocessor platform configuration shown in Table 7.1:

```
sudo qemu-system-x86_64 -enable-kvm -cpu host  -smp 16 -hda /home/
    sdas/iso/ubuntu.backing.qcow2.img -m 8192 -nographic -kernel /home
    /sdas/litmus13/litmus-rt/arch/x86_64/boot/bzImage -append "console
    =ttyS0,115200 root=/dev/hda1 rootfstype=ext4 nmi_watchdog=0" -gdb
    tcp::12345 -net nic -net tap,ifname=tap0,script=no  -net user,
    hostfwd=tcp::2222-:22
```

By the same token LITMUS^{RT} was installed on the physical machine using the following shell script which might include some irrelevant drivers and modules provided with the default operating system ISO by Kubuntu:

```
make include/config/kernel.release
VER=`cat include/config/kernel.release`
CC=gcc
THREADS=`cat /proc/cpuinfo | grep processor | wc -l`

echo "Building kernel ${VER} with ${CC} on ${THREADS} threads"

if [ $(/usr/bin/id -u) -ne 0 ]; then
        echo "Please run this script as root."
        exit 2
fi

check_errors() {
        # Function. Parameter 1 is the return code
        # Para. 2 is text to display on failure.
        if [ "${1}" -ne "0" ]; then
                echo "ERROR # ${1} : ${2}"
                # as a bonus, make our script exit with the right
                    error code.
                exit ${1}
        fi
}
make -j$THREADS bzImage CC=$CC
check_errors $? "make bzImage failed"
make -j$THREADS modules CC=$CC
check_errors $? "make modules failed"
sudo make modules_install
check_errors $? "make modules_install failed"
sudo make headers_install INSTALL_HDR_PATH=/usr/src/linux-headers-"
    $VER"
check_errors $? "make headers_install failed"
sudo make install
check_errors $? "make install failed"
sudo update-grub
check_errors $? "grub update failed"
```

Both of the multiprocessor simulation test-bed configurations are shown in Table 7.1.

	QEMU emulator configuration	Physical machine configuration
Architecture	x86_64	x86_64
CPU op-mode(s)	32-bit, 64-bit	32-bit, 64-bit
Byte Order	Little Endian	Little Endian
CPU(s)	16	4
On-line CPU(s) list	0-15	0-3
Thread(s) per core	1	2
Core(s) per socket	1	2
CPU socket(s)	16	1
NUMA node(s):		1
Vendor ID	GenuineIntel	GenuineIntel
CPU family	6	6
Model	37	37
Stepping	2	2
CPU MHz	2260.996	2260.835
BogoMIPS	4521.99	4521.67
Virtualization	VT-x	VT-x
Hypervisor vendor	KVM	
Virtualization type	full	
L1d cache	32K	32K
L1i cache	32K	32K
L2 cache	4096K	256K
L3 cache:		3072K
NUMA node0 CPU(s):		0-3

Table 7.1: Simulation platform configuration of virtual and physical machines

7.2 Experimental Task Sets

We used an open source tool *experiment-scripts* to generate our experiment tasksets. As our selected algorithm is PD^{2*}, it actually differs from PD^2 in the context of grouping the tasks as heavy and light based on their utilization $> .5$ or $< .5$ respectively. Along with provided uniform distribution we have added another uniform utilization bound $[0.1, 0.9]$ (uni-range). $[0.0001, 0.001]$ (uni-very-light) to analyze the performance of simulated algorithms in the context of tasksets with very low individual overhead. The ranges for the uniform distributions are uni-very-light, uni-light, uni-medium, uni-heavy, uni-mixed, uni-range presented in Table 7.2

Utilization Type	Range
uni-very-light	0.0001, 0.001
uni-light	0.001, 0.1
uni-medium	0.1, 0.4
uni-heavy	0.5, 0.9
uni-mixed	0.1, .4
uni-range	0.1, 0.9

Table 7.2: Uniform distribution of utilization used in task set generation

As we have already discussed that, experiment-scripts come with the functionality of running the simulation and parsing the simulated data and also converting the data from the binary format to CSV format. We have extended this experiment-scripts to achieve compatibility with our new implemented scheduler plug-in.

In this thesis work, we have simulated multi-core scheduling algorithms in several combinations.

1. For both PD^2 and PD^{2*}, we have simulated tasksets consisting of randomly generated 24 tasks for each of the uniform distributions presented in 7.2 generated using experiment-script. In total, the number of tasksets in the experiment is $2 \times 6 = 12$. Also, we have simulated tasksets described above both in the QEMU emulator and on the physical machine.

2. Again, for both PD^2 and PD^{2*}, we have simulated a sequence of randomly generated 20 tasksets for each of the uniform distributions presented in 7.2, where task numbers in tasksets were from 5 to 100 in steps of 5, generated using experiment-script. In total, the number of tasksets in the experiment is $40 \times 6 = 240$.

3. We organized another simulation for all of the schedulers in LITMUS^{RT} which are GSN-EDF, PSN-EDF, C-EDF, PD^2 as PFAIR and PD^{2*} as PFAIR23. In this case we consider a sequence of 20 tasksets where utilization distributions are uni-light, uni-heavy and uni-rang. In total, the number of tasksets in the experiment is $100 \times 3 = 300$.

7.3 Algorithm Implementation

In Chapter 5, we have discussed both the Pfair variants PD^2 and PD^{2*}. As PD^{2*} proposed by Nelissen et al. [2014] is a generalized version of PD^2, the whole structure of PD^{2*} does not differ from PD^2 very much, except for the group deadline parameters and functionality. Though PD^{2*} is only slightly different from PD^2, in order to run a number of simulations simultaneously we have to implement PD^{2*} as separate scheduler plugin to LITMUS^{RT} with the support of existing data structures for PFAIR. In existing LITMUS^{RT} PD^{2*} is named PFAIR, to keep our implementation simple we named PD^{2*} PFAIR23.

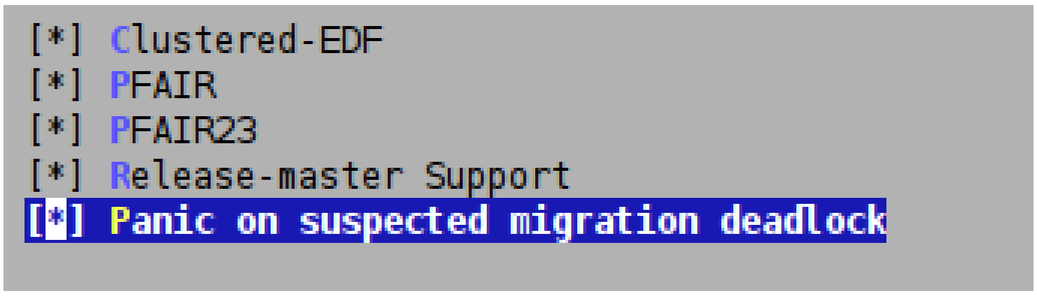
To make PFAIR23 available in the kernel configuration, we extended the existing *Kconfig* with the following configuration:

```
config PLUGIN_PFAIR23
	bool "PFAIR23"
	depends on HIGH_RES_TIMERS && HZ_PERIODIC && HZ = "1000"
	default y
	help
	  Include the PFAIR23 plugin (i.e., the PD^2* scheduler) in
	     the kernel.
	  The PFAIR23 plugin requires high resolution timers (for
	     staggered
	  quanta) and also requires HZ_PERIODIC (i.e., periodic timer
	      ticks
	  even if a processor is idle, as quanta could be missed
	     otherwise).
	  Further, the PFAIR23 plugin uses the system tick and thus
	     requires
```

```
          HZ=1000 to achive reasonable granularity.

          If unsure, say Yes.
```

After this extension of the kernel configuration, our PFAIR23 became visible in the *Scheduling* plugin option in *LITMUS*^{RT} as below:



```
[*] Clustered-EDF
[*] PFAIR
[*] PFAIR23
[*] Release-master Support
[*] Panic on suspected migration deadlock
```

Figure 7.1: Plugin

7.4 Experiment Results

Before continuing with the simulation data, it is worth mentioning that all overhead such as context switching, scheduling, releases, etc. is measured in microseconds, only release latency is measured in nanoseconds. The parse scripts from the experiment-script scale from processor cycles (feather trace traces events in terms of cycles) to microseconds LITMUS^{RT} [2013].

Execution of experiments was done by run_exps.py to simulate tasksets and trace the overheads. It follows the state sequence below:

1. Loading experiment [NUMBER] of [TOTAL NUMBER OF EXPERIMENT].
2. Enabling Overhead Trace.
3. Enabling Sched Trace.
4. Enabling Trace-cmd / Kernelshark.
5. Enabling Logger.
6. Writing 1 proc entries.
7. Starting 3 regular tracers; Overhead Trace, Sched Trace and Trace-cmd.
8. Switching to [ACTIVATED SCHEDULER PLUGIN].
9. Starting [NUMBER] tasks in the selected experiment taskset.
10. Sleeping until tasks are ready for release... .
11. Starting 1 released tracers.
12. Releasing [NUMBER OF TASKS] tasks in the selected experiment taskset.
13. Waiting for program to finish... .
14. Stopping exact tracers.

15. Saving results in [LOCATION TO SAVE ALL TRACE DATA].

16. Stopping regular tracers.

17. Switching back to Linux scheduler.

18. Experiment done!/interrupted.

Out of the resulting six pairs of results of our first set of simulation based on processor utilization distribution described in Table 7.2, we presented the result of one pair with very light utilization scale. Also for both QEMU emulator and physical machine the details are found in the following tables:

- For QEMU emulator with very light processor utilization distribution, the results of PD^2 are presented in Table 7.3 and PD^{2*} in Table 7.4

- For physical machine with processor utilization distribution, the results of PD^2 are presented in Table 7.5 and of PD^{2*} in Table 7.6

	Avg	Max	Min	Var
CXS	3.500	1976.344	0.687	99.953
RELEASE	7.152	11317.965	0.922	40382.454
RELEASE_LATENCY	1181.014	18275.324	4.300	3451880.861
SCHED	24.676	29250.326	2.837	163587.753
SCHED2	5.401	182.545	2.165	9.907
TICK	9.709	41742.031	0.717	45000.009
record-loss	0.931	0.981	0.816	0.002

Table 7.3: Simulation data of PD^2 on QEMU emulator with uni-very-light processor utilization distribution and 24 tasks

	Avg	Max	Min	Var
CXS	3.450	202.267	0.692	7.939
RELEASE	2.662	626.708	0.935	60.690
RELEASE_LATENCY	1216.406	27254.463	4.649	3743596.048
SCHED	19.238	14295.758	2.924	25519.474
SCHED2	5.397	138.003	2.179	8.132
TICK	10.897	69877.859	0.812	132953.676
record-loss	0.933	0.987	0.825	0.002

Table 7.4: Simulation data of PD^{2*} on QEMU emulator with uni-very-light processor utilization distribution and 24 tasks

Using feather-trace-tools the number of context switches is calculated for both PD^2 and PD^{2*} as listed in Table 7.7

	Avg	Max	Min	Var
CXS	1.063	15.747	0.281	0.131
RELEASE	0.658	1.659	0.294	0.072
RELEASE_LATENCY	1099.309	8408.448	0.296	6413152.000
SCHED	9.322	69.452	1.963	14.204
SCHED2	2.913	15.369	1.611	0.391
TICK	12.496	28.488	0.428	42.608
record-loss	0.912	1.000	0.510	0.022

Table 7.5: Simulation data of PD^2 on physical machine with uni-very-light processor utilization distribution and 24 tasks

	Avg	Max	Min	Var
CXS	1.089	46.780	0.273	0.189
RELEASE	0.756	1.775	0.329	0.067
RELEASE_LATENCY	12.166	36.162	0.297	176.488
SCHED	9.361	77.735	1.966	18.022
SCHED2	2.924	15.425	1.596	0.532
TICK	12.384	34.526	0.384	43.661
record-loss	0.845	1.000	0.505	0.026

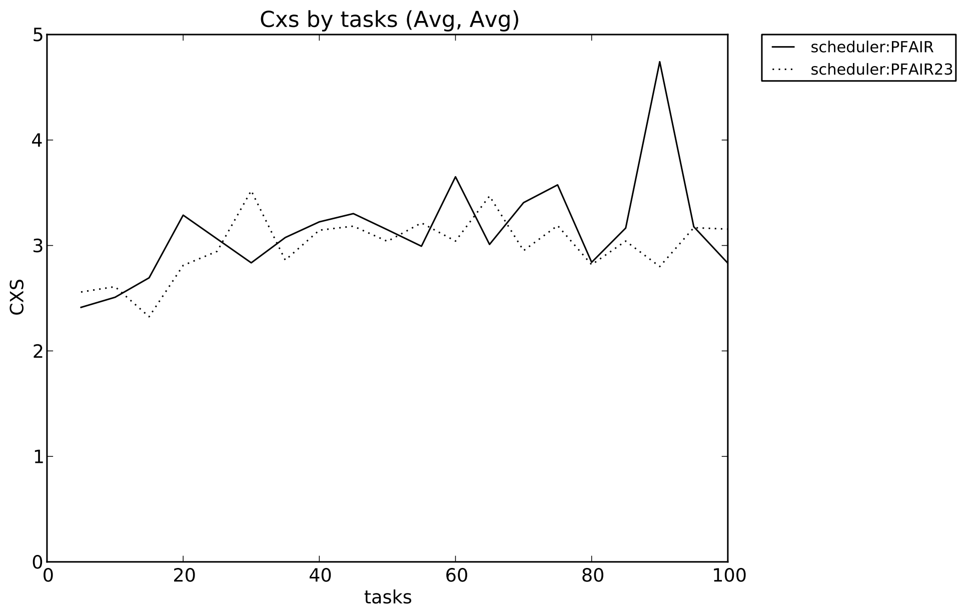
Table 7.6: Simulation data of PD^{2*} on physical machine with uni-very-light processor utilization distribution and 24 tasks

	QEMU emulator		Physical Machine	
Number of CXS start	PD^2	PD^{2*}	PD^2	PD^{2*}
Total	1166427	1171742	900336	936803
Skipped	3	3	3	3
Avoided	0	0	0	0
Complete	9452	9499	82078	82079
Incomplete	326	353	2207	1922
Non RT	43226	43465	22917	28248
Interleaved	0	0	0	0
Interrupted	9	15	35	103

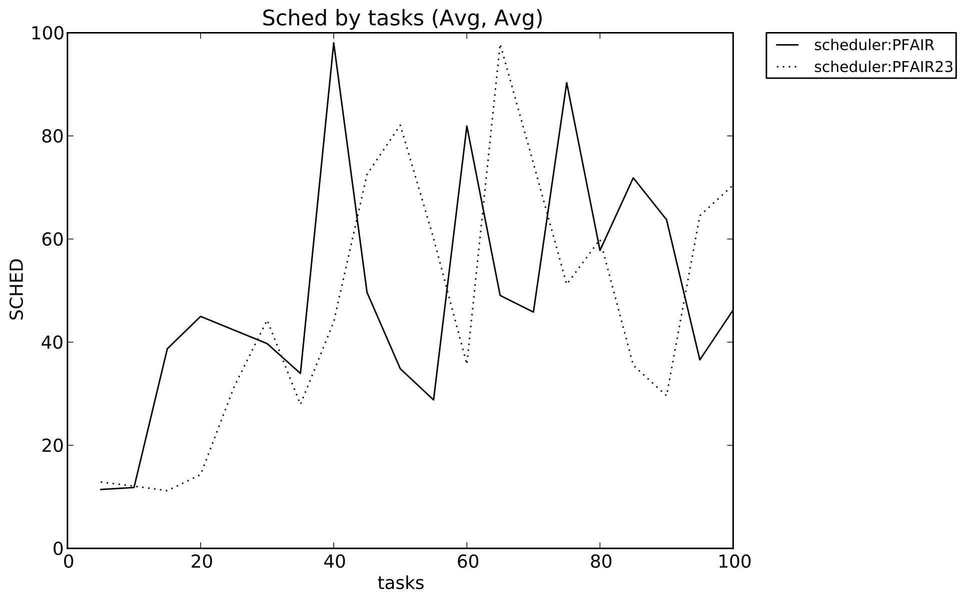
Table 7.7: Number of context switches incurred for PD^2 and PD^{2*} with the taskset of 24 tasks, both in QEMU emulator and on the physical machine

Our second set of simulation runs is based on processor utilization distribution described in Table 7.2, where the task number in the taskset increased from 5 to 100 in steps of 5, generated approximately 28 GB of raw data. Consequently parsed CSV's are also populated with huge amounts of data which led us to represent the results in the form of plots. The objective of this simulation was to observe the behavior of PD^2 and PD^{2*} with all the individual utilization scales listed in Table 7.2. Among the generated plot diagrams coming from the simulation data, CXS_task_Avg_Avg represents the average of average context switch overhead measured for each tested taskset.

Results with very-light utilization and full range utilization are presented in the following Figure 7.2 and Figure 7.3, which represent the average of average context switch and scheduling overheads for tasksets mentioned above:

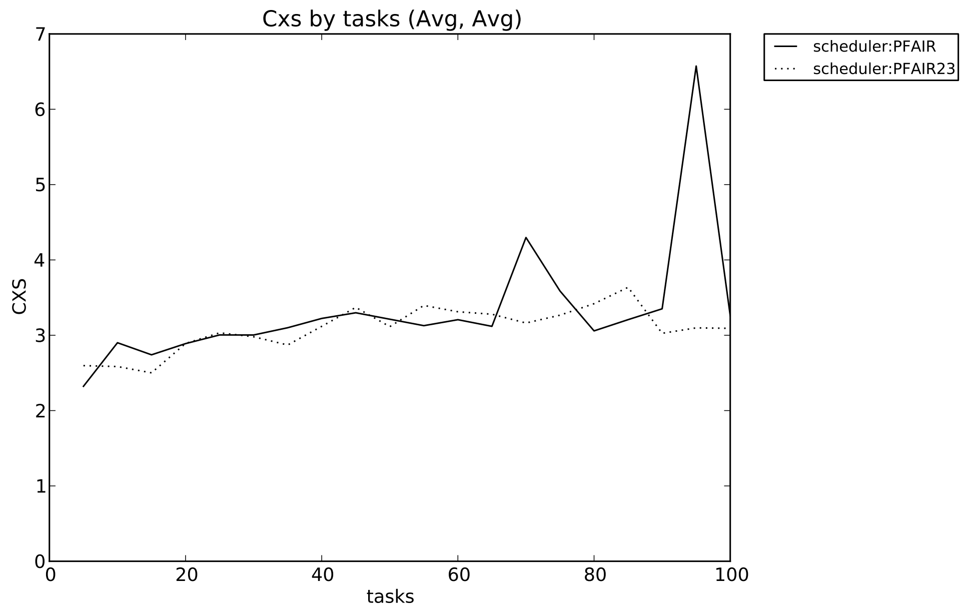


(a) CXS for tasksets with very-light utilization

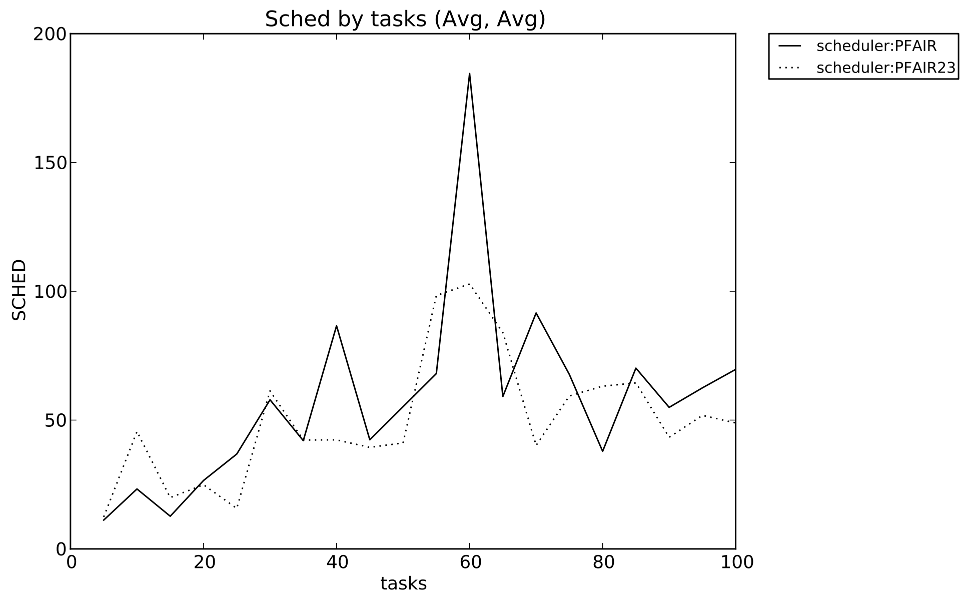


(b) SCHED for tasksets with very-light utilization

Figure 7.2: Context switching (μs) and scheduling overheads (μs) of PD^2 as PFAIR and PD^{2*} as PFAIR23 algorithms with very-light processor utilization distribution for tasksets with task number increasing from 5 to 100 in steps of 5



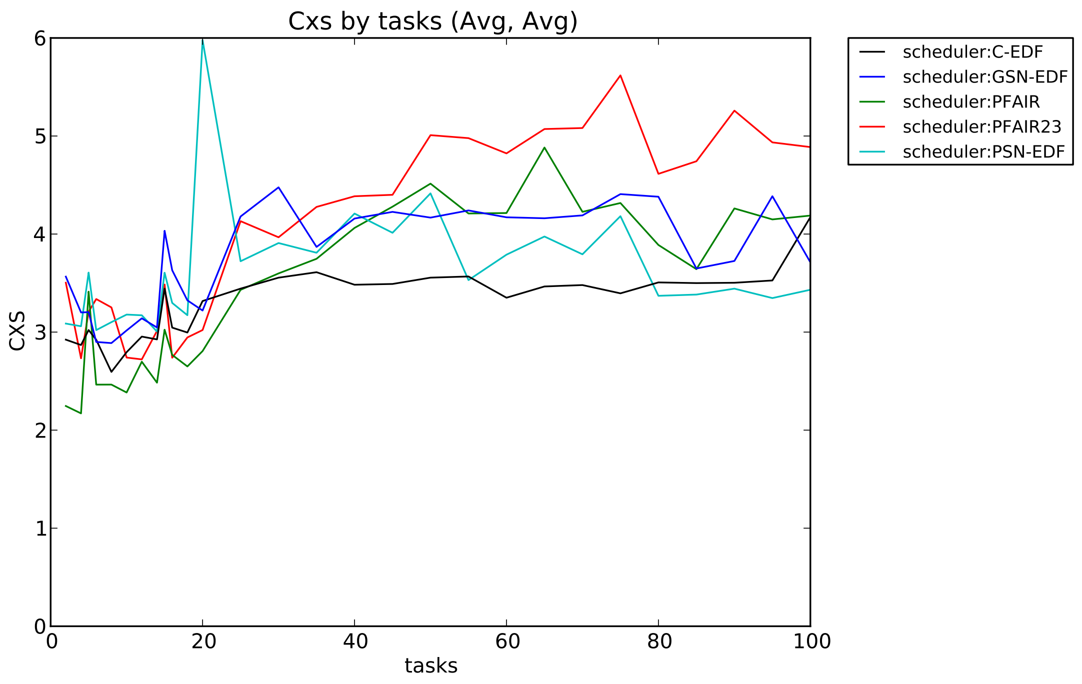
(a) CXS for tasksets with full range utilization



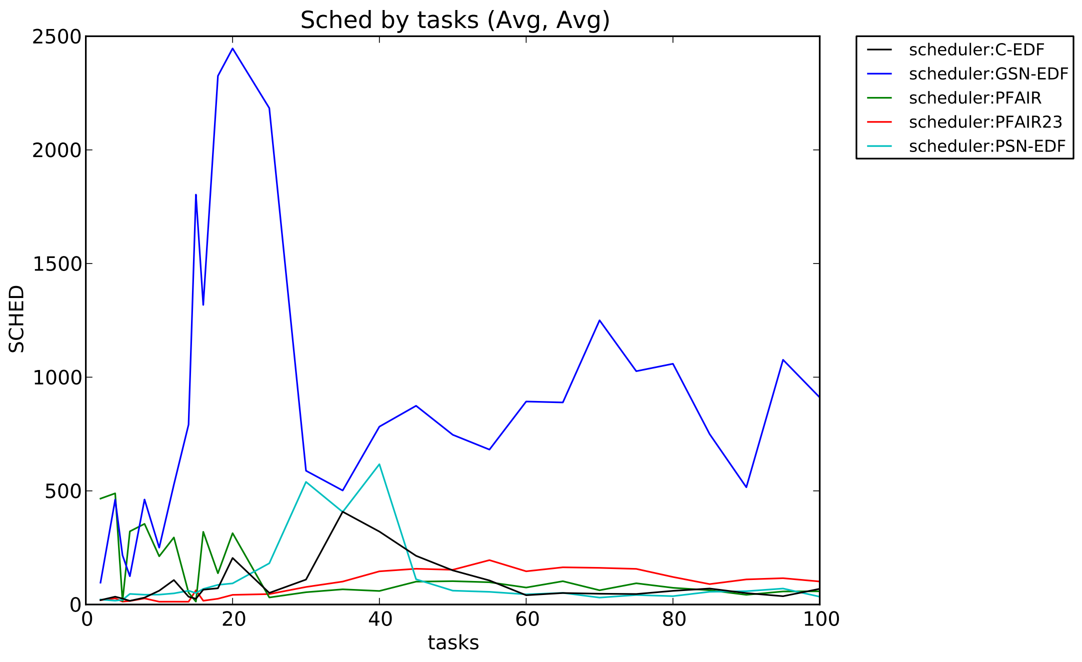
(b) SCHED for tasksets with full range utilization

Figure 7.3: Context switching (μs) and scheduling overheads (μs) of PD^2 as PFAIR and PD^{2*} as PFAIR23 algorithms with full range processor utilization distribution for tasksets with task number increasing from 5 to 100 in steps of 5

Following in the footsteps of the previous results, our third simulation also generated a large amount of raw data which consist of tasksets with the task number increasing from 5 to 100 in steps of 5 and includes all the scheduler plug-ins. This simulation was conducted to compare the scheduler behavior for all of the tasksets. Results are plotted in Figure 7.4 and Figure 7.5 for uniform distribution of full range tasks.

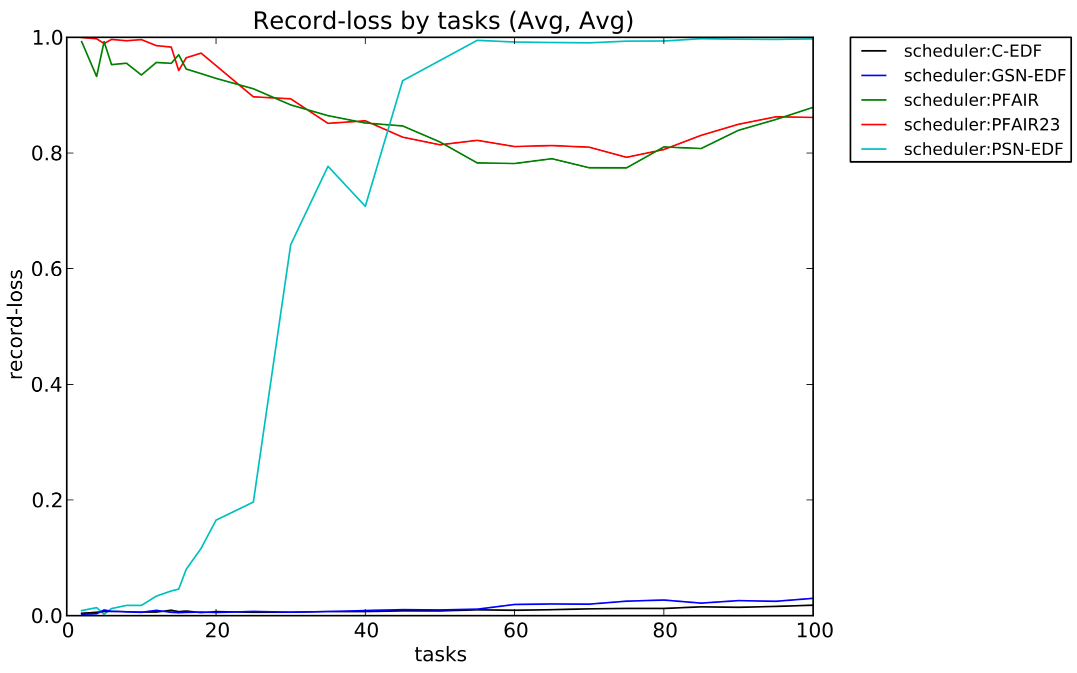


(a) CXS for tasksets with full range utilization



(b) SCHED for tasksets with full range utilization

Figure 7.4: Context switching (μs) and scheduling overhead (μs) of PSN-EDF,GSN-EDF,C-EDF,PFAIR,PFAIR23 algorithms for tasksets with tasks increasing from 5 to 100 in steps of 5, utilization distribution: full range



(a) RECORD LOSS for tasksets with full range utilization

Figure 7.5: Record loss rations (%) of PSN-EDF,GSN-EDF,C-EDF,PFAIR,PFAIR23 algorithms for tasksets with tasks increasing from 5 to 100 in steps of 5, utilization distribution: full range.

7.5 Summary

In our simulation, we confirmed relevance of the previous research results presented in Brandenburg et al. [2008], proposed by Nelissen et al. [2014] as discussed in Chapter 5 which validates our implementation. Considering the form factor of the simulation platform formerly used by researchers like Sun UltraSPRAC Niagra multi-core platform or multiple Intel chip based Xeon E5405 multi-core platform, we have to restrict our number of experiments. All worst-case overheads must be known in advance to satisfy the definition of hard real-time which is not possible in currently available Linux. Many sources of unpredictability are present, i.e., interrupt handlers and priority inversions within the kernel. Furthermore, the lack of determinism is also caused by the hardware platforms on which Linux runs where cache contention, bus , and bus locking and atomic operations are different in each specific case [Brandenburg et al., 2008].

In our first simualtion attempt with a taskset consisting of 24 tasks using the utilization distributions shown in 7.2, we found different results for context switching overhead for PD^2 and PD^{2*} when utilization was very light. The simulation result shows that PD^{2*} has lower context switching overhead than PD^2 for very light utilization distribution. As our first attempt was also conducted both in QEMU emulator and on the physical machine, we also have a different result for the total number of context switching. Though the total number of context switches observed on the physical machine was less than in QEMU emulator, the number of non real-time context switches in QEMU emulator was greater than on the physical machine. Also, the total number of interrupted context switches in QEMU emulator is less than on the physical machine. This led us to run our further simulation experiments in QEMU emulator.

Our second simulation attempt concentrated on both PD^2 and PD^{2*} where task number in a taskset increased from 5 to 100 in steps of 5. As a consequence of this simulation, we found that for PD^2 and PD^{2*} in case of a large number of tasks in a taskset, total context switching overheads of PD^{2*} are almost similar to those of PD^{2*}, and total scheduling overhead is higher for PD^{2*} than that for PD^2. This is consistent with the results proposed by Nelissen et al. [2014].

The third simulation attempt concentrated on all the available schedulers including our integrated one, PD^{2*} where the task number in a taskset was similar to the second attempt and increased from 5 to 100 in steps of 5. In this simulation we found that, as expected context switching overheads for both of the schedulers PD^2 and PD^{2*} are higher than the others, although the individual context switching overheads of PD^{2*} are higher than those of PD^{2*} compared to the previous simulation. By giving a closer look to Figure 7.2 (b) and Figure 7.4 (b), we found that the for taskset with more than 25 tasks (app.) scheduling overhead for PD^{2*} was higher than PD^2 compared to scheduling overhead measured in our third attempt simulation, where scheduling overhead for PD^{2*} and for PD^2 was interchanged by the variation of the number of tasks in taskset in our second attempt simulation, which correlates the variation in simulation results in our current simulation attempt. Considering scheduling analysis focused on global and partitioned approaches, apart from the higher percentage of record loss 7.5, PSN-EDF was more effective than others. From the scheduling point of view, scheduling overhead of GSN-EDF is very high. Furthermore, C-EDF was also highly effective. This confirms the measurement presented by Brandenburg et al. [2008]

8. Conclusion and Future Work

This thesis work is an effort combining both analysis and simulation of scheduling techniques for multi-core scheduling embedded architectures. The analysis of scheduling techniques is augmented by simulation results while the analysis of simulation test-beds concentrates on the possibility of simulating the analyzed techniques.

In our analysis of scheduling techniques, we confirmed the performance measurement conducted by Brandenburg et al. [2008]. It was observed that from the point of view of overhead staggered Pfair performed much better that pure Pfair. Our result also showed that the Global EDF scheduling algorithm performed poorly due to the overhead incurred by the manipulation of a lengthy global queue accessed by all processors. Furthermore, for hard real-time tasksets Partitioned EDF performed best except when the tasks had high individual utilization. In a performance measurement experiment by Nelissen et al. [2014], confirming the same conclusion of Brandenburg et al. [2008], it was shown that BF^2 performed better than PD^2 in terms of overhead. It was also mentioned that in terms of preemption and migration the results of PD^{2*} were similar to PD^2, though its scheduling overhead was worse, PD^{2*} being a slight variation of PD^2 [Nelissen et al., 2014]. In our work, we organized the performance measurement and comparison of PD^{2*} based on the proposal of Nelissen et al. [2014]. Consequently, the need to organize performance measurement led us to analyze and compare available multi-core real-time test-beds as well as real-time operating systems for multi-core embedded architectures while searching for a suitable one.

In our analysis of real-time test-beds, we found that there are two basic categories of test-beds available based on their application: simulation test-beds for industrial purposes and simulation test-beds for research purposes. Both types have advantages and disadvantages considering their architectural implementation, provided APIs, affiliated tools and user support, as discussed in the Chapter 6. To simulate our selected multi-core scheduling algorithm PD^{2*}, we chose LITMUS^{RT} as the simulation test-bed.

Though PD^{2*} is rather similar to PD^2, we had to implement the algorithm proposed by Nelissen et al. [2014] as a new plug-in on LITMUS^{RT} to be able to simulate it along with other scheduling algorithms. To implement this algorithm, we created a new plug-in named PFAIR23 while adopting the structures and functions of the existing PD^2 plug-in named PFAIR. This was made possible by virtue of its being FOSS (Free Open Source Software). Also, for acquisition and analysis of simulation data, we extended the user space tools provided by LITMUS^{RT} to make the new plug-in compatible with the existing simulation test-bed.

To see the difference in behavior of scheduling algorithms based on the distinction between system architectures, we executed our simulation test cases in both QEMU emulator and on the physical machine. We observed that scheduling algorithms return different results for different platforms. Possible reasons might be the cache size of the system or, for the global approach, from which cache level processors are accessing data. Also, hardware interrupts like NMI (non-maskable interrupts) or system management interrupts may play a role. This observation prioritizes the reliability of simulation data produced on the real physical machine that is needed before utilizing any scheduling algorithm on any particular platform for industrial purposes, specifically on hard real-time embedded systems.

In our second simulation attempt, the total context switching overheads of PD^{2*} are almost similar to those of PD^2 while the total scheduling overhead is higher for PD^{2*}, which is consistent with the results proposed by Nelissen et al. [2014]. In our third simulation we found that, as expected, context switching overheads for both of the schedulers PD^2 and PD^{2*} are higher than for those of other algorithms, which corresponds to the result proposed by Nelissen et al. [2014]. Also, the scheduling overhead of GSN-EDF is very high. Furthermore, C-EDF was also highly efficient, which also confirms the measurement presented by Brandenburg et al. [2008].

In real-time simulation, the simulation data is also produced in real-time. Thus, efficient storing of simulation data comes into consideration for reduction of the entire load of the simulator. For instance, the simulation process runs much faster when the generated data is stored into system swap memory rather than in VFS (Virtual File System). Depending on the tasksets to be simulated, if the generated data is stored in the network file system, the total simulation overhead is much higher than that in any of the previous cases. As a consequence of this simulation overhead, simulation data might be lost, which leads to the variation of the actual result from the expected one.

There are numerous directions for future work. In particular, given our investigation of the scheduling algorithms, scheduling algorithms BF and BF^2 could be implemented on both conventional SMPs, i.e., multi-core x86 machine, as well as a multi-core embedded platform, i.e., multi-core ARM architecture. Also, the enhancement of the simulation test-bed with more convenient data structure that could help generate simulation data of much smaller size would allow to conduct simulation on systems with much smaller form factor, thus facilitating the ongoing research on multi-core scheduling techniques on embedded architectures.

Bibliography

G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 2nd ed. Springer, 2004.

A. Mohammadi and S. G. Akl, “Technical report no. 2005-499 scheduling algorithms for real-time systems ,” 2005.

Nelissen, Geoffrey, Su, Hang, Guo, Yifeng, Zhu, Dakai, Nélis, Vincent, Goossens, and Joël, “An optimal boundary fair scheduling,” *Real-Time Systems*, pp. 1–53, 2014. [Online]. Available: http://dx.doi.org/10.1007/s11241-014-9201-0

P. Yiqiao, Z. Qingguo, S. Kairui, and W. Zhangjin, “Various freed multi-cores rtos based linux,” in *IT in Medicine and Education, 2008. ITME 2008. IEEE International Symposium on*, Dec 2008, pp. 900–905.

ChronOS, “Chronos real-time linux,” 2013. [Online]. Available: http://chronoslinux.org/wiki/Main_Page

K. Juvva, “Real-time systems ,” 1998. [Online]. Available: http://www.ece.cmu.edu/~koopman/des_s99/real_time/

A. Burns and A. Wellings, *Real-Time Systems and Programming Languages: Ada 95, real-time Java and real-time POSIX*, 3rd ed. Harlow, UK: Addison-Wesley, 2001.

R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/1978802.1978814

P. Marwedel, “Embedded system design,” Dordrecht, The Netherlands, 2006.

P. L. Holman, “On the implementation of pfair-scheduled multiprocessor systems,” Ph.D. dissertation, 2004, aAI3140333.

S. Baruah and N. Fisher, “The partitioned multiprocessor scheduling of sporadic task systems,” in *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, Dec 2005, pp. 9 pp.–329.

K. Ramamritham, J. Stankovic, and P.-F. Shiah, “Efficient scheduling algorithms for real-time multiprocessor systems,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 1, no. 2, pp. 184–194, Apr 1990.

M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York, NY, USA: W. H. Freeman & Co., 1979.

C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973. [Online]. Available: http://doi.acm.org/10.1145/321738.321743

J. Leung and J. Whitehead, “On the complexity of fixed-priority scheduling of periodic real-time tasks,” Performance Evaluation, Vol. 2, No. 4, pp. 237—250, December 1982.

M. L. Dertouzos, “Control robotics: The procedural control of physical processes,” in *In Proceedings of the International Federation for Information Processing Working Conference on Data Semantics.807–813.*, 1974.

S. Dhall and C. Liu, “On a real-time scheduling problem. ; 26: 127-140.” Operations Research ; 26, 1, 127–140, 1978.

Oh, Yingfeng, Son, and S. H., “Tight performance bounds of heuristics for a real-time scheduling problem,” Charlottesville, VA, USA, Tech. Rep., 1993.

Y. Oh and S. H. Son, “Allocating fixed-priority periodic tasks on multiprocessor systems,” *Real-Time Syst.*, vol. 9, no. 3, pp. 207–239, Nov. 1995. [Online]. Available: http://dx.doi.org/10.1007/BF01088806

A. Burchard, J. Liebeherr, Y. Oh, and S. Son, “New strategies for assigning real-time tasks to multiprocessor systems,” *Computers, IEEE Transactions on*, vol. 44, no. 12, pp. 1429–1442, Dec 1995.

B. Andersson, S. Baruah, and J. Jonsson, “Static-priority scheduling on multiprocessors,” in *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, Dec 2001, pp. 193–202.

D.-I. Oh and T. P. Bakker, “Utilization bounds for n-processor rate monotonescheduling with static processor assignment,” *Real-Time Syst.*, vol. 15, no. 2, pp. 183–192, Sep. 1998. [Online]. Available: http://dx.doi.org/10.1023/A:1008098013753

B. Andersson and J. Jonsson, “The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%,” in *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, July 2003, pp. 33–40.

J. Lopez, M. Garcia, J. Diaz, and D. Garcia, “Worst-case utilization bound for edf scheduling on real-time multiprocessor systems,” in *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, 2000, pp. 25–33.

S. Baruah and N. Fisher, “The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems,” *Computers, IEEE Transactions on*, vol. 55, no. 7, pp. 918–923, July 2006.

S. K. Baruah and N. W. Fisher, “The partitioned dynamic-priority scheduling of sporadic task systems,” *Real-Time Syst.*, vol. 36, no. 3, pp. 199–226, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1007/s11241-007-9022-5

B. Andersson and J. Jonsson, “Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition,” in *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, 2000, pp. 337–346.

A. Srinivasan and S. Baruah, “Deadline-based scheduling of periodic task systems on multiprocessors,” *Information Processing Letters*, vol. 84, no. 2, pp. 93 – 98, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020019002002314

J. Goossens, S. Funk, and S. Baruah, “Priority-driven scheduling of periodic task systems on multiprocessors,” *Real-Time Systems*, vol. 25, no. 2-3, pp. 187–205, 2003. [Online]. Available: http://dx.doi.org/10.1023/A%3A1025120124771

T. Baker, “An analysis of edf schedulability on a multiprocessor,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, no. 8, pp. 760–768, Aug 2005.

M. Bertogna, M. Cirinei, and G. Lipari, “Improved schedulability analysis of edf on multiprocessor platforms,” in *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*, July 2005, pp. 209–218.

T. P. Baker and S. K. Baruah, “Schedulability analysis of multiprocessor sporadic task systems,” in *Handbook of Realtime and Embedded Systems.* CRC Press, 2007.

M. Bertogna, “Real-time scheduling analysis for multiprocessor platforms,” Ph.D. dissertation, Scuola Superiore Sant’Anna, Pisa, Italy, 2007.

S. Baruah and J. Goossens, “Rate-monotonic scheduling on uniform multiprocessors,” in *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, May 2003, pp. 360–366.

M. Bertogna, M. Cirinei, and G. Lipari, “New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors,” in *Proceedings of the 9th International Conference on Principles of Distributed Systems*, ser. OPODIS’05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 306–321. [Online]. Available: http://dx.doi.org/10.1007/11795490_24

B. Andersson and J. Jonsson, “Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling,” in *Proceedings of the 21st IEEE Real-Time Systems Symposium – Work-in-Progress session, Orlando, Florida, November 29, 2000, s. 53–56*, 2000.

S. Baruah and N. Fisher, “Global static-priority scheduling of sporadic task systems on multiprocessor platforms.” In Proceedings of the International Conference on Parallel and Distributed Computing and Systems., 2006.

S. K. Baruah, “Schedulability analysis of global deadline monotonic scheduling,” Tech. rep., University of North Carolina, Chapel Hill, NC.http://www.cs.unc.edu/ baruah/Pubs.shtml, Tech. Rep., 2007.

N. Fisher, “The multiprocessor real-time scheduling of general task systems,” Ph.D. dissertation, Department of Computer Science, The University of North Carolina at Chapel Hill, Chapel Hill, NC, 2007.

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

J. Anderson and A. Srinivasan, “Mixed pfair/erfair scheduling of asynchronous periodic tasks,” in *Real-Time Systems, 13th Euromicro Conference on, 2001.*, 2001, pp. 76–85.

J. H. Anderson and A. Srinivasan, “A new look at pfair priorities,” Technical Report TR00-023,Departement of Computer Science, University of North Carolina, Tech. Rep., September 1999.

A. Srinivasan and J. H. Anderson, “Optimal rate-based scheduling on multiprocessors,” in *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, ser. STOC ’02. New York, NY, USA: ACM, 2002, pp. 189–198. [Online]. Available: http://doi.acm.org/10.1145/509907.509938

S. Baruah, J. Gehrke, and C. Plaxton, “Fast scheduling of periodic tasks on multiple resources,” in *Parallel Processing Symposium, 1995. Proceedings., 9th International*, Apr 1995, pp. 280–288.

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.

P. Holman and J. H. Anderson, “Adapting pfair scheduling for symmetric multiprocessors,” *J. Embedded Comput.*, vol. 1, no. 4, pp. 543–564, Dec. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1233791.1233800

D. Zhu, D. Mosse, and R. Melhem, “Multiple-resource periodic scheduling problem: how much fairness is necessary?” in *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, Dec 2003, pp. 142–151.

H. Kim and Y. Cho, “A new fair scheduling algorithm for periodic tasks on multiprocessors,” *Information Processing Letters*, vol. 111, no. 7, pp. 301 – 309, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020019010003996

H. Cho, B. Ravindran, and E. Jensen, “An optimal real-time scheduling algorithm for multiprocessors,” in *Real-Time Systems Symposium, 2006. RTSS ’06. 27th IEEE International*, Dec 2006, pp. 101–110.

K. Funaoka, S. Kato, and N. Yamasaki, “Work-conserving optimal real-time scheduling on multiprocessors,” in *Real-Time Systems, 2008. ECRTS ’08. Euromicro Conference on*, July 2008, pp. 13–22.

S. Funk, “Lre-tl: An optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines,” *Real-Time Syst.*, vol. 46, no. 3, pp. 332–359, Dec. 2010. [Online]. Available: http://dx.doi.org/10.1007/s11241-010-9109-2

S. K. Lee, “On-line multiprocessor scheduling algorithms for real-time tasks,” in *TENCON ’94. IEEE Region 10’s Ninth Annual International Conference. Theme: Frontiers of Computer Technology. Proceedings of 1994*, Aug 1994, pp. 607–611 vol.2.

S. Kato and N. Yamasaki, “Global edf-based scheduling with efficient priority promotion,” in *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA ’08. 14th IEEE International Conference on*, Aug 2008, pp. 197–206.

B. Brandenburg, J. Calandrino, and J. Anderson, “On the scalability of real-time scheduling algorithms on multicore platforms: A case study,” in *Real-Time Systems Symposium, 2008*, Nov 2008, pp. 157–169.

M. Dellinger, A. Lindsay, and B. Ravindran, “An experimental evaluation of the scalability of real-time scheduling algorithms on large-scale multicore platforms,” *J. Exp. Algorithmics*, vol. 17, pp. 4.3:4.1–4.3:4.22, Oct. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133803.2345677

"Ieee standard for information technology- standardized application environment profile (aep)-posix realtime and embedded application support," pp. i–164, 2004.

MARTE, "Marte," 2011. [Online]. Available: http://marte.unican.es/

W. Betz, M. Cereia, and I. Bertolotti, "Experimental evaluation of the linux rt patch for real-time applications," in *Emerging Technologies Factory Automation, 2009. ETFA 2009. IEEE Conference on*, Sept 2009, pp. 1–4.

A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio, "Performance comparison of vxworks, linux, rtai, and xenomai in a hard real-time application," *Nuclear Science, IEEE Transactions on*, vol. 55, no. 1, pp. 435–439, Feb 2008.

RTAI, "Rtai home page," 2014. [Online]. Available: https://www.rtai.org/

Xenomai, "Xenomai home page," 2014. [Online]. Available: http://www.xenomai.org/

D. P. Bovet and M. Cesati, *Understanding the Linux Kernel.* Beijing: O'Reilly, 2001. [Online]. Available: http://swbplus.bsz-bw.de/bsz088883213cov.htm

Lehrbaum and Rick, "Using linux in embedded and real-time systems," *Linux J.*, vol. 2000, no. 75es, Jul. 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=349516.349542

S. Bicer, F. Pilhofer, G. Bardouleau, and J. Smith, "Nextgeneration hard real-time on posix-based linux," 2006.

V. Yodaiken, "The rtlinux manifesto," in *In Proc. of The 5th Linux Expo*, 1999.

S. Kairui, B. Shuwei, Z. Qingguo, N. Mc, and L. Lian, "Analyzing rtlinux/gpl source code for education," *8th Real Time Linux Workshop*, 2006.

RTLinux, "Rtlinux," 1999. [Online]. Available: http://en.wikipedia.org/wiki/RTLinux

P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo, "A new kernel approach for modular real-time systems development," in *Real-Time Systems, 13th Euromicro Conference on, 2001.*, 2001, pp. 199–206.

L. Abeni and G. Buttazzo, "Support for dynamic qos in the hartik kernel," in *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, 2000, pp. 65–72.

S.Ha.R.K, "S.ha.r.k.: Soft hard real-time kernel," 2008. [Online]. Available: http://shark.sssup.it/

P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour, "Diapm-rtai position pape," *2nd Real Time Linux Workshop, Orlando, FL*, November 2000.

P. Mantegazza, E. L. Dozio, and S. Papacharalambous, "Rtai: Real time application interface," *Linux J.*, vol. 2000, no. 72es, Apr. 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=348554.348564

P. Gerum, "Xenomai - implementing a rtos emulation framework on gnu/linux," 2004. [Online]. Available: http://www.xenomai.org/

M. Masmano, I. Ripoll, and A. Crespo, “An overview of the xtratum nanokernel,” in *Workshop on Operating Systems Platforms for Embedded Real-Time applications*, 2005.

P. S., M. M., R. Ismael, and C. A., “Partikle os, a replacement of the core of rtlinux,” in *9th Real-Time Linux Workshop*, 2007.

M. Dellinger, P. Garyali, and B. Ravindran, “Chronos linux: A best-effort real-time multiprocessor linux kernel,” in *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, June 2011, pp. 474–479.

J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, “Litmus^{RT} : A testbed for empirically comparing real-time multiprocessor schedulers,” in *Real-Time Systems Symposium, 2006. RTSS ’06. 27th IEEE International*, Dec 2006, pp. 111–126.

LITMUS^{RT}, “Litmus^{RT} : Testbed for multiprocessor scheduling in real-time system,” 2013. [Online]. Available: http://www.litmus-rt.org/

E. Jensen and J. Northcutt, “Alpha: a nonproprietary os for large, complex, distributed real-time systems,” in *Experimental Distributed Systems, 1990. Proceedings., IEEE Workshop on*, Oct 1990, pp. 35–41.

Y.-C. Wang and K.-J. Lin, “Implementing a general real-time scheduling framework in the red-linux real-time kernel,” in *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, 1999, pp. 246–255.

B. B. Brandenburg, A. D. Block, J. M. Calandrino, U. Devi, H. Leontyev, and J. H. Anderson, “Litmus rt: A status report,” 2007.

B. Brandenburg and J. Anderson, “Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors,” in *Real-Time Systems, 2007. ECRTS ’07. 19th Euromicro Conference on*, July 2007, pp. 61–70.

J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson, “Rtos support for multicore mixed-criticality systems,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, April 2012, pp. 197–208.

B. B. Brandenburg, “Scheduling and locking in multiprocessor real-time operating systems,” Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.

Schedcat, “The schedulability test collection and toolkit,” 2011. [Online]. Available: http://www.litmus-rt.org/

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature