Supporting Intra-Task Parallelism in Real-Time Multiprocessor Systems

José Carlos Nunes da Fonseca

Dissertação para a obtenção do Grau de Mestre em
Engenharia Informática

Área de especialização em **Sistemas Gráficos e Multimédia**

Orientador
Doutor Luís Miguel Pinho Nogueira

Júri
Presidente: Doutor Luís Miguel Moreira Lino Ferreira,
Professor Adjunto no Departamento de Engenharia Informática
do Instituto Superior de Engenharia do Porto
Vogais: Doutor Luís Miguel Rosário da Silva Pinho,
Professor Coordenador no Departamento de Engenharia Informática
do Instituto Superior de Engenharia do Porto
Doutor Luís Miguel Pinho Nogueira,
Professor Adjunto no Departamento de Engenharia Informática
do Instituto Superior de Engenharia do Porto

Porto, Outubro de 2012

Resumo Alargado

Os sistemas de tempo real modernos geram, cada vez mais, cargas computacionais pesadas e dinâmicas, começando-se a tornar pouco expectável que sejam implementados em sistemas uniprocessador. Na verdade, a mudança de sistemas com um único processador para sistemas multiprocessador pode ser vista, tanto no domínio geral, como no de sistemas embebidos, como uma forma eficiente, em termos energéticos, de melhorar a performance das aplicações.

Simultaneamente, a proliferação das plataformas multi-processador transformaram a programação paralela num tópico de elevado interesse, levando o paralelismo dinâmico a ganhar rapidamente popularidade como um modelo de programação. A ideia, por detrás deste modelo, é encorajar os programadores a exporem todas as oportunidades de paralelismo através da simples indicação de potenciais regiões paralelas dentro das aplicações. Todas estas anotações são encaradas pelo sistema unicamente como sugestões, podendo estas serem ignoradas e substituídas, por construtores sequenciais equivalentes, pela própria linguagem. Assim, o modo como a computação é na realidade subdividida, e mapeada nos vários processadores, é da responsabilidade do compilador e do sistema computacional subjacente.

Ao retirar este fardo do programador, a complexidade da programação é consideravelmente reduzida, o que normalmente se traduz num aumento de produtividade. Todavia, se o mecanismo de escalonamento subjacente não for simples e rápido, de modo a manter o *overhead* geral em níveis reduzidos, os benefícios da geração de um paralelismo com uma granularidade tão fina serão meramente hipotéticos.

Nesta perspetiva de escalonamento, os algoritmos que empregam uma política de *work-stealing* são cada vez mais populares, com uma eficiência comprovada em termos de tempo, espaço e necessidades de comunicação. Contudo, estes algoritmos não contemplam restrições temporais, nem outra qualquer forma de atribuição de prioridades às tarefas, o que impossibilita que sejam diretamente aplicados a sistemas de tempo real. Além disso, são tradicionalmente implementados no *runtime* da linguagem, criando assim um sistema de escalonamento com dois níveis, onde a previsibilidade, essencial a um sistema de tempo real, não pode ser assegurada.

Nesta tese, é descrita a forma como a abordagem de *work-stealing* pode ser resenhada para cumprir os requisitos de tempo real, mantendo, ao mesmo tempo, os seus princípios fundamentais que tão bons resultados têm demonstrado. Muito resumidamente, a única fila de gestão de processos convencional (*deque*) é substituída por uma fila de *deques*, ordenada de forma crescente por prioridade das tarefas. De seguida, aplicamos por cima o conhecido algoritmo de escalonamento dinâmico G-EDF, misturamos as regras de ambos, e assim nasce a nossa proposta:

o algoritmo de escalonamento RTWS.

Tirando partido da modularidade oferecida pelo escalonador do Linux, o RTWS é adicionado como uma nova classe de escalonamento, de forma a avaliar na prática se o algoritmo proposto é viável, ou seja, se garante a eficiência e escalonabilidade desejadas. Modificar o núcleo do Linux é uma tarefa complicada, devido à complexidade das suas funções internas e às fortes interdependências entre os vários subsistemas. Não obstante, um dos objetivos desta tese era ter a certeza que o RTWS é mais do que um conceito interessante. Assim, uma parte significativa deste documento é dedicada à discussão sobre a implementação do RTWS e à exposição de situações problemáticas, muitas delas não consideradas em teoria, como é o caso do desfasamento entre vários mecanismo de sincronização.

Os resultados experimentais mostram que o RTWS, em comparação com outro trabalho prático de escalonamento dinâmico de tarefas com restrições temporais, reduz significativamente o *overhead* de escalonamento através de um controlo de migrações, e mudanças de contexto, eficiente e escalável (pelo menos até 8 CPUs), ao mesmo tempo que alcança um bom balanceamento dinâmico da carga do sistema, até mesmo de uma forma não custosa. Contudo, durante a avaliação realizada foi detetada uma falha na implementação do RTWS, pela forma como facilmente desiste de roubar trabalho, o que origina períodos de inatividade, no CPU em questão, quando a utilização geral do sistema é baixa.

Embora o trabalho realizado se tenha focado em manter o custo de escalonamento baixo e em alcançar boa localidade dos dados, a escalonabilidade do sistema nunca foi negligenciada. Na verdade, o algoritmo de escalonamento proposto provou ser bastante robusto, não falhando qualquer meta temporal nas experiências realizadas. Portanto, podemos afirmar que alguma inversão de prioridades, causada pela sub-política de roubo BAS, não compromete os objetivos de escalonabilidade, e até ajuda a reduzir a contenção nas estruturas de dados. Mesmo assim, o RTWS também suporta uma sub-política de roubo determinística: PAS. A avaliação experimental, porém, não ajudou a ter uma noção clara do impacto de uma e de outra. No entanto, de uma maneira geral, podemos concluir que o RTWS é uma solução promissora para um escalonamento eficiente de tarefas paralelas com restrições temporais.

Palavras-chave: Sistemas multi-processador, escalonamento de tempo real, *intra-task parallelism*, EDF, *work-stealing*, Linux

Abstract

Multiple programming models are emerging to address the increased need for dynamic task-level parallelism in applications for multi-core processors and shared-memory parallel computing, presenting promising solutions from a user-level perspective. Nonetheless, while high-level parallel languages offer a simple way for application programmers to specify parallelism in a form that easily scales with problem size, they still leave the actual scheduling of tasks to be performed at runtime. Therefore, if the underlying system cannot efficiently map those tasks on the available cores, the benefits will be lost.

This is particularly important in modern real-time systems as their average workload is rapidly growing more parallel, complex and computing-intensive, whilst preserving stringent timing constraints. However, as the real-time scheduling theory has mostly been focused on sequential task models, a shift to parallel task models introduces a completely new dimension to the scheduling problem.

Within this context, the work presented in this thesis considers how to dynamically schedule highly heterogeneous parallel applications that require real-time performance guarantees on multi-core processors. A novel scheduling approach called RTWS is proposed. RTWS combines the G-EDF scheduler with a priority-aware work-stealing load balancing scheme, enabling parallel real-time tasks to be executed on more than one processor at a given time instant. Two stealing sub-policies have arisen from this proposal and their suitability is discussed in detail.

Furthermore, this thesis describes the implementation of a new scheduling class in the Linux kernel concerning RTWS, and extensively evaluate its feasibility. Experimental results demonstrate the greater scalability and lower scheduling overhead of the proposed approach, comparatively to an existing real-time deadline-driven scheduling policy for the Linux kernel, as well as reveal its better performance when considering tasks with intra-task parallelism than without, even for short-living applications.

We show that busy-aware stealing is robust to small deviations from a strict priority schedule and conclude that some priority inversion may be actually acceptable, provided it helps reduce contention, communication, synchronisation and coordination between parallel threads.

Keywords: Multiprocessor systems, real-time scheduling, intra-task parallelism, work-stealing, EDF, Linux

Acknowledgements

I am indebted to many people who helped me in manifold large and small ways over the last year. First of all, I would like to thank my advisor, Luís Nogueira, for his infinite availability, wisdom, fellowship, and guidance throughout the course of this thesis. Luís is one of very few professors I have met who is just as human and real as he is dedicated and accomplished.

I would also like to express my deepest thanks and appreciation to Cláudio Maia, Paulo Baltarejo, Miguel Pinho and André Pedro. André Pedro, for his Latex lessons; Miguel Pinho, for his advices and experience; Paulo Baltarejo, for being my Linux kernel's mentor; and Cláudio Maia, well, his invaluable contribution cannot be put into words. Without all their unrelenting support, I probably would not have made my way here. Many thanks are also due to the friendly people at CISTER, and in particular Inês Almeida who assisted me in all bureaucratic stuff.

Outside school, foremost, I am grateful to my parents Elvira and Manuel, and my sister Helena, for their endless support, encouragement, understanding and love. A special thanks goes to my closest friends, mainly Marcos Sousa, Tiago Ribeiro, and Ana Silva, for keeping me sane. When the work was driving me crazy, you guys were there to provided me what I needed most: relax and fun.

Last but not least, this work was partially supported by the EU ARTEMIS JU funding, within RECOMP project, ref. ARTEMIS/0202/2009, JU Grant nr. 100202.

Contents

List of Figures

List of Tables

Acronyms

BAS	Busy-Aware Stealing
BF	Best-Fit
CFS	Completely Fair Scheduler
DAG	Directed Acyclic Graph
DM	Deadline Monotonic
FF	First-Fit
FIFO	First-In First-Out
FP	Fixed-Priority
G-EDF	Global Earliest Deadline First
GPL	GNU General Public License
GPOS	General-Purpose Operating System
HRT	Hard Real-Time
JLFP	Job-Level Fixed-Priority
LIFO	Last-In First-Out
OS	Operating System
PAS	Priority-Aware Stealing
POSIX	Portable Operating System Interface
RM	Rate Monotonic
RTOS	Real-Time Operating System
RTS	Real-Time System
RTWS	Real-Time Work-Stealing

SMP	Symmetric Multiprocessing
SPMD	Single-Programming-Multiple-Data
SRT	Soft Real-Time
WCET	Worst-Case Execution Time

Chapter 1

Introduction

It is expected that parallel workloads to become rather common as multi-core platforms become ubiquitous. In contrast to prior work on real-time scheduling of parallel workloads, this thesis considers a more general model of parallel real-time tasks where dynamically generated threads can take arbitrarily different amounts of time to execute. It proposes a novel scheduling policy that combines the Global Earliest Deadline First (G-EDF) scheduler with a priority-based work-stealing policy, allowing parallel real-time tasks to be executed in more than one processor at a given time. To the best of our knowledge, we are the first to: (i) deal with real-time priorities in a work-stealing scheduler; and (ii) to actually implement support for parallel real-time computations in the Linux kernel.

1.1 Motivation

The advent and ubiquity of multi-core technologies has opened the door for a wide-range of general-purpose applications to effectively harness the increasing processing capability through parallelization. From a user-level perspective, dynamic intra-task parallelism is steadily gaining popularity as a programming model for multi-core processors. Parallelism is easily expressed by spawning threads that the implementation is allowed, but not mandated, to execute in parallel, using frameworks such as OpenMP [ARB], Cilk [Frigo et al., 1998], Intel's Parallel Building Blocks [Corporation, a], Java Fork-join Framework [Lea, 2000], Microsoft's Task Parallel Library [Corporation, b], or StackThreads/MP [Taura et al., 1999].

These high-level parallel frameworks seek to reduce the complexity of multicore programming by giving programmers abstract execution models, such as implicit threading, where programmers annotate their applications to suggest the parallel decomposition. Implicitly-threaded applications, however, do not specify the actual decomposition of computations or the mapping from computations to cores[1]. In fact, the annotations act simply as hints that can be ignored and safely replaced with sequential counterparts. The parallel decomposition itself is the responsibility of the language implementation and, more specifically, of the runtime scheduler. Further-

[1] In the context of this work, we will use the terms processor, core and CPU interchangeably.

more, the actual scheduling depends on the underlying system which by turn heavily influences any application speed up.

Unfortunately, scalable performance is only one facet of the problem in embedded multi-core real-time platforms. Predictability and computational efficiency are often conflicting goals, as many performance enhancement techniques aim at boosting the average expected execution time, without considering potentially adverse consequences on worst-case execution time. Thus, applications with strong predictability requirements often tend to underuse hardware resources [Colin and Petters, 2003]. Such a waste of resources can only be justified for very critical systems in which a single missed deadline may cause catastrophic consequences.

Therefore, the growing importance of parallel programming models introduce a new dimension to real-time multi-core scheduling, with many open issues to be studied. Recent works on real-time scheduling of parallel tasks define a task as a collection of several regions, both sequential and parallel [Lakshmanan et al., 2010, Saifullah et al., 2011]. A task always starts with a sequential region, which then forks into several parallel independent threads (the parallel region) that finally join in another sequential region. However, these models require that each region of a task contains threads of execution that are of equal length.

In contrast, in this thesis we consider a more general model of parallel real-time tasks where threads can take arbitrarily different amounts of time to execute. That is, different regions of the same parallel task can contain different numbers of threads, regions can contain more threads than the number of cores, and threads can have arbitrarily different execution needs. Therefore, this model is more portable.

Indeed, there are many applications for which this condition holds, and it is this kind of dynamic and irregular parallelism that is of primary interest for us. The distribution of work and data in such applications cannot be characterised a priori because these quantities are input-dependent and evolve with the computation itself. In practice, such real-time applications span a wide spectrum, including radar tracking, autonomous driving, and video surveillance. Applications with these properties pose significant challenges for high-performance parallel implementations, where equal distribution of work over processors and locality of reference are desired within each processor. Nevertheless, as the problem sizes scale and processor speeds saturate, the only way to meet deadlines in such systems is to parallelize the computation.

Implicit threading also encourage the programmer to divide the program into short-living threads because doing so increases the flexibility to distribute work evenly across processors. The downside of such fine-grained parallelism is that the total scheduling cost can be significant. The best way to reduce the total scheduling cost is to find the sub-costs that matter most and focus on reducing them.

One of the simplest, yet best-performing, dynamic load-balancing algorithms for shared-memory architectures is work-stealing [Blumofe and Leiserson, 1999]. The principle of work-stealing is that idle cores, which have no useful work to do, should bear most of the scheduling costs, and busy cores, which have useful work to do, should focus on finishing that work. Blumofe and Leiserson have theoretically proven that the work-stealing algorithm is optimal for scheduling fully-strict computations, *i.e.* computations in which all join edges from a thread go to its parent

thread in the spawn tree [Blumofe and Leiserson, 1999]. Under this assumption, an application running on P processors achieves P-fold speed-up in its parallel part, using at most P times more space than when running on one CPU. These results are also supported by experiments [Saha et al., 2007].

However, the need to support task priorities fundamentally distinguishes the problem at hand in this thesis from other work-stealing choices previously proposed in the literature [Guo et al., 2010, Vrba et al., 2009, 2010]. With classical work-stealing, threads waiting for execution in a deque may be repressed by new threads, which are enqueued at the bottom of the worker's deque. As such, a thread at the top of a deque might never be executed if all workers are busy. Consequently, there is no upper bound on the response time of a multi-threaded real-time job.

1.2 Contributions

Motivated by these observations, the work presented throughout this thesis breaks new ground in several ways, focusing on supporting intra-task parallelism in real-time multiprocessor systems, both in theory and practise:

- While several others have previously considered work-stealing as a load balancing mechanism for parallel computations, we are the first to do so considering different task priorities.
- We propose Real-Time Work-Stealing (RTWS), a novel real-time scheduling approach that combines the G-EDF scheduler with a priority-based locality-aware work-stealing scheme, allowing parallel real-time tasks to be executed in more than one processor at a given time instant. To the best of our knowledge, no research has ever focused on this subject.
- Our work is the first to actually implement support for parallel real-time computations in the Linux kernel through the development of a new scheduling class (SCHED_RTWS) and respective system calls. At the time of this writing, neither any RTOS natively supports such scheduling nor any known extension does so.

Importantly, the research work described in this thesis has resulted in two scientific publications. The paper entitled *Real-Time Scheduling of Parallel Tasks in the Linux Kernel* [Fonseca et al., 2012] has been published in the 4th Informatics Symposium (INForum 2012), while the paper entitled *Dynamic Global Scheduling of Parallel Real-Time Tasks* [Nogueira et al., 2012] has been accepted at the 10th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC 2012).

1.3 Institutional support

This research work was developed in the context of the RECOMP European project, from the ARTAMIS program, held at CISTER (Research Centre in Embedded Real-Time Computing Systems). CISTER is a top-ranked research unit associated with the INESC-TEC, from the School of Engineering (ISEP) of the Polytechnic Institute of Porto (IPP), Portugal. The research unit focuses its

activity in the analyses, design and implementation of real-time and embedded computing systems. Back in the 2004 evaluation process, CISTER was the only research unit in Portugal, in the areas of computer and electrical engineering and computer science, to be awarded the top-level rank of Excellent. This outstanding rating was confirmed in the last evaluation process (2007). CISTER has grown to become one of the leading European research units in the area, contributing with seminal research works in numerous subjects. Since mid-2011, CISTER is an autonomous research unit associated to INESC-TEC.

1.4 Outline

The rest of this document is structured as follows:

- Chapter 2 introduces the real-time concepts and scheduling theory on which this work is fundamentally based, with emphasis on the periodic task model and EDF algorithms.
- Chapter 3 is devoted to provide the remainder necessary background directly related to the main contributions of this thesis. It starts by discussing parallel computations and how they can be expressed, modelled and scheduled, with particular focus on the work-stealing scheduler. It continues by analysing the current modular framework of the Linux scheduler, and it finishes by covering briefly relevant real-time implementations on the Linux kernel.
- Chapter 4 discusses design and implementation of the RTWS scheduler. First, it dives deep in the state-of-art of parallel real-time scheduling, with some insights on the current challenges in supporting task-level parallelism in real-time multiprocessor systems being given as well. Then it presents our system model and addresses the problem of adapting work-stealing to real-time. The major rules and flow of RTWS are described next. Last but not least, it explains how this scheduling algorithm was implemented in the Linux kernel, and how one can use it from user-space.
- In Chapter 5, we evaluate the scalability, effectiveness and efficiency of our RTWS implementation, mostly by comparing it to other real-time scheduling policy through experimental results. The nature of the experiments is also explained herein.
- Finally, Chapter 6 sums up results, offers some concluding remarks and suggests possible future extensions to our work.

Chapter 2

Real-Time Systems

Real-time computing is becoming increasingly important and pervasive, as more and more industries, infrastructures, and even ordinary people depend on them. Naturally, with the general proliferation of multi-core platforms, real-time applications started to be massively deployed on such platforms. A key factor for that, among other reasons, is the considerable boost in processing capacity in a relatively cheap, small, and low power consuming chip. Therefore, they offer an opportunity to maximise performance and, through parallelism, execute more complex and computing-intensive tasks whose stringent timing constraints cannot be guaranteed on uniprocessor systems.

However, most research in traditional multiprocessor real-time scheduling is still limited to sequential task models and ignore task-level parallelism. Such model scales poorly and is unable to effectively exploit the potential of multi-core platforms. Thus, a dramatic change in programming models and scheduling paradigms is undeniably demanded.

This chapter discusses representative research efforts and gives a special focus to the real-time scheduling theory, as both are directly related to the main contributions of this thesis. We also briefly present real-time systems' concepts and contextualise them within the conducted work.

2.1 Definition

A Real-Time System (RTS) is any information processing system where the correctness of each computation depends not only on the logical results it provides but also on the time instant at which these results are produced [Stankovic, 1988]. A late response time (*i.e.* the time taken for the system to generate output from some associated input) is as bad as a wrong response since it may provoke an unexpected behaviour, which might lead to a system failure. Hence, RTSs must respond in a timely predictability way to externally generated input stimuli, even under transient overload. An automobile airbag system, one of the most safety-critical features in a modern car, is a simple example of a real-time computing system — the strict real-time constraint in this

system is the time interval in which the airbag must be deployed in order to prevent the driver from getting severely hurt. No matter what non-critical operation is taking place at that instant, the RTS will put it on hold and will immediately deploy the airbag as soon as it receives a signal from the sensors detecting the collision.

In contrast, a system is said to be *non-real-time* whenever one cannot guarantee a response time under any circumstance, even if rather often the outcome respects the timing boundaries. Analogously, if a car is equipped with a non-RTS it might deploy the airbag after finishing the request to power the stereo, which by coincidence happened to come right before the self-triggered critical request. Needless to say, an airbag system deployed even 0.01 seconds later than the demanded time may have catastrophic consequences. In fact, for certain RTSs few microseconds separate the success from the disaster.

Nonetheless, a RTS is not a fast computing system, as oftentimes mistakenly deemed so. Its response time scale magnitude can indeed range from a microsecond in a radar data acquisition to an hour in a chemical reaction. Thus, no matter how fast hardware or algorithms are, its performance has to always be guaranteed against the characteristics of the surrounding execution environment. Here the key property is predictability, *i.e.* the logical and timing behaviour must be as deterministic as required to fulfill system specifications, and not speed. Undoubtedly high-speed computing helps to minimise the average response time of a task set or even to meet some stringent individual timeliness requirements, but it solely does not assure the overall system correctness.

Guaranteeing real-time performance, while most effectively exploiting the available resources, demands the appliance of efficient scheduling algorithms, properly supplemented by schedulability analysis or similar techniques. Such techniques must provably assure that timing constraints will always be met by a given scheduler during system's activity. For better understanding the scheduling theory referred all over this document, next section introduces scheduling and real-time terminology.

2.2 Terminology and periodic task model

The term *job* refers to a schedulable and executable unit of work. *Schedulable* means that it can be allocated to a resource (*e.g.* processor) in a particular sequence determined by the scheduling algorithm being used and it will meet its timing constraints. A set of related jobs defines a *task*, while a collection of tasks is called *task set*.

The necessary time to run a single job on a given platform is called *execution time*. The time instant at which a job is required to complete its execution is denominated as *deadline* or as absolute deadline, as it is successively calculated for each job. A relative deadline, in turn, is its maximum allowable response time. Additionally, the recurrent nature of real-time activities is expressed by a *period*. The period represents the expected time of arrival between jobs, whether they are cyclic or event-driven. The moment a job becomes available for execution is called *release time*. However, for the particular case of the first job release, that time instant is denominated *offset*.

2.2. TERMINOLOGY AND PERIODIC TASK MODEL

The fraction of one processor's capacity that must be allocated to a task is its *utilisation*. This does not mean, however, that a task can only execute on one processor. Straightforwardly, the sum of all tasks' utilisation within a task set gives its denoted *total utilisation*. A procedure that determines if a task set is schedulable under a given scheduling algorithm is a *schedulability test*. Whenever exists an algorithm able to deem a task set schedulable, this task set becomes *feasible*. A scheduling algorithm can be considered optimal if, on a m processors system, a task set with total utilisation at most m is schedulable.

Depending on the consequences of missing timing constraints, real-time tasks are commonly classified as either[1] *hard* or *soft*. An Hard Real-Time (HRT) task must always meet its deadline due to its critical nature where an overrun in response time may lead to a fatal flaw, *e.g.* loss of life or big financial damage. Hence, a judicious Worst-Case Execution Time (WCET) has to be assigned to them. When deadline violations are tolerable to a limited extent (*tardiness*[2] must be bounded to be schedulable), but not desirable, as they entail performance degradation, a task is said to be Soft Real-Time (SRT). This type of tasks does not require a execution time so rigid, therefore it employs an average execution time. For instance, the airbag feature mentioned above clearly fits in the former classification, whereas a multimedia interactive game suits the latter one, provided an underlying failure (perceived as sluggishness) does not have catastrophic consequences, although it results in a not smooth gameplay, and consequently in unsatisfied end-users.

Table 2.1: A summary of the periodic task model's constraints and notation

Notation	Interpretation	Constraint / Definition
τ	A task set	$\tau = \tau_1, \ldots, \tau_n$
τ_i	The i^{th} periodic task	$1 \leq i \leq n$
$J_{i,j}$	The j^{th} job of task τ_i	$j \geq 1$
J_i	An arbitrary job of T_i	
C_i	$\tau_i's$ per-job WCET	$C_i > 0$
O_i	$\tau_i's$ offset	$O_i \geq 0$
T_i	$\tau_i's$ period	$P_i > C_i$
D_i	$\tau_i's$ relative deadline	$D_i \geq C_i$
u_i	$\tau_i's$ utilisation	$u_i = C_i/T_i$
$a_{i,j}$	$J'_{i,j}s$ release time	$a_{i,j} \geq a_{i,j-1} + T_i$
$d_{i,j}$	$J'_{i,j}s$ absolute deadline	$d_{i,j} = a_{i,j} + D_i$
$f_{i,j}$	$J'_{i,j}s$ completion time	$f_{i,j} \geq a_{i,j}$

Besides criticalness, tasks can also be classified based on their periodicity. Tasks which exhibit irregular activations are called *aperiodic*, whilst *periodic* are the ones requiring symmetrical arrival times. Periodic tasks are typically used in control and signal-processing applications and often have hard deadlines, since they have to be executed at constant ratios for stability and update purposes. On the other hand, aperiodic tasks commonly have soft deadlines and are used to handle random processing requirements such as displaying activities. When aperiodic tasks have hard deadlines they are denominated *sporadic*. Note that for these tasks, period is replaced by a minimum interarrival time in order to enable deadlines' fulfillment [Mok, 1983]. In this thesis,

[1] More specific classifications can be found in the literature.

[2] Tardiness refers to how far after deadline a task has finished its execution.

we address a task model for parallel HRT tasks similiar to the periodic one, whose notation is shown in Table 2.1. Henceforth, every time we mention RTS we refer to HRT scenarios, unless we specifically say otherwise.

Furthermore, literature differentiates three levels of constraint on task deadlines:

- *Constrained deadlines* - Task deadlines cannot be greater than their periods ($D_i \leq T_i$).
- *Implicit deadlines* - All task deadlines must be equal to their periods ($D_i = T_i$).
- *Arbitrary deadlines* - Task deadlines may take any value.

When considering a RTS as a whole, there are several important aspects that should be taken into consideration in order to ensure the timeliness of all tasks with timing requirements. In particular, the Operating System (OS) plays a major role in the management of all concurrent activities running on a single or multiprocessor device, both taking care of task management, through the use of scheduling mechanisms that handle the priority of each task, and managing memory allocations, by taking into account the timing requirements of the tasks. When designing a RTS, every detail must be carefully analysed in order to make it as deterministic and predictable as possible, both in terms of time and space.

2.3 Real-time scheduling

In any multitasking RTS, scheduling is the fundamental component since it is responsible for: (i) providing an algorithm that defines a set of rules concerning how to commit resources (mostly processors) between tasks; (ii) establishing whether a temporal specification is guaranteed to be satisfied under such algorithm, through exhaustive worst-case behaviour analyses; (iii) maximising system utilisation; and (iv) ideally minimising each task's response time. Therefore, it is of paramount importance to understand its nomenclature, proposed approaches, and problems facing its theory for multi-core processor systems.

A standard set of simplifications are commonly assumed to eliminate every potential source of unpredictability when devising an algorithm and developing corresponding schedulability analysis:

- **Every task is independent** - besides processors, no hardware or software resources are shared.
- **Deterministic timing behaviour** - there is no drift on tasks' timing behaviour. Tasks are release at, and execute for, exactly the time they are supposed to.
- **Jobs do not self-suspend** - a pending job is always either executing or ready for execution.
- **No runtime overheads** - migrations, context switches and other scheduling decisions take negligible time or are subsumed into the WCET of each task.

Nevertheless, this idealised task behaviour does not hold in practise and it is indeed problematic. Although these simplifying assumptions definitely help to formally express an algorithm's logic and perform schedulability tests on it, as one very hardly would get anywhere if he tried to weight all unpredictable factors. Then, upon implementation, these can be attenuated by adding extra features to deal with them. In fact, as stated in Chapter 1, this implementation awareness is a driving motivation underlying the work presented herein.

Generally, scheduling algorithms are categorised as *static* or *dynamic*, depending on the method used for task priority assignment. Static schedulers, also known as Fixed-Priority (FP) schedulers, enact at design time a constant priority to each task, which is then applied to all of its jobs. In contrast, dynamic schedulers assign at runtime a priority directly to the jobs based on the current system state. Basically, this categorisation affects when and in what order each job shall execute.

Furthermore, scheduling algorithms can also be classified, as follows, according to when preemptions are enable.

- *Preemptive* - jobs may be preempted by higher priority ones at any time instant.
- *Non-preemptive* - preemption is not allowed and, therefore, once a job is scheduled for execution it will not be swapped out until completion.
- *Cooperative* - there are specific preemptable sections within a job execution.

In this thesis, we focus on dynamic and preemptive scheduling algorithms for implicit-deadline real-time tasks. Since we have also restricted our work to homogeneous multiprocessor systems (*i.e.* systems with identical processors), we only briefly address uniprocessor real-time scheduling for contextualisation and completeness. A detailed historical perspective of the most important research advances in this field can be found in [Sha et al., 2004].

The seminal research into uniprocessor real-time scheduling dates back to the late 1960s and early 1970s, and it was primarily applied to schedule computer programs during the first manned space flight to the moon [Liu, 1969, Liu and Layland, 1973]. Remarkably, Liu and Layland [1973] introduced provably optimal[3] static and dynamic algorithms for the scheduling of periodic tasks, which later became known as Rate Monotonic (RM) and EDF. respectively. During the 1980s and 1990s, these policies were improved to adopt more realistic models of synchronisation [Sha et al., 1990], timing constraints [Lehoczky, 1990] and overheads [Katcher et al., 1993], for example. Today, this theory can be considered mature and successfully put in practise for industrial purposes.

As a reference point, and since we have chosen G-EDF as our task-level policy, this section will relate with EDF schedulers, whenever feasible, to illustrate or describe how differently scheduling algorithms can be designed, extended, and implemented, and how that will affect the system's performance and its schedulability boundaries.

EDF is the most studied dynamic, or Job-Level Fixed-Priority (JLFP) as oftentimes referred, real-time scheduling algorithm. It is very intuitive, since it schedules in order of urgency. That

[3] Regarding the specific scenario it is intended to.

is, in contrast to RM which prioritises tasks based on their periods, EDF assigns priorities to tasks according to the deadlines of their current requests, in a form that the task with the nearest deadline becomes the highest priority task in the system and, therefore, the one to be selected for execution. Regardless of priorities changing at runtime, no manual assignment is required.

The acceptance test

$$u_{sum}(\tau) = \sum_{i}^{n} u_i \leq 1 \tag{2.1}$$

clearly shows that EDF is optimal in an HRT context as it fully utilises the processor capacity, unlike RM whose maximum demand of processor time is limited to $ln(2) \approx 69,3\%$. Moreover, EDF is also optimal for SRT constraints, seeing that a task set HRT schedulable implies bounded tardiness.

A simple example may clarify how EDF works. Let us consider the task set detailed in Table 2.2, which has four tasks and utilisation: $u_{sum}(\tau) = \frac{2}{13} + \frac{3}{13} + \frac{2}{15} + \frac{3}{17} = 72.2\%$. Fig. 2.1 shows the timeline execution for the first job of each task. The only task released at instant 0 is τ_4, so it starts executing immediately. At instant 1, τ_3 arrives with an earlier deadline. Since τ_4 needs more 2 times units to finish its instance, it is preempted by τ_3. It goes like this until instant 6, when τ_1 finishes his job. Now that the remaining three tasks are ready, the earliest deadline task is selected for execution: τ_3. The schedule goes on this descending way until instant 10 when the last first job terminates.

Table 2.2: A task set example for EDF schedule

Task	C_i	T_i	D_i	O_i
τ_1	2	11	11	3
τ_2	3	13	13	2
τ_3	2	15	15	1
τ_4	3	17	17	0

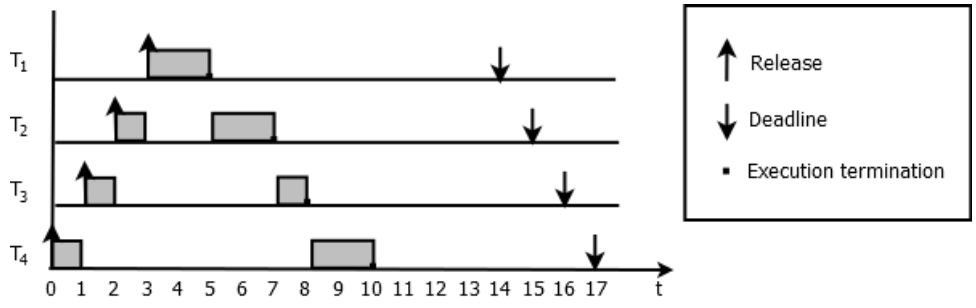


Figure 2.1: An EDF schedule example

Nonetheless, EDF is not preferable over RM for practical uses. One plausible reason is the conceptual difficulty associated to an efficient implementation of EDF [Short, 2010], mainly because it is not straightforward the mapping of deadlines to priority arrays or bitmaps pervasively used in OS for scheduling purposes, and when attempting to do so it demands frequent and costly recomputations. Supposed RM advantages in practise, namely less runtime overhead and

more predictability under overload, arose from misconceptions or specific situations as Buttazzo [2005] conclusively debunked. Hence, there is no reasonable justification, quite the contrary, for the absence of EDF-alike schedulers in a Real-Time Operating System (RTOS).

Unfortunately, multiprocessor real-time scheduling theory has not yet enjoyed such a success as it did on a uniprocessor. As early as in the 1969, Liu [1969] observed the intrinsic complexity of multi-core scheduling and how hardly uniprocessor algorithms could be extended to it:

> *"Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors."*

Table 2.3: A task set example causing the Dhall effect

Task	C_i	T_i	u_i
τ_1	2ϵ	1	$\rightarrow 0$
τ_2	2ϵ	1	$\rightarrow 0$
τ_3	1	$1+\epsilon$	$\rightarrow 1$

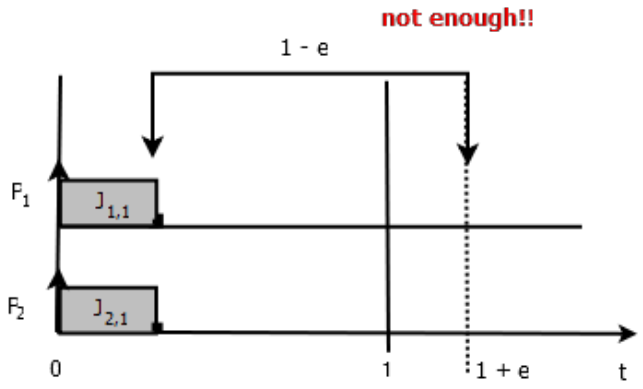
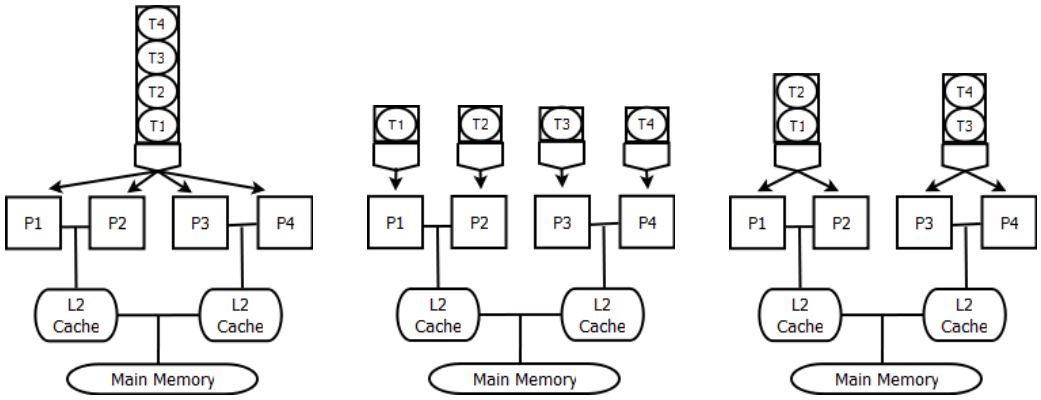


Figure 2.2: A Dhall effect schedule example

In fact, few years later, Dhall [1977] reported that when globally enforcing a RM or EDF scheme on a multi-core host, some task sets may miss deadlines even though low system utilisation is requested. To provide an understanding of the so-called *Dhall effect*, let us consider an example. Consider a system with 2 processors ($m = 2$) and 3 implicit-deadline tasks ($n = 3$), as specified by Table 2.3, to be scheduled according to the EDF policy. Since all tasks are released at $t = 0$, the first job of τ_1 and τ_2 with deadline 1 will have higher priority over the first job of τ_3, whose deadline is 1 + ϵ. Consequently, processors P_1 and P_2 are assigned to $J_{1,1}$ and $J_{2,1}$ during the time interval $[0, 2\epsilon]$, leaving a maximum of 1 - ϵ time units for $J_{3,1}$ before its deadline, which is not enough for it to be completely executed (see Fig. 2.2). Hence, this task set cannot be feasibly schedule by the EDF scheduling algorithm on a 2-processor computing system although $\sum_i^n u_i < 2$, as $\epsilon \rightarrow 0$, $\sum_i^n u_i \rightarrow 1$.

This finding led research community to look at global scheduling algorithms, where tasks can execute in any processor as an obsolete approach and, therefore, guided its course to partitioned ones, where tasks are statically allocated to processors in a fixed manner. Global scheduling algorithms recovered their popularity two decades later when it was realised that the Dhall effect is mostly related to heavy tasks scheduling, *i.e.* tasks with high utilisation, and not intrinsically a global approach problem [Phillips et al., 1997].



(a) Global scheduling **(b)** Partitioned scheduling **(c)** Clustered scheduling

Figure 2.3: Multi-core scheduling approaches for 4 CPUs that share L2 chaches in pairs of two

As slightly mentioned before, there are two fundamental classes of multi-core scheduling schemes: *global* and *partitioned*. However, not every scheduling scheme fits into one of these distinct categories but instead employ both [Carpenter et al., 2004], as depicted in Fig. 2.3c. Due to their wide variety, such hybrid approaches have many classifications (*e.g.* clustered, task-splitting), being *semi-partitioned* the prevalent term for them. In the general case, each τ_i may execute on a subset $P(\tau_i)$ of P, with overlapping permitted. Whenever $|P(\tau_i)| = 1$ partitioning is at the table, while $|P(\tau_i)| = m$ implies global scheduling. Thus, global and partitioned schemes are restricted instances of the above model.

2.3.1 Global

Under global scheduling, there is a single priority-ordered queue serving the entire system, where all ready jobs are stored (see Fig. 2.3a). At any time instant, the global scheduler can then select for execution the highest priority pending jobs since it has a full overview of the system and every job may migrate among processors. Clearly, two pivotal benefits arise from this broad knowledge and centralisation: optimal scheduling decisions are easily achieved and load balancing is automatically handled. Moreover, queueing theory results report that better average response times are produced by a single-queue scheduling than queue-per-core scheduling [Kleinrock, 1976]. Therefore, analytically speaking, global schedulers are superior to any partitioned algorithm as even optimality can be accomplished (for implicit-deadline tasks at least).

A class of global rate-based[4] schedulers called *Proportionate Fair (Pfair)* scheduling, intro-

[4]Rate-based means that the scheduler is invoked at steady points in time, which are pre-computed based on integers multiples of an input quantum.

duced by Baruah et al. [1996], provides the only known optimal method for scheduling HRT tasks on multiprocessors. The idea behind Pfair is that each task progresses proportionate to its utilisation and not only based on its deadline. For that, Pfair algorithms break a task into many unit-size sub-jobs, assign an individual deadline to them, and finally schedule them sequentially following a pure EDF strategy. In order to excel in performance, in the sense that if tasks request no more than the available processor capacity, and task set's utilisation is at most m, then all deadlines are met, an appropriate granularity must be defined. Unfortunately, if the execution time of a task is large compared to this unit-size then the preemption overhead becomes unreasonably large, which makes Pfair scheduling unfeasible in practise.

On the other hand, let us consider the scheduling algorithm G-EDF, where the uniprocessor EDF scheduler is globally applied to a single shared queue. Despite G-EDF is vulnerable to severe algorithmic capacity loss in the HRT case, since it is subject to the Dhall effect, resulting in an total utilisation bounded by $(m+1)/2$ for periodic task sets [Andersson et al., 2001], which is also extensible to any global JLFP scheduler, for SRT systems G-EDF is optimal because it guarantees bounded tardiness for any sporadic task set as long as $u_{sum}(\tau) \leq m$ [Devi and Anderson, 2008].

Although the theoretical worst-case performance of G-EDF in an HRT context cannot be higher than $(m+1)/2$, when $u_{max}(\tau)$ is considerably less than one, a higher utilisation guarantee can be assured. Thus, new schedulability tests based on the presence of high-utilisation tasks have been derived. The first, and of primary interest to us, was introduced by Goossens et al. [2003], who showed that a set of independent periodic tasks with implicit-deadlines can be successfully schedulable by G-EDF on m processors if

$$u_{sum}(\tau) \leq m - (m-1)u_{max}(\tau). \tag{2.2}$$

Furthermore, several tweaks to the G-EDF algorithm and respective worst-case analysis were developed with the same principle in mind. Srinivasan and Baruah [2002] proposed EDF-US[ζ], an algorithm that assigns the highest (fixed) priority to task of utilisation greater than some threshold ζ, and schedule the remaining tasks according to the standard EDF policy. By setting ζ to $m/(2m-1)$, an utilisation bound $u_{max}(\tau)$ free is obtained:

$$u_{sum}(\tau) \leq m^2/(2m-1). \tag{2.3}$$

Besides deriving a utilisation bound and showing that it is tight, Goossens et al. [2003] also proposed an algorithm that sets as highest priority tasks the k ones with highest utilisation. This approach was named EDF(k), and a sufficient schedulability condition for it was shown to be

$$m \geq (k-1) + \lceil \frac{u_{sum}(\tau) - u_k}{1 - u_k} \rceil, \tag{2.4}$$

where u_k is given by the kth task utilisation with tasks order by decreasing utilisation.

Either EDF-US[ζ] and EDF(k) were examined by Baker [2005], who showed that the optimal threshold used in EDF-US[ζ] with respect to maximising the utilisation bound is $1/2$, as it results in a sufficient test equally to the maximum possible bound for this class of scheduling algorithms:

$$u_{sum}(\tau) \leq (m+1)/2. \tag{2.5}$$

Concerning EDF(k), Baker [2005] revealed that there exists a minimum value of k (k_{min}) for which the worst-case guaranteed schedulable utilisation in Equation 2.6 also holds. Nevertheless, when accounting the number of task sets schedulable, EDF(k_{min}) outperforms EDF-US[$1/2$]

However, in practise, global scheduling algorithms are traditionally eschewed by OS developers due to the non-deterministic contention, potentially excessive overheads, implementation complexity, scalability issues and cache invalidation, whose impact and costs scheduling theory become accustomed to neglect.

As mentioned earlier, Pfair algorithms are impractical because making scheduling decisions (*e.g.* preemptions, migrations) at each tight timeslice, further the associated loss of cache affinity, plus the general communication and synchronization required, entail very high overheads. On the other hand, G-EDF does not incur such problematic overhead but still encompasses a single centralised queue whose access is disputed by m processors. Global structures like this must be protected by a lock mechanism to prevent concurrent data manipulation (*race-conditions*), which translates directly into serious contention and lack of scalability when the number of processors competing for the resource increases significantly.

These inherent issues constitute the reasons why global algorithms have drawn little interest from research community and have been discarded from most modern implementation. Although global scheduling remains controversial as a concept it is extremely appealing.

2.3.2 Partitioned

The alternative to global scheduling is partitioned scheduling, in which each processor has its own private queue and tasks are statically and permanently allocated to them during an offline phase such that no overload occurs (see Fig. 2.3b). This permits schedulability to be verified using a wealth of thoroughly studied real-time scheduling analyses techniques for uniprocessor systems, as well as eliminates scalability bottlenecks. Precisely, as partitioned scheduling is simple and scalable, the Linux scheduler was rewritten to adopt this approach in kernel v2.6, significantly boosting its performance for many-cores machines. Yet, no true real-time scheduling policies are natively supported by Linux.

However, the moment you treat each processor as an isolated domain and you are forced to choose *a priori* where to allocate the tasks, you run into a bin-packing problem which is known to be NP-hard in the strong sense [Garey and Johnson, 1990]. Heuristics must then be used in order to find a fast satisfactory solution, as an exhaustive search for an optimal one is impractical. Most common ones are First-Fit (FF) and Best-Fit (BF). FF selects the first non-empty queue with enough resources remaining, while BF looks for the queue where the least amount of resources are left after allocations.

Still, even an optimal allocation may leave some processors partially idle. Hence, most partitioned schedulers employ load balancing mechanisms (inter-domain migrations) for distributing the work evenly between domains and handling load-transients. Needless to say, this clashes

with the partitioning philosophy itself, since potentiates cache misses and overheads, causing yet another non-deterministic latency.

Following the global algorithm trend, there exists task sets with $u_{sum}(\tau)$ at most $(m+1)/2+\epsilon$ that cannot be schedulable on m processors by partitioned algorithms regardless of the allocation heuristic used. However, partitioned scheduling has reached the best possible results. López et al. [2000] showed that when using EDF the lowest utilisation bound of any reasonable allocation algorithm is equal to Equation 2.2, while the highest utilisation bound for the same scenario is given by

$$u_{sum}(\tau) \leq \frac{(\lceil 1/u_{max}(\tau) \rceil m + 1)}{(\lceil 1/u_{max}(\tau) \rceil + 1)}, \tag{2.6}$$

where it is assumed that $n > m/(\lceil 1/u_{max}(\tau) \rceil)$, being n the number of task in τ.

They also proved that EDF-FF and EDF-BF, like all reasonable allocation algorithms that order tasks by decreasing utilisation, achieve the higher limit. For the unrestricted case, where $u_{max}(\tau) = 1$, Equation 2.6 is attained. Therefore, EDF-FF and EDF-BF are optimal partitioning approaches in the limited sense that their guaranteed tight utilisation bound is as large as it could feasibly be.

2.4 Summary

This chapter introduced RTSs, where the key concept is not to be fast, but deliver determinism and predictability to real-time applications with stringent timing constraints. An HRT system cannot miss deadlines under any circumstance, whereas a SRT system may tolerate short latencies. Afterwards, relevant real-time terminology and the periodic task model were presented. Finally, we addressed real-time scheduling theory by discussing main scheduling concepts and by proving a brief historical overview about real-time scheduling algorithms and their schedulability tests. Emphasis was given to EDF schedulers, since our work embraces G-EDF.

Chapter 3

Background

Now that embedded, mainstream, and high-end computers are being deployed on multi-core chips, the huge challenge facing parallel programming for performance and productivity improvements has taken on a new urgency. Many high-level parallel programming models, languages, and tools have emerged in order to exploit parallelism in the most efficient way by easing programmers' burden when transforming or writing applications in a simple, well-defined, scalable, and portable multi-threaded form.

However, these high-level frameworks leave the actual scheduling of resulting threads to be performed at runtime. Therefore, if the underlying system cannot efficiently map those threads on the available cores, then the performance achieved will be significantly lower than the desired one.

This chapter is divided in two major sections. In section 3.1, we discuss parallel computing benefits and concerns, and we justify our approach to schedule fine-grained parallel applications. We address, roughly speaking, two main ways of expressing parallelism by covering some particular models. Finally, this section presents work-stealing, a provably efficient scheduling algorithm for dynamic and irregular parallel computations. Section 3.2 introduces the Linux scheduler, focuses on the essentials of the modular scheduling framework internals, and finishes by presenting supplementary patches that provide enhanced real-time scheduling capabilities.

3.1 Parallel computing

Parallel computing is more than just a promising approach to boost applications' performance, or to meet the demanding modern computational requirements, by executing each application simultaneously on multiple processors. It is a compelling vision for how computation can seamlessly scale from a single processor to virtually limitless computing power [Dongarra et al., 2003]. Unfortunately, expressing and achieving an highly efficient parallel computation is not trivial. In fact, the scaling of applications' performance to match the anytime available parallelism is a long-lasting open problem with many related issues that need to be appropriately addressed,

namely: (i) how to design parallel algorithms, (ii) when to partition an application into threads[1] and to what amount, (iii) when, and in what way, do threads coordinate, communicate, and synchronise, and (iv) how to schedule threads onto the processors [Gajski and Peir, 1985, Quinn, 1994]. Therefore, the development of parallel applications relies largely on the availability of suitable software tools and environments. Consequently, much of the parallelizing burden and responsibility falls on the application's developer.

In this sense, there are two main strategies to develop parallel applications [Diaz et al., 2012]: *automatic parallelisation* and *parallel programming*. In the former, existing sequential source code is automatically parallelized by a proper compiler. Thus, it relieves the programmer from the parallelizing burden as all it takes is the code recompilation. Nevertheless, the amount of parallelism reached by current compiler technology is considerably low since such generic automatic conversion is extremely complex to obtain. In contrast, the latter involves developing a parallel application from scratch. This allows programmers to efficiently express parallelism and also to freely choose the programming model and the language. However, such coding is difficult, sometimes unproductive and painful, as data partitioning highly depends on algorithms design, and compiler assistance techniques have limited applicability. All in all, parallel programming leads to a better performance than automatic parallelization but at the expense of more programming efforts.

Parallel programming itself may also differ in ease and efficiency depending on the approach adopted. *Implicit threading* abstracts the programmer from task decomposition and placement details, as these are left to the compiler and runtime system. Thus the programmer just has to identify and annotate potential parallel regions on the application. Such annotations act simply as hints that can be ignored and safely replaced with sequential counterparts whenever the compiler finds them not worthwhile. Instead, *explicit threading* assumes that the programmer is wise enough to be the best judge of how a particular application can be parallelized and integrated in the system in order to extract the best attainable performance. Hence, the programmer takes full control and responsibility for partitioning the computation into threads, mapping them onto processors, defining the communication structure, *etc.*.

3.1.1 Parallel programming models

A parallel programming model is an abstract parallel machine describing how parallelism can be expressed, managed, and matched to the underlying system. Is is designed to separate software-development concerns from effective parallel-execution concerns, providing abstraction and stability [Skillicorn and Talia, 1998]. Hence, it is not tied to any specific type of machine: any model can (theoretically) be implemented on any underlying hardware.

However, unlike sequential programming, where the von Neumann model dominates, several different models can be found in different parallel computations. This is a natural outcome when modelling such an isolation layer because the level of abstraction employed may vary significantly (*e.g.* closer to particular existing hardware architectures). Furthermore some parallel

[1] A thread refers to any independent flow of control within an application. In a parallel real-time task model, each job spawns several threads, becoming itself the master thread.

algorithms are easier to express in certain models. In addition, one or several parallel programming languages, or libraries, are often associated with the parallel programming model that they realise. Thus, the choice of model is determined by the available parallel computing resources, by the ultimate goal of the system, and by the type of parallelism inherent to the problem.

Due to the heterogeneity of levels of abstraction involved, it is extremely hard to categorise and compare parallel programming models neatly. In this thesis, we just consider the most relevant ones within a classification based on *process communication* and *computation decomposition* properties. For a comprehensive presentation or a thorough classification, the reader is referred to literature such as Maggs et al. [1995], Skillicorn and Talia [1998], Asanovic et al. [2006] and Diaz et al. [2012].

Process communication

Process communication relates to the mechanisms by which parallel processes are able to interact with each other. The most common models of communication are *shared memory* and *message passing*. In the shared memory model, a set of threads, created when the computation enters a parallel region, have access to a common memory. Threads communicate implicitly by writing to and reading from a shared address space. However, as threads run asynchronously, coordination must be handled by the programmer, and the system underneath, to manage potentially conflicting accesses. Despite the necessary synchronisation constructs for concurrent threads, a user-friendly programming perspective to memory is provided, since it can be seen as an extension of sequential programming methodology. Moreover, data sharing between processes is both fast and uniform due to the proximity of processors to memory. Nevertheless, as processors must contend for access to the physical memory (typically via bus), adding processors increases memory latency as well as traffic associated with cache management, which naturally affects scalable performance. Performance also suffers from the lack of locality exploitation. This model is a natural match for a shared memory architecture (illustrated in Fig. 3.1), where a single global address space exists in which all data resides, as the one present in Symmetric Multiprocessing (SMP) systems, commonly used in today's desktops.

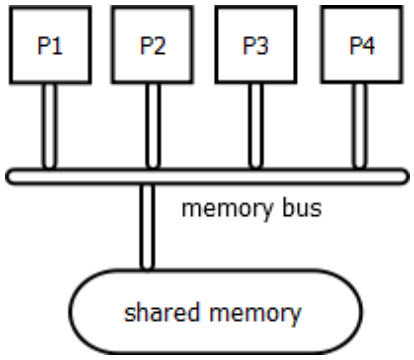


Figure 3.1: Shared memory multiprocessor

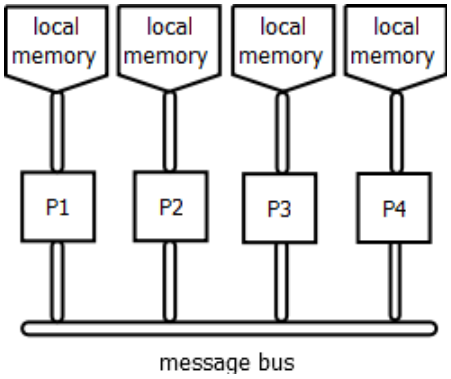


Figure 3.2: Distributed memory multiprocessor

On the other hand, in the message passing model, a set of processes have their local private

data structures, which belongs and can be addressed only by the corresponding processor. Any communication between processes has to be explicitly performed by exchanging messages with special send and receive commands. Data distribution must be carefully handled. As processors do not share an address space, they do not have to worry about concurrent accesses or external data manipulation from other processors. Therefore, the concept of cache coherency does not apply. Furthermore, the lack of a common bus translates in no inherent limitation on the number of processors; the size of the system becomes constrained only by the network structure used to connect processors to each other. The major drawback of this model is precisely the difficulty and costs involved in interprocessor communications, and consequently in programming, as the programmer is responsible for defining how and when data is communicated. Distributed memory architectures (illustrated in Fig. 3.2), such as supercomputer clusters, where each processor has its own local memory, are a natural match for the message passing model.

Naturally, hybrid models do exist, where a global address space is logically partitioned into portions, and each portion is local to one processor. The goal is to combine the productivity of the shared memory model with the performance of the message passing one.

Computation decomposition

Any parallel application is composed of simultaneously executing processes. Computation decomposition relates to the way in which these processes are formulated and several models can be employed for that matter. Here, we discuss the traditional *Single-Programming-Multiple-Data (SPMD)* and the increasingly popular *task parallelism*. An application following the SPMD model executes on multiple processors, but each processor deals with different portions of data, though the code is the same. The number of parallel activities (*e.g.* processes, threads) remains constant throughout application execution. This is, after the initial distribution, no further parallelism can be expressed. In this model, the programmer has the responsibility for mapping the parallel activities onto the available processor and load balancing. While this somehow limits flexibility and is more cumbersome at development time, it indeed reduces runtime overhead, since dynamic scheduling is no longer necessary. The standard MPI is based on SPMD as well as some parallel programming languages such as UPC.

An application under the task parallelism model spawns parallel activities dynamically according to the complexity of the problem faced by it. This is, the number of parallel activities may vary largely during execution (so does the amount of work contained by each one of them) and thereby adapts to the currently available parallelism. Hence, the programmer focus on decomposing the application into sub-computations that can, but are not mandated to, run in parallel. Thus, all these activities need to be mapped to processors at runtime by either the language's runtime system, the OS, or even a thread package. The programmer is then released from the onus of scheduling and balancing the load. Therefore, task parallelism is steadily gaining popularity as a parallel programming model, as demonstrates its implementation in the standard OpenMP, in several languages (*e.g.* Cilk, Chapel) and libraries (*e.g.* TBB, StackThreads/MP), and the introduction and dissemination of lightweight processes packages such as Portable Operating System Interface (POSIX) threads.

3.1.2 Fine-grained parallelism

Despite the large availability of implicit-threading technologies with lightweight processes implementations, most parallel applications are still written in a coarse-grained manner, typically with one thread per core - *task-level parallelism*. Each thread is relatively big in terms of code size and execution time, so data is transferred among cores infrequently. In contrast, a fine-grained application dynamically spawns threads according to the problem size, rather than the number of cores, commonly resulting in a large amount of short-living threads - *data-level parallelism*. Nevertheless, other levels of parallelism can be detected in an application (see Table 3.1).

Table 3.1: Parallelism granularity

Grain size	Level of parallelism	Code example	Mostly parallelised by
Very fine	Instruction-level	Operation	Processor
Fine	Data-level	Loop	Compiler
Medium	Control-level	Function	Programmer
Coarse	Task-level	Heavyweight process	Programmer

In order to attain the best speed-up, the best trade-off between scheduling flexibility and overheads needs to be found. If the granularity[2] is too fine, the performance may be limited by poor locality or excessive communication. On the other side, if the granularity is too coarse, the performance may be limited by load imbalance.

In this thesis, we focus on moderate fine-grained parallelism, where intra-task parallelism is expressed at a reasonable granularity, to amortize thread operation costs (*e.g.* creation and synchronisation), provide locality, and yet yield enough flexibility for good load balancing. Nonetheless, this still leads to a large number of threads creation, and has the following advantages over coarse-grained approaches [Narlikar and Blelloch, 1998]:

- **Simplicity** - Programmers can express all the worthwhile parallelism in the form of lightweight threads, without specifying their mapping to cores. This results in a simpler, shorter, clearer code, particularly for applications with irregular and dynamic parallelism.

- **Portability** - The resulting application is architecture independent (as long as the language in which it was written also is), since the parallelism is not statically mapped to a fixed number of cores.

- **Load balance** - Since the number of threads spawned is often of a much higher degree than the number of cores that will be used, the load can be transparently and effectively balanced by the implementation.

- **Flexibility** - Unlike coarse-grained applications, where any change to the execution order of heavyweight threads may involve considerable programming efforts because it is explicitly coded, fine-grained ones can be dynamically rescheduled just by tuning the underlying scheduler.

[2] In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

Although the fine-grained threads model allows programmers to easily expose all the parallelism in the application, scalable performance is not guaranteed. Actually, it heavily relies on the underlying system because an efficient scheduler to map threads to processors at runtime is mandatory. Typically, such schedulers focus on providing good data locality, keeping the overall overhead low, and balancing the workload to deliver good time performance [Chandra et al., 1993, Hummel and Schonberg, 1991].

However, if the same schedulers do not take into consideration the potential memory usage of parallel applications, a dynamic, fine-grained one may end up generating excessive active parallelism, which leads to a huge space requirement [Blumofe and Leiserson, 1993, Narlikar and Blelloch, 1998]. Moreover, a space-inefficient scheduler oftentimes degrades applications performance due to more memory page misses and consequently more memory-related system calls. Hence, reducing the memory requirements of a parallel computation is as important as reducing the executing time itself.

Work-stealing is a well-studied runtime scheduling paradigm that can both analytically and empirically provide a fair combination of the above demands. Due to its high success in scheduling dynamically growing multi-threaded applications, we decided to extend it to the real-time realm.

3.1.3 Work-stealing scheduler

Work-stealing by Blumofe and Leiserson [1999] is a simple scheduling algorithm for fully-strict[3] multi-threaded computations which is provably efficient in terms of time, space, and communication. Unlike its variant *work-sharing*, where newly spawned threads are distributed amongst (hopefully idle at that moment) processors, in work-stealing idle processors take the initiative: they attempt to ”steal” threads from other processors. Thus, when all processors have work to do, there is no need to migrate threads, and when they do not, most of the effort involved with acquiring more work is undertaken by the idle ones.

A work-stealing scheduler employs a fixed number of worker threads (henceforth referred as just workers), usually and preferably one per core to minimise the overhead for context switching. Each of those workers has a local double-ended queue, called deque, to store ready threads. As soon as a master thread is assigned to a worker and starts to be executed, it can enter a parallel region at anytime. Newly spawned threads are enqueued at the head of the worker’s deque. For example, as Fig. 3.3 depicts, task 1 spawned three new threads, which were enqueued at the head of deque A, while task 2 is still on a sequential region. When a worker finishes or suspends the execution of a thread, it looks for more work at the head of its deque. Therefore, workers treat their own local deques as a stack, pushing and popping threads from the bottom in a Last-In First-Out (LIFO) order. Consequently, since most threads (primarily in fine-grained applications) share some data with their parents, it is very likely that the data required by a recently created thread is still in cache [Acar et al., 2000].

So far, all operations performed by the workers are completely local and no synchronisation

[3] All data dependency edges from a thread go to the thread’s parent.

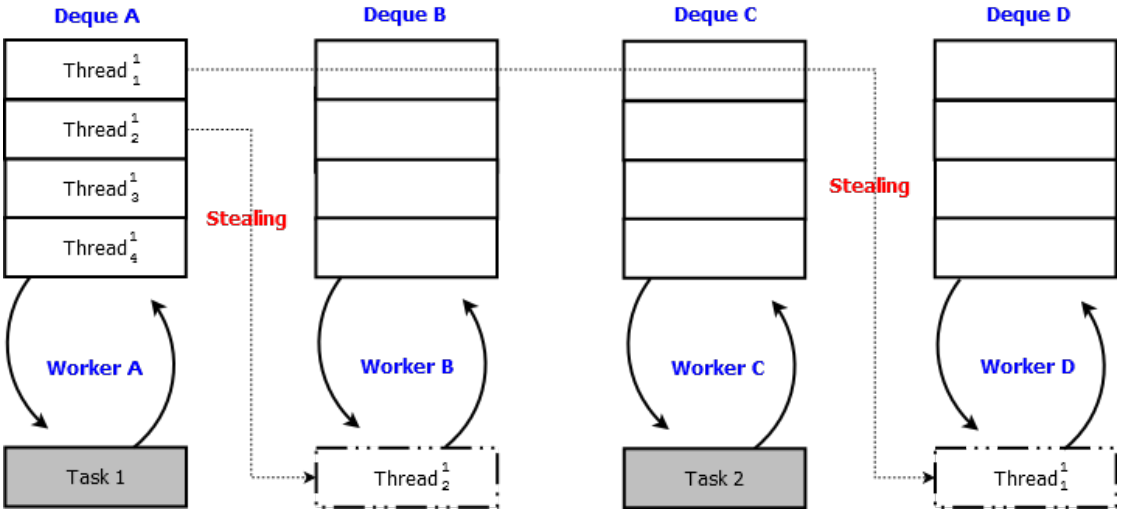


Figure 3.3: Work-stealing scheduler on a 4-core system

is necessary. Interaction between workers is required only when a deque runs out of work. Thus, threads created on a processor remain stored there unless load balance is demanded, which effectively increases scheduling granularity, and hence provides good data locality and low scheduling contention [Narlikar, 1999]. In this case, the idle worker becomes a thief and attempts to work-steal from a victim worker randomly chosen. If the victim's deque is not empty, then the thief dequeues the thread at the tail and starts executing it; else, the thief restarts the process, selecting another victim uniformly at random to steal from. The principle is to move load balancing costs from the busy worker to the idle one, which would otherwise be wasting CPU cycles anyway. In Fig. 3.3, workers B and D each steal a thread from deque A. Note that the order is totally unpredictable as randomness is the key property on the stealing strategy in order to reduce contention, which is aggravated when many processors are idle at the same time. Locality is favoured again by stealing in a First-In First-Out (FIFO) manner because the first threads are the ones with higher probability to generate future workloads [Frigo et al., 1998]. Furthermore, by having thieves operating on the opposite end of the deque than the worker they are stealing from, non-blocking deques can be implemented [Arora et al., 1998, Chase and Lev, 2005, Hendler et al., 2006] to minimise the synchronisation cost. Clearly, all deque manipulations run in constant-time O(1), independently of the number of threads in the deque.

Following Blumofe and Leiserson [1993], we denote T_∞ as the minimum execution time of a fully strict computation on an infinite number of processors and T_1 as its minimum serial execution time. It is proved that the expected time T_p to execute the multi-threaded computation, on an ideal machine with no scheduling overhead, on p processors verifies Equation 3.2.

$$T_p \leq \frac{T_1}{p} + T_\infty. \tag{3.1}$$

This time appears asymptotically optimal in the case of very parallel applications where $T_\infty \leq T_1$. Moreover, Blumofe and Leiserson [1993] proved that the necessary space S_p for the execution satisfies

$$S_p \leq \frac{S_1}{p}, \tag{3.2}$$

whereas the expected total communication of the algorithm is at most $T_\infty S_{max} P$, being S_{max} the largest activation record of any thread.

One approach to schedule parallel applications using work-stealing is to include the calls to a user-space runtime library that manages the threads themselves explicitly in the application. This technique places a lot of onus on the programmer, requiring that the programmer is fully aware of the runtime library and the details of scheduler, which in turn affects the productivity. Hence, work-stealing schedulers generally resort to an alternate approach where the parallelism is expressed at a higher-level of abstraction using some parallel constructs in a language. This code is then transformed into an equivalent version with appropriate calls to the work-stealing runtime library using a compiler. Several frameworks for parallel programming, such as TBB and Cilk, employ this technique. However, the compiler needs to do a good job of mapping the threads appropriately in order to match the performance of a good hand-written application with direct calls to runtime.

Therefore, implementing a work-stealing scheduler at the kernel level, by exploiting the OS's capabilities, allows one to finally switch from the current support of user-space runtime libraries or compilers to native support from the operating system. Furthermore, existing user-level work-stealing schedulers are not effective in the increasingly common setting where multiple applications time-share a single multi-core, suffering from both system throughput and fairness problems [Ding et al., 2012].

3.2 The Linux scheduler

Linux is, in simplest terms, a non-commercial General-Purpose Operating System (GPOS). It was originally developed by Linus Torvalds, in 1991, specifically for the Intel 80386 microprocessor. Since then, Linux has evolved and grown at a spectacularly high pace due to the early adoption of the GNU General Public License (GPL), which makes its source code open and available to anyone to study and modify (as hundreds of developers worldwide do and as we did in this work). Witnessing this tremendous success is the fact that, today, Linux runs on more than 90% of the 500 fastest supercomputers, leads the servers' segment, and has a strong presence on embedded systems such as smartphones (yes, Android is built on Linux!), watches, televisions and network routers.

The Linux kernel is the heart of every Linux system. The kernel is the lowest-level software layer that interfaces with the hardware, and expertly manages the limited resources. One of the most important kernel subsystems is the process scheduler, or simply the *scheduler* as hereinafter designated. The scheduler decides which process[4] to run at any time instant, and it is its responsibility to share the finite resource of CPU time among all runnable processes in the system.

[4] In this thesis, *task* and *process* are used as synonyms.

How the scheduler works affects how the system behaves. Because Linux is a multitasking system, the scheduler must give to users the impression that the CPU is always available. Even on a multi-core machine, where processes can actually execute concurrently, when there are more processes than CPUs, the scheduler is responsible for switching between processes at very short time frames to give the illusion of simultaneous processing. Of course different processes have different needs, and the scheduler has to play with that in an unnoticeable way. Yet, a scheduling policy may favour task switching in order to provide an interactive system, it may privilege batch processes and hence allow them to run longer, it may also decide that some processes are vital for the system and should never be blocked by non-critical ones. A real-time scheduler forcibly follows this last strategy.

In the remainder of this section, we cover the essentials of the scheduler internals[5], with emphasis on its modular design, and we discuss several real-time extensions to the Linux kernel proposed by research institutions and independent developers. The purpose of this section is neither delve deep into the core scheduler logic nor describe the implementation of the scheduling policies. For that, and much more about the Linux kernel, the reader is referred to these two outstanding books by Bovet and Cesati [2005] and Mauerer [2008].

3.2.1 Modular scheduler core

The Linux scheduler was completely redesigned by Ingo Molnar as a scalable and modular scheduling framework, which makes the core scheduler quite extensible in a hierarchical manner. This new modular scheduler was introduced in the kernel 2.6.23, replacing the old O(1) scheduler, and become known as the Completely Fair Scheduler (CFS). However it does not mean that the scheduler is broken into loadable modules, as the word ”modular” traditionally suggests. There is no mechanism to add modules *on-the-fly*. Each of these modules translates in a *scheduling class* that encapsulates specific scheduling policies logic about which the core scheduler does not assume much. The core scheduler is ”just” a dispatcher that drives the overall flow and performs low-level task switches. Scheduling policies rule how and when tasks will be scheduled. While a scheduling class may be responsible for several policies, a task belongs exactly to a single policy.

As the core scheduler hierarchically queries the scheduling classes which task is supposed to execute next, without any knowledge about their internals, they have to provide a generic bind between the core scheduler logic and individual scheduling strategies. Thus, each operation that can be requested by the scheduler is represented by one function pointer, independently on how they are (if they are) actually implemented by each class. The set of function pointers available is collected in a special data structure called `sched_class`[6]. Without extensions necessary for multi-core systems (we will talk about this later), the operations that can be provided are as follows:

- **enqueue_task()** adds a new task to the runqueue. This function is called whenever a task enters a runnable state.

[5] All references to the kernel content relate to its status in the version studied (2.6.36), not the current one.
[6] Defined in include/linux/sched.h.

- **dequeue_task()** removes a task from the runqueue. This function is invoked whenever a task switches from a runnable into a not runnable state.
- **yield_task()** yields the CPU giving room for the execution of other tasks. This function is called whenever a task wants to relinquish control of the CPU voluntary.
- **check_preempt_curr()** checks whether the currently running task should be preempted. This function is invoked after every enqueue operation.
- **pick_next_task()** selects the most appropriated task eligible to be executed next. This function is called after a task has been taken away from the CPU.
- **put_prev_task()** makes a executing task no longer executing. This function is invoked before the currently running task is replaced with another one.
- **set_curr_task()** is mostly called whenever the scheduling policy of a task is changed.
- **task_tick()** is invoked by the scheduler at a very short periodic rate, which is defined by the HZ macro.
- **task_fork()** is triggered whenever a running task spawns a new task.

Besides these function pointers, a `sched_class` instance also contains a pointer, called `next`, which establishes how classes are related in a flat priority hierarchy. As Fig. 3.4 depicts, the stock kernel is released with the core scheduler logic plus three scheduling classes, supporting five scheduling policies in total. The real-time class deals with POSIX FP real-time scheduling and, therefore, is the highest priority one, followed by the CFS class which provides fairness to regular tasks by picking, at any moment, the task with the gravest need for executing (*i.e.* priorities are adjusted periodically). The idle class is the last one to be invoked by the core scheduler as it holds no scheduling policy but handles logic for idle tasks that are active on a CPU when there is nothing better to run. A brief description of each scheduling class is given below.

1. **SCHED_RR**. A round robin real-time policy that will let a task run until it has exhausted its time slice if no higher priority task becomes runnable in the meanwhile. When a task exhausts its time slice, it gets inserted at the end of its runqueue level. This way it ensures fair assignment of CPU time to all SCHED_RR tasks of the same priority but blocks any task below it.
2. **SCHED_FIFO**. A first-in, first-out real-time policy whose behaviour is identical to SCHED_RR but it has no concept of time slice. Thus, as long as a task is not blocked by a higher priority one it will execute for as long as it wishes and then leaves its runqueue.
3. **SCHED_NORMAL**. The default policy in a Linux system and the reason why the Linux scheduler is today called CFS. The idea here is to run normal tasks concurrently at precise weighted speeds so that each task receives a fair amount of processor share.

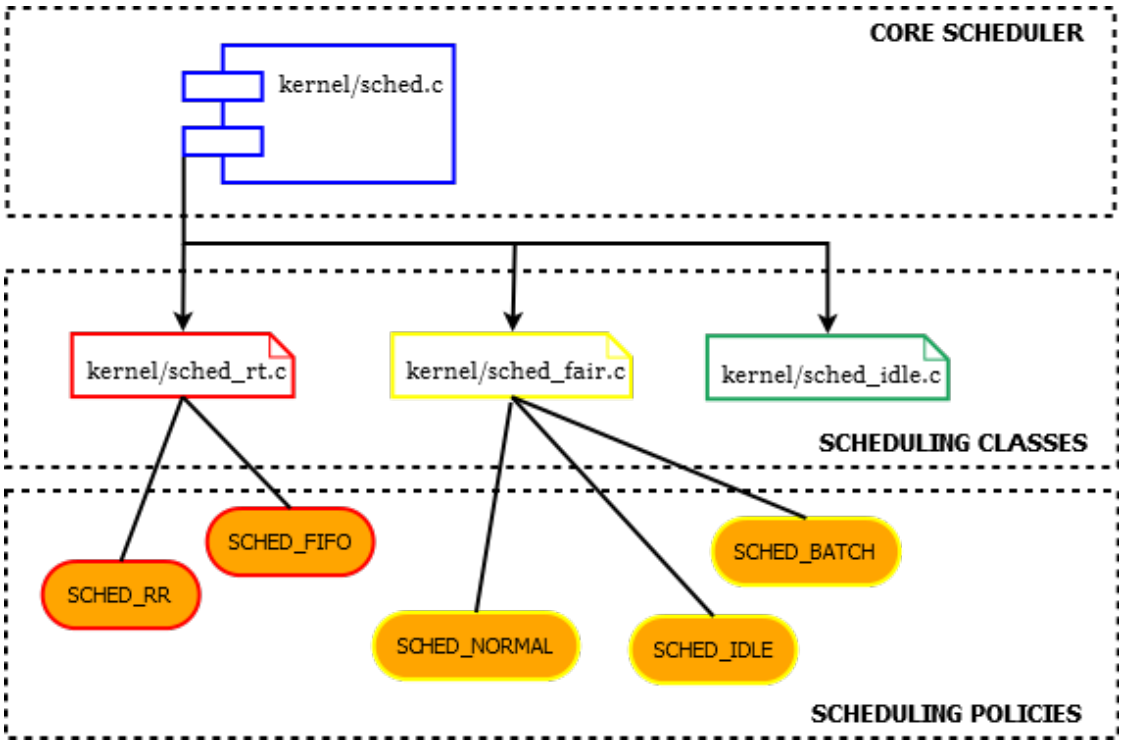


Figure 3.4: The linux modular scheduling framework

4. **SCHED_BATCH**. A policy for CPU-intensive batch tasks which do not require interactivity. Since these tasks want to execute for a long period of time, they cannot disturb interactive tasks. Hence they are disfavoured in scheduling decisions and typically remitted to the background.

5. **SCHED_IDLE**. The last policy to be handled by the CFS class as its tasks always have a minimal relative weight (low importance). Note that SCHED_IDLE has ,despite its name, a different purpose than the idle class.

3.2.2 Main scheduling structures

The scheduler contains a series of data structures to represent, sort, track and manage the tasks in the system. How the scheduler operates is strictly linked with the design of these structures. The most important ones are: *process descriptor*, *scheduling entity*, and *runqueue*.

The runqueue is the key data structure of the scheduler since it manages all active tasks. In this new scheduler, each CPU has its own runqueue data structure called `rq`[7]. Nevertheless, each active task appears on one, and just one, runqueue. Indeed, it is not possible to run a task on several CPUs at the same time unless this task is parallelized. In this case, the task spawns threads which are allowed to execute on different CPUs, as task scheduling makes no relevant distinction between tasks and threads - they are both scheduling entities. Furthermore, a runnable task can only be executed by the CPU owning the runqueue to which that task is associated. However,

[7] Defined in /kernel/sched.c.

a runnable process may migrate to other runqueue than the one originally assigned, mostly for load balancing purposes. Some interesting fields that can be found inside `rq` are:

- `lock` is a spin lock that protects the integrity of the runqueue and its tasks.
- `nr_running` accounts the runnable tasks in the runqueue.
- `curr` is a pointer for the currently executing task on the CPU.
- `clock` provides a per-runqueue time.

The most important fields on the runqueue are those that somehow relate to the set of runnable tasks in the system. Yet, tasks are not directly managed by the general elements of the runqueue. Instead, a class-specific sub-runqueue is embedded into the main runqueue, so each scheduling class can implement it on a different way. For example, `struct cfs_rq` holds anytime sub-runqueue status of the CFS class as well as the disposal of its enqueued tasks. `struct rt_rq` works analogously. To highlight, `cfs_rq` uses a time-ordered red-black tree to store runnable tasks and consequently build a "timeline" of future task execution. Fig. 3.5 shows all this at first sight confusing things.

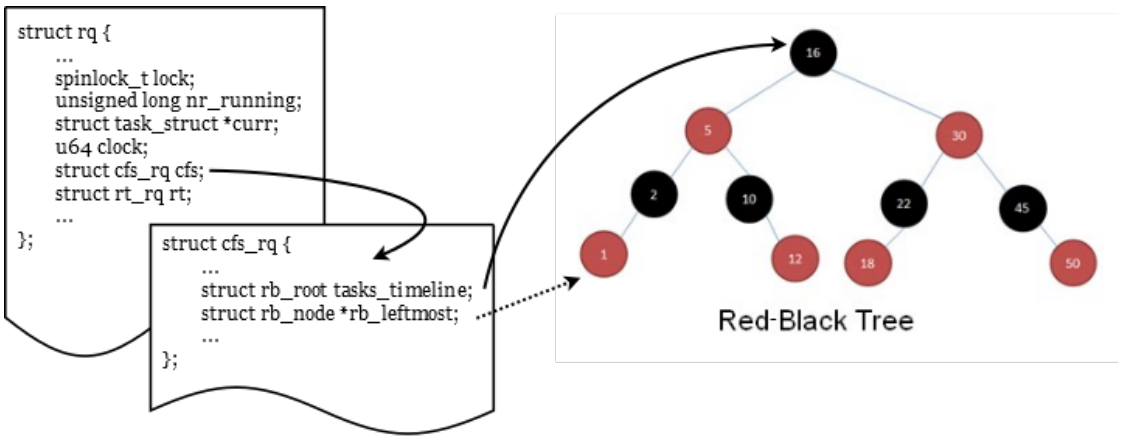


Figure 3.5: The CFS runqueue

In a nutshell, a red-black tree is a type of self-balancing binary search tree whose nodes are sorted by a key. The leftmost node is then the one with a lowest key value. Red-black trees allow for efficient management of the nodes they contain, and their typically operations (*i.e.* insertion, lookup and deletion) take $O(log_n)$ time to complete, where n here is the number of elements present in the tree. The Linux kernel provides this data type as a standard.

Each task is represented by an instance of a structure denominated `task_struct`[8], the process descriptor, which maintains up-to-date information about it. There are several scheduling-relevant fields included in a `task_struct`; among others:

- `state` describes the current state of the task. Fig. 3.6 depicts the main process states and transitions.

[8]Defined in /include/linux/sched.h.

- `prio` and `normal_prio` denote the dynamically computed priorities of the task, whereas `static_priority` is the relative priority assigned to the task when it was created (it can be modified by the user but not by the kernel). There is also `rt_priority` which is a static priority for a real-time task.
- `sched_class` as we have already seen connects the task to its scheduling class.
- By turn, `policy` denotes the scheduling policy applied to the task.

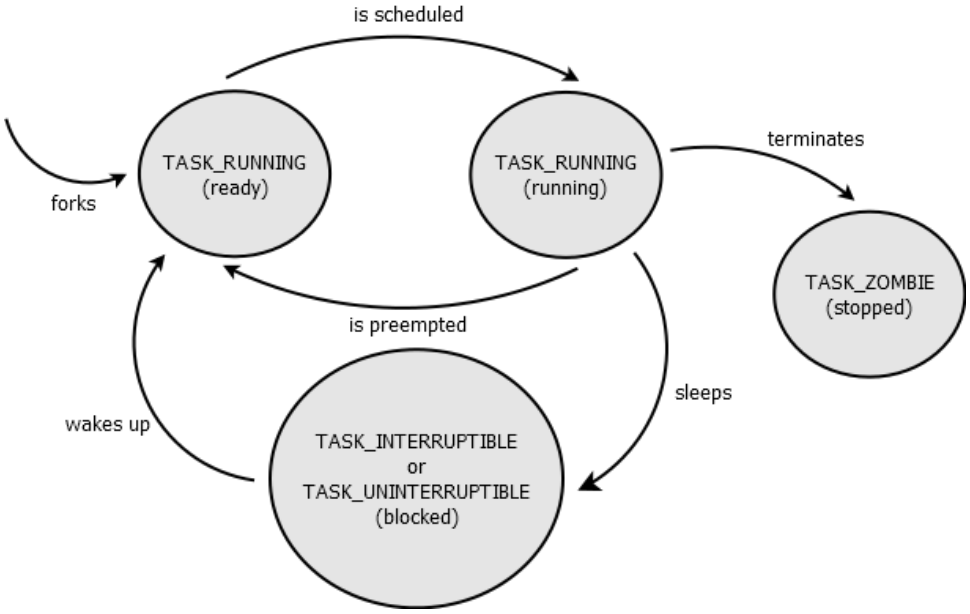


Figure 3.6: Transitions between process states

However, the scheduler does not operate directly on tasks because is not restricted to schedule tasks. In fact, it can schedule a whole group of them. The concept of scheduling entity denotes this generality. Such an entity is implemented in a modular fashion as well due to the inevitable class-dependency. Therefore each processor descriptor contains an instance of `sched_entity` and `sched_rt_entity` structures, which serve the CFS class and the real-time one, respectively. These structures typically encompass statistical elements, group scheduling fields and, of course, the actual and some historical task details. For instance, in `sched_entity`, `on_rq` indicates if the entity is currently enqueued in a runqueue, while `sum_exec_runtime` records the consumed CPU time when the entity is executing.

Note that despite a task is necessarily a scheduling entity the inverse statement is not true in general. In our work we equate both since we are concerned only with task scheduling.

3.2.3 Multiprocessor-dedicated logic

So far, all that has been said is totally general and, therefore, can be applied to single core and multi-core systems as well. Naturally, Linux provides several pivotal enhancements to efficiently make use of multiprocessor machines, whatever form they come. Notice, however, that these

enhancements, specially scheduling related ones, add much complexity to the scheduler, so they must be anytime addressed carefully. Here we will just consider some mechanism in a simple way to show the essential principle.

In order to ensure good scheduling on multi-core systems, the scheduler must address a few additional issues:

- As we have discussed in previous sections, the CPU load must be distributed as evenly as possible over the available cores. It is a completely waste of resources, and a significantly decrease in throughput, if four concurrent applications are assigned to one CPU, while there is one dealing with the idle task.
- It has to be possible to set the affinity of a task to a specific CPU or a subset of CPUs. This allows one, for example in a 4-cores system, to dedicate one CPU to a single batch application, whilst binding the remaining tasks to the others three CPUs.
- Last but not least, the scheduler must be able to migrate tasks across CPUs. However, this feature may severely impair performance if used in an ad-hoc manner. For instance, cache misses are the biggest concern on a small SMP system, whereas on a large system a CPU can be located literally some meters away from the target memory, resulting in a extremely costly access operation.

Needless to say that a multiprocessor Linux kernel (one configured with `CONFIG_SMP`) requires extensions to the afore-mentioned data structures to satisfy the above conditions.

`task_struct` includes the `cpus_allowed` field which is a bit mask representing the affinity of a task to particular CPUs. By turn, `sched_class` is augmented by additional functions:

- **select_task_rq()** selects the best suited runqueue for a task. This function is invoked whenever a new task enters the system or wakes-up.
- **set_cpus_allowed()** is called to modify a given task's CPU affinity. Depending on the new parameters, it may be responsible for initiating a task migration.
- **load_balance()** checks if the runqueue is balanced within its scheduling domain (explained afterwards); attempts to move tasks when the answer is negative.
- **pre_schedule()** performs scheduling decisions before the actual schedule. This function is invoked inside the main schedule routine.
- **post_schedule()** differs from the previous function only in the invocation moment, which is after the actual schedule.

Linux sticks to the SMP model in a sense that the kernel should not have any bias toward one CPU with respect to the remaining ones. Nonetheless, as multi-core machines come in many different flavours (*e.g.* hyper-threading chips, SMP and NUMA architectures, permutations between the three), the scheduler behaves accordingly for system performance benefit. This is,

in order to extract the best perform out of a multi-core system the scheduler sophisticatedly takes into consideration the topology of the CPUs, specially for load balancing purposes, so it can migrate tasks intelligently. For that, the notion of *scheduling domains* is supported by the kernel, and each runqueue (CPU) is associated to one scheduling domain through the addition of a `sched_domain`[9] structure pointer (field `sd`) inside `rq`.

Long-story short, a scheduling domain is a set of CPUs, which share some hardware characteristics, and whose workloads should be kept balanced by the scheduler. Scheduling domains are hierarchically organised: a multi-level system will have many levels of domains, and each level may contain different domains. A small SMP system, like the one considered in our work, typically has a single domain which spans every CPU available. Thanks to this hierarchy, the runqueue balancing algorithm can be easily tuned for any type of multi-core architectures, or technologies, and therefore it can be performed in a rather efficient way.

3.2.4 Real-time scheduling on Linux

The existing real-time scheduling policies perform very well in their own domain of application, however, they cannot provide the timing guarantees a real-time system requires as no concept of actual timing constraints (*e.g.* deadlines) can be associated to tasks. Moreover, the latency that may be experienced by a task cannot be bounded, since it highly depends on the number of runnable tasks assigned to that particular scheduling policy at that time. These issues are of paramount criticalness when running time-sensitive or control applications. Therefore, without a true real-time scheduler, one cannot derive a feasibility analysis of the system under development.

Due to this lack of real-time support in the mainstream, some companies started deploying modified versions of the Linux kernel with enhanced real-time capabilities. Although, these non-standard versions of Linux have commercial purposes. Thus, they are not free and their development is restricted to a small community. Fortunately, following the GNU spirit, several real-time extensions have been proposed to the Linux kernel mainly by research institutions and independent developers. Among these research projects, which have been invaluable in demonstrating the capabilities and limitations of new multi-core resource allocation techniques on actual hardware, the works more related (so-to-say) to our proposal of supporting full deadline scheduling for real-time parallel computations in the Linux kernel are LITMUS^{RT} and SCHED_DEADLINE.

LITMUS^{RT} [Calandrino et al., 2006] is a plugin-based scheduling framework for the Linux kernel, which supports the sporadic task model under a wide variety of implemented real-time policies, targeting both global and partitioned scheduling. The project focus primarily on the experimental evaluation of multiprocessor scheduling algorithms and synchronisation protocols for real-time system, from a research point of view. In that regard it simplifies such prototyping by providing abstractions and interfaces within the kernel.

SCHED_DEADLINE (originally named SCHED_EDF) [Faggioli et al., 2009] is a scheduling class for the Linux kernel that mimics the standard real-time class but employs an EDF policy. It im-

[9] Defined in include/linux/sched.h.

plements partitioned, global and clustered scheduling by applying CPU affinities and by allowing dynamic task migrations across CPUs, using push and pulls operations. This scheduling policy can handle periodic, sporadic or aperiodic tasks once it uses the Constant Bandwidth Server (CBS) [Abeni and Buttazzo, 1998] to provide bandwidth isolation (*i.e.* no task is permitted to execute longer than its budget every deadline length time interval). Therefore hard and soft real-time tasks can cohabit in the same environment as they do not interfere with each other even when they misbehave.

Nevertheless, none of those patches directly supports parallel real-time tasks. It has also to be said that both of them haven't become part of the official Linux kernel yet. While this is clearly the aim of SCHED_DEADLINE, as its implementation is (at the time of this writing) being kept lined up with the mainstream kernel and is POSIX-compliance, LITMUS^{RT} does not share this concern which eventually make it obsolete by now.

Despite any real-time scheduler whatsoever being added, Linux intrinsically presents some limitations for real-time systems since as a GPOS its primary design goal is to optimise the average throughput. Namely unpredictable latencies, non-preemptable sections, and coarse-grained timing resolution are potential issues for real-time applications [Scordino and Lipari, 2006]. Thankfully some meaningful efforts have been redirected into this direction.

In fact, even HRT tasks can be scheduled on Linux by adopting the so called *interrupt abstraction* approach. This approach consists of creating an abstraction layer of virtual hardware between the standard Linux kernel and the computer hardware. The resulting system is a multithreaded RTOS in which the standard Linux kernel is the lowest priority thread, therefore, it executes only when the real-time kernel is inactive. The main advantage is to attain very low latencies, hence it is efficient, whereas the major drawback is its invasiveness. RTLinux, Xenomai and RTAI are notable examples where this solution was successfully implemented.

PREEMPT_RT is a quickly evolving set of patches maintained by a restrict group of skilled kernel developers, currently led by Thomas Gleixner. The philosophy is to minimize the amount of kernel code that is non-preemptible, while also minimising the amount of code that must be changed in order to provide this added preemptibility. In order to accomplish an almost fully preemptible kernel, most kernel spinlocks are replaced by mutexes that support priority inheritance protocol [Sha et al., 1990], which solves the problem of unbounded *priority inversion*. Moreover, all interrupts are moved to kernel threads so they become schedulable. By attaining a predictable behaviour in critical kernel activities, a more deterministic Linux kernel is obtained, which is the most important property of any RTOS.

A priority inversion happens when a higher priority task is blocked on a shared resource owned by a lower priority task. If the lower priority task task is preempted by a medium priority task while holding the resource, the higher priority one will have to wait for an unbounded time.

Some features from the PREEMP_RT patch series, such as generic IRQs and hrtimers, have found their way into the mainline kernel. Other useful features remain as add-ons because, while increasing determinism, they often result in higher kernel overheads, and consequently lower throughput, which goes against the GPOS principles governing Linux.

3.3 Summary

In this chapter we saw how parallelism can be explored by programmers and in what way that may affect productivity and performance. As we are concerned with the scheduling of highly heterogeneous real-time parallel applications for shared memory architectures, we highlighted the characteristics of the followed models to generate work, namely the task parallelism model and shared memory one. In order to efficiently schedule such fine-grained and dynamic parallel computations, a time-, space-, and communication-aware scheduler must be employed. Work-stealing, which we explained in detail, not only provably assures that, but also automatically balances the workload in the system. The scheduler we present in the next chapter is a variant of this scheduling algorithm, typically implemented in a language runtime system.

In this chapter, we investigated how to implement our proposal in Linux, due to its free and open-source nature. Namely, we discussed the Linux modular scheduling framework, we looked at the main data structures of the Linux scheduler, as well as its support for multiprocessor systems. At the end, we pointed out few of the limitations it faces regarding real-time support, while mentioning some patches that attempt to overcome those cases. The SCHED_DEADLINE scheduling class, whose implementation inspired our work, was briefly described here.

Chapter 4

Real-Time Work-Stealing

The motivation for this project was outlined in Chapter 1. Based on that, the next chapters discussed the main theory behind our proposal. In Chapter 2, we covered the real-time scheduling world, pointing out the chosen directions for our model. Chapter 3 introduced parallel computing strategies to effectively exploit parallelism, explaining how we can efficiently map threads to cores. This guided our design from top to bottom. Furthermore, Chapter 3 has also laid down the foundation for our implementation by describing the Linux scheduler. It is now time to bringing it all together.

Meant to be used natively as an OS scheduler, RTWS is a novel scheduling approach, which combines the G-EDF policy with a priority-based locality-aware work-stealing load balancing scheme, enabling parallel real-time tasks to run on more than one processor at a given time instant.

In this chapter, we provide a detailed description of all the work devised regarding the RTWS scheduler and justify our options. The next section describes the state-of-art in parallel real-time scheduling, shortly comparing to the system model we present in Section 4.2. Then, Section 4.3 follows by discussing the algorithm design, with emphasis being given to: (i) data structures, (ii) major rules, and (iii) sub-policies. The last major contribution of our work is the RTWS implementation in the Linux kernel. The core of this complex proceeding is analysed in Section 4.4.

4.1 Related work

Task-level parallelism is a form of parallelization of code across multiple processors in parallel computing environments. Many real-time applications have a lot of potential parallelism which is not regular in nature and which varies with the data being processed. Parallelism in these applications is often expressed in the form of dynamically generated threads of work that can be executed in parallel. The goal is to allow the programmer to express all the available parallelism and let the runtime system execute the program efficiently.

Considerable work on scheduling of parallel tasks can be found in [Agrawal et al., 2008, Arora

et al., 1998, Blelloch et al., 1999, Hummel and Schonberg, 1991, Polychronopoulos and Kuck, 1987, Turek et al., 1994]. However, it cannot be applied to real-time systems since timing constraints are not contemplated. Real-time scheduling of parallel tasks started to be addressed in 1989 when Han and Lin [1989] have shown the NP-hardness of preemptive scheduling parallel jobs, and the intractability of many parallel scheduling problems. The non-preemptive case was later studied by Wang and Cheng [1992] which proposed a heuristic based on the makespan metric. Ludwig and Tiwari [1994] also took makespan into consideration for scheduling parallel *malleable* tasks and their relation to *non-malleable* ones. However, these early works impose many limitations on the number and configuration of processors allotted to a task.

From an optimisation point of view, some research has studied cache-aware schedulers for multi-threaded tasks [Anderson and Calandrino, 2006, Calandrino and Anderson, 2009]. Anderson and Calandrino [2006] consider Pfair algorithm and encourage tasks of the same weight to be co-scheduled in order to minimise cache misses. Calandrino and Anderson [2009] show a significant performance improvement, with a slight overhead trade-off, when their cache-aware scheduler does accurately profiling. Nevertheless, in both works the parallelism degree of a job cannot be greater than the number of processors in the system.

Most prior work in parallel real-time scheduling makes simplifying assumptions about task models [Collette et al., 2008, Jansen, 2004, Kato and Ishikawa, 2009, Lee and Lee, 2006, Manimaran et al., 1998], assuming that the parallelism degree of jobs is known beforehand and using this information when making scheduling decisions. In practice, this information is not easily discernible, and in some cases can be inherently misleading. For instance, Jansen [2004], Lee and Lee [2006] and Collette et al. [2008] focus on *malleable* tasks, where tasks can efficiently execute on any number of processors and change it at runtime. On the other hand, Manimaran et al. [1998] and Kato and Ishikawa [2009] investigate the scheduling of *moldable* tasks, where the number of processors allotted to a task is defined before execution. The latter work, in its Gang EDF algorithm, also restricts the number of parallel threads within a task to its associated number of processors, while the former work considers non-preemptive EDF scheduling but does not allow the number of processors simultaneously used by a task to be posteriorly changed.

Recently, Lakshmanan et al. [2010] proposed a scheduling technique for a synchronous parallel task model. In this model, every task is an alternate sequence of parallel and sequential regions, with each parallel region consisting of multiple threads of equal length that synchronise at the end of the region. In their model, all parallel regions are assumed to have the same number of parallel threads, which must be no greater than the number of processors. Saifullah et al. [2011] considered a more general task model, allowing different regions of the same parallel task to contain different numbers of threads and regions to contain more threads than the number of processor cores. It still requires, however, that each region of a task contains threads of execution that are of equal length. In contrast, this thesis considers a more general model of parallel real-time tasks where threads can take arbitrarily different amounts of time to execute.

Furthermore, both works handle scheduling parallel tasks by decomposing them into sequential subtasks. In [Lakshmanan et al., 2010], this technique requires a resource augmentation bound of 3.42 under partitioned Deadline Monotonic (DM) scheduling. For the synchronous

model with arbitrary numbers of threads in parallel regions, the work in [Saifullah et al., 2011] proves a resource augmentation bound of 4 and 5 for G-EDF and partitioned DM scheduling, respectively. Instead, we try to minimise the scheduling overhead by generating parallelism only when required, *i.e.* when a processor becomes idle.

We believe that achieving predictable good performance for fine-grained task-level parallelism in embedded real-time systems is important for several reasons: (i) an efficient implementation of fine-grained parallelism allows more parallelism to be exploited, which is especially important with the expected increase in core counts in future processors; (ii) the programming model is simplified if programmers do not need to avoid spawning small tasks, which is very difficult when task execution times can not be predicted in advance; and (iii) many real-time systems have periodic serialisation points when input is consumed and output is produced. A natural way to program such a system is to parallelize each interval, which then becomes a parallel region.

4.2 System model

We consider the scheduling of implicit-deadline periodic independent real-time tasks on m identical processors $p_1, p_2, \ldots, p_m$ using G-EDF. With G-EDF, each task ready to execute is placed in a system-wide queue, ordered by non-decreasing absolute deadline, from which the first m tasks are extracted to execute on the available processors.

We primarily consider a synchronous task model, where each task $\tau_1, \ldots, \tau_n$ can generate a virtually infinite number of multi-threaded jobs. A multi-threaded job is a sequence of several regions, and each region may contain an arbitrary number of parallel threads which synchronise at the end of the region (see Fig. 4.1). For any region with more than one thread, the threads on that region can be executed in parallel on different cores. All parallel regions in a task share the same number of processors and threads inherit the parent's deadline. For now, our work is focused on systems where all parallel threads are fully independent, *i.e.* except for the m-cores there are no other shared resources, no critical sections, nor precedence constraints.

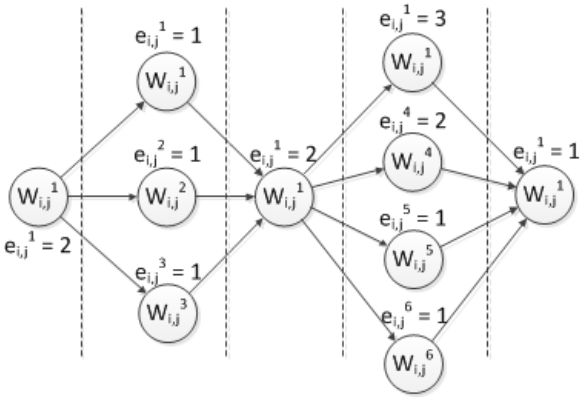


Figure 4.1: A multi-threaded job with 5 regions

The j^{th} job of task τ_i arrives at time $a_{i,j}$, is released to the G-EDF queue at time $r_{i,j}$, starts to be executed at time $s_{i,j}$ with deadline $d_{i,j} = r_{i,j} + T_i$, with T_i being the period of τ_i, and finishes

its execution at time $f_{i,j}$. These times are characterised by the relations $a_{i,j} \leq r_{i,j} \leq s_{i,j} \leq f_{i,j}$. Successive jobs of the same task are required to execute in sequence.

During the course of its execution the j^{th} job of task τ_i can enter in a parallel region and dynamically generate an arbitrary number of parallel threads which synchronise at the end of that region. A thread is denoted $w_{i,j}^k, 1 \leq k \leq n_i$, where n_i is the total number of threads belonging to the j^{th} job of task τ_i. We assume $n_i \geq 2$ holds for at least one task τ_i in the system. Otherwise, the considered task set does not have intra-task parallelism.

The execution requirements of a thread $w_{i,j}^k$ of task τ_i is denoted by $e_{i,j}^k$. Therefore, the WCET) C_i of task τ_i on a multi-core platform is the sum of the execution requirements of all of its threads, if all threads are executed sequentially in the same core.

Contrary to regular jobs of a task, dynamically generated parallel threads are not pushed to the G-EDF queue, but instead maintained in a local priority-based work-stealing double-ended queue (deque) of the core where the job is currently being executed, thus reducing contention on the global queue. For any busy core, parallel threads are pushed and popped from the bottom of the deque and these operations are synchronisation-free.

The fraction of the capacity of one processor that is assigned to a task τ_i is defined as its utilisation $u_i = \frac{C_i}{T_i}$. We further define $U_\Pi = \sum_i^n u_i$ as the system utilisation on the identical multiprocessor platform Π comprised of m unit-capacity processors and $u_\Pi = max_{1 \leq i \leq n} u_i$ as the maximum task utilisation.

A task set Γ is said to be schedulable by algorithm $\mathcal{A}$, if $\mathcal{A}$ can schedule Γ such that every $\tau_i \in \Gamma$ can meet its deadline d_i. With G-EDF, a task τ_i executed on the identical multiprocessor platform Π comprised of m unit-capacity processors never misses its scheduling deadline under the following conditions Goossens et al. [2003]:

$$u_\Pi \leq 1;$$

$$U_\Pi \leq m - u_\Pi(m-1) \tag{4.1}$$

Naturally, if only soft real-time tasks are considered, jobs may miss their deadlines by bounded amounts, eliminating such restrictive utilisation limits. It has been shown that, when using G-EDF to schedule sporadic soft real-time tasks on m processors, deadline tardiness is bounded, provided total utilisation is at most m Valente and Lipari [2005].

4.3 Design

Dynamic scheduling of parallel computations by work-stealing [Blumofe and Leiserson, 1999] has gained popularity in academia and industry for its good performance, ease of implementation and theoretical bounds on space and time. Work-stealing has proven to be effective in reducing the complexity of parallel programming, especially for irregular and dynamic computations, and its benefits have been confirmed by several studies Navarro et al. [2009], Neill and Wierman

[2009].

However, the need to support tasks' priorities and deadlines fundamentally distinguishes the problem at hand in this thesis from other work-stealing choices previously proposed in the literature Guo et al. [2010], Vrba et al. [2009, 2010]. With classical work-stealing, threads waiting for execution in a deque may be repressed by new threads, which are enqueued at the bottom of the worker's deque. As such, a thread at the top of a deque might never be executed if all workers are busy. Consequently, there is no upper bound on the response time of a multi-threaded real-time job.

Therefore, considering threads' priorities and using a single deque per core would require, during stealing, that a worker iterate through the threads in all deques until the highest priority thread to be stolen was found. This cannot be considered a valid solution since it greatly increases the theft time and, subsequently, the contention on a deque.

Using a single global concurrent priority-based deque is also not viable. While priority queues are often used in single core schedulers, when moving to a parallel context, concurrent priority queues are hard to make both scalable and fast Lenharth et al. [2011]. Furthermore, the semantics of priority queues naturally suggest an ordered insertion method, which is against the work-stealing deque philosophy.

Our proposal is to replace the single per-core deque of classical work-stealing with a per-core priority queue, each element of which is a deque. A deque holds one or more threads of the same priority. At any time, a core picks the bottom thread from the highest-priority non-empty deque. If it finds its queue empty, it steals a thread from the top of the highest-priority non-empty deque of the chosen core's queue. Fig. 4.2 provides a first depiction of the overall design.

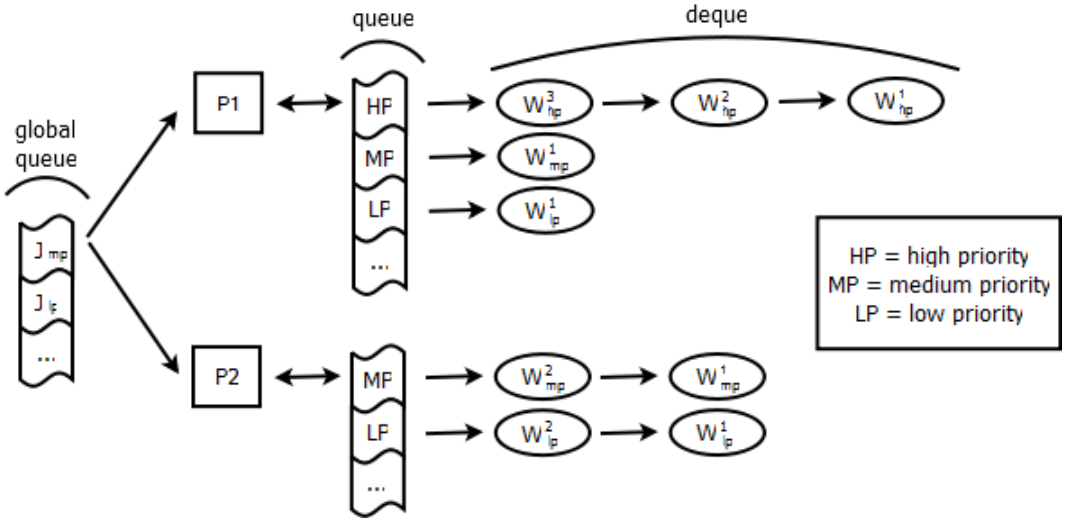


Figure 4.2: Overview of the RTWS data structures design

Notice that with this design all queue manipulations are straightforward since empty deques do not actually remain stored (we just mention non-empty deques for ease of understanding). Thus, no benefit from traditional work-stealing properties is lost while we assure determinism and predictability.

Among the various possible alternatives for designing a global real-time scheduler [Brandenburg and Anderson, 2009], the simplest and most commonly used ones are: (i) a single global queue from where tasks are consecutively dispatched to cores, and (ii) a distributed approach where each core has its own queue and tasks are dynamically allotted to those queues through migrations. Advantages of the former are the easy management of the unique queue, no need to synchronise between clocks of different CPUs, and, most of all, optimal picking of work because the scheduler has not to decide where to enqueue ready tasks. Moreover, the selection of what task to run next is straightforward, assuming a somehow ordered global queue. Nevertheless, such an approach has a serious drawback: performance degradation when the number of cores accessing it increases. This happens because in order to keep the queue consistent (*i.e.* to ensure that only one core concurrently manipulates the queue), it must be protected by a lock mechanism. Naturally, as a SMP system gets larger, the lock contention overhead considerably gets higher, eventually becoming the scalability bottleneck.

On the other hand, the latter case has the benefit of solving this scalability problem since each core selects runnable tasks only from its queue. Hence, contention received by any local queue is much lower and independent of the addition of CPUs. However, as a distributed approach implies the allotment of tasks to CPUs in the first place, this raises several disadvantages. Queue management is rather costly and complex due to the indispensable dynamic task migration and consistence of scheduling data information. Furthermore, making a good global scheduling decision is technically difficult due to the lack of synchronisation between CPUs' clocks.

As depicted in Fig. 4.2, our proposal adopts a single global queue for job-level scheduling, and, in last resort, a global distributed approach alike for parallel threads scheduling (this will be explained in the next section). This way the probability of acquiring a contended lock is minimised and threads are seldom migrated (only when a CPU would otherwise be idle). Thus, we mitigate both approaches drawbacks, while we conciliate and extract the best out of them.

4.3.1 Rules

The correctness and efficiency of a scheduling algorithm cannot be assured just by the data structures used by it and the flow connecting them. A set of rules to determine which m tasks must be executed on the m available CPUs is compulsory.

The proposed RTWS scheduler encompasses a G-EDF scheduling policy combined with a priority-based work-stealing load balancing scheme, used to allow parallel tasks to execute on more than one processor at any moment. The goals are to fit a wide-range of parallel real-time systems, reduce scheduling overheads, improve system performance by efficiently managing dynamic parallelism, and guarantee the schedulability of the system by G-EDF. Needless to say, in order to accomplish such goals some rules must be defined. We describe the major ones below.

- **Rule A:** a single global ready queue exists in the system, ordered by non-decreasing absolute deadlines. At each instant, the higher priority (with shorter absolute deadline) jobs are scheduled for execution.
- **Rule B:** whenever a job of a task τ_i being executed at a processor p enters a parallel region

and dynamically generates a set of parallel threads, those threads are not pushed to the G-EDF queue but instead maintained in the processor's local priority queue.

- **Rule C:** as soon as a job spawns parallel threads, it starts to be handled as a thread.
- **Rule D:** each entry in the processor's local priority queue is a deque, holding one or more threads of the same priority. At any time, a processor first looks into its local queue, picking the bottom thread from the highest-priority non-empty deque.
- **Rule E:** if the local queue is empty and there is no thread to pick, then a processor searches for jobs in the G-EDF queue.
- **Rule F:** still, if there is no eligible job in the G-EDF queue, the processor will steal the earliest deadline eligible thread from the top of the chosen busy processor's deque.
- **Rule G:** threads will never preempt any other entity. Only arriving jobs may cause a preemption.
- **Rule H:** opposed to a local thread, a stolen thread preempted by a job with a shorter deadline is enqueued in the G-EDF queue (like a preempted job is) and not back to the respective deque of the processor's local priority queue.

Each released job is enqueued in a system-wide global queue ordered by non-decreasing absolute deadlines, with ties broken by FIFO. At $t = 0$, all the m cores are idle and the m higher priority jobs are selected for execution. By following a global approach, cores are responsible for dequeuing the highest priority jobs from the global queue, therefore, eschewing the bin-packing problem of partitioned approaches, and achieving optimal scheduling decisions.

When entering a parallel region, a job generates an arbitrary number of threads, possibly with different execution requirements. To reduce contention on the global queue and to avoid uncontrolled priority inversion when stealing, each core has a deadline-ordered queue, each element of which is a deque. Therefore, each dynamically generated thread is enqueued, following a LIFO order, in the bottom of the respective deque, so that data locality is favoured and communication and synchronisation among cores are minimised.

For each core, the local deques are the first place to look for work, not only due to the fact that if they have work it means that there is a deadline to be met, but also to take advantage of caches and keep overhead low. If the local deques are empty, the global queue is searched. This step assumes that no matter how many threads the other cores in the system still have to execute, they are able to finish their work within the deadline (the schedulability of the task set is assured by G-EDF). Clearly, this step favours jobs in the G-EDF queue, with respect to parallel threads generated on other cores, by reducing their latency. Recall that we try to minimise the scheduling overhead by generating parallelism only when required, *i.e.* when a processor would be otherwise idle. Moreover, we focus on reducing the worst-case response time of the tasks and not the best, since real-time is not about fast computing but computing every task in time.

Finally, if no work has yet been found, a stealing operation takes place, ensuring that the top-right parallel thread (*i.e.* the oldest highest priority thread), in the chosen core is stolen. As the oldest element in its deque, it is a good candidate for stealing because it is likely that related-data is no longer cached. This last step helps to reduce the overall average response time and to keep the load balanced. By having a thief operating on the opposite end of the deque than the victim, both can perform actions on the deque concurrently as synchronisation-free mechanisms can be implemented. Furthermore, the load balancing operation cost is imputed to a core that would otherwise be wasting CPU cycles. The process flow diagram for this task selection procedure is shown in Fig. 4.3.

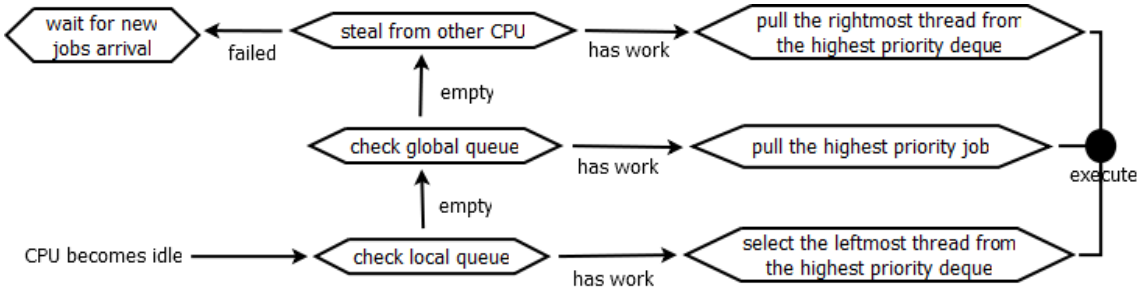


Figure 4.3: Process flow diagram representing rule E and F

Whenever a new job is released and enqueued in the G-EDF queue, and all cores are busy, the scheduler verifies if the core executing the lowest priority job/thread, among all the executing jobs/threads, has a higher deadline than the newly arrived job. If this condition is true, the job/thread is preempted. One of three possible situations occurs, depending on the properties of the preempted entity:

1. A **job** is enqueued back in the global queue because it has not yet entered a parallel region.
2. A **local thread** (*i.e.* a thread currently running on the core where it was spawned) is enqueued back in the respective deque in the core's local priority queue. Moving all related parallel threads would be too costly. This is the reason why we have a per-core queue of deques.
3. A **stolen thread** is enqueued in the global queue in order to prevent starvation and, therefore, a possible deadline miss.

The process flow diagram for this task preemption procedure is shown in Fig. 4.4. Note that spawned threads will never cause a preemption because system predictability does not rely on their parallel execution. This substantially reduces the number of context switches, while also contributes to retard accesses to the global queue.

4.3.2 Sub-policies

In designing a work-stealing scheduler there are two scheduling sub-policies to consider: *work-first* and *help-first*. Under the work-first policy, as soon as a job spawns a thread, it will be

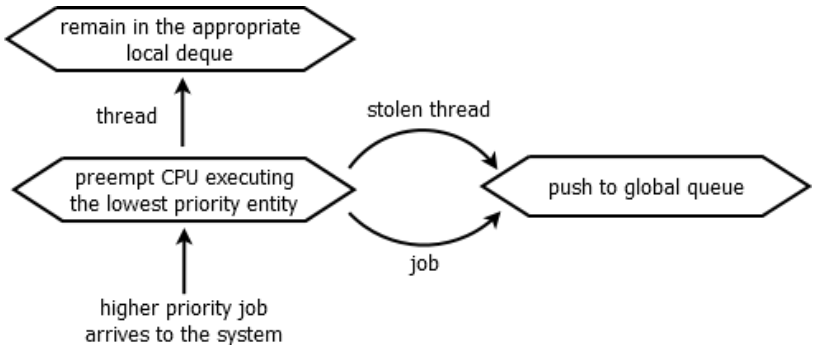


Figure 4.4: Process flow diagram representing rule G and H

swapped out, so the respective core starts working on the spawned thread eagerly. Consequently, unless the spawned thread creates more threads itself, there is only the master thread available for work-stealing. Work-first shines when computations are recursive (*e.g.* following divide-and-conquer paradigm).

In contrast, the help-first policy dictates that a core continues executing the master thread and leaves spawned threads to be stolen, so as many idle cores as spawned threads may immediately participate on the computation execution. This strategy fits better in computations that present flat parallelism (*e.g.* following a basic fork-join model).

RTWS supports both work-first and help-first scheduling sub-policies. However, since nested parallelism is beyond the scope of our work, we will neglect work-first in the remaining of this thesis, with an exception raised for the implementation discussion.

So far we have not discussed how do we elect the processor to steal from. Two approaches are possible for selecting the victim: (i) a probabilistic approach, where the victim is chosen randomly; or a (ii) deterministic approach, where the core is chosen by the priorities of the threads waiting to be executed in the deques.

Blumofe and Leiserson [1999] demonstrate that a random choice of the stolen core is fair and presents the advantage that the choice of the target does not require more information than the total number of cores in the execution platform. However, random selection, while fast and easy to implement, may not always select the best victim to steal from. As core counts increase, the number of potential victims also increases, and the probability of selecting the best victim decreases. This is particularly true under severe cases of work imbalance, where a small number of cores may have more work than others [Bhattacharjee et al., 2011]. Moreover, when a thief cannot obtain tasks quickly, the unsuccessful steals it performs waste computing resources, which could otherwise be used to execute waiting threads. In fact, if unsuccessful steals are not well controlled, applications can easily be slowed down by 15%–350% [Blumofe and Leiserson, 1999].

It is crystal clear that a blind probabilistic approach (*i.e.* a random choice where all cores are considered) is not suitable for a real-time scheduler. Nevertheless, since in our model the schedulability of the task set is guaranteed by G-EDF, no specific task needs to be executed in parallel. In other words, even executing sequentially every task in the system is guaranteed to

meet its deadline under any circumstance, which makes RTWS robust to small deviations from a strict priority schedule. In fact, some priority inversion may be actually acceptable, provided it helps reduce contention, as well as synchronisation and coordination between parallel threads. Thus, if we discard idle cores and steal randomly only among busy cores, applications will not suffer any performance loss. Hereinafter, we will refer to this as Busy-Aware Stealing (BAS).

Naturally, a deterministic approach (henceforth called PAS) is an obvious solution when real-time scheduling is at stake. Priority-Aware Stealing (PAS) can be defined as follows.

Definition 1 *The set of processors P_s eligible for work-stealing among the set of m identical processors $P = \{p_1, p_2, \ldots, p_m\}$ is given by $P_s = \{P_s | P_s \in P, n_{p_i} \geq 1\}$, where n_{p_i} is the number of threads in the local priority queue of processor p_i.*

Having P_s, an idle processor steals the earliest deadline thread w_{edf} among the ones in the top of the highest-priority non-empty deques (first entry in each of the processor's local priority queue) from the set of eligible processors P_s.

Definition 2 *The earliest deadline thread w_{edf} from the set of eligible processors P_s is defined as $\exists^1 w_{edf} \in P_s : min_{d_k^r}(P_s), P_s \neq \emptyset$.*

Note that the $\exists^1$ relation is guaranteed by the min function which, whenever there is more than one thread with the same earliest deadline, always returns the first thread on the list.

However, this determinism may turn in large contention overhead, affecting performance scalability. For instance, if at the same time instant 10 CPUs become idle, and there are several CPUs with ready threads, all idle ones will disputed the access to a single queue, resulting in considerable blocking time for 9 CPUs which could undoubtedly be better availed. A scenario like this is much unlikely to happen using BAS.

As BAS and PAS fit well in different real-time systems, our proposal encompasses these two stealing sub-policies. Yet, both will always select the rightmost thread from the highest-priority non-empty deque of the target queue.

4.3.3 Scheduling multi-threaded jobs with RTWS

Consider the following task set, described by WCET and period, $\tau_1 = (5, 10)$, $\tau_2 = (10, 20)$, and $\tau_3 = (4, 19)$. Task τ_1 executes sequentially for three time units and then spawns two threads which have an execution requirement of one time unit each. Task τ_2 has a sequential execution requirement of two time units and then spawns four threads, with the first and third threads having an execution requirement of one time unit, whereas the second and fourth threads have an execution requirement of three time units. Finally, task τ_3 only executes sequentially. Note that the task set is schedulable under G-EDF, $u_\Pi = 0.5$ and $U_\Pi = 1.21$.

Fig. 4.5 depicts a possible schedule generated by RTWS for those three tasks in two identical processors, when applying help-first and PAS sub-policies.

All tasks are released at $t = 0$. The ones with a lower deadline, τ_1 and τ_3, are selected for execution in the two cores. In the interval $t = [0, 5]$ none of the cores is idle. Therefore, task

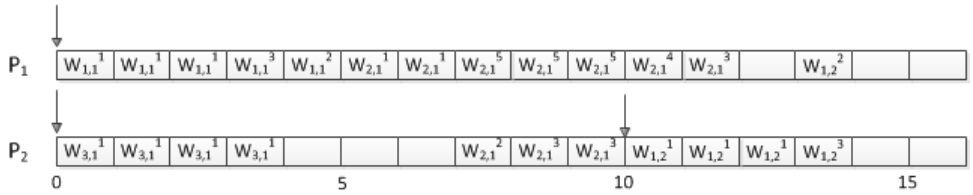


Figure 4.5: A RTWS schedule example

τ_1 executes sequentially, although it spawns parallel threads. At $t = 5$, task τ_2 is scheduled for execution in core 1. Its sequential part executes until $t = 7$ and then it spawns four threads. As core 2 is idle at time $t = 7$ and there is pending work in the priority queue of core 1, it is able to work-steal. Therefore, at $t = 7$, core 2 steals $w^2_{2,1}$ from the highest-priority non-empty deque of core 1.

At $t = 10$, a job from task τ_1 is released and preempts $w^3_{2,1}$, which has a lower priority. According to the RTWS policy, $w^3_{2,1}$ is enqueued in the global queue until one of the cores is able to finish its execution. In the depicted example, $w^3_{2,1}$ is executed at $t = 11$ in core 1.

As core 2 is idle after $t = 12$, threads generated by the second job of task τ_1 can be executed in parallel by both cores, by work-stealing at time $t = 13$.

4.4 Implementation

Based on the design principles presented in section 4.3, we have implemented RTWS in the standard Linux kernel 2.6.36 as a new scheduling class called SCHED_RTWS. In this section, we will dive into the code: (i) presenting the added data structures, (ii) analysing the main implementation logic, and (iii) showing the differences between theory and practise.

As we have seen in section 3.2.1, the Linux kernel has three native scheduling classes, hierarchically organised to establish a priority order between them. In order to create our new scheduler module, we need to code it in a separate file (`kernel/sched_rtws.c`) and position it anyhow in the module's hierarchy. Not surprisingly, the RTWS class is placed on the top of the hierarchy, becoming the highest priority module in the system, as shown in Fig. 4.6. The reason is because we will be dealing with time-sensitive real-time parallel tasks which cannot be delayed by ordinary tasks.

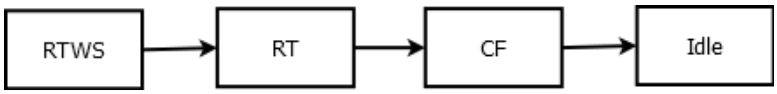


Figure 4.6: Priority hierarchy of scheduler modules

Before informing the core scheduler about the new highest priority module, a set of functions specified in the `sched_class` structure must be implemented. Listing 4.1 shows the definition of `rtws_sched_class`, which realises the RTWS scheduler module.

The first field (`next`) is a pointer to the second highest priority scheduling class in the hierarchy. Accordingly, `rt_sched_class`, which implements the two POSIX real-time policies, will be

queried every time RTWS fails to return a task. The other fields are functions that act as callbacks to specific events. Section 4.4.2 will narrowly analyse the most relevant ones. The reader may wonder why there is no `CONFIG_SMP` directive isolating the multiprocessor functions. Well, we neither intent to merged this first approach into mainline Linux, nor POSIX-compliance is a goal, so we just focused on the scheduling features for simplicity.

```
static const struct sched_class rtws_sched_class = {
  .next = &rt_sched_class,
  /* main functions */
  .enqueue_task = enqueue_task_rtws,
  .dequeue_task = dequeue_task_rtws,
  .check_preempt_curr = check_preempt_curr_rtws,
  .pick_next_task = pick_next_task_rtws,
  /* secondary functions */
  .put_prev_task = put_prev_task_rtws,
  .set_curr_task = set_curr_task_rtws,
  .task_tick = task_tick_rtws,
  .task_fork = task_fork_rtws,
  .task_dead = task_dead_rtws,
  .switched_from = switched_from_rtws,
  .switched_to  = switched_to_rtws,
  /* multiprocessor functions */
  .set_cpus_allowed = set_cpus_allowed_rtws,
  .task_woken = task_woken_rtws,
};
```

Listing 4.1: RTWS scheduling class

To differentiate tasks bound to our scheduling policy from other tasks in the system, we refer to them as RTWS tasks, or RTWS jobs due to these tasks continuous recurrency. Further, we use the term *pjob* when referring to a parallel thread of a job. Recall that RTWS tasks are periodic and, therefore, are potentially endlessly releasing new instances. They present a code structure similar to the algorithm present in Listing 4.2 because their periodicity is typically time-triggered and not event-triggered as most sporadic tasks. Note that, in this example, we ignore the computation itself (*i.e.* no parallelism is expressed), and focus only on the periodic behaviour.

```
start := time_now() + offset;
while (true) {
  delay_until(start);
  compute();
  start := start + period;
}
```

Listing 4.2: RTWS task algorithm example

4.4.1 Data structures

Following the scheduler code convention, we do not embed required data fields directly on the existing structures but instead create our own. Thereby, each process descriptor is provided with

a `struct sched_entity_rtws` which is an entity to schedule RTWS tasks (detailed in Listing 4.3). Remember that we neglect group scheduling, so an entity equals to a task, or a thread, as in Linux there is no substantial difference, they are both represented by a `task_struct`. This entity manages the general parameters of a RTWS task and some information about its status. Furthermore, an additional data structure is created to manage RTWS job specific parameters (`struct rtws_job`), tough a task only tracks the current job.

```
struct rtws_job {
  atomic_t nr; /* task instance number */
  u64 deadline; /* absolute deadline */
  u64 release; /* absolute release time */
};

struct sched_rtws_entity {
  struct hrtimer timer;
  struct rb_node task_node;
  u64 rtws_deadline; /* relative deadline */
  u64 rtws_period;  /* relative period */
  struct sched_stats_rtws stats;
  struct rtws_job job;
  unsigned long nr_pjobs;  /* number of spawned threads */

  /* specifying the scheduler behaviour: */
  unsigned int flags;
  int help_first, throttled, stolen;

  /* parallel threads fields: */
  struct rb_node pjob_node;
  struct rb_node stealable_pjob_node;
  struct sched_rtws_entity *parent; /* pointer to the RTWS entity that spawned
       it */
};

struct task_struct {
  volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
  ...
  struct sched_entity se;
  struct sched_rt_entity rt;
  struct sched_rtws_entity rtws;
  ...
};
```

Listing 4.3: RTWS scheduling entity

`struct hrtimer` represents a high resolution timer which is used to set and trigger tasks' periodicity at precise instants. It also contains a pointer to a callback function, so actions can be performed as soon as a task is released. Note that all timing parameters are set using nanosecond time unit. `task_node`, `pjob_node` and `stealable_pjob_node` are required to organise RTWS on three red-black trees serving different purposes. `struct sched_rtws_stats` holds statistic

information about the successive jobs execution of a RTWS task. `help_first` and `stolen` act as binaries to indicate whether an entity employs the help-first scheduling sub-policy and has been stolen, respectively, while `flags` and `throttled` are barely used to boost or throttle entity status. The remaining fields are quite self-explanatory.

Each per-processor main runqueue is provided with a `struct rtws_rq` which is a sub-runqueue holding all RTWS runnable entities assigned to that processor (detailed in Listing 4.4). Each `rtws_rq` points to a global runqueue (`struct global_rq`) where all RTWS ready jobs are maintained before they get scheduled. Although we could access `global_rq` directly, we decided to embed it for the sake of consistency. Notice, however, that any inner affiliation is just logical because it boils down to sort pointers: the actual process descriptors are all, with no exception whatsoever, stored in a circular doubly-linked list, called the task list.

```
struct global_rq {
  raw_spinlock_t lock; /* global runqueue lock */
  struct rb_root tasks;
  struct rb_node *leftmost_task;
  unsigned long nr_running; /* number of ready jobs */
  int deterministic;
};

struct rtws_rq {
  struct global_rq *global;
  struct rb_root pjobs;
  struct rb_node *leftmost_pjob;
  unsigned long nr_running; /* number of currently stored tasks */

  struct rb_root stealable_pjobs;
  struct rb_node *leftmost_stealable_pjob;
  u64 earliest_dl;

  struct rq_stats_rtws stats; /* accounting for runqueue operations */
};

struct rq {
  raw_spinlock_t lock; /* runqueue lock */
  ...
  struct cfs_rq cfs;
  struct rt_rq rt;
  struct rtws_rq rtws;
  ...
};
```

Listing 4.4: RTWS runqueues

All ready RTWS jobs are stored and sorted by increasing absolute deadline, with ties broken by FIFO, in a red-black tree represented by its root tasks. This is the first difference between theory and practise, because while queues (translated in linked lists under Linux) are much easier to understand, red-black trees are more efficient for priority-ordered data management. Another

big difference relates to the fact that each local queue of deques is implemented as two red-black trees (`pjobs` and `stealable_pjobs`). One red-black tree is not enough because it would required a thief to do a depth search in order to find the suitable thread, and once found, it would demand a costly rearrangement of the tree balance.

`pjobs` red-black tree is ordered by increasing absolute deadline, with ties broken by LIFO, and it contains all local pending threads plus the entity currently executing. Therefore, unless there is a context switch taking place, the leftmost element is always the entity running on that particular CPU. Note that, in theory, this aspect is omitted since many sources of overhead are considered non-existent. In the other hand, `stealable_pjobs` red-black tree is also sorted by increasing absolute deadline, but ties are broken by FIFO and the entity currently executing is left out. This way the leftmost element is assured to be the top-right thread from the design previously discussed. Thus, the desired thread from both corresponds to a leaf and pick it is straightforward because `leftmost_task`, `leftmost_pjob` and `leftmost_stealable_pjob` operate like a cache for the respective leftmost element.

Only one stealing sub-policy is adopted by all tasks. Whether it is PAS or BAS depends on the binary behaviour applied to `determinism`, which matches to 1 and 0, respectively. `earliest_dl` is not always the deadline of `leftmost_task`. It is used to keep the previous earliest deadline until we perform the last update on `rtws_rq` global status. Finally, `lock` fields are spinlocks to effectively synchronise runqueues. As these lock mechanisms keep spinning until acquire the resource (they do not sleep like semaphores), great care must be taken in order not to delay real-time scheduling decisions. The same goes for hierarchical locking, as deadlock situations may arise due to interrupts being enable or concurrent inverse lock acquisitions.

4.4.2 Features

Let us now turn our attention to how the scheduling features provided by the RTWS scheduler are implemented. First of all, in practise, we do not straightly insert every arriving job in the global queue, waiting for CPUs to pick work, as the model suggests. Since a timer interrupt is individually handled by a CPU, the other CPUs have no idea about the arrival of a new job. In fact, they can notice it by checking constantly the global queue. However, constantly means at each local tick, which might be considerably late compared to the time when the job was released. And we all know by now that even very short delays matter in a RTS. Therefore, as Fig. 4.7 illustrates, we employ a dispatching mechanism that we called *dispatcher agent*.

The dispatcher agent starts by verifying if the current CPU is free of RTWS tasks, so that the released job can be schedule right away. In case of failing, it verifies whether there is any idle `rtws_rq` or a CPU executing a lower priority task. When both conditions return false, the job is enqueued in `global_rq` and waits for its turn. However, if one condition is satisfied, the job is enqueued on the eligible `rtws_rq` and the kernel native `resched_task` function is invoked on that specific CPU to perform a task switch. It must be said that to the idle condition is given preference over the other one. By adopting this particular sequence of steps, we assure that jobs are scheduled when they should and where they are less costly. Thus, function

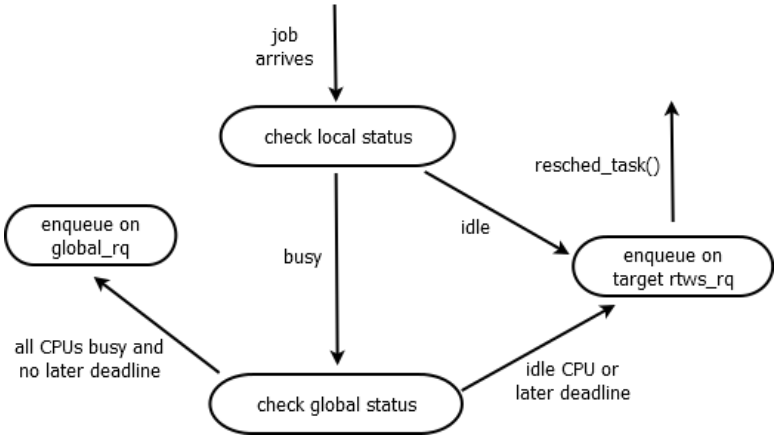


Figure 4.7: Dispatcher agent role

`check_preempt_curr_rtws` just has to check preemptions locally.

It goes without saying that the dispatcher agent also has to deal with waking up tasks, since in practise many kernel subsystems rely on wait mechanisms to deliver correctness and performance. Nevertheless, it is not worth to be illustrated here to avoid too much confusion.

Two functions are available to move elements to and from the `rtws_rq`: `enqueue_task_rtws` and `dequeue_task_rtws`. Let us concentrate only on placing new tasks on the runqueue because removing is basically the inverse but way simpler. Fig. 4.8 shows the code flow diagram for `enqueue_task_rtws`.

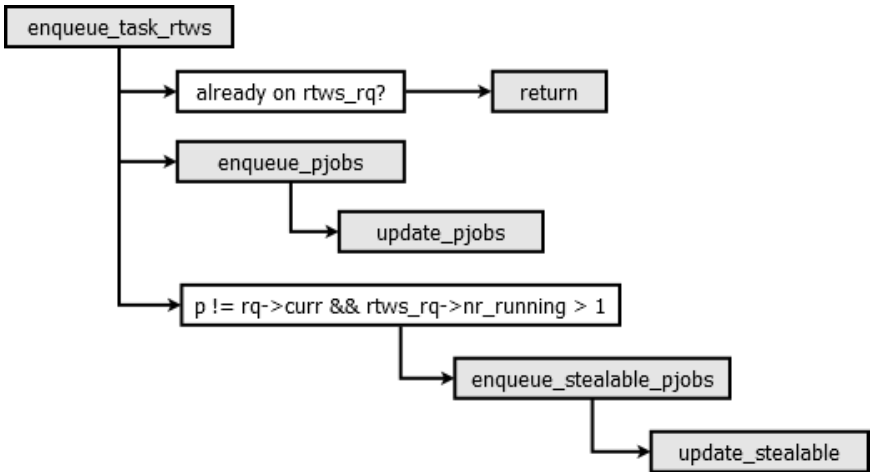


Figure 4.8: Code flow diagram for enqueue_task_rtws

If the task is already stored, nothing needs to be done. Otherwise, we proceed inserting the task on `rtws_rq` with `enqueue_pjobs`, where the scheduler takes the opportunity to update: (i) the `leftmost_pjob` in case the task at hand has higher priority; (ii) runqueue related statistics; and (iii) global information about current earliest task in this CPU. Then, if our queueing task is not being executed and there are at least two RTWS runnable tasks, we also add it to to the `stealable_pjob` red-black tree and analogously perform updates, so that it becomes available to be stolen.

Selecting the next task to run is performed in `pick_next_task_rtws`. This procedure is very similar to the theoretical design. The code flow diagram is shown in Fig. 4.9.

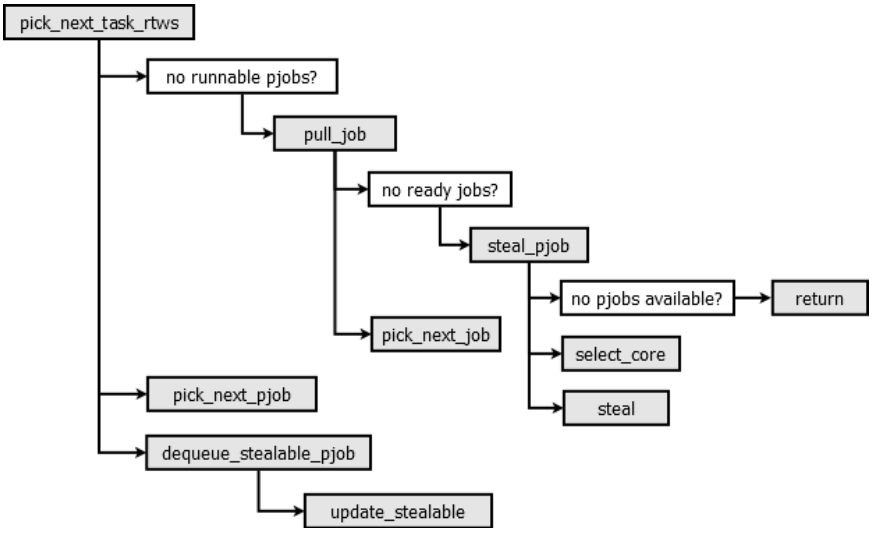


Figure 4.9: Code flow diagram for pick_next_task_rtws

If no RTWS tasks are currently pending on this CPU as indicated by an empty `nr_running` counter, the work is delegated to `pull_job` which retrieves a job from `global_rq` if its field `nr_running` is higher than zero; else `steal_pjob` is invoked. We give up and end all the process, passing the initiative to the real-time class, whether there is no eligible CPU for work-stealing. Otherwise, we choose the CPU victim according to `determinism` value, and steal the leftmost element from it. Note that it is implicit both pull methods being responsible for triggering selected task dequeuing on target, queueing on source, and then update data.

In contrast, if `leftmost_task` is available at first place, `pick_next_pjob` extracts `sched_entity_rtws` from that red-black tree. This is done using the `container_of` mechanism since any RTWS red-black tree manages instances of `rb_node` that are embedded in those scheduling entities. Now the task has been picked, but some more work is required to make it unavailable for stealing in order to prevent concurrent execution of the same process descriptor, which would crash the system. This is handled by `dequeue_stealable_pjob`.

Another key `sched_class`-specified function to respect RTWS rules is `put_prev_task_rtws` because it is its responsibility to dispatch tasks to the proper runqueues when they are withdrawn from CPU. Fig. 4.10 presents the code flow diagram.

If the task is not on `rtws_rq`, then we do nothing because it certainly finished its execution, and any necessary clean up or statistical accounting regarding task termination can be done in `task_dead_rtws`. Otherwise, a preemption occurred, and when we are dealing with a pjob spawned on this CPU, we call `enqueue_stealable_pjob` for the aforementioned reason. However, if that is not the case, we push task away to `global_rq` by invoking `dequeue_pjob` and `enqueue_job`, respectively. Although, in between those operations, we boost pjob status by setting `flags` to `RTWS_SPECIAL` if `stolen` equals to 1.

Before we look at how do we link the RTWS scheduler to user-space, a word must be said

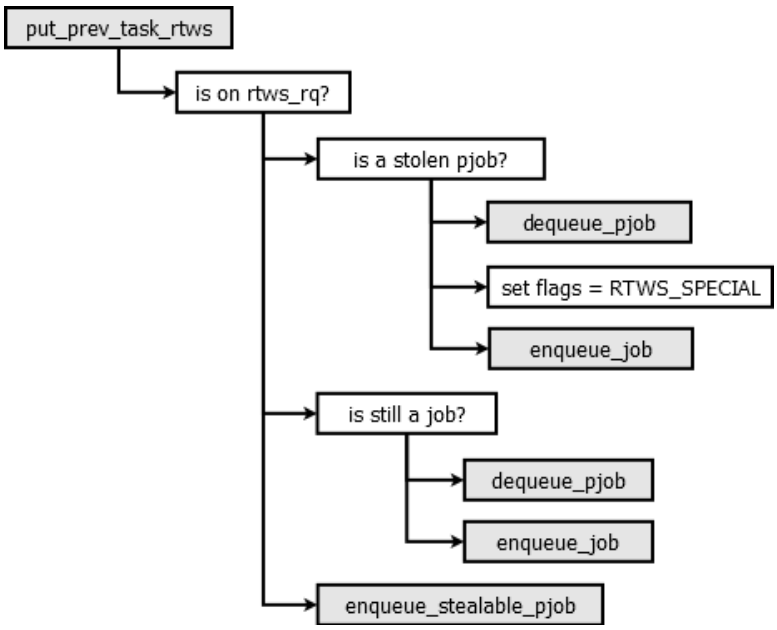


Figure 4.10: Code flow diagram for put_prev_task_rtws

about `set_curr_task_rtws`. While the content of other functions implemented by `sched_class_rtws` that have not deserved our attention is quite generic and, therefore, has low relevance, `set_curr_task_rtws` one is vital because it sets the absolute deadline for the first instance, which may start immediately (*i.e.* task offset is not defined), thus not triggered by the timer. Anyway, both timer callback and `set_curr_task_rtws` update scheduling parameters by calling `update_task_rtws`:

```
1  static inline void update_task_rtws(struct rq *rq,
                struct sched_rtws_entity *rtws_se)
3  {
     ...
5    atomic_inc(&rtws_se->job.nr); /* increment jobs counter */
     /* reseting flags */
7    rtws_se->nr_pjobs = 0;
     ...
9    /* update absolute deadline */
     rtws_se->job.deadline = rq->clock + rtws_se->rtws_deadline;
11 }
```

Listing 4.5: update_task_rtws function

Note that the job absolute deadline is always set as the sum of current time and relative deadline, not release time plus relative deadline as our system model states. We do so to avoid suffering from cumulative timer drift [Burns and Wellings, 2007]. While a production-quality RTS cannot take this shortcut because it would be cheating the true timing constraints (that's one of the reasons to use a RTOS), we just want to validate RTWS in practise. Moreover, SCHED_DEADLINE also follows this approach, and it is important that we set equals grounds as we will experimental compare both scheduling policies in Chapter 5.

4.4.3 System calls

A *system call* is the standard way of allowing user-space code to trigger kernel events in order to exploit the special capabilities of the kernel. They enable the kernel to be a transparent system layer from the view of user applications - it is always there but never really noticed. System calls in Linux are fast, mostly because the infrastructure is very efficient, but also due to their reduced number. Hence, adding a system call must be a thoughtful and last resort decision. Despite the internal kernel API is declared unstable as a design-feature, the external API cannot be broken under any circumstance. Therefore, once a system call is added neither it can ever be removed nor its signature can ever be changed.

In this project, we only had access to x86 hardware. In x86, we need to modify the file `arch/x86/kernel/syscall_table_32.S` in order to register new system calls. All new entries must be placed on the bottom of the list, so we do not break user-space compatibility by changing the unique identifier given to each system call. The system call name must be given the prefix `sys_`, whereas the function created to trap the interrupt must use a special macro where all input-arguments are specified. For instance, the correct macro for a system call requiring three arguments is `SYSCALL_DEFINE3`. The macro wraps the actual function, which for scheduler-related system calls is typically placed in `sched.c`. It must be said that for other platforms the process differs little, since the only required change is where we add the table-lookup address.

Each system call must inform the user application if its routine was executed and with which result. This is accomplished by means of its return code. Generally, negative return values denote an error, whilst positive return values (and 0) indicate successful termination. In order to copy data safely from user-space to kernel-space, and vice-versa, functions like `copy_from_user` and `copy_to_user`, respectively, must be used.

```
/**
 * sys_sched_setscheduler_ex - set/change the scheduler policy but with
    extended sched_param dedicate to real-time timing constraints.
 * @pid: the pid in question.
 * @policy: new policy.
 * @len: size of data pointed by param_ex.
 * @param: structure containing the extended parameters.
 */
SYSCALL_DEFINE4(sched_setscheduler_ex, pid_t, pid, int, policy,
    unsigned, len, struct sched_param_ex __user *, param_ex)
{
  if (policy < 0)
    return -EINVAL;

  return do_sched_setscheduler_ex(pid, policy, len, param_ex);
}
```

Listing 4.6: sched_setscheduler_ex system call

RTWS implementation provides three system calls:

1. **sched_setscheduler_ex():** initially, a RTWS task is created as any task in the system, using either fork or clone system calls. After that, it may change its policy by invoking the native `sched_setscheduler` system call. However, `sched_setscheduler` has no argument which supports real-time timing constraints parameters. `sched_setscheduler_ex` solves this issue by replacing the traditional `param` structure with a extended one (`param_ex`), where D_i, T_i and C_i can be specified. A description of the remaining arguments and the system call implementation can be found above, in Listing 4.6. Notice that all parameters validation and actual policy change are delegated to `do_sched_setscheduler_ex` which we is also not supported natively.

2. **sched_setsubpolicies_rtws():** allows one to change scheduling sub-policy for a particular task and stealing sub-policy for the overall system. It takes three arguments: `pid_t pid`, `int helpfirst`, and `int pas`. `pid` identifies the task in question, `helpfirst` replaces the given `task rtws_se->helpfirst`, whereas `pas` sets `global_rq->determinism`. As we have seen in Section 4.4.1, 0 and 1 are indeed the only acceptable values for these last two arguments, and the system call returns an error otherwise. By default both are set to 1, meaning that work-first and PAS are the sub-policies enforced.

3. **sched_delay_until_rtws():** is responsible for setting the current task's timer to expire at a specific point in time, and for putting the task to sleep until then. Therefore, this system call is intended to simulate a task periodicity. Since the periodic task model dictates that the first job release can be delayed by an offset, `sched_delay_until_rtws` provides an argument (`const struct timespec __user * release`) where the user can define the first release time. After that the kernel takes full control over the timing details, guaranteeing periodic correctness, so that the user just has to blindly invoke this system call at the end of each task instance. Any attempt to set the timer on the past, or invoke the system call on a parallel thread, will output an error. As soon as the timer expire, the task is woken up and the timer callback `new_job_rtws` is triggered.

4.5 Summary

In this chapter we presented the RTWS scheduler, which combines the G-EDF policy with a priority-based locality-aware work-stealing load balancing scheme, enabling parallel real-time tasks to run on more than one processor at a given time instant. We introduced the model that supports RTWS applications domain, we provided the nitty-gritty details and justifications about its design, which was guided with a distinct purpose: to bring predictability to the provably efficient work-stealing scheduling algorithm. At last, a thorough discussion concerning RTWS implementation in the Linux kernel was given, in which we focused on showing the differences between theory and practise.

Chapter 5

Experimental Evaluation

In the preceding chapter, we introduced the RTWS scheduler for heterogeneous real-time parallel tasks. As stated, theoretical and practical design decisions were taken to provide efficient scheduling decisions regarding dynamic intra-task parallelism, without jeopardising real-time guarantees, rather than increase the system's utilisation bound or boost the performance of applications. By efficient we mean a scheduling policy able to minimise implementation's sources of overhead.

Therefore, an overhead-aware evaluation of the proposed scheduling policy is required to assess its practicality. Naturally, we also have to evaluate if mixing real-time principles with parallel computing features is worthwhile. Thus, in this chapter, after we explain our experimental scenario, a discussion on the results collected is presented, mainly regarding two major sources of scheduling overhead: migrations and context switches. In order to have a comparison base, we present an evaluation of SCHED_DEADLINE as well, under the same circumstances. Furthermore, we investigate the scalability of our approach, and comment on the load balance discrepancy.

Note, however, that the target of the following analysis is not to prove that our RTWS implementation is better than other real-time policies because they serve different purposes. Moreover, a scheduling algorithm performance analysis may be influenced by a number of subtle events that affect how the system behaves, introducing unexpected noise in the collected data.

5.1 Scenario

The experiments reported in this thesis were conducted in a machine equipped with 16 GB of main memory and an eight-core processor, where each of the cores is running at 2.0 GHz. All assessments were carried out under both RTWS stealing sub-policies. Hereinafter, we use the terms RTWS-PAS and RTWS-BAS to distinguish the experiments. Every time we want to make no distinction, we simply use RTWS.

The Linux kernel 2.6.36 was configured as follows: disabled group scheduling, CPU frequency

scaling, hyper-threading, and tickless system; HZ macro set to 1000; preemptible kernel selected as preemption model. Since our evaluation is also based in a comparison to SCHED_DEADLINE (version 3), we have disabled bandwidth management on it to set equal grounds.

A set of three major experiments was conducted, where in each of the experiments twenty random task sets were used [Sousa et al., 2011], running in 2, 4 and 8 cores. In order to dynamically generate the task sets, we have defined the minimum task utilisation (u_{min}) equal to 0.1, the maximum task utilisation (u_{max}) equal to 0.5, a minimum period (T_{min}) of 700 ms, and a maximum period (T_{max}) of 800 ms. The period T_i of each task was computed as $T_i = T_{min} + x * (T_{max} - T_{min})$, where x denotes a random value between 0 and 1.

In order to analyse the scalability of the proposed approach with respect to the number of tasks/threads in the system, until the maximum system utilisation calculated by Equation 4.1 is reached, three utilisation windows ($[U_{\Pi min}, U_{\Pi max}]$) were chosen: $[0.38, 0.40]$, $[0.58, 0.60]$ and $[0.73, 0.75]$. The tightness of the chosen intervals is justified by the need to ensure similarities between task sets within the same experiment. With these parameters, we compute each task utilisation as follows: u_i is given by $u_i = u_{min} + x * (u_{max} - u_{min})$, where $\sum_{k=1}^{n} u_k \geq U_{\Pi min}$ and $\sum_{k=1}^{n} u_k \leq U_{\Pi max}$. Finally, C_i is given by $C_i = T_i * u_i$.

The number of parallel threads per task was dynamically derived as $n_i = x*(m*2)$, whereas the number of tasks (n) was totally dynamic, based on the system utilisation window condition being satisfied (please refer to Table 5.1). Note that as we keep increasing $U_{\Pi max}$, and u_{max} remains constant, n scales. We strongly believe that these parameters can deeply assess our scheduler features.

Table 5.1: Composition of each experiment

m	Total tasks			Total threads		
	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%
2	50	82	98	128	218	216
4	102	158	193	457	703	866
8	217	320	401	1736	2720	3491

Each task was a simple fork-join application whose actual work was limited to a series of NOP instructions to avoid memory and cache interferences. Even though RTWS is specially designed to explore data locality, we let that aside because we will not evaluate cache misses. Each of the task's jobs (i) executes sequentially; (ii) splits into multiple parallel threads; and (iii) synchronises at the end of the parallel region, resuming the execution of the master thread. Sequential, parallel, and total execution times were derived randomly, with the actual total execution time upper bounded by C_i.

5.2 Overheads

Data was collected and averaged concerning the number of context switches and migrations, parameters which represent the main sources of scheduling overhead. Fig. 5.1 depicts the average number of migrations that occurred for each scheduling policy when all cores were online. In the case of RTWS, the number of migrations refers to the number of steals performed by the idle

cores, while the values collected for SCHED_DEADLINE refer to pure migrations that occurred between cores.

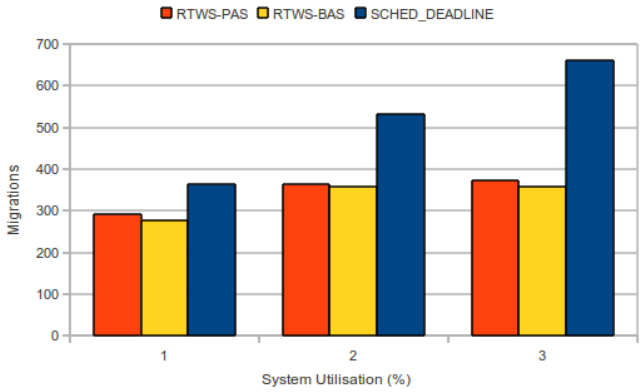


Figure 5.1: Average number of migrations on the 8-core experiments

The overall results show that RTWS outperforms SCHED_DEADLINE in every experiments. These results can be explained by our decision to favour data locality, generating parallelism only when strictly required, *i.e.* when a core becomes idle. In fact, the results are far better for medium/high workloads since load balancing calls are more frequently required on SCHED_DEADLINE with the greater number of tasks. Remarkably, the number of migrations barely increases on RTWS under such heavy circumstances. For lower workloads, the difference becomes slighter mainly because on our scheduling policy the system lacks parallel threads to keep all cores busy. Surprisingly, RTWS-PAS caused more migrations than RTWS-BAS; we expected it to be the other way around due to the lesser contention time RTWS-BAS is subject. However, the difference is so small we cannot conclude anything but blame Linux kernel's predictability gap.

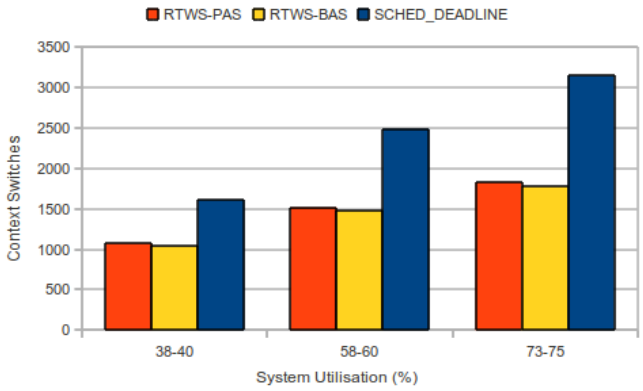


Figure 5.2: Average number of context switches on the 8-core experiments

Regarding the average number of context switches, depicted in Fig. 5.2, no matter the considered workload rate, RTWS also outperforms SCHED_DEADLINE on the eight-core experiments. SCHED_DEADLINE blindly assigns new jobs of a task to the core where the last job of that task was executed, which rather frequently leads to a preemption of the running job. Contrariwise, in RTWS, preemptions are minimised because a released job is assigned to a idle core (if available)

or inserted into the global queue when its priority is lower than the ones currently executing. Moreover, we do not allow parallel threads to preempt other threads or jobs, unless they have been stolen. Even though the number of context switches increases with higher system utilisations, values indicate a less than linear scalability for both policies, which can be seen as a good behaviour. Stealing sub-policies have no impact on the number of context switches besides the one directly related with the variance on the number of migrations. Hence, it is easily understandable why RTWS-PAS is shown to trigger more context switch operations. There is no need to blame Linux kernel again.

5.3 Scalability

Before analysing the scalability results introduced by Tables 5.2 and 5.3, let us clarify that the RTWS-PAS two-core experiments are treated as the base case and, therefore, every other single experiment relates to that base case resulting in a factor - the scale up ratio. For example, a scale up ratio of 2 means that the considered metric has doubled. Furthermore, note that this kind of scalability is strictly and peculiarly linked to the values presented in Table 5.1 because the amount of tasks and parallel threads has greater impact on the number of migrations, and context switches, than a core increase itself.

Table 5.2: Scale up ratios on number of migrations

m	RTWS_PAS			RTWS_BAS			SCHED_DEADLINE		
	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%
2	1	1	1	1.13	1.10	1.08	2.75	3.27	3.08
4	5.88	5.55	5.92	6.13	5.55	5.67	11.38	12	13
8	36.38	33	31.08	34.75	32.46	29,83	45.38	48.36	55.08

According to the values reported in Table 5.2, it becomes crystal clear that the obtained results suffered from some unexpected noise: even in the two-core experiments, where stealing randomly or deterministically produces the same outcome, differences between RTWS-PAS and RTWS-BAS can be noticed.

Still, considering the properties of our experiments, one can conclude that the number of migrations is largely influenced by the number of dynamically generated parallel threads. Provided that we create more tasks when m is increased, the number of threads exponentially grows as can be easily seen in Table 5.1. Nonetheless, this growth factor is not directly proportional to the scale up ratio. Note the reaction triggered by C_i being constant in every experiment: the more we parallelize, the less executing time will be assigned to each thread, faster threads will finish, migrations will scale.

Thereby, we have to multiply the ratio of the system's total number of threads by the ratio of each task's maximum number of threads to be able to find the linear scalability value. For example, for $m = 4$ and a utilisation interval $[0.38, 0.40]$, the scale up ratio is expected to be $\frac{457}{128} * \frac{8}{4} = 7.14$. After analogously calculating for the remaining cases, it is clear that RTWS efficiently scales as respects to the number of migrations.

Under G-EDF, context switches occur either when a job is released or when it completes.

Table 5.3: Scale up ratios on number of context switches

m	RTWS_PAS			RTWS_BAS			SCHED_DEADLINE		
	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%
2	1	1	1	1.01	1.02	0.99	1.35	1.40	1.33
4	2.91	2.80	3	2.89	2.78	2.95	4.43	4.45	4.74
8	10.60	9.97	10.75	10.26	9.70	10.43	15.93	16.36	18.51

However, not every job release will swap the currently executing job. Thus, the number of context switches over a time interval of length L is upper bounded by twice the number of jobs' releases during that interval. As every experiment has lasted exactly the same time and its periodicity parameters were constant, the scale up ratio on the number of jobs is given by the scale up ratio on the number of tasks. Intuitively, for $m = 4$, RTWS scales in a very efficient manner, as Table 5.3 reflects, since there are approximately twice more tasks (*e.g.* $\frac{158}{82} = 1.93$) but the scale up ratios on the number context switches are lower than the upper bounded value of 4.

Following the same logic, for $m = 8$ our scheduling algorithm appears to scale poorly because the amount of tasks is almost four times higher ($\frac{217}{50} \approx \frac{320}{82} \approx \frac{401}{98} \approx 4$). Nevertheless, recall that, in RTWS, stolen parallel threads may also preempt any schedulable entity, plus we still have to account each thread's completion as a context switch, seriously inflating the upper bounded scale up ratio from G-EDF. In this case, it is particularly noticeable by having to dispatch an incredibly high number of threads, which in turn also potentiates work-stealing (please refer to Table 5.1 and Fig. 5.2 again).

It must be said that scalable efficiency by itself is meaningless in a RTS. That is, it does not really matter if a real-time scheduling algorithm has negligible overhead but is unable to meet all deadlines. Oppositely, a overhead increase is justified by a gain in schedulability. However this observation does not hold for this experimental analysis. Besides being overhead-aware, both RTWS-PAS and RTWS-BAS did not miss any deadline, whilst SCHED_DEADLINE missed a couple.

5.4 Load imbalance

In a RTS, load balancing is not a requirement. As long as a scheduler delivers predictability to be able to scheduled every feasible task set, it could even execute all tasks in a single processor. Nonetheless, for several reasons, included energy-wise which is of paramount importance specially on embedded systems, it is preferable that real-time schedulers assure both. Fig. 5.3 shows the average load imbalance (in terms of overall execution times) registered in the 8-cores experiments.

Although for low system utilisations RTWS is unable to distributed the workload with more efficiency than SCHED_DEADLINE, the reporting results are quite interesting. Among other things, work-stealing is known as a load balancing scheme for parallel computations, so at first glance it might be hard to understand why, in some scenarios, it fails to overcome a real-time scheduling policy which has no particular feature addressing intra-task parallelism. Well, the answer is not on the work-stealing design but on our implementation. Since Linux scheduler is a modular framework, every time RTWS, on a certain CPU, fails to find a *stealable* task, it passes the lead

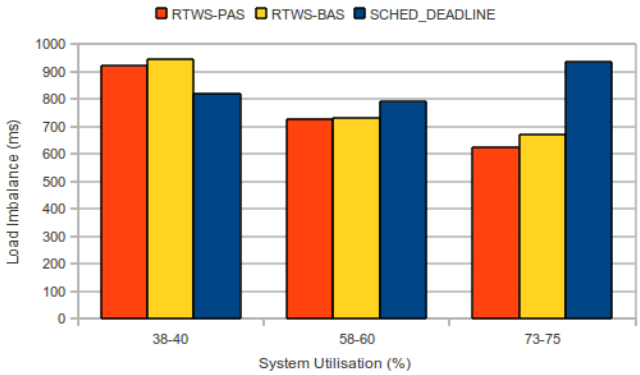


Figure 5.3: Average load imbalance on the 8-core experiments

to the other scheduling classes, which have no knowledge about the global status of RTWS run-queues. RTWS is invoked again only when RTWS tasks are assigned to that same CPU. Naturally, this results in scheduling downtime when the work available is scarce.

5.5 Response time

The experimental results presented so far where inconclusive in order to understand if one of the proposed sub-policies outshines the other, or whether they offer a fair trade-off between determinism and low lock contention. Therefore, in this section we turn our attention to evaluating tasks' response time.

As we have seen early in Chapter 2, response time denotes the time elapsed between the moment a job becomes ready to be scheduled, and the moment when it finishes its execution. Therefore, when we consider a task's response time using its average value, we get an idea about how efficiently that task is being executed (parallel vs sequential performance). On the other hand, when a task's response time is measured by its worst-case value, it becomes clear how strictly the schedule is being respected and how far from the deadline that task is (*laxity*).

In this sense, we have measured both ways of perceiving a task's response time, not only for RTWS-PAS and RTWS-BAS, but also for a pure G-EDF approach, ignoring intra-task parallelism and executing sequentially for an equivalent amount of time. For the remainder of this section, we refer to this last approach as RTWS-SEQ, and it will enlighten us whether generating short-living threads and scheduling them under RTWS is worthwhile for real-time systems.

The obtained results are depicted in Tables 5.4 and 5.5. Note that, in both tables, a scale up ratio of 0.5 means that the considered metric has reduced to an half comparatively to the base case which is RTWS-SEQ two-core experiments.

Table 5.4: Scale up ratios on the average response time

m	RTWS-SEQ			RTWS-PAS			RTWS-BAS		
	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%
2	1	1	1	0.87	0.88	0.87	0.87	0.88	0.87
4	1.04	0.99	0.97	0.80	0.82	0.80	0.82	0.82	0.81
8	0.98	0.98	0.96	0.69	0.73	0.75	0.70	0.73	0.76

It was expected that, when considering average response times, RTWS-BAS would outperform RTWS-PAS. However, once again, both sub-policies show identical results all over the experiments. By now, it is safe to say that the conducted experiments do not led to concurrent stealing operations as many times as we wished. Nevertheless, RTWS always achieves better performance when intra-task parallelism is expressed than when sequential execution is considered. Moreover, its performance increases as more cores become available for work-stealing. This allow us to conclude that RTWS provides an efficient scheduling environment for fine-grained parallel real-time tasks.

Table 5.5: Scale up ratios on the worst-case response time

m	RTWS-SEQ			RTWS-PAS			RTWS-BAS		
	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%
2	1	1	1	0.91	0.91	0.91	0.91	0.91	0.91
4	1.04	1.01	0.99	0.93	0.89	0.89	0.93	0.90	0.90
8	0.99	1.01	0.99	0.87	0.91	0.91	0.89	0.91	0.91

Following the previous reasoning, similarities between RTWS-BAS and RTWS-PAS prevail also for worst-case response times. Nevertheless, worst-case response times relate to a wiser choice of the victim core when considering task priorities since the earliest deadline ready thread has less flexibility to support waiting times. Also, unlike average response times, worst-case response times do not scale since, with work-stealing, we do not force parallelism, but instead it only takes place only when a core would otherwise be idle. Recall that we favour predictability over performance. Yet, the multi-threaded version of the experiments always outperforms its sequential counterpart.

One final note about the obtained results with RTWS-SEQ. The experimental results were almost constant, but not strictly identical, because there are no threads involved and the number of tasks n adapts to the increasing number of cores m.

5.6 Summary

In this chapter, we presented the experimental results collected from the Linux kernel 2.6.36, regarding dynamic generated task sets running under RTWS-PAS and RTWS-BAS. An overhead-aware and a scalability evaluation were discussed by comparing RTWS to SCHED_DEADLINE. RTWS was shown to outperform the latter scheduling policy, and to efficiently schedule the experiments provided, at least up to 8 cores. However, due to the underlying OS's unpredictability, no conclusions about the impact of the two stealing sub-policies can be taken.

Furthermore, by assessing the load imbalance of both schedulers, we found that RTWS implementation needs to be tweaked in order to be pro-active concerning the work-stealing strategy. RTWS was also shown to provide better performance when considering tasks with intra-task parallelism than without, through response time analysis.

Chapter 6

Conclusion

High-level parallel languages offer a simple way for application programmers to specify parallelism in a form that easily scales with problem size, leaving the scheduling of the tasks onto processors to be performed at runtime. This thesis demonstrated how to schedule highly heterogeneous parallel applications that require real-time performance guarantees on multi-core processors. In contrast to prior work on real-time scheduling of parallel workloads, a more general model of parallel real- time tasks where dynamically generated threads can take arbitrarily different amounts of time to execute was considered.

This chapter resumes its most relevant contributions and highlights some lines of future work.

6.1 General conclusions

Modern RTSs increasingly generate heavy and highly varying workloads and it is rapidly becoming unreasonable to expect to implement them as single core systems. In fact, a general shift from single to multi-core processors can be seen both in the general purpose and embedded domains as an energy-efficient way to boost applications' performance.

Simultaneously, the proliferation of multi-core platforms have transformed parallelism into a main concern, and dynamic task-level parallelism is steadily gaining popularity as a programming model. The idea behind that model is to encourage application developers to expose every opportunity for parallelism by just pointing out potentially parallel regions within the code. All annotations provided act simply as hints that can be ignored and safely replaced with sequential counterparts by the language implementation. Hence, how computations are actually decomposed and mapped to processors is the responsibility of the compiler and runtime systems.

By easing the developer from this burden, programming complexity is considerably reduced, which usually translates in increased productivity. Nevertheless, if the scheduling mechanism underneath is not simple and fast to keep the overall overhead low, such fine-grained parallelism is not worthwhile, and all benefits will be lost.

From a scheduling perspective, work-stealing algorithms are increasingly popular, and are

considered a promising approach to address the software challenge in the ongoing trend for massive parallelism due to their provably time, space, and communication efficiency. However, they do not contemplate timing constraints or any other form of prioritising tasks, which prevents them for being applied to a RTS. Moreover, they are traditionally employed on the language runtime, creating a two-level scheduling system, where predictability cannot be ensured.

In this thesis, we described how work-stealing can be redesigned to fulfill real-time requirements, maintaining its basic principles. Long-story short, conventional deques are replaced by a queue of deques ordered by increasing priority. We further applied the well-known G-EDF policy on top of it, mixed the rules, and RTWS was born.

Taking advantage of the modularity offered by the Linux scheduler, we added RTWS to it as a new scheduling class, in order to practically assess if our approach is viable (*i.e.* provides efficiency and schedulability). Enhance the Linux kernel is a tremendous task, due to the complexity of the kernel internals and high interdependence between various subsystems. Nevertheless, we wanted to make sure RTWS is more than a interesting concept. Moreover, despite Linux is not a RTOS, it supplies the tools and documentation we needed to get started, and is open-source. A representative part of this thesis was dedicated to discuss RTWS implementation, and state issues like synchronisation drifts that are not address in theory.

Experimental results showed that RTWS, in comparison to other practical work, significantly reduces the scheduling overhead through an efficient and scalable (at least up to 8 cores) control of migrations and context switches, while still achieves good dynamic load balancing even with low communication costs. Furthermore, RTWS was also shown to provide better performance when considering tasks with intra-task parallelism than without, even for short-living computations. However, during evaluation we realised that RTWS implementation has a flaw, causing unacceptable scheduling downtime when the system utilisation is low.

Although we focused on keeping the overhead low and on achieving good data locality, system's schedulability was never neglect by us. In fact, our scheduling algorithm proved to be very robust as we did not get any deadline miss on the performed experiments. Therefore, we can pronounce that some priority inversion caused by the BAS stealing sub-policy does not compromise the schedulability goals, and it even helps to reduce contention as well as to keep global accounted information to a minimum. Yet, RTWS supports a deterministic stealing sub-policy: PAS. The experimental evaluation did not help to have a clear picture about PAS and BAS consequences.

All in all, we can conclude that RTWS is a promising solution to efficiently schedule highly heterogeneous and dynamic parallel real-time tasks, assuming the restrictions defined in our system model.

6.2 Summary of the main contributions

In contrast to prior work on real-time scheduling of parallel tasks, this thesis considered a more general and portable model of parallel real-time tasks, where dynamically spawned threads may take arbitrarily different amounts of time to execute. That is, any task may be composed by sev-

eral sequential and parallel regions, where each parallel regions may contain an arbitrary number of threads (is not limited to the cores count), and each one of those threads may have arbitrarily different execution needs.

Targeting the aforementioned model, we proposed RTWS, a novel scheduling algorithm that combines the G-EDF scheduler with a priority-based locality-aware work-stealing load balancing policy, allowing parallel real-time tasks to be executed in more than one processor at a given time instant. The ultimate goal is to provide efficient low-level support for the scheduling of parallel real-time, mixing real-time determinism and predictability with work-stealing space and communication awareness.

Towards this, we implemented RTWS in the standard Linux kernel just as a proof of concept, since Linux is not a RTOS and, therefore, is not reliable for time-sensitive applications. To the best of our knowledge, we are the first to: (i) deal with real-time priorities (deadlines) in a work-stealing scheduler; and (ii) to actually implement support for parallel real-time computations in the Linux kernel. Last but not least, this research work has resulted in two scientific publications.

6.3 Future work

The research on this topic is all but over. First of all, we will address the implementation flaw detected. One possible way to sort things out is to retry the steal operation for a static pre-defined number of times. Another important topic is to come up with a solution, both theoretical and practical, for the nested parallelism limitation. One possible direction is to consider parallel multi-threaded tasks to be represented as a Directed Acyclic Graph (DAG) where nodes represent threads and edges represent dependences between those threads. In its current state, RTWS does not support this task model. If a stolen thread is able to spawn new threads on a CPU different than the one who assured its schedulability, whenever a preemption occurs it would be too costly to move them all to the global queue.

Furthermore, several improvements on efficiency of the presented implementation, namely lock acquisition points and data structures, should be deeply studied to further reinforce our results. A key change would be to port RTWS to recent Linux kernel versions and apply it on top of PREEMP_RT patch set.

Many more metrics, such as cache misses and latencies, are possible to be collected. Numerous experimental analyses should be considered to clarify how the peculiarities of each task set may influence our scheduler goodness or, at least, enlighten about which stealing sub-policy suits better a generic RTS. It would also be of great interest to test real-world applications to see if RTWS misbehaves.

With the complexity of multi-core systems growing, it may be interesting to evaluate RTWS in large multi-core systems that are likely to have hierarchical cache layouts. One possible extension to RTWS for such systems could be a scheduling approach that mixes aspects of partitioning and global scheduling. In particular, while task migrations within a cluster of cores that share some lower level cache might be acceptable, migrations among processors that are “far apart” in the cache hierarchy may be too expensive.

Bibliography

Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, page 4, Madrid, Spain, December 1998.

Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Theory of Computing Systems*, pages 1–12, 2000.

Kunal Agrawal, Charles E. Leiserson, Yuxiong He, and Wen Jing Hsu. Adaptive work-stealing with parallelism feedback. *ACM Trans. Comput. Syst.*, 26(3):7:1–7:32, September 2008.

James H. Anderson and John M. Calandrino. Parallel real-time task scheduling on multicore platforms. In *PROC. OF THE 27TH IEEE REAL-TIME SYSTEMS SYMP*, pages 89–100. IEEE, 2006.

Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *In Proc. 22nd IEEE Real-Time Systems Symposium*, pages 193–202. Society Press, 2001.

OpenMP ARB. Openmp. Available at `http://www.openmp.org/`.

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th annual ACM symposium on Parallel algorithms and architectures*, pages 119–129, New York, NY, USA, 1998. ACM.

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

T.P. Baker. An analysis of edf schedulability on a multiprocessor. *Parallel and Distributed Systems, IEEE Transactions on*, 16(8):760 – 768, aug. 2005.

S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

Abhishek Bhattacharjee, Gilberto Contreras, and Margaret Martonosi. Parallelization libraries: Characterizing and reducing overheads. *ACM Transactions on Architecture and Code Optimization*, 8(1):5:1–5:29, February 2011.

Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, March 1999.

Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the 25th ACM symposium on Theory of computing*, pages 362–371, New York, NY, USA, 1993. ACM.

Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. ISBN 0596005652.

Björn B. Brandenburg and James H. Anderson. On the implementation of global real-time schedulers. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, pages 214–224, Washington, DC, USA, 2009. IEEE Computer Society.

Alan Burns and Andy Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, New York, NY, USA, 3rev ed edition, 2007. ISBN 0521866979, 9780521866972.

Giorgio C. Buttazzo. Rate monotonic vs. edf: judgment day. *Real-Time Syst.*, 29(1):5–26, January 2005.

John M. Calandrino and James H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*, ECRTS '09, pages 194–204, Washington, DC, USA, 2009. IEEE Computer Society.

John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. Litmus^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 111–126, 2006.

John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in cool. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 249–259, New York, NY, USA, 1993. ACM.

David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 21–28, 2005.

Antoine Colin and Stefan M. Petters. Experimental evaluation of code properties for wcet analysis. In *Proceedings of the 24th IEEE RTSS*, pages 190–199, December 2003.

Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106:180–187, May 2008.

Intel Corporation. Parallel building blocks. Available at `http://software.intel.com/en-us/articles/intel-parallel-building-blocks/`, a.

Microsoft Corporation. Task parallel library. Available at `http://msdn.microsoft.com/en-us/library/dd460717.aspx`, b.

Umamaheswari C. Devi and J. H. Anderson. Tardiness bounds under global edf scheduling on a multiprocessor. *Real-Time Syst.*, 38(2):133–189, February 2008.

Sudarshan Kumar Dhall. *Scheduling periodic-time - critical jobs on single processor and multiprocessor computing systems.* PhD thesis, Champaign, IL, USA, 1977. AAI7714943.

Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23:1369–1386, 2012.

Xiaoning Ding, Kaibo Wang, Phillip B. Gibbons, and Xiaodong Zhang. Bws: balanced work stealing for time-sharing multicores. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 365–378, New York, NY, USA, 2012. ACM.

Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White, editors. *Sourcebook of parallel computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 1-55860-871-0.

Dario Faggioli, Michael Trimarchi, and Fabio Checconi. An implementation of the earliest deadline first algorithm in linux. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1984–1989, March 2009.

José Carlos Fonseca, Luís Nogueira, Cláudio Maia, and Luís Miguel Pinho. Real-time scheduling of parallel tasks in the linux kernel. In *Proceedings of the 4th INForum*, Lisbon, Portugal, September 2012.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, 1998.

D. D. Gajski and Jib-Kwon Peir. Essential issues in multiprocessor systems. *Computer*, 18(6):9–27, June 1985.

Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. ISBN 0716710455.

Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems Journal*, 25:187–205, September 2003.

Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, April 2010.

C.-C. Han and K.-J. Lin. Scheduling parallelizable jobs on multiprocessors. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 59 –67, dec 1989.

Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. *Distributed Computing*, 18:189–207, February 2006.

S. F. Hummel and E. Schonberg. Low-overhead scheduling of nested parallelism. *IBM J. Res. Dev.*, 35(5-6):743–765, September 1991.

Klaus Jansen. Scheduling malleable parallel tasks: An asymptotic fully polynomial time approximation scheme. *Algorithmica*, 39(1):59–81, January 2004.

D. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Trans. Softw. Eng.*, 19(9):920–934, September 1993.

S. Kato and Y. Ishikawa. Gang edf scheduling of parallel task systems. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 459 –468, December 2009.

Leonard Kleinrock. *Queueing Systems*, volume II: Computer Applications. Wiley Interscience, 1976. (Published in Russian, 1979. Published in Japanese, 1979.).

K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 259 –268, December 2010.

Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, 2000.

Wan Yeon Lee and Heejo Lee. Optimal scheduling for real-time parallel tasks. *Transactions on Information and Systems*, E89-D:1962–1966, June 2006.

J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 201–209, 1990.

Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Priority queues are not good concurrent priority schedulers. Technical Report TR-11-39, The University of Texas at Austin, Department of Computer Sciences, November 2011.

C. L. Liu. Scheduling Algorithms for Multiprocessors in a Hard Real-Time Environment. *JPL Space Programs Summary 37-60*, II:28–31, 1969.

C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1(20):40–61, 1973.

J. M. López, M. García, J. L. Díaz, and D. F. García. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. In *Proceedings of the 12th Euromicro conference on Real-time systems*, Euromicro-RTS'00, pages 25–33, Washington, DC, USA, 2000. IEEE Computer Society.

Walter Ludwig and Prasoon Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, SODA '94, pages 167–176, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: a survey and synthesis. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, HICSS '95, pages 61–, Washington, DC, USA, 1995. IEEE Computer Society.

G. Manimaran, C. Siva Ram Murthy, and Krithi Ramamritham. A new approach for scheduling of parallelizable tasks inreal-time multiprocessor systems. *Real-Time Systems Journal*, 15:39–60, July 1998.

Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, UK, 2008. ISBN 0470343435, 9780470343432.

A.K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

Girija J. Narlikar. Scheduling threads for low space requirement and good locality. In *In Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 83–95, 1999.

Girija J. Narlikar and Guy E. Blelloch. Pthreads for dynamic and irregular parallelism. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–16, Washington, DC, USA, 1998. IEEE Computer Society.

Angeles Navarro, Rafael Asenjo, Siham Tabik, and Călin Caşcaval. Load balancing using work-stealing for pipeline parallelism in emerging applications. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 517–518, New York, NY, USA, 2009. ACM.

Daniel Neill and Adam Wierman. On the benefits of work stealing in shared-memory multiprocessors. Technical report, Department of Computer Science, Carnegie Mellon University, 2009.

Luís Nogueira, José Carlos Fonseca, Cláudio Maia, and Luís Miguel Pinho. Dynamic global scheduling of parallel real-time tasks. In *Proceedings of the 10th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, Paphos, Cyprus, December 2012.

Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 140–149, New York, NY, USA, 1997. ACM.

C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, December 1987.

M.J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill computer science series: Networks–parallel and distributed computing. McGraw-Hill, 1994. ISBN 9780070512948.

Bratin Saha, Ali-Reza Adl-Tabatabai, Anwar Ghuloum, Mohan Rajagopalan, Richard L. Hudson, Leaf Petersen, Vijay Menon, Brian Murphy, Tatiana Shpeisman, Eric Sprangle, Anwar Rohillah, Doug Carmean, and Jesse Fang. Enabling scalability and performance in a large scale cmp environment. *ACM SIGOPS Operating Systems Review*, 41(3):73–86, June 2007.

Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 217 –226, Vienna, Austria, December 2011.

Claudio Scordino and Giuseppe Lipari. Linux and real-time: Current approaches and future opportunities. In *IEEE International Congress ANIPLA*, 2006.

L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronisation. *IEEE Transaction on Computers*, 39(9):1175–1185, 1990.

Lui Sha, Tarek Abdelzaher, Karl-Erik Arzen, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Syst.*, 28(2-3):101–155, November 2004.

Michael Short. Improved task management techniques for enforcing edf scheduling on recurring tasks. In *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '10, pages 56–65, Washington, DC, USA, 2010. IEEE Computer Society.

David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, June 1998.

Paulo Baltarejo Sousa, Björn Andersson, and Eduardo Tovar. Implementing Slot-Based Task-Splitting Multiprocessor Scheduling. In *of 6th IEEE International Symposium on Industrial Embedded Systems (SIES 11)*, 2011.

Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Inf. Process. Lett.*, 84(2):93–98, October 2002.

John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. Stackthreads/mp: integrating futures into calling standards. *ACM SIGPLAN Notices*, 34(8):60–71, 1999.

John Turek, Uwe Schwiegelshohn, Joel L. Wolf, and Philip S. Yu. Scheduling parallel tasks to minimize average response time. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, SODA '94, pages 112–121, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

P. Valente and G. Lipari. An upper bound to the lateness of soft real-time tasks scheduled by edf on multiprocessors. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 311–320, December 2005.

Željko Vrba, Håvard Espeland, Pål Halvorsen, and Carsten Griwodz. Limits of work-stealing scheduling. In *Proceedings of the 14th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 280–299, May 2009.

Zeljko Vrba, Paal Halvorsen, and Carsten Griwodz. A simple improvement of the work-stealing scheduling algorithm. In *Proceedings of the 4th International Conference on Complex, Intelligent and Software Intensive Systems*, pages 925–930, February 2010.

Qingzhou Wang and Kam Hoi Cheng. A heuristic of scheduling parallel tasks and its analysis. *SIAM J. Comput.*, 21(2):281–294, April 1992.